Synchronizing Processes

Thesis by
H. Peter Hofstee

In Partial Fulfillment of the Requirements
for the Degree of
Doctor of Philosophy

California Institute of Technology
Pasadena, California

1995
(Submitted 30 September 1994)

Acknowledgements

Without Jan van de Snepscheut this thesis could have never been written. He took me on as a graduate student in Groningen late 1988 and encouraged me to come to Caltech with him in 1989. For more than five years, he was a superb advisor and a dear friend.

I thank Mani Chandy for graciously accepting me as his student after Jan died. He gave me the support and encouragement I needed to complete the work, and he proofread the material, often on the same day he received it.

I thank the other members of my committee, Rajive Bagrodia, Alain Martin, Yaser Abu-Mostafa, and Chuck Seitz, for their careful review of this dissertation.

Through their classes, Alain, Chuck, Mani, and Ralph Back have deepened my interest in concurrency, communication, and VLSI, and have had a significant influence on the direction taken in this thesis.

I have benefited from discussions with (in more or less chronological order) Hans Marks, Ico van den Born, Bert Hof, Peter Hilbers, Johan Lukkien, Klaas Esselink, Jan-Eppo Jonker, Steve Burns, José Tierno, Marcel van der Goot, Drazen Borkovic, Tony Lee, Rustan Leino, Ulla Binau, Christian Nielsen, Rob Harley, Berna Massingill, John Thornley, Paul Sivilotti, Adam Rifkin, Rajit Manohar, Svetlana Kryukova, and Andrew Lines. Within this group José, Rustan, Marcel, Rajit, Johan, and Berna deserve special mention. I thank all others with whom I interacted for their friendship, and for making life at Caltech enjoyable.

I thank the current staff of the Caltech Computer Science Department, Cindy Ferrini, Diane Goodfellow,Chris Lee, Gail Stowers, and Nancy Zachariasen and those who kept the place running in the past, in particular Arlene DesJardins, Dian DeSha, Tanya Erdmann, and Patty Renstrom, for providing me with an environment in which I could work efficiently. Alice Nelson cleaned my office for most of these years, and kept my plants from dying.

Thanks to Andy Fyfe and Jeffrey Prisbrey for preparing the LaTeXthesis style files.

I thank Bill and Delores Bing for the Caltech music program, and Wen-King Su for Friday night movies. Together they ensured that I relaxed at regular intervals.

I thank Frieda for her love and support, for taking care of me when I didn't, and for giving me a place to come home to.

I thank my parents and family for their love, for their never fading support and encouragement, and for fostering my interest in academia.

The research described in this thesis was funded in part by two I.B.M. graduate fellowships, the Advanced Research Projects Agency, and the Air Force under grant AFOSR 91-0070. I thank them all for their support.

Abstract

In this monograph we develop a mathematical theory for a concurrent language based on angelic and demonic nondeterminism. An underlying model is defined with sets of sets of sequences of synchronization actions. A refinement relation is defined for the model, and equivalence classes under this relation are identified with processes. Processes, together with the refinement relation, form a complete distributive lattice.

We define a language with parallel composition, sequential composition, angelic and demonic nondeterminism, and an operator that connects pairs of synchronization actions into synchronization statements and hides these actions from observation. Also, angelic and demonic iteration are defined. All operators are monotonic with respect to the refinement ordering. Many algebraic properties are proven from these definitions. We study duals of processes and prove that they can be related to the most demonic environment in which a process will not deadlock. We give a simple example to illustrate the use of duals.

We study classes of programs for which angelic choice can be implemented by probing the environment for its next action. To this end specifications of processes are extended with simple conditions on the environment. We give a more elaborate example to illustrate the use of these conditions and the compositionality of the method.

Finally we briefly introduce an operational model that describes implementable processes only. This model mentions probes explicitly. Such a model may form a basis for a language that is less restrictive than ours, but that will also have less attractive algebraic properties.

Contents

List of Figures

Glossary of symbols

Symbol	Meaning
$a, b, c, a_0, \ldots$	synchronization statements
$a.b, \ldots$	synchronization pairs
$\overline{a}$	a probe
x, y, z	variables over synchronizations
X, Y, Z	sets of synchronizations or pairs
r, s, t	traces of synchronization statements and pairs
R, S, T	sets of traces
$\mathcal{R}, \mathcal{S}, \mathcal{T}$	sets of sets of traces
$\alpha(\mathcal{S})$	set of statements in $\mathcal{S}$
$\sigma(\mathcal{S})$	set of statements and pairs $\mathcal{S}$
$\downarrow$	projection
$\uparrow$	ejection
$\Downarrow$	projection (with pairs)
$\Uparrow$	ejection (with pairs)
$\leq$	prefix
$<$	strict prefix
$\circ$	remainder after
$\sqsupseteq$	refines, implements
$\sqsupset$	strictly refines
$\simeq$	is equivalent to
$\sqcap$	demonic nondeterminism
$\sqcup$	angelic nondeterminism
$[]$	ALT composition from CSP
$\parallel$	parallel composition
;	sequential composition

