Process Spaces [†]

Radu Negulescu

Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada, N2L 3G1

radu@maveric.uwaterloo.ca
ftp://cs-archive.uwaterloo.ca/cs-archive/CS-95-48/CS-95-48.ps.Z

December 1995

Abstract

This paper introduces process spaces, a unified theory of interacting systems. The main new trait, abstract executions, leads to a simple and general set formalism. For concurrent systems (including digital circuits), process spaces apply to diverse correctness concerns and yield a new classification of liveness and progress faults. The resulting studies of different correctness concerns are decoupled and homogeneous (i.e., they do not interfere with each other and they have the same algebraic structure). Applications to other interacting systems, such as electrical networks and dynamical systems, are also possible. Process spaces have many meaningful properties; here, some results from concurrency theory are generalized and simplified, and some new results are found.

1 Introduction

Concurrent systems are practically everywhere; some examples are digital circuits (synchronous and asynchronous), distributed programs and data structures, communication protocols, and work flows in a factory.

By *interacting systems* we mean systems that can be coupled and compared. This paper introduces *process spaces*, a unified theory of interacting systems, including concurrent systems as particular cases. Also, this paper starts to investigate the applications and the algebraic properties of this theory.

[†] This research was supported by a grant and a scholarship from the Information Technology Research Centre of Ontario, by an Ontario Graduate Scholarship, and by Grant No. OGP0000871 from the Natural Sciences and Engineering Research Council of Canada.

In process spaces, the notion of 'execution' is abstract, and is a primitive notion. An execution is not necessarily a sequence of events, a function of time, etc.; a priori, an execution has no structure. As a result, process spaces may model the entire spectrum of discrete-state, continuous-state, and hybrid systems, and allow for discrete and continuous time as well. Another consequence of abstract executions is a simple set-theoretic formalism. Many meaningful properties can be simply checked by Venn diagrams.

For concurrent systems, process spaces apply to diverse correctness concerns: safety, liveness, progress, timing, and even absence of dangling inputs. These applications can be obtained as separate instances of the same theory, by choosing a suitable type of execution for each correctness concern. These applications are completely decoupled (e.g., the study of liveness has no safety or connectivity restrictions) and homogeneous (i.e., they have precisely the same algebraic properties). A comparison between the process space studies of liveness and progress yields a formal classification of liveness and progress faults (called 'lock faults').

Process spaces have strong relationships with several theories of concurrency (see Section 3); however, we are not aware of previous work with any of the characteristics mentioned above.

We first present the process space formalism (Section 2). Then, we show how process spaces can be used to study the behavior of concurrent systems; we provide several examples and a classification of lock faults (Section 3). We give several algebraic properties of process spaces and discuss some of their practical and theoretical significance; some results from concurrency theory are generalized and simplified (Section 4). We discuss further applications to electrical networks, dynamical systems, behavior analysis in cases where ports can change direction (input/output lines), a connectivity concern for concurrent systems (input control), and frameworks for studying timing and true concurrency (Section 5). Finally, we summarize the contributions of this paper and their significance, and we indicate directions for further work (Section 6).

2 The Process Space Formalism

In this section, we present the main concepts of process spaces. We offer some intuition, but no justifications. The justification for this formalism is 'because it works'. This will be substantiated in the rest of the paper, where particular examples of applications and interpretations are given. We recommend that the readers consider at least the examples in Subsections 3.2, 5.1, and 5.4 in order to understand how the model can be interpreted and applied. However, we chose to omit examples from this section, because the model itself is quite simple, and because we should not create a bias towards the types of systems which are more familiar to the author.

Our model consists of the following definitions of execution, process, and process space. Let $\mathcal{E}$ be an arbitrary set; we refer to the elements of $\mathcal{E}$ as *executions*. A *process* over $\mathcal{E}$ is a pair (X, Y) of subsets of $\mathcal{E}$ such that $X \cup Y = \mathcal{E}$. The set of all processes over $\mathcal{E}$ is called the *process space* of $\mathcal{E}$ and is denoted by $S_{\mathcal{E}}$.

Our main intuition for this model is the agreement pattern presented below; particular interpretations are discussed in other sections. A process can describe a device by means of an *agreement* between the device and its environment, regarding executions. Notice that a process partitions $\mathcal{E}$ into three disjoint sets: $\overline{X}$, $X \cap Y$, and $\overline{Y}$ (where $\overline{\ }$ denotes the complement with respect to $\mathcal{E}$). The agreement stipulates that only executions from $X \cap Y$ are allowed to occur in the presence of the device. Nevertheless, all executions in $\mathcal{E}$ are considered to be possible: all can occur, although some may be forbidden by the agreement. The agreement also qualifies the executions that are not allowed to occur, by assigning the 'blame' to either the device or the

environment. Set $\overline{X}$ contains the executions in which the device violates the agreement, while set $\overline{Y}$ contains the executions in which the environment violates the agreement. Accordingly, X contains executions where the device respects the agreement (but the environment may or may not violate it), while Y contains executions in which the environment respects the agreement (but the device may or may not violate it). Note that the case where both the device and the environment violate the agreement is not possible, because the condition $X \cup Y = \mathcal{E}$ implies $\overline{X} \cap \overline{Y} = \varnothing$. Thus, the 'blame' cannot be assigned to both the device and the environment for the same execution.

We denote processes by p, q, …. For process $p = (X, Y)$, we use the following terminology and notation (see Figure 1):

$\mathbf{as}\,p = X$ = the *accessible* set of p,
$\mathbf{at}\,p = Y$ = the *acceptable* set of p,
$\mathbf{v}p = \overline{X} \cup \overline{Y}$ = the *violation* set of p,
$\mathbf{r}p = \overline{Y}$ = the *reject* set of p,
$\mathbf{c}p = X \cap Y$ = the *contract* set of p,
$\mathbf{e}p = \overline{X}$ = the *error* set of p.

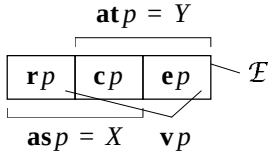


Figure 1: Execution sets of a process.

These terms refer to the agreement described above, as seen from the perspective of the device (rather than the environment). For example, the acceptable executions are acceptable to the device.

Refinement is a binary relationship $\sqsubseteq$ on $S_{\mathcal{E}}$ such that

$$(X_1, Y_1) \sqsubseteq (X_2, Y_2) \;\Leftrightarrow\; X_1 \supseteq X_2 \;\wedge\; Y_1 \subseteq Y_2.$$

Intuitively, refinement represents a *relative* notion of correctness: $p \sqsubseteq q$ means that p can be replaced by q without bad effects. The directions of the set inequalities can be understood by considering that the occurrence of an execution constitutes a danger of misusing certain processes (the danger of misuse is formalized by the presence of that execution in the reject sets of those processes). A better process accepts more executions, thus it is less exposed to misuse. Also, a better process accesses fewer executions, thus it is less likely to misuse other processes. Examples will be given in Sections 3 and 5, for particular applications of the formalism.

Reflection is a unary operation $-$ on $S_{\mathcal{E}}$ such that

$$-(X, Y) = (Y, X).$$

Informally, if a process p is viewed as an agreement between a device and its environment from a device point of view (as described above), then reflection turns the table: $-p$ represents the corresponding agreement from the environment point of view. The environment undertakes not to misuse the device; thus, the device rejects become the environment errors. Also, the environment demands that the device does not err; thus, the device errors become the environment rejects. In this sense, $-p$ models the 'matching environment' of a device represented by p.

Product, written $\times$, and *exclusive sum*, written $\oplus$, are binary operations on $S_{\mathcal{E}}$ such that

$$(X_1, Y_1) \times (X_2, Y_2) = (X_1 \cap X_2,\; Y_1 \cap Y_2 \cup \overline{X_1 \cap X_2}),$$
$$(X_1, Y_1) \oplus (X_2, Y_2) = (X_1 \cap X_2 \cup \overline{Y_1 \cap Y_2},\; Y_1 \cap Y_2).$$

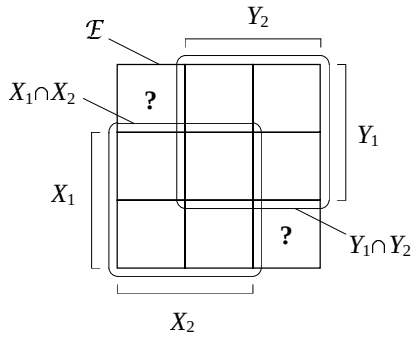


Figure 2: Deriving the product and exclusive sum operators.

Informally speaking, the product models a system formed by two devices operating jointly. The new system's accessible executions include all executions that are accessible to *both* participating devices, and its acceptable executions include all executions that are acceptable to *both* devices. In Figure 2, we see that the executions marked with '?' are so far not accounted for. These executions are errors for the resulting system, because they should be avoided by one of the devices. Accordingly, these executions are acceptable to the resulting system. In the definition of product, notice that $\overline{X_1 \cap X_2} \setminus (Y_1 \cap Y_2) = \overline{X_1} \cap \overline{Y_2} \cup \overline{Y_1} \cap \overline{X_2}$, i.e., the acceptable set is augmented from $Y_1 \cap Y_2$ by precisely the executions marked '?'. The exclusive sum is similar to the product, except for the executions marked '?'. Each of these executions should be avoided not only by one of the devices, but also by the environment of the other. In the exclusive sum, we consider these executions to be rejects, rather than errors, for the resulting system. (This amounts to blaming the environments, rather than the devices, for not avoiding these executions. In this sense, the exclusive sum assumes that the environments, rather than the devices, operate jointly.) Accordingly, these executions are accessible to the exclusive sum. Notice that $\overline{Y_1 \cap Y_2} \setminus (X_1 \cap X_2) = \overline{X_1} \cap \overline{Y_2} \cup \overline{Y_1} \cap \overline{X_2}$.

Statement 1 *For processes p, q and r, we have*

(a)	$p \times p = p$,	**(a′)**	$p \oplus p = p$,	(*idempotency of* $\times$ *and* $\oplus$)
(b)	$(p \times q) \times r = p \times (q \times r)$,	**(b′)**	$(p \oplus q) \oplus r = p \oplus (q \oplus r)$,	(*associativity of* $\times$ *and* $\oplus$)
(c)	$p \times q = q \times p$,	**(c′)**	$p \oplus q = q \oplus p$.	(*commutativity of* $\times$ *and* $\oplus$)

Idempotency ensures that connecting a device with an identical replica of itself produces an identical device. Commutativity and associativity ensure that the order of connecting devices (in parallel) does not matter.