$\mathcal{S}^*$	angelic iteration
$\mathcal{S}^\dagger$	demonic iteration
$OP(vars : range : expr)$	quantified expression
μ	least fixed point
ν	largest fixed point

Chapter 1

Introduction

1.1 Semantics

This monograph is concerned with giving a precise meaning, or semantics, for a class of concurrent programs. Without a semantics, one cannot reason that a program meets a specification, hence a programming language without a semantics is useless. Of course all programming languages assign meaning to programs written in them, but the degree of formality differs vastly from one notation to another. In the worst case, the definition is: "the meaning of a program written in this language is the result you get by running the (compiled) program on this machine." Such a specification, although precise, is next to useless, and a programmer who has to rely on it cannot but produce "soft"-ware.

In the best case the semantics is made mathematically precise. One way to do this is to identify a program with mathematical formulas, and to relate the formulas in a convincing way to the operation of a machine. Similarly, in the best case, a specification is given with mathematical precision, or as a mathematical formula, and a set of rules is given for verifying whether a program meets a specification or not. Modulo the relation between the operation of the machine and the formula, which should be simple enough to convince, one can then *prove* that a program meets its specification.

Once such a semantics, and a set of rules, is given for a language, a notion of *refinement* follows immediately. One program refines another if it meets every specification the other program meets. The programming language can be chosen to be large enough so that specifications are programs as well. To be useful as a specification language, the language has to be a real extension of the language used to describe executable programs, i.e., the language has to contain statements that do not admit (direct) implementation. Given such a language, we can state the job of the programmer thus: "transform the specification statement through a series of refinements into a program that admits an implementation." We can supply the programmer with example transformations that are meant to capture frequently occurring programming problems together with a parameterized proof. If we have a set of transformations that is complete, we can use programs (tools) that assist the programmer in constructing programs by automatically carrying out the (correct) transformations the programmer selects. If the term were not already in use "hard"-ware (i.e., "solid"-ware) would have been a good word to describe programs constructed in this fashion. We believe that parameterized proofs are the basic concept that underlies almost all teaching in computer science. It forms the formal basis of the notion of "archetypes" (templates) [10] and was our goal in [24].

If our programs are formulas, and if the rules for correctness are needed only to define the refinement relation, a good way of going about it is to define the refinement relation directly. This is what we do in this monograph. We define concurrent communicating processes through their communication behaviors, and we define a refinement relation. Everything else follows.

1.2 Goal

This monograph contains an attempt to do for a class of concurrent programs what the predicate-transformer semantics [5, 48] has done for sequential programs. Originally [12] sequential programs were identified with functions from predicates (postconditions) to predicates (weakest preconditions) that satisfy the certain "healthiness conditions" that were meant to guarantee that these functions, called "predicate transformers," correspond to implementable programs. Since then, it has been observed [5, 36, 34] that there are good reasons for replacing the healthiness conditions with a single condition: monotonicity. Not all monotonic predicate transformers correspond to executable programs, but including them all makes it possible to unify partial programs [4], specifications [34], and executable programs [14] in one framework. Furthermore, the fixed-point theorem used to define iteration turns out to rely on monotonicity alone [15].

Monotonic predicate transformers, with the ordering relation defined pointwise from the implication ordering on predicates, form a partial order. This ordering relation is the basis of the refinement calculus [4, 35] , that allows programmers to transform specifications into programs. Because of the firm mathematical foundation of the refinement calculus, program-refinement calculations can be mechanically verified, and hence it is possible to obtain programs that not only provably meet their specification, but for which the possibility of error in the proof has been all but eliminated. In recent years remarkable progress in the area of proof verification systems [48, 1, 45] suggests that this is likely to affect programming not only in principle, but also in practice.

The second area of progress is more technical, and stems from the fact that monotonic predicate transformers with the refinement ordering form a complete lattice [7]. The fixed points of a monotonic function on a complete lattice form a complete lattice themselves, which implies that there is a unique least (and largest) fixed point. This makes it particularly easy to define iteration or recursion for semantics where the underlying model is a lattice; a definition as a least fixed point of a recursive equation is guaranteed to define the construct properly and uniquely. Since all interesting programming languages contain iteration, recursion, or both, it is important to be able to extend models to include such constructs.

Having seen the successes of the lattice-theoretical approach to sequential programming, we try to identify a lattice structure that can capture the meaning of concurrent programs. An important class of concurrent programs consists of programs whose semantics is determined by their communication behaviors [33]. In this monograph we construct a semantics starting with a model and a refinement relation. The refinement relation is inspired by reformulating and extending the partial order for sequential programs. We choose a model that describes processes with undirected point-to-point communication. The model with the refinement relation forms a complete lattice. By examining operators on the model we define the language *synchronizing processes*. The algebraic properties of *synchronizing processes* bear a strong resemblance to those of theoretical CSP (TCSP) [8, 23], but, because we start with refinement, the model is very different.

We motivate our interest in processes with synchronizing communication as follows. Even though communication channels with bounded slack [29] does not seem to be the abstraction used in most high-level programming languages, it remains the abstraction of choice for the specification of concurrent programs that are to be implemented in hardware. The reason is that slack in channels requires buffers and consumes chip area. Since chip area is a resource that needs to be managed, the language needs to address it. Methods exist [43, 9, 32, 39, 41] for translating CSP-like specifications into self-timed [38] (asynchronous, data-driven) VLSI circuits, based on either CSP [23] or UNITY [11]. We are concerned with transformations between CSP-like programs, and between specifications and CSP-like programs.

1.3 Refinement relation

To establish the link between sequential and concurrent programs, we first study a model for sequential programs that concentrates on input-output behavior. In the predicate transformer model, interaction with the environment is limited to initial and final states, and hence we characterize sequential programs by sequences of length two. Identifying a program with a set of such sequences amounts to identifying programs with relations on the state space [16, 13]. As explained in Chapter 2, such a model suffices to describe programs with either angelic or demonic nondeterminism, but not both. We extend the model to sets of sets of pairs, and we show how this model corresponds to monotonic predicate transformers. We reformulate the refinement relation for this model.

In later chapters this refinement relation is extended to sequences of length greater than two. This, however, is not the only difference between sequential and concurrent programs. We also need to reexamine angelic and demonic nondeterminism. Demonic nondeterminism indicates that the execution mechanism is free to choose any of the alternatives. No fairness is assumed. Demonic nondeterminism remains unchanged in the context of concurrent programs. Angelic nondeterminism is the dual of demonic nondeterminism; it specifies that the choice will be made to accommodate demonic nondeterminism in the environment. We maintain this interpretation for concurrent programs, but there is an important new aspect. In the case of sequential programs, nontrivial angelic choice cannot be implemented. Because there is no interaction with the environment during execution of a sequential program, the execution mechanism has no way of telling what choice the environment expects. In the case of *synchronizing processes*, however, some programs involving angelic choice can be implemented. The reason is that synchronization actions involve the environment, and hence the choice can sometimes be made by probing [31, 46] the environment for its next action. We give a mathematical characterization of a class of implementable programs and study their composition. In Section 4.7 the mathematical characterization is related to operational considerations.

1.4 Model

As explained in the previous section, the model for *synchronizing processes* is sets of sets of sequences of undirected actions. Because we want to be able to treat parallel composition and synchronization separately, and because we want to combine synchronization and hiding, we choose a step-trace semantics [47, 42] rather than a purely interleaving semantics. A semantics that allows synchronization within a process [25] differs from the more standard models in that parallel composition is no longer equivalent to a choice among interleavings. This then is another point where the work in this monograph differs from work with similar goals in the context of CCS [33], CSP [23], or process algebras such as ACP [6]. Because we choose to concentrate on processes with point-to-point communication only, a step is either a single communication action (one half of a synchronization, or "port action") or a pair of such actions (a synchronization action). We show that even though this approach complicates the model in comparison to an interleaving semantics, the algebra remains simple.

We define a simple programming language based on this model. The basic process is a port action. Processes may be composed into bigger ones by angelic or demonic composition, by sequential composition, and by parallel composition for processes with disjoint alphabets of port actions. We also define angelic and demonic iteration. The final ingredient of the language is a "connect" operator, that synchronizes pairs of ports, and then hides the ports from observation.

The choice of model and language is motivated by very operational considerations stemming from experience with designing circuits. Parallel composition corresponds to putting two circuits into one box and should therefore be an algebraically trivial operation (nothing really happens). If two channels (corresponding to a pair of wires) are connected, and if only point-to-point connections are used, connection synchronizes the circuits and hides the connected channels from the environment.

The language is simple enough to allow rigorous mathematical treatment, yet large enough so that interesting programming and system-design problems can be formulated in it. After we have introduced the complete language in Chapter 4, we discuss some examples: a data-driven adder and a pipelined memory.

Languages such as CSP [23] contain a choice construct that is neither angelic nor demonic. We show that this construct is essential to describe a nondeterministic merge process, or, in circuit design terminology, an arbiter. The last example in Chapter 4, which discusses mutual exclusion, contains such a merge, and therefore we cannot reason about it in our model. In Chapter 5 we therefore introduce a different operational model; one that can also be used to describe the arbiter.

1.5 Organization of this monograph

Chapter 2 discusses a model for generalized sequential programs and the corresponding refinement ordering. Chapter 3 introduces the step-trace semantics that forms the basis of our model for concurrent programs. We also discuss a class of implementable processes. Chapter 4 introduces demonic nondeterminism and a refinement ordering that is an extension of the ordering in Chapter 2. We discuss operational considerations and give some programming examples. In Chapter 5 we briefly introduce an operational model that describes a larger class of processes.