Top, denoted by $\top$, is the process $(\varnothing, \mathcal{E})$, where $\varnothing$ is the empty set. *Bottom*, denoted by $\bot$, is the process $(\mathcal{E}, \varnothing)$. *Void*, denoted by Φ, is the process $(\mathcal{E}, \mathcal{E})$. Void has no rejects and no errors.

A process (X, Y) is *robust* if $Y = \mathcal{E}$. A process (X, Y) is *chaotic* if $X = \mathcal{E}$.

Informally, a process (X, Y) guarantees that certain executions do not occur in its presence (its errors) and requires that other executions must not occur in its presence (its rejects). In this interpretation, robust and chaotic processes are pure guarantees and pure requirements, respectively. Robustness also represents an *absolute* notion of correctness. The fact that a process has no rejects means that it needs no guarantees from the environment and can operate autonomously, hence the term 'robust'. Chaotic processes are so called because any execution is accessible to them. Void is both robust and chaotic.

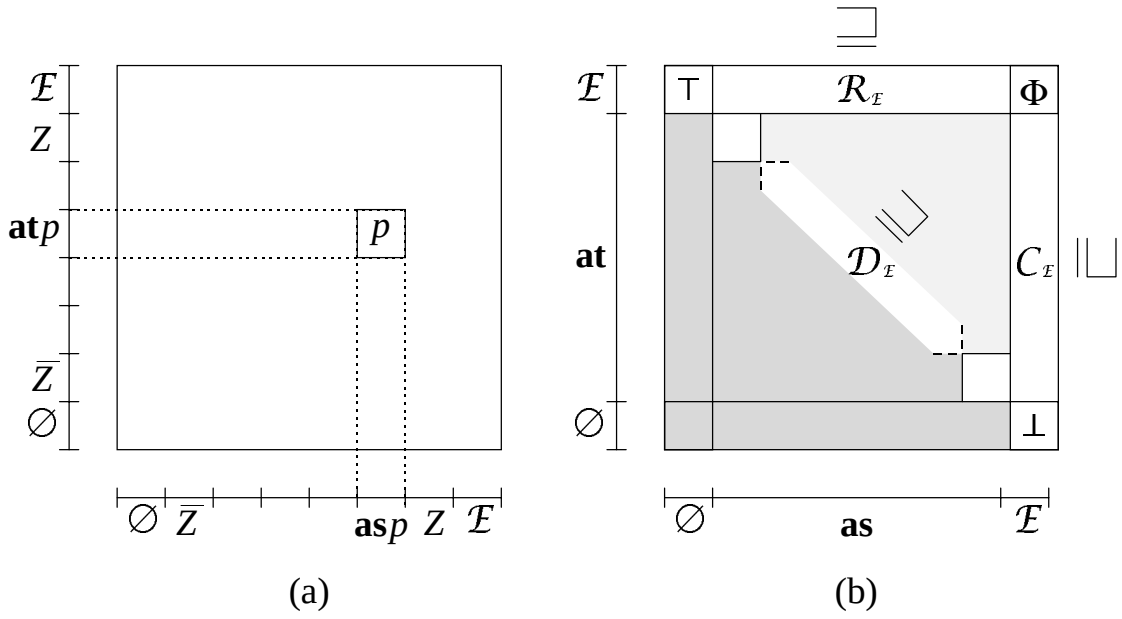


Figure 3: Charting a process space (and beyond).

Figure 3 (b) charts a process space and Figure 3 (a) illustrates the notation. 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 3 (a). Complementary subsets are represented by segments equidistant from the middle of an axis. Notice the segments for $\overline{Z}$ and Z in Figure 3 (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, **as**p, and **at**p in Figure 3 (a). With these conventions, a process space $S_{\mathcal{E}}$ is represented in Figure 3 (b): $S_{\mathcal{E}}$ 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 3 (b). The upper and right borders in Figure 3 (b) are the sets of robust and chaotic processes, denoted by $\mathcal{R}_{\mathcal{E}}$ and $C_{\mathcal{E}}$, respectively. 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 $\mathcal{R}_{\mathcal{E}}$, $C_{\mathcal{E}}$ and $\mathcal{D}_{\mathcal{E}}$.

3 Behavior of Concurrent Systems

In this section, we apply the process space formalism to particular correctness concerns regarding the behavior of concurrent systems. The only parameter of a process space is the execution set; we instantiate the whole construction by choosing an execution set. For each application, refinement and robustness have specific meanings as conditions for safety, liveness, etc. In examples, we apply refinement and robustness to detect diverse faults in concurrent systems.

Concurrent systems have discrete state spaces and operate in continuous time. Some examples of types of concurrent systems were given in Section 1. The state spaces can be finite or countable. The state transitions are called *events* (however, events do not necessarily change state, because a state transition may be from a state into the same state). For simplicity, in this section events are assumed to be instantaneous and not simultaneous; however, process spaces are by no means bound to this point of view or even to concurrent systems. In Section 5, we discuss other possible applications and points of view.

For this type of concurrent systems, the process space product resembles the '$\|$' operator in Hoare's communicating sequential processes [Ho85] (see the laws for 'failures' and 'divergences'); the process space refinement is similar to the '$\sqsubseteq$' ordering of 'non-deterministic processes' in [Ho85] (in terms of 'failures' and 'divergences'); and the process space reflection is similar to the 'reflection' in Ebergen's method for the design of delay-insensitive circuits [Eb89, Eb91]. Some other formalisms of concurrent systems define similar operators for parallel composition (product), comparison of an implementation to a specification (refinement), or the matching environment of a specification (reflection), in terms of particular types of executions ('partial' and 'complete' executions, 'computations', 'traces', etc.). For example, we mention: the failure models of Brookes, Hoare, and Roscoe [BHR84, BR85]; the testing equivalences of de Nicola and Hennessy [dNH83]; Dill's trace theories [Di89]; Josephs' receptive process theory [Jo92]; and Verhoeff's models of delay-insensitive systems [Ve94b]. (A complete literature survey is, unfortunately, beyond the scope of this paper.) However, the process space operators are simpler (e.g., they have no connectivity restrictions, and they do not need action sets) and more general (they are not bound to a particular type of executions, and they apply wherever process spaces apply). Exclusive sum of processes does not seem to have a precedent.

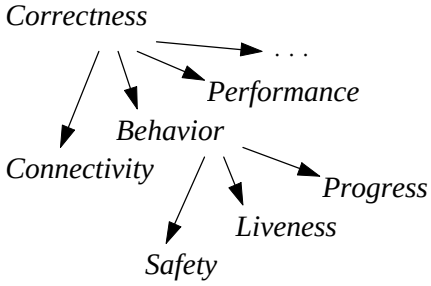


Figure 4: Correctness concerns for concurrent systems.

Concurrent systems can have diverse pathologies: hazards, deadlock, livelock, starvation, etc. Three main classes of correctness concerns for concurrent systems are connectivity, behavior, and performance. Performance concerns involve bounds or other relationships on quantitative parameters such as delays and probabilities. Roughly speaking, behavior concerns regard those aspects of the operation of a concurrent system that can be determined without specifying quantitative parameters[1]. Connectivity concerns regard proper connections of ports. There can be other concerns, such as maintainability, cost, and fault tolerance. Both performance and behavior concerns are topics of active research, and even the connectivity concerns need some work. We further categorize behavior concerns as safety (absence of hazards, wrong outputs, etc.), liveness (absence of deadlock, starvation, etc.) and progress (absence of deadlock (again), livelock, etc.).

[1] Reference to parameters seems to be the distinction between 'metric-containing' and 'metric-free' concerns [Mo95].

For the examples of concurrent systems, we use the following terminology and notation. Let $\mathcal{U}$ be a set, called the *action universe*. An *alphabet* is a subset of $\mathcal{U}$. A *word* over an alphabet Σ is a (finite or infinite) sequence of actions from Σ. Concatenation of a finite word with an arbitrary word is denoted by their juxtaposition. The empty word is ε. For words u and v, we write $u \leqslant v$ if u is a prefix of v, i.e., if there exists word w such that $uw = v$, or if v is infinite and $u = v$.

A *language* is a set of words. We use the following notation for languages: **pref** is prefix-closure (the set of all prefixes of the words in a language), * is Kleene closure (defined if all words in the language are finite), $\setminus$ is set difference, $\cup$ is union, $\cap$ is intersection, $\cdot$ or juxtaposition is concatenation (defined if all words in the first operand are finite), action x can represent language $\{x\}$, and alphabet Σ can represent the language of single-action words with actions from Σ. A language L is *prefix-closed* if $L =$ **pref** L. Throughout this paper, the unary operators have higher precedence than the binary operators.

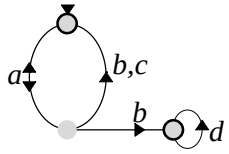


Figure 5: A finite automaton.

A regular language of finite words is represented by a finite automaton over $\mathcal{U}$. We render an automaton as follows. The initial state is marked with an incoming arrow. The accepting states are circled. Each edge has a label from $\mathcal{U}$. Double arrows on an edge represent two edges with the same label heading in opposite directions. Multiple labels on an edge represent several edges, between the same two states, each having a single label. For example, the language of the automaton in Figure 5 is $(a\ (a \cup b \cup c))^* \cdot (\varepsilon \cup abd^*)$. Word ε is in the language, because it leads to a circled state, the initial state. Word aba is not in the language, because it leads to a state that is not circled.

For language L, L^ω is the set of all concatenations of infinitely many words from L, and L^∞ is the set of all concatenations of (finitely or infinitely many) words from L. We have $L^\infty = L^* \cup L^\omega$. For finite word u, u^ω is the infinite word $uuu\cdots$. The precedence of unary language operators is * (highest), $^\omega$, $^\infty$, **pref** (lowest).

The *projection* of a word u on an alphabet Σ is a word $u{\downarrow}\Sigma$ obtained by deleting from u all actions which are not in Σ. For language L and alphabet Σ, the *expansion* of L from Σ is the language $L{\uparrow}\Sigma = \{ u \in \mathcal{U}^\infty \mid u{\downarrow}\Sigma \in L \}$, i.e., the set of all words whose projections on Σ are in L. The precedence of binary operators on languages and alphabets is $\cdot$ (highest), $\downarrow$, $\uparrow$, $\setminus$, $\cap$, $\cup$ (lowest).

To represent concurrent systems, each event (state transition) is associated with the ‘occurrence’ of an action. Several events, possibly in different concurrent systems, can be associated with the same action.

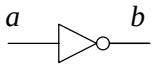


Figure 6: Inverter.