Chapter 6 contains conclusions and a discussion of possible extensions of this work.

We haven chosen to give proofs of all theorems in this thesis, but to do so in appendices. We feel that even though the proofs are important, and reading them, or even better redoing them, will lead to greater understanding, the sheer number of them would render the thesis nearly unreadable. When we give a programming example, the calculations are included in the main text.

Chapter 2

Sequential programs

2.1 Introduction

In this chapter we give an operational semantics for sequential programs based on their input-output behaviors. Because we want to relate our semantics to the weakest-precondition semantics of statements, which does not mention intermediate states, we define our semantics in terms of initial and final states only. A second reason for not trying to extend the semantics to intermediate states, as is done in [28, 44], is that, as a long term goal, we are interested in semantics of CSP-like languages. State in such languages is not shared between processes; hence, intermediate states are not observable.

The inspiration for our semantics is the operational interpretation of a sequential program as a game of angel versus demon described in [2]. The work may be seen as an extension of relational program semantics [13, 16] to programs with angelic as well as demonic nondeterminism. We do not attempt to develop a calculus; the goal of this chapter is merely to introduce a model and refinement relation similar to the model and refinement relation used in later chapters for concurrent programs.

We show the correspondence between the semantics defined here and the predicate-transformer semantics [48]. We also show how the refinement relation defined in this chapter corresponds to refinement in the predicate-transformer calculus.

2.2 Demonic relations

Before we attempt to define a model in terms of pairs of states for arbitrary sequential programs, we examine two semantics that define programs as a set of states, i.e., a relation. If, for a given initial state, the relation contains only one state pair with that state as its first component, the operational interpretation seems clear. Executing the program from such an initial state terminates in the state that is the second component of the pair. If, however, there is more than one pair with the same initial state, we face a choice in our interpretation. If we want to model programs we can implement, it makes sense to define the execution mechanism as terminating in any one of the states that appear as the second components of the pairs with the same initial states.

This then leads to the following formula for the weakest precondition for relation R and postcondition q. The predicate is defined in terms of its characteristic set.

$$wp.R.q = \{x : x \in \Sigma \ \wedge \ \forall(y : \langle x, y\rangle \in R : q.y) : x\} \tag{2.1}$$

Unless indicated otherwise, we use lower case letters x, y, z to denote variables of type Σ, that is, states. We use lower case letters p, q to denote variables of type $\mathcal{P}(\Sigma)$, that is, predicates. We use lower case letters $r,\ s,\ t,\ u$ to denote variables of type $\Sigma \times \Sigma$, that is, pairs. We use upper case letters $R,\ S,\ T,\ U$ to denote variables of type $\mathcal{P}(\Sigma \times \Sigma)$, that is, sets of pairs of states.

The problem with this definition of wp is that nonterminating programs cannot be represented. The empty relation does not correspond to **abort**, but to **magic** as the following calculation shows.

$$\begin{array}{ll}
& wp.\emptyset.q \\
= & \{ \text{ Definition } wp \ \ \} \\
& \{x : x \in \Sigma \ \wedge \ \forall(y : \langle x, y\rangle \in \emptyset : q.y) : x\} \\
= & \{ \text{ Empty range } \} \\
& \{x : x \in \Sigma : x\} \\
= & \{ \text{ Predicates range over } \Sigma \ \} \\
& true
\end{array}$$

In [16] the problem is solved by adding a special state, "infinity", that indicates nontermination. If we require that postcondition q never contains the infinity state, and if we also require that if the first component of a pair is infinity, the second is infinity as well, we do not have to modify our formula for the weakest precondition. Here we have made a different choice than [16] where wp is defined for postconditions that may include the infinity state.

In Figure 2.1 we have represented some familiar statements assuming a state space of size two (labeled 0 and 1). An edge between initial and final state indicates the pair is in the relation.

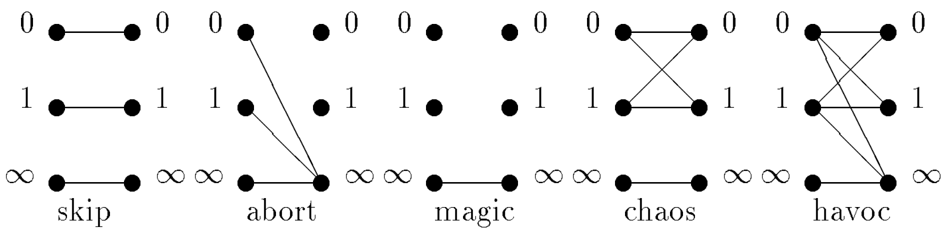


Figure 2.1: Demonic relations.

The operational interpretation of these statements is as follows. Execution of **skip** corresponds to "do nothing." It terminates in the state in which it is started. Execution of **abort** never terminates, independent of the state in which it is started. Execution of **magic** terminates magically; any desired property holds upon termination. Executing **chaos** terminates, but the final state is arbitrary. Executing **havoc** may or may not terminate, and if it terminates the final state is arbitrary.

We now define the constant statements hinted at in Figure 2.1 and the assignment statement. Σ is the set of states. $\Sigma^+ = \Sigma \cup \{\infty\}$.

- **skip** $= \{x : x \in \Sigma^+ : \langle x, x\rangle\}$
- **abort** $= \{x : x \in \Sigma^+ : \langle x, \infty\rangle\}$
- **magic** $= \{\langle\infty, \infty\rangle\}$
- **chaos** $= \{x, y : x, y \in \Sigma : \langle x, y\rangle\} \cup \{\langle\infty, \infty\rangle\}$
- **havoc** $= \{x, y : x, y \in \Sigma : \langle x, y\rangle\} \cup \{x : x \in \Sigma^+ : \langle x, \infty\rangle\}$

- $v := e = \{x : x \in \Sigma : \langle x, x[v := e]\rangle\} \cup \{\langle\infty, \infty\rangle\}$

We may think of the state as a vector of the values of the variables. $x[v := e]$ denotes the vector x with the values of components v replaced by e.

We leave the proofs that these definitions correspond to the usual definitions as predicate transformers to the reader. Very similar proofs are carried out in detail in Section 4 of this chapter. The model and the relational calculus are studied in detail in [16].

2.3 Angelic relations

An alternative interpretation of a relation is to consider pairs with the same initial state to model angelic choice. This leads to the following definition of the *wp*.

$$wp.R.q = \{x : x \in \Sigma \ \wedge \ \exists(y : \langle x, y\rangle \in R : q.y) : x\} \tag{2.2}$$

The problem with this definition is that we cannot model miraculous termination. The statement **magic** "terminates" satisfying the postcondition from any initial state, even if there is no final state that satisfies the postcondition, i.e., $wp.\mathbf{magic}.\mathit{false} = \mathit{true}$, whereas in the formula above $wp.R.\mathit{false} = \mathit{false}$. We solve the problem by extending the state space with a special "happy" state, that indicates miraculous termination. We could maintain our definition of the *wp* and require that any postcondition include the happy state, but we prefer to let predicates range over the "normal" state space and rewrite *wp* as follows.

$$wp.R.q = \{x : x \in \Sigma \ \wedge \ \exists(y : \langle x, y\rangle \in R : y = \ddagger \ \vee \ q.y) : x\} \tag{2.3}$$

With this definition we obtain a model for a language with angelic choice only, much like one of the languages studied in [3]. As healthyness conditions we require that all relations contain the pair $\langle\ddagger, \ddagger\rangle$ and that if the initial state of a pair is the happy state, the final state is also the happy state. Predicates range over Σ. Σ^+ now indicates $\Sigma \cup \{\ddagger\}$. In Figure 2.2 we represent some constant statements for a state space with two states only. The operational

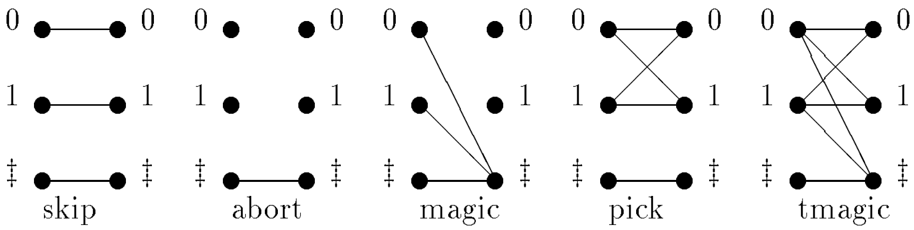


Figure 2.2: Angelic relations.

interpretation of the two new statements is as follows. Executing **pick** terminates in any desired state. Executing **tmagic** terminates miraculously, or in any desired state. We define the constant statements from Figure 2.2, and the assignment statement as follows.

- **skip** $= \{x : x \in \Sigma^+ : \langle x, x \rangle\}$
- **abort** $= \{\langle \ddagger, \ddagger \rangle\}$
- **magic** $= \{x : x \in \Sigma^+ : \langle x, \ddagger \rangle\}$
- **pick** $= \{x, y : x, y \in \Sigma : \langle x, y \rangle\} \cup \{\langle \ddagger, \ddagger \rangle\}$
- **tmagic** $= \{x, y : x, y \in \Sigma : \langle x, y \rangle\} \cup \{x : x \in \Sigma^+ \langle x, \ddagger \rangle\}$
- $v := e = \{x : x \in \Sigma : \langle x, x[v := e] \rangle\} \cup \{\langle \ddagger, \ddagger \rangle\}$

We defer further study of this model until the next section, where an extension of the model is defined.

2.4 Angelic and demonic nondeterminism

We have seen how a relational model can describe statements with either demonic or angelic nondeterminism, but not both. In this section we present a model that can describe statements with both kinds of nondeterminism. Though we do not attempt to give a full calculus or include iteration here, we do study this model in some detail.