Some aspects of the behavior of a concurrent system are well described by two alphabets and a language. Actions that are controlled by that concurrent system are called *outputs*; actions that are controlled by the environment are called *inputs*; and, the set of finite sequences of inputs and outputs that can occur up to a certain time in correct operation of concurrent system *s* is called the *language* of *s*. In this section, we shall sometimes describe devices by their inputs and outputs, shown in a figure, and by their languages. For example, the inverter in Figure 6 has input *a* and output *b*, and we may assign to it the language **pref**(*ba*)*. Events correspond to changes in the voltage signals on the terminals of the inverter. In determining the language, we assume that initially all signals are low (power off).

On the other hand, such descriptions are insufficiently flexible: e.g., they do not permit a terminal to be sometimes an input and other times an output, and they do not contain *explicit* liveness or progress information. (One can attach certain *implicit* liveness or progress properties to such descriptions (see [NB95a, NB95b]), but that is done at the expense of flexibility.) Here, we use such descriptions only informally, to introduce examples. The only formal representations in this paper are by processes.

3.1 Safety

Informally speaking, safety properties of concurrent systems assert that 'something bad does not happen' [LL90]. Safety violations include hazards, illegal output events, and illegal input events.

For studying safety, we take the execution set to be $\mathcal{U}^*$ and we consider that each concurrent system *s* is represented by its *safety process*, a process σs over $\mathcal{U}^*$. A *partial execution* is a finite sequence of actions that can be observed up to a certain time. The safety processes are determined by the agreement pattern described in Section 2, applied to partial executions. A theory of safety, containing all operators and properties of process spaces, is obtained by just choosing the execution set to be $\mathcal{U}^*$ and by applying the agreement pattern to partial executions. Since the operators and properties are for arbitrary processes over $\mathcal{U}^*$, this theory has no connectivity restrictions or interference from any other correctness concerns. The following examples illustrate safety processes and meanings of robustness and product in the safety interpretation.

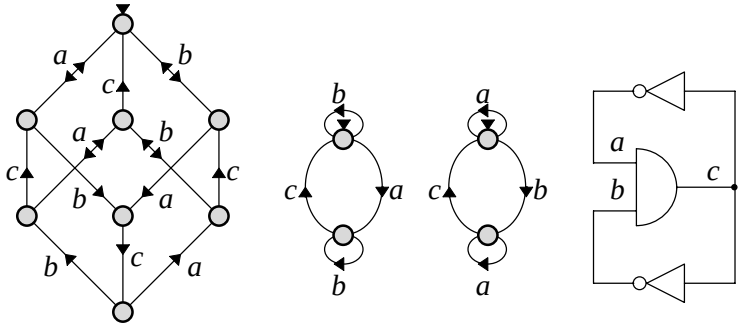


Figure 7: Contract sets and circuit for Example 1.

Example 1 The robustness condition can be used, for instance, to detect hazards in a digital circuit. Hazards can be represented by employing particular models of gate behavior in which certain transitions are declared illegal. Also see [Di89], p. 46, and the 'stability' condition in [St94], p. 165. For definitions and other techniques for the analysis of hazards see, e.g., [BS95].

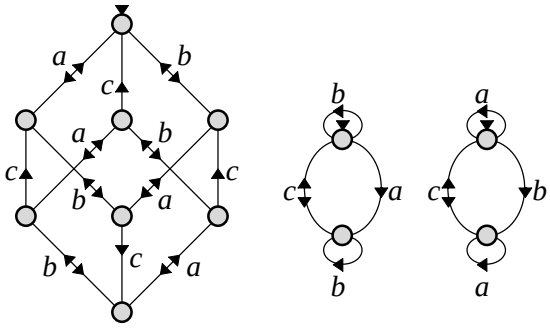
We first construct safety processes representing the components of a circuit. Then, we check robustness of their product; violations of this condition constitute safety faults.

Let σ_{INV1}, σ_{AND}, and σ_{INV2} be the safety processes of the three gates in Figure 7 (d) (from top to bottom). Assuming $\mathcal{U} = \{a, b, c\}$, the contract sets of these processes are represented by the finite automata in Figure 7 (b), (a), and (c), respectively. Each event corresponds to a change of the logical value of the corresponding voltage signal in the circuit. We consider all signal voltages to be low in the initial state (power off). For example, the contract aca of σ_{INV1} specifies that a goes from low to high, then c goes from low to high, then a goes from high to low. Hazards are ruled out by omitting those input events that disable output events. For example, two consecutive transitions on c are omitted from the contract set of INV1, because the second c would disable the output event a and thus constitutes a hazard. The rejects of each of σ_{AND}, σ_{INV1}, and σ_{INV2} are finite words u such that there exist finite word v and input a such that va is a prefix of u, v is in the contract set of the respective process, and va is not in the contract set. Informally, the rejects of these processes are finite words that get out of the contract set by an invalid input. For example, $abbcabc$ is a reject of σ_{AND}, because the second b is an invalid input, but ab is a contract. Dually, the errors of these processes are finite words that get out of the contract language by an invalid output. For example, $abcbaabc$ is an error of σ_{INV1}, because the third a is an invalid output.

Notice that the contract sets of σ_{INV1} and σ_{INV2} are different from the languages of INV1 and INV2, because of actions that are neither inputs nor outputs. Events whose actions are neither inputs nor outputs of a gate can occur arbitrarily in the contracts, rejects, and errors of the safety process of that gate (such events are not 'seen' by the gate), but do not appear in the language of that gate. Events from outside the alphabets of a gate produce self-loops in every state.

Next, we check the robustness of the product of the safety processes representing the parts of the circuit. For the circuit in Figure 7 (d), the condition is not satisfied. Let $u = abcac$. We note that $u \in \mathbf{as}\sigma_{\text{INV1}} \cap \mathbf{as}\sigma_{\text{AND}}$, since u is in the contract sets in Figure 7 (a) and (b). Also, $u \in \mathbf{as}\sigma_{\text{INV2}}$, since u gets out of the contract set in Figure 7 (c) by an input to INV2 (the second c), and thus u is a reject for σ_{INV2}. We also note that $u \notin \mathbf{at}\sigma_{\text{INV2}}$, because u is a reject for σ_{INV2}. After some set manipulations, it follows that $u \notin \mathbf{at}(\sigma_{\text{INV1}} \times \sigma_{\text{AND}} \times \sigma_{\text{INV2}})$, and thus $\sigma_{\text{INV1}} \times \sigma_{\text{AND}} \times \sigma_{\text{INV2}} \notin \mathcal{R}_{u^*}$. (One can check the last sentence directly or by applying Statement 12 (a) from Section 4.) Therefore, the safety condition is violated.

The violation can be interpreted as follows. The offending word represents a hazard. After $abca$, both the input and the output signals of INV2 are high, and an output event is enabled. However, another input transition c can occur first, changing the input voltage to low and disabling the output transition. Also see the 'oscor' example in [St94], p. 167.



(a) AND (b) INV1 (c) INV2

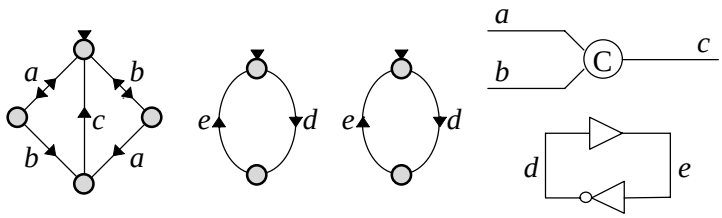
Figure 8: Contract sets for Example 2.

Example 2 For the circuit in Figure 7 (d) (Example 1), it is up to the users whether or not to consider hazards to be safety violations (or to choose which types of hazards constitute safety violations). One may, for instance, permit arbitrary input transitions under an inertial gate model. The contract sets of the modified safety processes of INV1, AND, and INV2 are shown in Figure 8 (b), (a), and (c), respectively. The rejects are still the finite words that get out of the contract set by an input event, and the errors are still the finite words that get out of the contract set by an output. Note that any input event is allowed from any state; therefore the reject sets are empty. Thus, the safety processes of the three gates are robust. It follows that the product of these safety processes is also robust. (One can check the last sentence directly or by applying Statement 17.) Thus, there is no safety fault.

3.2 Liveness

Informally speaking, liveness properties of concurrent systems assert that 'something good eventually does happen' [LL90]. Examples of liveness faults include deadlock and starvation.

For studying liveness, we consider that each concurrent system *s* is represented by its *liveness process*, a process λs over $\mathcal{U}^\infty$. A *complete execution* is a finite or infinite sequence of actions that can be observed until the 'end of time'. The liveness processes are determined by the agreement pattern described in Section 2, applied to complete executions. A theory of liveness, containing all operators and properties of process spaces, is obtained by just choosing the execution set to be $\mathcal{U}^\infty$ and by applying the agreement pattern to complete executions. Since the operators and properties are for arbitrary processes over $\mathcal{U}^\infty$, this theory has no safety restrictions or interference from any other correctness concerns. The following examples illustrate liveness processes and meanings of refinement and product in the liveness interpretation.



(a) C (b) BUF (c) INV (d) circuit

Figure 9: Languages and circuit for Example 3.

Example 3 The refinement relationship on liveness processes can be used to verify faults such as deadlock ('wait-for' cycle) and unfairness in a digital circuit. In this example, we check whether the circuit in Figure 9 (d) is a correct implementation of the AND gate in Example 1.

Let C (a C-element), BUF (a buffer), and INV (an inverter) denote the three components in Figure 9 (d) (from top to bottom). The languages of C, BUF, and INV are shown in Figure 9 (a), (b), and (c), respectively. In determining the liveness processes we will consider the liveness properties of the components, as explained below.

We take

$$\mathbf{as}\lambda\,\mathrm{BUF} = ((de)^{\infty} \cup (de)^{*}dd\{d, e\}^{\infty})\uparrow\{d, e\},$$
$$\mathbf{at}\lambda\,\mathrm{BUF} = (\mathbf{pref}(de)^{\infty} \cup (de)^{*}e\{d, e\}^{\infty})\uparrow\{d, e\}.$$

Recall that $(de)^{\infty} = (de)^{*} \cup (de)^{\omega}$ and that $(de)^{\infty} \subset \mathbf{pref}(de)^{\infty}$. The finite words in $(de)^{*}$ are contracts of $\lambda\,\mathrm{BUF}$ because, at any time, the environment of BUF may stop producing d events. The infinite word $(de)^{\omega}$ is a contract of $\lambda\,\mathrm{BUF}$ because the environment and BUF need not stop at all. (We have used 'strong liveness with respect to outputs' [NB95a, NB95b] as a guide for determining the contracts of $\lambda\,\mathrm{BUF}$.) 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 words in $(de)^{*}d$ are errors, since BUF should eventually produce an e after receiving a d. The words in $(de)^{*}e\{d, e\}^{\infty}$ are also errors, since the environment does not expect two consecutive e events. Notice that the errors in $(de)^{*}e\{d, e\}^{\infty}$ are not only safety violations, but also liveness violations, because, after two consecutive e events, the environment may demand, for instance, events that BUF cannot produce (e.g., events whose actions are not in the output alphabet of BUF). Also, arbitrary interleavings of actions from outside $\{d, e\}$ can occur in the accessible and acceptable executions of $\lambda\,\mathrm{BUF}$, hence the expansions from $\{d, e\}$.