We define a program to be a set of sets of pairs of states. A set of pairs of states can be thought of to represent a program, possibly angelic, as in the previous section. The set of such sets models demonic nondeterminism.

This leads us to define a function *wp* as follows.

$$wp.\mathcal{S}.q = \tag{2.4}$$
$$\{x : x \in \Sigma \wedge \forall(S : S \in \mathcal{S} : \exists(s : s \in S \ \wedge \ s.ini = x : q.s.fin \ \vee \ s.fin = \ddagger)) : x\}$$

$r.ini$ is the first component of the pair r, and $r.fin$ the second. We use calligraphic letters $\mathcal{R}$, $\mathcal{S}$, $\mathcal{T}$, $\mathcal{U}$ to denote variables of type $\mathcal{P}(\mathcal{P}(\Sigma^+ \times \Sigma^+))$, or programs.

Of course we could have chosen to extend the model from Section 2 instead, and obtained equivalent results. The choice was made by looking ahead to upcoming chapters. In standard trace theory a component with a larger traceset is considered to implement one with a smaller. This model corresponds to the approach taken in Section 3.

We require the following "healthiness conditions" of all programs $\mathcal{S}$.

- $\mathcal{S} \neq \emptyset$
- $\forall(S : S \in \mathcal{S} : \langle \ddagger, \ddagger \rangle \in S)$
- $\forall(S : S \in \mathcal{S} : \forall(s : s \in S : s.ini = \ddagger \Rightarrow s.fin = \ddagger))$

We define some constants.

- **magic** $= \{\{x : x \in \Sigma^+ : \langle x, \ddagger \rangle\}\}$
- **tmagic** $= \{\{x, y : x, y \in \Sigma : \langle x, y \rangle\} \cup \{x : x \in \Sigma^+ : \langle x, \ddagger \rangle\}\}$
- **abort** $= \{\{\langle \ddagger, \ddagger \rangle\}\}$
- **havoc** $= \{y : y \in \Sigma : \{x : x \in \Sigma : \langle x, y \rangle\} \cup \{\langle \ddagger, \ddagger \rangle\}\} \cup \{\{\ddagger, \ddagger\}\}$
- **pick** $= \{\{x, y : x, y \in \Sigma : \langle x, y \rangle\} \cup \{\langle \ddagger, \ddagger \rangle\}\}$
- **chaos** $= \{y : y \in \Sigma : \{x : x \in \Sigma : \langle x, y \rangle\} \cup \{\langle \ddagger, \ddagger \rangle\}\}$
- **skip** $= \{\{x : x \in \Sigma^+ : \langle x, x \rangle\}\}$

The assignment statement is defined as follows.

- $v := e = \{\{x : x \in \Sigma : \langle x, x[v := e] \rangle\} \cup \{\langle \ddagger, \ddagger \rangle\}\}$

We show that these constructs have the wp's that we expect.

$$
\begin{array}{ll}
& wp.\mathbf{magic}.q \\
= & \{ \text{ Definition } \mathbf{magic}, wp \ \} \\
& \{z : z \in \Sigma \ \wedge \ \forall(S : S \in \{\{x : x \in \Sigma^+ : \langle x, \ddagger\rangle\}\} : \\
& \quad \exists(s : s \in S \ \wedge \ s.ini = z : q.s.fin \ \vee \ s.fin = \ddagger)) : z\} \\
= & \{ \text{ Calculus } \} \\
& \{z : z \in \Sigma \wedge \\
& \quad \exists(s : s \in \{x : x \in \Sigma^+ : \langle x, \ddagger\rangle\} \ \wedge \ s.ini = z : q.s.fin \ \vee \ s.fin = \ddagger) : z\} \\
= & \{ \text{ Calculus } \} \\
& true
\end{array}
$$

$$
\begin{array}{ll}
& wp.\mathbf{tmagic}.q \\
= & \{ \text{ Definition } \mathbf{tmagic}, wp \ \} \\
& \{z : z \in \Sigma \wedge \\
& \ \forall(S : S \in \{\{x, y : x, y \in \Sigma : \langle x, y\rangle\} \cup \{x : x \in \Sigma^+ : \langle x, \ddagger\rangle\}\} : \\
& \quad \exists(s : s \in S \ \wedge \ s.ini = z : q.s.fin \ \vee \ s.fin = \ddagger)) : z\} \\
= & \{ \text{ Calculus } \} \\
& \{z : z \in \Sigma \wedge \\
& \ \exists(s : s \in \{x, y : x, y \in \Sigma : \langle x, y\rangle\} \cup \{x : x \in \Sigma^+ : \langle x, \ddagger\rangle\} \ \wedge \ s.ini = z : \\
& \quad q.s.fin \ \vee \ s.fin = \ddagger) : z\} \\
= & \{ \text{ Calculus } \} \\
& true
\end{array}
$$

We see that **magic** and **tmagic** specify the same predicate transformer. This can be understood as follows. The predicate-transformer semantics is concerned only with guaranteed behavior, not with possible behavior. The "best" choice for the program with regard to its predicate-transformer properties is to always terminate magically if possible.

The following calculation shows that also **abort** and **havoc** specify the same predicate transformer.

$$
\begin{array}{ll}
& wp.\mathbf{abort}.q \\
= & \{ \text{ Definition } \mathbf{abort}, wp \ \}
\end{array}
$$

$$
\begin{aligned}
& \{z : z \in \Sigma \ \wedge \ \forall (S : S \in \{\{\langle \ddagger, \ddagger \rangle\}\} : \\
& \qquad \exists (s : s \in S \ \wedge \ s.ini = z : q.s.fin \ \vee \ s.fin = \ddagger)) : z\} \\
= & \ \{ \text{ Calculus } \} \\
& \{z : z \in \Sigma \ \wedge \ \exists (s : s \in \{\langle \ddagger, \ddagger \rangle\} \ \wedge \ s.ini = z : q.s.fin \ \vee \ s.fin = \ddagger) : z\} \\
= & \ \{ \text{ Calculus } \} \\
& \{z : z \in \Sigma \ \wedge \ z = \ddagger : z\} \\
= & \ \{ \ \ddagger \notin \Sigma \ \} \\
& \mathit{false}
\end{aligned}
$$

$$
\begin{aligned}
& \mathit{wp}.\mathbf{havoc}.q \\
= & \ \{ \text{ Definition } \mathbf{havoc}, \mathit{wp} \ \} \\
& \{z : z \in \Sigma \ \wedge \\
& \ \ \forall (S : S \in \{y : y \in \Sigma : \{x : x \in \Sigma : \langle x, y \rangle\} \cup \{\langle \ddagger, \ddagger \rangle\}\} \\
& \cup \{\{\ddagger, \ddagger\}\} : \\
& \qquad \exists (s : s \in S \ \wedge \ s.ini = z : q.s.fin \ \vee \ s.fin = \ddagger)) : z\} \\
\Rightarrow & \ \{ \text{ Calculus } \} \\
& \{z : z \in \Sigma \ \wedge \ \exists (s : s \in \{\langle \ddagger, \ddagger \rangle\} \ \wedge \ s.ini = z : q.s.fin \ \vee \ s.fin = \ddagger) : z\} \\
= & \ \{ \text{ Calculus } \} \\
& \{z : z \in \Sigma \ \wedge \ z = \ddagger : z\} \\
= & \ \{ \ \ddagger \notin \Sigma \ \} \\
& \mathit{false}
\end{aligned}
$$

Once again the difference may be understood in terms of guaranteed behavior. In the case of **havoc** the program may or may not terminate from any initial state. But since the demon may always make the "worst" choice, termination cannot be guaranteed.

$$
\begin{aligned}
& \mathit{wp}.\mathbf{pick}.q \\
= & \ \{ \text{ Definition } \mathbf{pick}, \mathit{wp} \ \} \\
& \{z : z \in \Sigma \ \wedge \ \forall (S : S \in \{\{x, y : x, y \in \Sigma : \langle x, y \rangle\} \cup \{\langle \ddagger, \ddagger \rangle\}\} : \\
& \qquad \exists (s : s \in S \ \wedge \ s.ini = z : q.s.fin \ \vee \ s.fin = \ddagger)) : z\} \\
= & \ \{ \text{ Calculus } \} \\
& \{z : z \in \Sigma \ \wedge \ \exists (s : s \in \{x, y : x, y \in \Sigma : \langle x, y \rangle\} \cup \{\langle \ddagger, \ddagger \rangle\}
\end{aligned}
$$

$\wedge\ s.ini = z : q.s.fin\ \vee\ s.fin = \ddagger) : z\}$

$= \{$ Calculus $\}$

$\{z : z \in \Sigma\ \wedge\ (z = \ddagger\ \vee\ q \neq \mathit{false}) : z\}$

$= \{\ \ \}$

$q \neq \mathit{false}$

$wp.\mathbf{chaos}.q$

$= \{$ Definition $\mathbf{chaos}, wp$ $\}$

$\{z : z \in \Sigma\ \wedge\ \forall(S : S \in \{x :: \{y :: \langle y, x\rangle\} \cup \{\langle\ddagger,\ddagger\rangle\}\} :$

$\exists(s : s \in S\ \wedge\ s.ini = z : q.s.fin\ \vee\ s.fin = \ddagger)) : z\}$

$= \{$ Calculus $\}$

$\{z : z \in \Sigma\ \wedge\ \forall(x :: \exists(s : s \in \{y :: \langle y, x\rangle\} \cup \{\langle\ddagger,\ddagger\rangle\}\ \wedge\ s.ini = z :$