By similar considerations, we take

$$\mathbf{as}\lambda\,\mathrm{INV} = ((de)^{*}d \cup (de)^{\omega} \cup (de)^{*}e\{d, e\}^{\infty})\uparrow\{d, e\},$$
$$\mathbf{at}\lambda\,\mathrm{INV} = (\mathbf{pref}(de)^{\infty} \cup (de)^{*}dd\{d, e\}^{\infty})\uparrow\{d, e\},$$

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

Let $u = abca(de)^{\omega}$. We have $u \in \mathbf{as}\lambda_{\mathrm{C}} \cap \mathbf{as}\lambda_{\mathrm{BUF}} \cap \mathbf{as}\lambda_{\mathrm{INV}}$. On the other hand, $u \notin \mathbf{as}\lambda_{\mathrm{AND}}$ 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 $\mathbf{as}(\lambda_{\mathrm{C}} \times \lambda_{\mathrm{BUF}} \times \lambda_{\mathrm{INV}}) \not\subseteq \mathbf{as}\lambda_{\mathrm{AND}}$, and, therefore, $\lambda_{\mathrm{AND}} \not\pitchfork \lambda_{\mathrm{C}} \times \lambda_{\mathrm{BUF}} \times \lambda_{\mathrm{INV}}$.

The violation can be interpreted as follows. Word *u* causes a deadlock (a 'wait-for cycle' occurs). After *abca*, C waits for the environment of AND to send another input event, while the environment of AND waits for C to produce an output event. Note that deadlock occurs despite the fact that some parts in the circuit never stop producing events *d* and *e*.

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

Figure 10: Specification for Example 4.

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

task body P1 is
begin
   loop
     Non_Critical_Section_1;
     C1 := 0;
     loop
       exit when 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 when C1 = 1;
       C2 := 1;
       C2 := 0;
     end loop;
     Critical_Section_2;
     C2 := 1;
   end loop;
end P2;
```

Figure 11: Implementation for Example 4.

Example 4 Let us consider a classical example of starvation adapted from [Be90] (p. 35). The concurrent program in Figure 11 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 10. Let the actions be

r_{xyz} = P*x* reads C*y* = *z*,
w_{xyz} = P*x* sets C*y* to *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*.

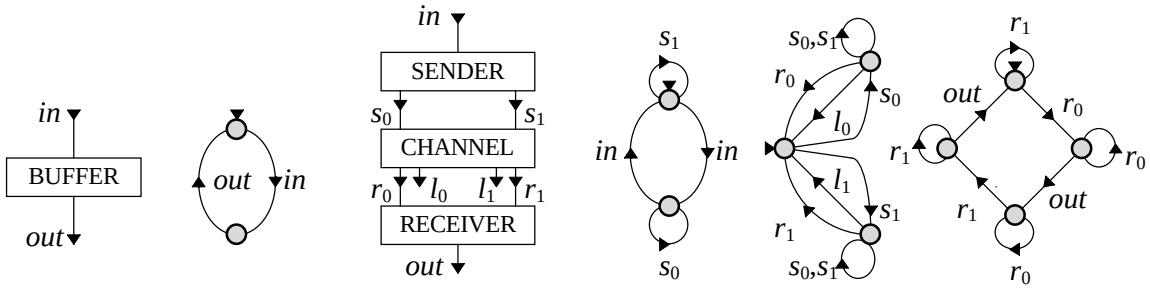
Consider infinite execution $u = encs_2 lncs_2 (encs_1 lncs_1 w_{110} r_{121} w_{220} r_{210} ecs_1 lcs_1 w_{111} w_{221})^{\omega}$. Word *u* is an accessible complete execution for P1, because P1 executes its main loop, in which it can stay forever. Word *u* is also an accessible complete execution 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 execution 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 will eventually choose ecs_2. (We have used 'strong liveness with respect to outputs' [NB95a, NB95b] as a guide for determining the contracts of λ_{P1}, λ_{P2}, and λ_{P0}.) Thus, $u \in \mathbf{as}(\lambda_{\mathrm{P1}} \times \lambda_{\mathrm{P2}})$ and $u \notin \mathbf{as}\lambda_{\mathrm{P0}}$. Consequently, we have $\mathbf{as}\lambda_{\mathrm{P0}} \not\supseteq \mathbf{as}(\lambda_{\mathrm{P1}} \times \lambda_{\mathrm{P2}})$ and thus $\lambda_{\mathrm{P0}} \not\pitchfork (\lambda_{\mathrm{P1}} \times \lambda_{\mathrm{P2}})$.

The violation can be interpreted as follows. Word *u* causes starvation because it never allows P2 to enter its critical section.

3.3 Progress

Informally speaking, progress properties of concurrent systems assert that 'something good does happen within a bounded time' (our interpretation). (Progress is a behavior concern rather than a performance concern, because it does not refer to the values of the delay bounds, but only to their existence. The values of the delay bounds need not be known for progress analysis.) Examples of progress faults include deadlock and livelock.

For studying progress, we consider that each concurrent system s is represented by its *progress process*, a process πs over $\mathcal{U}^\infty$. An *unbounded execution* is a finite or infinite sequence u of actions with the property that, after every prefix of u, the execution point can continue to follow u (can remain in u) for any amount of time. (An unbounded execution u may or may not be complete, as the execution point may or may not follow u forever.) The progress processes are determined by the agreement pattern described in Section 2, applied to unbounded executions. A theory of progress, containing all operators and properties of process spaces, is obtained by just choosing the execution set to be $\mathcal{U}^\infty$ and by applying the agreement pattern to unbounded executions. Since the operators and properties are for arbitrary processes over $\mathcal{U}^\infty$, this theory has no safety restrictions or interference from any other correctness concerns. The following examples illustrate progress processes and meanings of refinement and product in the progress interpretation.



(a) (b) BUFFER (c) (d) SENDER (e) CHANNEL (f) RECEIVER

Figure 12: Communication protocol for Example 5.

Example 5 Consider the following communication protocol. The specification of the communication protocol is a 1-bit buffer, as in Figure 12 (a). The language of the buffer is represented in Figure 12 (b), but more explanations of its operation are necessary. 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 12 (c). The languages of the components are represented in Figure 12 (c), (d), and (e), but more explanations are necessary. After *in*, SENDER sends s_0 messages. Some may be lost, causing l_0 events which reset CHANNEL and are not seen by RECEIVER. Some may get through, causing r_0 events to be fired. 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. Some allowances are made for out of place s_0, r_0, s_1, and r_1 events, which do not change the state of RECEIVER when repeated.

Consider $u = in(s_0l_0)^\omega$. Word u is an accessible unbounded execution of SENDER, because the environment is under no obligation to provide another *in*. It is also an accessible unbounded execution of CHANNEL: although CHANNEL cannot choose l_0 over r_0 infinitely many times, CHANNEL can choose l_0 any finite number of times. Thus, the execution point of CHANNEL can

stay in u for an unbounded time after any prefix of u. Word u is also an accessible unbounded execution of RECEIVER, since RECEIVER remains in its initial state, where it is not expected to issue an output event. Since $u \in \mathbf{as}\,\pi\text{SENDER} \cap \mathbf{as}\,\pi\text{CHANNEL} \cap \mathbf{as}\,\pi\text{RECEIVER}$, we have $u \in \mathbf{as}\,(\pi\text{SENDER} \times \pi\text{CHANNEL} \times \pi\text{RECEIVER})$. On the other hand, u is not an accessible unbounded execution of BUFFER, because an *out* event must follow *in* within a bounded time. Hence, $u \notin \mathbf{as}\,\pi\text{BUFFER}$. Therefore, $\pi\text{BUFFER} \not\subseteq \mathbf{as}\,(\pi\text{SENDER} \times \pi\text{CHANNEL} \times \pi\text{RECEIVER})$.

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 The deadlock fault in Example 3 also constitutes a violation of progress. The progress processes of the components in Example 3 are exactly the same as their liveness processes. (This is not always true. For instance, the liveness and progress processes of CHANNEL in Example 5 are different.) Hence, the fault in Example 3 can be detected with progress processes in the same manner as with liveness processes.

3.4 Classification of Lock Faults

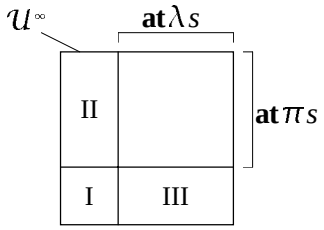


Figure 13: Absolute lock faults.

The fact that the progress execution set is the same as the liveness execution set ($\mathcal{U}^\infty$) invites a comparison between liveness and progress. By this comparison, we obtain a classification of liveness and progress faults (called 'lock faults'), in absolute and relative versions.

Let s be a concurrent 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 define these situations as *absolute locks of type* I, II, and III, respectively (see the diagram in Figure 13 (a)). *There are no other liveness or progress faults.*

Lock faults can also be considered in a relative sense, whereby two concurrent 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 positions with respect to $\mathbf{as}\,\lambda s_1$, $\mathbf{as}\,\lambda s_2$, $\mathbf{as}\,\pi s_1$, and $\mathbf{as}\,\pi s_2$, as shown in the diagram in Figure 14 (a). (Figure 14 (b) is similar to Figure 14 (a), except that s_1 and s_2 have been swapped.) We define *relative locks of type* I, II, and III as the existence of a word in at least one of the regions marked 'I', 'II', and 'III', respectively, in either Figure 14 (a) or (b).

Locks of type I, II, and III appear to generalize the notions of deadlock, starvation, and livelock, respectively. For instance, the deadlock fault in Example 3 and Example 6 is a relative

lock of type I; the starvation fault in Example 4 is relative lock of type II; and, the livelock fault in Example 5 is a relative lock of type III. Denoting by s_1 the respective specifications and by s_2 the respective implementations, we have: in Example 3 and Example 6, $u \in \mathbf{as}\lambda s_2$, $u \in \mathbf{as}\pi s_2$, $u \notin \mathbf{as}\lambda s_1$, and $u \notin \mathbf{as}\pi s_1$; in Example 4, $u \in \mathbf{as}\lambda s_2$, $u \in \mathbf{as}\pi s_2$, $u \notin \mathbf{as}\lambda s_1$, and $u \in \mathbf{as}\pi s_1$ (we consider that the execution point of the specification can follow u for an unbounded time because the option ecs_1 can be chosen any finite number of times in a row); and, in Example 5, $u \notin \mathbf{as}\lambda s_2$, $u \in \mathbf{as}\pi s_2$, $u \notin \mathbf{as}\lambda s_1$, and $u \notin \mathbf{as}\pi s_1$.

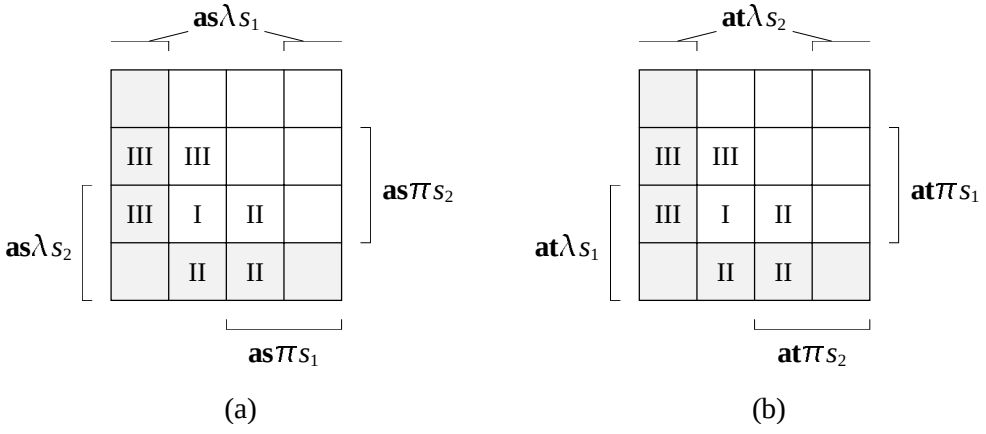


Figure 14: Relative lock faults.

Relationships between relative and absolute lock faults follow from a relationship between refinement and robustness, which will be given later in this paper (Statement 13, Subsection 4.3). For example, a relative lock of type I can be viewed as an absolute lock of type I between an implementation and the environment of a specification.

The diagrams for relative lock faults can be simplified by the following observation. In many concurrent systems, every complete execution is also an unbounded execution (if the execution point can stay in execution u forever, it can also stay for an unbounded time after any prefix of u). Therefore, we often have, for concurrent 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 14 (a) and (b) are void.

4 Process Space Structure

In this section, we discuss several process space properties and their significance.

In Subsection 4.1, we address some basic questions about the algebraic structure of process spaces. Some highlights are the level of abstraction of our theory (see comment after Statement 2), the lattice structure, and a duality principle.

In Subsection 4.2, the product and exclusive sum operators are extended to arbitrary sets of processes, in order to deal with possibly infinite systems.

In Subsection 4.3, properties with a more practical meaning are derived. We show how process spaces allow for structured verification. We link the process space notions of absolute and relative correctness, and the verification and testing viewpoints. A design equation for process spaces is stated and solved. A decomposition into robust and chaotic processes and a separate treatment of robust and chaotic processes are proposed.

4.1 Basic Algebraic Properties

Statement 2 *For processes p, q and r,*

(a) $p \sqsubseteq p$, (*reflexivity of* $\sqsubseteq$)
(b) $p \sqsubseteq q \ \wedge \ q \sqsubseteq r \ \Rightarrow \ p \sqsubseteq r$, (*transitivity of* $\sqsubseteq$)
(c) $p \sqsubseteq q \ \wedge \ q \sqsubseteq p \ \Rightarrow \ p = q$. (*antisymmetry of* $\sqsubseteq$)

Statement 2 shows that refinement is a partial order. Reflexivity and transitivity are commonsense properties of a worse-or-as-good-as relationship. Antisymmetry ensures that process spaces are a fully abstract model (with respect to the refinement relationship).

Statement 3 (*Monotonicity*) *For processes p, q and r,*

(a) $p \sqsubseteq q \ \Rightarrow \ p \times r \ \sqsubseteq \ q \times r$, (*monotonicity of* $\times$ *with respect to* $\sqsubseteq$)
(a′) $p \sqsubseteq q \ \Rightarrow \ p \oplus r \ \sqsubseteq \ q \oplus r$. (*monotonicity of* $\oplus$ *with respect to* $\sqsubseteq$)

In words, if p is refined by q, then p coupled with r is refined by q coupled with the same r.

Statement 3 (a) does not have restrictions on the ports of the devices represented by p, q, and r. (There are no ports in process spaces.) For the safety and liveness conditions deriving from $\sqsubseteq$, this absence of restrictions may be surprising. For example, the device of r may have common internal ports with the device of q, but not with the device of p; the property still holds. Also see the 'compatibility with union' theorems in [NB95a] and [NB95b].

Statement 4 *For processes p and q,*

(a) $--p = p$,
(b) $p \sqsubseteq q \ \Leftrightarrow \ -q \sqsubseteq -p$,
(c) $-(p \times q) \ = \ -p \oplus -q$, **(c′)** $-(p \oplus q) \ = \ -p \times -q$. (*de Morgan's laws for* $-$, $\times$, *and* $\oplus$)

Statement 5 (*Process Lattice*) *For subset* $\mathcal{B}$ *of* $S_{\mathcal{I}}$ *, there exist unique processes* $\sqcup\mathcal{B}$ *(the* join *of* $\mathcal{B}$*) and* $\sqcap\mathcal{B}$ *(the* meet *of* $\mathcal{B}$*) such that, for every process p,*

(a) $(\forall\ q \in \mathcal{B}:\ p \sqsupseteq q) \ \Leftrightarrow \ p \sqsupseteq \sqcup\mathcal{B}$, ($\sqcup\mathcal{B}$ *is the least upper bound of* $\mathcal{B}$)
(a′) $(\forall\ q \in \mathcal{B}:\ p \sqsubseteq q) \ \Leftrightarrow \ p \sqsubseteq \sqcap\mathcal{B}$. ($\sqcap\mathcal{B}$ *is the greatest lower bound of* $\mathcal{B}$)

Statement 5 shows that $\langle S_{\mathcal{I}}, \sqsubseteq \rangle$ is a complete lattice.

Join and meet are also defined as operators on processes. *Join*, written $\sqcup$, and *meet*, written $\sqcap$, are binary operations on $S_{\mathcal{I}}$ such that

$$(X_1, Y_1) \ \sqcup \ (X_2, Y_2) \ = \ (X_1 \cap X_2, \ Y_1 \cup Y_2) \quad \text{and}$$
$$(X_1, Y_1) \ \sqcap \ (X_2, Y_2) \ = \ (X_1 \cup X_2, \ Y_1 \cap Y_2).$$

Informally speaking, meet models the non-deterministic choice between two devices: the device $p \sqcap q$ can choose to act either like p or like q in deciding which executions are accessible or acceptable, and may take the choice that causes the most violations. For instance, if execution u is acceptable to p but not to q, it is not acceptable to $p \sqcap q$. Dually, join models the non-

deterministic choice between two environments. The device $p \sqcup q$ 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 $-q$.

Statement 6 *For processes p and q,*

(a) $p \sqcup q = \sqcup\{p, q\}$ **(a′)** $p \sqcap q = \sqcap\{p, q\}$.

Statement 6 shows the correspondence between the join and meet operators we define and the join and meet induced by the refinement order. The induced join and meet satisfy laws of associativity, commutativity, idempotency, absorption, etc. (see for instance [DP90], Theorem 5.2); thus, so do the join and meet we define.

Statement 7 *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* $\sqcap$)

In conjunction with Statement 6, Statement 7 shows that $\langle S_{\mathcal{I}}, \sqsubseteq \rangle$ is a distributive lattice.

Statement 8 *For process p,*

(a)	$\bot \sqsubseteq p \sqsubseteq \top$,			(*extremal elements for* $\sqsubseteq$)
(b)	$p \times \Phi = p$,	**(b′)**	$p \oplus \Phi = p$,	(*identity elements for* $\times$ *and* $\oplus$)
(c)	$p \sqcup \bot = p$,	**(c′)**	$p \sqcap \top = p$,	(*identity elements for* $\sqcup$ *and* $\sqcap$)
(d)	$p \times \top = \top$,	**(d′)**	$p \oplus \bot = \bot$,	(*dominant elements for* $\times$ *and* $\oplus$)
(e)	$p \sqcup \top = \top$,	**(e′)**	$p \sqcap \bot = \bot$.	(*dominant elements for* $\sqcup$ *and* $\sqcap$)

The identity element properties for $\times$ and $\oplus$ ensure that introducing a void device in a system does not change the system.

Statement 9

(a)	$-\Phi = \Phi$,	**(a′)**	$-\top = \bot$,	**(a′′)**	$-\bot = \top$,
(b)	$-S_{\mathcal{I}} = S_{\mathcal{I}}$,	**(b′)**	$\top = \sqcup S_{\mathcal{I}} = \sqcap \varnothing$,	**(b′′)**	$\bot = \sqcap S_{\mathcal{I}} = \sqcup \varnothing$,
(c)	$-\mathcal{D}_{\mathcal{I}} = \mathcal{D}_{\mathcal{I}}$,	**(c′)**	$-\mathcal{R}_{\mathcal{I}} = C_{\mathcal{I}}$,	**(c′′)**	$-C_{\mathcal{I}} = \mathcal{R}_{\mathcal{I}}$.

where, for process set $\mathcal{B} \subseteq S_{\mathcal{I}}$, $-\mathcal{B}$ denotes the process set $\{ -p \mid p \in \mathcal{B} \}$.

Remark (*Duality Principle*) Let X be a statement about process spaces. The *dual* of X is a statement X^{∂} obtained by replacing in X every occurrence of $\sqsubseteq$ by $\sqsupseteq$, of $\sqcup$ by $\sqcap$, of $\times$ by $\oplus$, of $\mathcal{R}_{\mathcal{I}}$ by $C_{\mathcal{I}}$, of **as** by **at**, of **r** by **e**, and conversely. ($-$, Φ, $S_{\mathcal{I}}$, $\mathcal{D}_{\mathcal{I}}$, **c**, and **v** are their own duals.) Notice that $X^{\partial\partial} = X$. Process spaces admit the following duality principle: if statement X holds, then X^{∂} holds, too. (Informally speaking, the duality principle is a consequence of the de Morgan's laws in Statement 4 (c) and (c'), which essentially say that reflection is an isomorphism of process spaces.)

4.2 Systems

In order to deal with possibly infinite systems, product and exclusive sum are extended to sets of processes in the natural way. For $\mathcal{B} \subseteq \mathcal{S}_{\mathcal{I}}$, the *product* and *exclusive sum* of $\mathcal{B}$ are, respectively,

$$\times \mathcal{B} \;=\; \Big(\bigcap_{p\in\mathcal{B}} \mathbf{as}p\,,\; \bigcap_{p\in\mathcal{B}} \mathbf{at}p \;\cup\; \overline{\bigcap_{p\in\mathcal{B}} \mathbf{as}p} \Big),$$
$$\oplus \mathcal{B} \;=\; \Big(\bigcap_{p\in\mathcal{B}} \mathbf{as}p \;\cup\; \overline{\bigcap_{p\in\mathcal{B}} \mathbf{at}p}\,,\; \bigcap_{p\in\mathcal{B}} \mathbf{at}p \Big).$$

A *system* is a set of processes. A system can be treated as a single 'composite' process that behaves exactly like the whole system. The definitions above determine composite processes from the processes in the system. Product can be regarded as the law of composition for devices, while exclusive sum can be regarded as the law of composition for environments. Also see the explanation for deriving the binary $\times$ and $\oplus$ operators (Section 2).

Statement 10 *For processes p and q,*

(a) $p \times q \;=\; \times\{p, q\}$ **(a′)** $p \oplus q \;=\; \oplus\{p, q\}$.

Statement 10 establishes the correspondence between the binary and extended product and exclusive sum operators.

The following two statements provide criteria for relative and absolute correctness (refinement and robustness) on systems.

Statement 11 (*System Refinement Lemma*) *For process sets $\mathcal{B}$ and C,*

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

(a′) $\bigcap_{p\in\mathcal{B}} \mathbf{as}p \;\supseteq\; \bigcap_{q\in C} \mathbf{as}q \;\wedge\; \bigcap_{p\in\mathcal{B}} \mathbf{at}p \;\subseteq\; \bigcap_{q\in C} \mathbf{at}q \;\Rightarrow\; \oplus\mathcal{B} \;\sqsubseteq\; \oplus C.$

Remark [Ve94b] mentions a difficulty regarding further model extensions for dealing with systems of infinitely many devices (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 $\times\{p_i \mid i\in\mathbb{N}\} \sqsubseteq \times\{q_i \mid i\in\mathbb{N}\}$? Our answer is yes, even for uncountable process sets; for such sets, we replace $\mathbb{N}$ by an arbitrary index set I.

For all 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 Statement 11 (a), we have $\times\{p_i \mid i\in I\} \sqsubseteq \times\{q_i \mid i\in I\}$.

□

Statement 12 (*System Robustness Lemma*) *For process set $\mathcal{B}$,*

(a) $\times\mathcal{B} \;\in\; \mathcal{R}_{\mathcal{I}} \;\Leftrightarrow\; \bigcap_{p\in\mathcal{B}} \mathbf{as}p \;\subseteq\; \bigcap_{p\in\mathcal{B}} \mathbf{at}p,$

(a′) $\oplus\mathcal{B} \;\in\; C_{\mathcal{I}} \;\Leftrightarrow\; \bigcap_{p\in\mathcal{B}} \mathbf{at}p \;\subseteq\; \bigcap_{p\in\mathcal{B}} \mathbf{as}p.$

4.3 Structured Manipulation of Processes

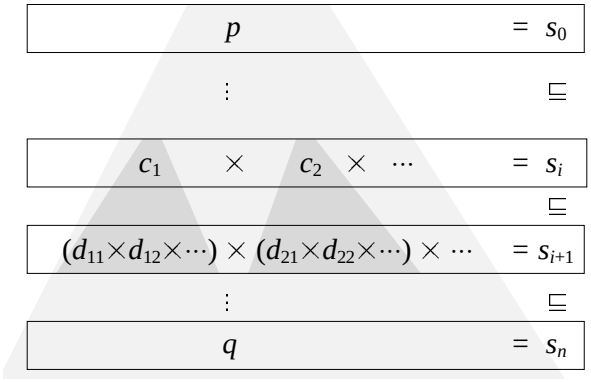


Figure 15: Modular and hierarchical verification.

Transitivity of $\sqsubseteq$ and monotonicity of $\times$ allow for structured (hierarchical and modular) verification. The problem is to determine whether $p \sqsubseteq q$, where p represents a specification and q an implementation of an interacting system. Typically, one devises a chain of intermediate specifications $s_0, s_1, \ldots, s_n$ such that $s_0 = p$ and $s_n = q$ (see Figure 15). 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 $\{0, \ldots, n-1\}$. By transitivity, $p \sqsubseteq q$ is established.

Structured verification can reduce computational costs by breaking the overall verification problem into smaller problems. Note that projection operators may be useful as constructors of intermediate specifications, but are not *necessary* for structured verification. Intermediate specifications may be guessed or derived by other methods.

Statement 13 (*Verification*) *For processes p and q,*

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

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

$$\begin{aligned}
& -p \times q \in \mathcal{R}_{\mathcal{I}} \\
\Leftrightarrow\;& \mathbf{at}(-p \times q) = \mathcal{E} \\
\Leftrightarrow\;& X_1 \cap Y_2 \cup \overline{Y_1 \cap X_2} = \mathcal{E} \\
\Leftrightarrow\;& (X_1 \cap Y_2) \cup \overline{Y_1} \cup \overline{X_2} = \mathcal{E} \\
\Leftrightarrow\;& (X_1 \cup \overline{Y_1} \cup \overline{X_2}) \cap (Y_2 \cup \overline{Y_1} \cup \overline{X_2}) = \mathcal{E} && \{\text{distributivity of } \cup \text{ through } \cap\} \\
\Leftrightarrow\;& (X_1 \cup \overline{X_2}) \cap (Y_2 \cup \overline{Y_1}) = \mathcal{E} && \{\overline{Y_1} \subseteq X_1 \;\wedge\; \overline{X_2} \subseteq Y_2\} \\
\Leftrightarrow\;& X_1 \cup \overline{X_2} = \mathcal{E} \;\wedge\; Y_2 \cup \overline{Y_1} = \mathcal{E} \\
\Leftrightarrow\;& X_1 \supseteq X_2 \;\wedge\; Y_1 \subseteq Y_2 \\
\Leftrightarrow\;& p \sqsubseteq q.
\end{aligned}$$

□

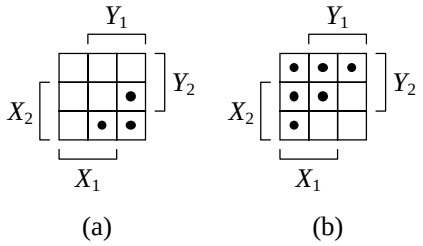


Figure 16: Diagrams for proving Statement 13.

Another proof of Statement 13 can be obtained by inspecting the Venn diagrams in Figure 16. The markers in Figure 16 (a) indicate set intersections that must be void for $p \sqsubseteq q$ to hold. For example, the two markers on the rightmost column of Figure 16 (a) mean that $X_2 \cap \overline{X_1} = \varnothing$, i.e., $X_2 \subseteq X_1$. The markers in Figure 16 (b) indicate the acceptable set of $-p \times q$. We check that the sets in Figure 16 (a) and (b) are complementary, meaning that the set in Figure 16 (a) is void if and only if the set in Figure 16 (b) is $\mathcal{E}$. Although this proof technique can be applied to most statements in this paper, we prefer logical deduction for a more insightful presentation.

Statement 13 links the autonomous and relative notions of correctness. Informally speaking, implementation q is correct with respect to specification p if and only if q operates correctly in the matching environment of p.

Statement 13 permits 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 [Eb89, Eb91] and further developed in [Di89], for their models.

Statement 14 (*Testing*) *For processes p and q,*

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

Proof By Statement 13 and Statement 4 (a), $r \times p \in \mathcal{R}_{\mathcal{E}} \;\Leftrightarrow\; -r \sqsubseteq p$ and $r \times q \in \mathcal{R}_{\mathcal{E}} \;\Leftrightarrow\; -r \sqsubseteq q$. Thus, it is sufficient to prove

$$p \sqsubseteq q \;\Leftrightarrow\; \forall\, r \in S_{\mathcal{E}}\colon\ (-r \sqsubseteq p \;\Rightarrow\; -r \sqsubseteq q).$$

(⇒) By transitivity of $\sqsubseteq$ (Statement 2 (b)), $p \sqsubseteq q \;\wedge\; -r \sqsubseteq p \;\Rightarrow\; -r \sqsubseteq q$.
(⇐) Let $r = -p$. By Statement 4 (a), $-r = p$. By reflexivity of $\sqsubseteq$ (Statement 2 (a)), $-r \sqsubseteq p$. By hypothesis, $-r \sqsubseteq q$. Since $-r = p$, we have $p \sqsubseteq q$.

□

Statement 14 is another link between the autonomous and relative notions of correctness. One can define refinement from a testing point of view: p is refined by q if q passes any test that p passes. Passing a test r can be viewed as the absence of rejects when the device is coupled with r. Statement 14 shows that this testing definition of refinement is equivalent to the direct definition we use.

The testing paradigm is commonplace in concurrency theory (see e.g. [dNH83]).

Statement 15 (*Design*) *For processes p, q and r,*

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

Proof

$$
\begin{aligned}
& p \sqsubseteq q \times r \\
\Leftrightarrow\ & -p \times q \times r \in \mathcal{R}_{\mathcal{F}} && \{\text{Statement 13}\} \\
\Leftrightarrow\ & -(p \oplus -q) \times r \in \mathcal{R}_{\mathcal{F}} && \{\text{Statement 4 (a) and (c}')\} \\
\Leftrightarrow\ & p \oplus -q \sqsubseteq r. && \{\text{Statement 13}\}
\end{aligned}
$$

□

The *design equation* 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. Statement 15 solves the design equation by showing that the minimal solution is $p \oplus -q$.

Related results can be found, for instance, in [Pr91], [Ve94a], and [Ve94b].

One possible application of the design equation 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 the 'glue' circuit.

By the duality principle (Subsection 4.1), we also claim the duals of the results above.

Many available parts and subsystems are robust (or sold as such), i.e., they are intended to be fool-proof and have a defined behavior in any environment. For example, voltage regulated sources often have overload protection. At the same time, the environments (e.g. users) should ideally be assumed to be chaotic. In these conditions, it is important to understand and exploit the characteristics of robust and chaotic processes.

Statement 16 (*RC Decomposition*)

(a) *For process p, there exist unique chaotic process q and robust process r such that*

$$q \times r = p.$$

(b) *For p, q, and r as in* Part (a), *we have*

$$q = p \sqcap \Phi \ \wedge\ r = p \sqcup \Phi.$$

Recall that robust processes can be regarded as pure guarantees, and chaotic processes as pure requirements. Statement 16 (a) shows that every process is the product of a pure guarantee and a pure requirement, while Statement 16 (b) provides a way to compute the factors. One application is that the robust and chaotic facets of a process can be 'dealt with' separately. For example, dynamic RAMs have a periodic refresh requirement in order to preserve valid data. Such a requirement can be satisfied by a subsystem designed specifically for that purpose, i.e., by a subsystem that refines the reflection of the chaotic part of the DRAM. The product of the DRAM with the refresh subsystem can then be used as a robust subsystem.

By duality, the same decomposition holds for the exclusive sum.

Statement 17 *$\mathcal{R}_\mathcal{E}$ is closed under $\times$, $\oplus$, $\sqcup$, and $\sqcap$.*

Statement 17 implies that the coupling of two robust devices or environments is also robust. A consequence is that one can ensure the robustness of a system simply by using robust components.

By duality, the same closure properties hold for $C_\mathcal{E}$.

Statement 18 (*Robustness*) *In the lattice $\langle S_\mathcal{E}, \sqcup, \sqcap \rangle$, $\mathcal{R}_\mathcal{E}$ is the principal filter generated by Φ.*

Statement 18 provides a characterization of robust processes as those processes better or identical to the void process.

By duality, $C_\mathcal{E}$ is the principal ideal generated by Φ, and the chaotic processes are those processes worse or identical to the void process.

5 Further Applications of Process Spaces

In this section, we propose other applications of the process space formalism, by instantiating the execution set.

5.1 Electrical Networks

Process spaces may also be useful in the study of electrical networks, especially if the precise values of the parameters (coefficients) are unknown, but only ranges are given. 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 agreement pattern described in Section 2.

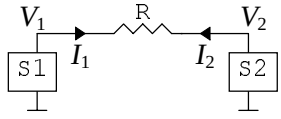


Figure 17: Network for Example 7.

Example 7 Figure 17 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 the sources by processes

$$\eta_{\mathtt{S1}} = \big(\{(V_1, I_1, V_2, I_2) \in \mathbb{R}^4 \mid V_{1\min} \leqslant V_1 \leqslant V_{1\max}\}, \{(V_1, I_1, V_2, I_2) \in \mathbb{R}^4 \mid I_{1\min} \leqslant I_1 \leqslant I_{1\max}\} \big),$$

$$\eta_{S2} = (\{(V_1, I_1, V_2, I_2) \in \mathbb{R}^4 \mid V_{2\min} \leqslant V_2 \leqslant V_{2\max}\}, \{(V_1, I_1, V_2, I_2) \in \mathbb{R}^4 \mid I_{2\min} \leqslant I_2 \leqslant I_{2\max}\}).$$

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

$$\eta_R = (\{(V_1, I_1, V_2, I_2) \in \mathbb{R}^4 \mid I_1 + I_2 = 0 \ \wedge \ r I_1 = V_1 - V_2\}, \ \mathbb{R}^4).$$

Now, we assume that $V_{1\min} = V_{2\min} = 4.9\text{V}$, $V_{1\max} = V_{2\max} = 5.2\text{V}$, $I_{1\max} = I_{2\max} = 25\text{mA}$, $I_{1\min} = I_{2\min} = -25\text{mA}$, 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 $z \in \mathbf{as}\,\eta_{S1} \cap \mathbf{as}\,\eta_{S2} \cap \mathbf{as}\,\eta_R$, but $z \notin \mathbf{at}\,\eta_{S1}$. It follows that $\eta_{S1} \times \eta_{S2} \times \eta_R \notin \mathcal{R}_{\mathbb{R}^4}$.

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

5.2 Dynamical Systems

Process spaces may also 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[2]) 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. The processes corresponding to dynamical systems are determined according to the agreement pattern described in Section 2.

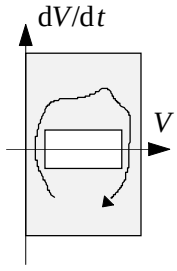


Figure 18: Requirement of Brockett ring type.

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 approximation (where states are assumed to be from a finite or countable set). [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 fast. This requirement (a Brockett ring requirement [Br89]) 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 18 (a region

[2] The time domain can also be discrete (e.g., the set of integers).

topologically isomorphic to an annulus, i.e., 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 contract 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 error. 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 approximation, 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.

5.3 Bi-directional Ports

In Section 3, we have assumed that the ports of a concurrent system are either inputs or outputs throughout the operation of that system. However, sometimes a port may change direction. For example, in larger digital components, a data line is sometimes an input (driven by the environment) and other times an output (driven by the device), in order to reduce the number of pins on the package. In such situations, one can treat an event as an input event if that event occurs while the corresponding action is an input action, and as an output event if that event occurs while its action is an output.[3]

Figure 19: Component for Example 8.

Example 8 The component P in Figure 19 has a control output *c* and a data line *d*. When *c* is high, *d* is an output; when *c* is low, *d* is an input. Signal *d* may change only once or may not change at all between two transitions on *c* or before the first transition on *c*. Both *d* and *c* are initially low. The safety process of component P is a process over $\mathcal{U}^*$, determined by the agreement pattern in Section 2.

$$\sigma_{\mathrm{P}} = \big(\mathbf{pref}(dc \cup c)^* \cup ((dc \cup c)(dc \cup c))^* \cdot dd\{c, d\}^* \\ , \mathbf{pref}(dc \cup c)^* \cup ((dc \cup c)(dc \cup c))^* \cdot (dc \cup c) \cdot dd\{c, d\}^* \big).$$

The contracts of σ_{P} are finite words that have no two consecutive *d* events. If two consecutive *d* events occur in a finite word, that word is a reject or an error according to whether signal *d* is an input or an output at the time the second *d* event occurs. For instance, execution *dd* is a reject because the second *d* occurs while *d* is an input, and the violation is due to the environment. Execution *cdd* is an error because the second *d* occurs while *d* is an output, thus the violation is due to the device.

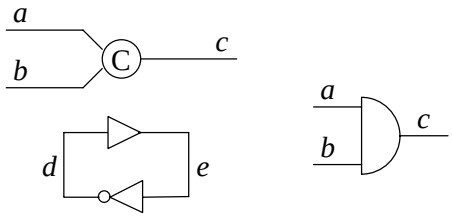
[3] Here, we assume that the direction switches are instantaneous, just as events are. More detailed treatments can also be obtained with process spaces, by a dynamical system or by 'true concurrency' execution models.

5.4 Input Control

Input control is a connectivity concern for concurrent systems which forbids dangling inputs.

In [Ve94a], studies of various connectivity concerns are proposed. Those studies are based on the 'testing paradigm', and thus have a high degree of similarity with the models in [Ve94b]. On the other hand, those studies are not homogeneous (e.g., the operators for parallel composition actually differ among the various models for connectivity and behavior concerns in [Ve94a, Ve94b]) and are not decoupled (e.g., absence of dangling inputs was not studied independently from other connectivity concerns).

For studying input control, we consider that each concurrent system s is represented by its *action control process*, a process κs over $\mathcal{U}$. By an *uncontrolled action* we mean an action that is not an output of a process. The action control processes are determined by the agreement pattern described in Section 2, applied to uncontrolled actions. The following examples illustrate action control processes and meanings of robustness and product in this interpretation.



(a) (b)

Figure 20: Circuit and gate for Example 9 and Example 10.

Example 9 Let the components of the circuit in Figure 20 (a) be C, BUF, and INV, from top to bottom. For each component in Figure 20 (a), we take the accessible uncontrolled actions to be all actions from $\mathcal{U}$ that are not outputs of that component, and the acceptable uncontrolled actions to be all actions that are not inputs of that component. Letting $\mathcal{U} = \{a, b, c, d, e\}$, we have (after some straightforward manipulations):

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

Note that $\mathcal{U} \setminus \mathbf{at}(\kappa\mathrm{C} \times \kappa\mathrm{BUF} \times \kappa\mathrm{INV}) = \{a, b\}$ indicates precisely the two dangling inputs in the circuit.

Example 10 Input control can also be considered in a relative sense. Let us check whether the circuit in Figure 20 (a) is a correct implementation of the AND gate in Figure 20 (b), with respect to input control. For that, we demand $\mathcal{K}\text{AND} \sqsubseteq \mathcal{K}\text{C} \times \mathcal{K}\text{BUF} \times \mathcal{K}\text{INV}$. The action control processes of the components are as in Example 9. We have

$$\begin{aligned}
&\mathbf{as}(\mathcal{K}\text{C} \times \mathcal{K}\text{BUF} \times \mathcal{K}\text{INV}) \\
&= \{a, b\} && \{\text{as in Example 9}\} \\
&\subseteq \{a, b, d, e\} \\
&= \mathbf{as}\,\mathcal{K}\text{AND}
\end{aligned}$$

$$\begin{aligned}
&\mathbf{at}(\mathcal{K}\text{C} \times \mathcal{K}\text{BUF} \times \mathcal{K}\text{INV}) \\
&= \{c, d, e\} && \{\text{as in Example 9}\} \\
&= \mathbf{at}\,\mathcal{K}\text{AND}
\end{aligned}$$

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

5.5 Timing

Timing properties can be decided by the time-stamped partial executions of a concurrent system, i.e., the finite or infinite sequences of pairs of actions and time-stamps representing events that occur up to a certain time. We take the time domain to be $[0,\infty)$, the set of non-negative real numbers.[4]

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 $\{i \in \mathbb{Z} \mid 0 \leqslant i < l\}$, where $\mathbb{Z}$ is the set of integer numbers and l is the length of x. Let the elements of x be $x_0, \ldots, x_{l-1}$. For example, the domain of indices of aba is $\{0, 1, 2\}$, $(aba)_1$ is b, and the domain of indices of ε is $\varnothing$. With this notation, we take the execution set to be $\mathcal{F} = \{x \in (\mathcal{U}\times[0,\infty))^* \mid \forall\ i, j \in \mathbf{d}x\colon (i \leqslant j \Rightarrow \mathbf{t}x_i \leqslant \mathbf{t}x_j)\}$. 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, we represent a concurrent system by its *timing process*, a process over $\mathcal{F}$. The timing processes are specified by the agreement pattern described in Section 2, applied to time-stamped partial executions. There are no other restrictions on specifying the timing processes.

Under this representation, timing faults are detected as violations of refinement or robustness.

5.6 True Concurrency

In Section 3, 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 more complicated point of view.

For a 'true concurrency' model, one can consider *actions* to be elements of $\mathcal{P}(\mathcal{U})$ (subsets of $\mathcal{U}$), as opposed to *atomic actions*, which are elements of $\mathcal{U}$. *Events* are occurrences of actions,

[4] Other time domains are also possible, such as the whole set of reals, the set of natural numbers, etc.

and *atomic events* are occurrences of atomic actions. The execution sets are determined by the agreement pattern in Section 2, applied to sequences over $\mathcal{P}(\mathcal{U})$. Contracts can be determined just like in Section 3, but they contain sequences over $\mathcal{P}(\mathcal{U})$ instead of over $\mathcal{U}$. Determining reject and error sets has a subtlety, if an illegal event contains both an illegal input event and an illegal output event. In such situations, the resulting execution is typically a reject rather than an error.

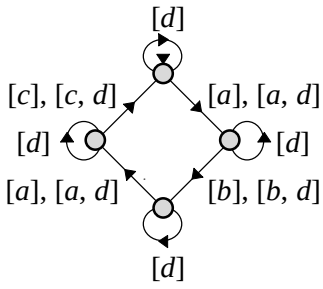


Figure 21: Contract set for Example 11.

Example 11 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 $\mathcal{P}(\mathcal{U})^*$. The contract set of σ'TOGGLE is as in Figure 21, where events 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 error of σ'TOGGLE, because [*a*, *c*] contains both an illegal input atomic event (no *a* is expected until an output atomic event) and an illegal output atomic event (it is not the turn of *c*).

6 Conclusions and Further Work

A new theory of interacting systems has been presented. Some of its novelties are abstract executions, totally decoupled and homogeneous correctness concerns, a unified treatment of discrete-state and continuous-state systems, the absence of connectivity restrictions, a representation of interacting systems by execution sets only (no states or action sets), the formal operators for this representation, the existence, meaning, and properties of an exclusive sum operation for interacting systems, a definition and characterization of robust and chaotic processes, the decomposition of a process into a 'pure guarantee' and a 'pure requirement', a classification of lock faults, an approach for dealing with bi-directional (input/output) ports, generalizations and simplifications of previous results from concurrency theory, new algebraic properties of concurrent systems and other interacting systems, a study of systems with possibly infinitely many components, a duality principle for interacting systems, a procedure for structured verification without connectivity restrictions, and diagrams for process spaces and for absolute and relative lock faults.

The process space product, refinement, and reflection, and some of their properties relate to operators introduced previously in concurrency theory (see Sections 3 and 4). However, we are not aware of a previous treatment of concurrency with any of the characteristics of process spaces we have mentioned in Section 1. In particular, it appears that the key idea of abstracting the notion of execution has not been proposed before. The 'lack of popularity' of this idea is not surprising. To eliminate the structure of executions, we have abandoned the notions of events, states, actions, and ports, and we have had to abandon all the related restrictions as well. These notions, in one form or another, stand at the basis of each of the previous treatments of concurrency known to us. Also, in process spaces there are no concepts of causality or even sequential ordering.

Process spaces are applied to safety, liveness, progress, and other correctness concerns and types of systems. For this, executions are finite words, infinite words, input and output actions, time-stamped sequences, vectors of real numbers, or functions of a real variable. More applications should be possible, for other types of executions; this is an important direction for further work.

From the study of liveness and progress by process spaces, a new classification of absolute and relative liveness and progress faults is obtained, which appears to generalize the notions of deadlock, starvation, and livelock. Another topic for further work may be the modularity and hierarchy properties for the proposed types of lock faults, following from the results in Section 4 applied to liveness and progress processes.

Another direction for further work is to identify and characterize classes of systems that occur often in practice, and to study the closure properties of such classes under the process space operators. For example, a frequently occurring relationship between the liveness and progress processes of a concurrent system has been presented at the end of Subsection 3.4.

The refinement partial order induces a lattice of processes which is complete (Subsection 4.1) and has all elements defined explicitly. (We did not need to introduce new elements by their operations to complete the structure.) It may be interesting to philosophize over the execution sets of top, bottom, and void, and what they stand for in 'real life'.

Process spaces admit a duality principle based on de Morgan's laws for product and exclusive sum (Subsection 4.1). Certain aspects of this duality have appeared before in concurrency theory, but, to the best of our knowledge, this duality has never been formally stated. Apparently, the existence of a dual operation for the parallel composition, or its de Morgan's laws, have not been mentioned before. The difference between the process space product and its dual (exclusive sum) is subtle enough, and they are the same, say, for robust processes; it is easy to confuse them at an intuitive level.

This paper only starts to explore the algebraic properties of process spaces. These algebraic properties and the related techniques for verification, design, etc. are inherited wherever process spaces apply. We are currently studying other algebraic properties of process spaces.

Automated manipulation of finite-state processes can be achieved by tools for regular languages. However, as in other concurrent system problems, one encounters the obstacle of state explosion: the number of states of the product of a system may grow exponentially with the number of processes in the system. Further work should explore and apply efficient methods for coping with state explosion. For now, the complexity of the verification problem can be reduced by applying the structured verification procedure we outline in Subsection 4.3. For the first time, we have eliminated all connectivity restrictions from structured verification.

Further work should also link process spaces to other models of interacting systems by assigning safety processes, liveness processes, etc., to objects in other models, and then relating the process space operators and relationships to their counterparts in other models. Such work may provide a unified point of view in concurrency theory and a basis of comparison among models. Moreover, such work may benefit the other models by ensuring that all process space properties

(present and future) apply to those models as well (perhaps under certain restrictions on the processes involved). This way, the users may apply process space techniques, while still enjoying advantages of other models. It is hoped that simplicity and a higher-level understanding will lead to a greater confidence in concurrent systems.

Acknowledgements I am grateful to Robert Berks, Jo C. Ebergen, Charles E. Molnar, and Tom Verhoeff for critical reviews of previous drafts. I am indebted to J. A. Brzozowski and Jo C. Ebergen for many insights into related topics. I am also indebted to J. A. Brzozowski for constant support, many critical reviews, and many comments and suggestions regarding the results and presentation of this paper.

References

[Be90] M. Ben-Ari. *Principles of Concurrent and Distributed Programming*. Prentice Hall, 1990.

[Br89] R. W. Brockett. Smooth dynamical systems which realize arithmetical and logical operations. In H. Nijmeijer and J. M. Schumacher, eds., *Three Decades of Mathematical Systems Theory: A Collection of Surveys at the Occasion of the 50th Birthday of J. C. Willems*, vol. 135 of *Lecture Notes in Control and Information Sciences*, pp. 19-30, Springer Verlag, 1989.

[BHR84] S. D. Brookes, C. A. R. Hoare, A. W. Roscoe. A theory of communicating sequential processes. *Journal of the ACM*, 31(7):560-599, 1984.

[BR85] S. D. Brookes and A. W. Roscoe. An improved failures model for communicating sequential processes. In *Proceedings NSF-SRC Seminar on Concurrency*, pp. 281-305, 1985.

[BS95] J. A. Brzozowski and C.-J. H. Seger. *Asynchronous Circuits.* Springer Verlag, 1995.

[DP90] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 1990.

[Di89] D. 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 Computer Science*, 34:83-133, 1983.

[Eb89] J. C. Ebergen. Translating programs into delay-insensitive circuits. CWI Tract 56, Centre for Mathematics and Computer Science, Amsterdam, The Netherlands, 1989.

[Eb91] J. C. Ebergen. A formal approach to designing delay-insensitive circuits. *Distributed Computing*, (5):107-119, 1991.

[GC94] M. R. Greenstreet and P. Cahoon. How fast will the flip flop? In *Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pp. 77-86, 215, 1994.

[He88] M. Hennessy. *Algebraic Theory of Processes*. Series in Foundations of Computing. The MIT Press, Cambridge, Mass., 1988.

[Ho85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[Jo92] M. B. Josephs. Receptive process theory. *Acta Informatica*, 29(1):17-31, 1992.

[JLUV94] M. B. Josephs, P. G. Lucassen, J. T. Udding, and T. Verhoeff. Formal design of an asynchronous DSP counterflow pipeline: a case study in Handshake Algebra. (Appendix: Handshake Algebra.) In *Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pp. 206-215, 1994.

[LL90] L. Lamport and N. Lynch. Distributed computing: models and methods. In J. van Leeuwen, ed., *Handbook of Theoretical Computer Science, vol. B, Formal Methods and Semantics*, pp. 1159-1196, The MIT Press - Elsevier, 1990.

[Ma86] A. Mazurkiewicz. Trace Theory. In W. Brauer, W. Reisig, and G. Rozenberg, eds., *Petri Nets, part II: Applications and Relationships to Other Models of Concurrency*, volume 255 of *Lecture Notes in Computer Science*, pp. 279-324, 1986.

[Mi89] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[Mo95] C. E. Molnar. Personal communication. 1995.

[NB95a] R. Negulescu and J. A. Brzozowski. Relative liveness: from intuition to automated verification. In *Proceedings of the Second Working Conference on Asynchronous Design Methodologies*, South Bank University, London, U.K., pp. 108-117, 1995.

[NB95b] R. Negulescu and J. A. Brzozowski. Relative liveness: from intuition to automated verification. Research report CS-95-32, University of Waterloo, Waterloo, Canada, 1995. `ftp://cs-archive.uwaterloo.ca/cs-archive/CS-95-32/CS-95-32.ps.Z`

[Pr91] I. S. W. B. Prasetya. *Solving the Design Equation in the Failures Model*. Master's thesis, Eindhoven University of Technology, Eindhoven, The Netherlands, 1991.

[St94] J. Staunstrup. *A Formal Approach to Hardware Design.* Kluwer Academic Publishers, 1994.

[Ud86] J. T. Udding. A formal model for defining and classifying delay-insensitive circuits and systems. *Distributed Computing* 1(4):197-204, 1986.

[vdS83] J. L. A. van de Snepscheut. *Trace Theory and VLSI Design*. Ph.D. Thesis, Eindhoven University of Technology, Eindhoven, The Netherlands, 1983.

[Ve94a] 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.

[Ve94b] T. Verhoeff. *A Theory of Delay-Insensitive Systems*. Ph.D. Thesis, Eindhoven University of Technology, Eindhoven, The Netherlands, 1994.