$q.s.fin\ \vee\ s.fin = \ddagger)) : z\}$

$= \{$ Calculus $\}$

$\{z : z \in \Sigma\ \wedge\ (z = \ddagger\ \vee\ q = \mathit{true}) : z\}$

$= \{\ \ \}$

$q = \mathit{true}$

$wp.\mathbf{skip}.q$

$= \{$ Definition $\mathbf{skip}, wp$ $\}$

$\{z : z \in \Sigma\ \wedge\ \forall(S : S \in \{\{x : x \in \Sigma^+ : \langle x, x\rangle\}\} :$

$\exists(s : s \in S\ \wedge\ s.ini = z : q.s.fin\ \vee\ s.fin = \ddagger)) : z\}$

$= \{$ Calculus $\}$

$\{z : z \in \Sigma\ \wedge$

$\exists(s : s \in \{x : x \in \Sigma^+ : \langle x, x\rangle\}\ \wedge\ s.ini = z : q.s.fin\ \vee\ s.fin = \ddagger) : z\}$

$= \{$ Calculus $\}$

$\{z : z \in \Sigma\ \wedge\ (z \in q\ \vee\ z = \ddagger) : z\}$

$= \{\ \ \}$

q

$$
\begin{aligned}
& wp.(v := e).q \\
= \{ & \text{ Definition v:=e}, wp \ \} \\
& \{z : z \in \Sigma \ \wedge \ \forall(S : S \in \{\{x :: \langle x, x[v := e]\rangle\} \cup \{\langle \ddagger, \ddagger \rangle\}\} : \\
& \quad \exists(s : s \in S \ \wedge \ s.ini = x : q.s.fin \ \vee \ s.fin = \ddagger)) : z\} \\
= \{ & \text{ Calculus } \} \\
& \{z : z \in \Sigma \ \wedge \ \exists(s : s \in \{x :: \langle x, x[v := e]\rangle\} \cup \{\langle \ddagger, \ddagger \rangle\} \ \wedge \ s.ini = z : \\
& \qquad\qquad\qquad q.s.fin \ \vee \ s.fin = \ddagger) : z\} \\
= \{ & \text{ Calculus } \} \\
& \{z : z \in \Sigma \ \wedge \ z[v := e] \in q \ \vee \ z = \ddagger : z\} \\
= \{ & \ \ \} \\
& q[v := e]
\end{aligned}
$$

We define the program constructors $\wedge$, $\vee$, and ; as follows.

- $\mathcal{S} \bigwedge \mathcal{T} = \mathcal{S} \cup \mathcal{T}$
- $\mathcal{S} \bigvee \mathcal{T} = \{S, T : S \in \mathcal{S} \ \wedge \ T \in \mathcal{T} : S \cup T\}$
- $\mathcal{S} \ ; \mathcal{T} = \bigwedge(S : S \in \mathcal{S} : \bigvee(s : s \in S :$
  $\bigwedge(T : T \in \mathcal{T} : \bigvee(t : t \in T \ \wedge \ s.fin = t.ini : \{\{\langle s.ini, t.fin\rangle\}\}))))$
  where $\vee$ over an empty set of statements is defined to be **abort**.

The following properties justify the choice of names.

$$
\begin{aligned}
& wp.\mathcal{S} \bigwedge \mathcal{T}.q.z \\
= \{ & \text{ Definition } \bigwedge, wp \ \} \\
& \forall(S : S \in \mathcal{S} \cup \mathcal{T} : \exists(s : s \in S \ \wedge \ s.ini = z : q.s.fin \ \vee \ s.fin = \ddagger)) \\
= \{ & \text{ Calculus } \} \\
& \forall(S : S \in \mathcal{S} : \exists(s : s \in S \ \wedge \ s.ini = x : q.s.fin \ \vee \ s.fin = \ddagger)) \wedge \\
& \forall(T : T \in \mathcal{T} : \exists(t : t \in T \ \wedge \ t.ini = x : q.t.fin \ \vee \ t.fin = \ddagger)) \\
= \{ & \text{ Definition } wp \ \} \\
& wp.\mathcal{S}.q.z \ \wedge \ wp.\mathcal{T}.q.z
\end{aligned}
$$

$$
\begin{array}{l}
\quad wp.\mathcal{S} \bigvee \mathcal{T}.q.z \\
= \{ \text{ Definition } \bigvee, wp \ \} \\
\quad \forall(S : S \in \{S, T : S \in \mathcal{S} \ \wedge \ T \in \mathcal{T} : S \cup T\} : \\
\qquad \exists(s : s \in S \ \wedge \ s.ini = z : q.s.fin \ \vee \ s.fin = \ddagger)) \\
= \{ \text{ Calculus } \} \\
\quad \forall(S, T : S \in \mathcal{S} \ \wedge \ T \in \mathcal{T} : \\
\qquad \exists(s : s \in S \cup T \ \wedge \ s.ini = x : q.s.fin \ \vee \ s.fin = \ddagger)) \\
= \{ \text{ Calculus } \} \\
\quad \forall(S, T : S \in \mathcal{S} \ \wedge \ T \in \mathcal{T} : \\
\quad\ \ \exists(s : s \in S \ \wedge \ s.ini = z : q.s.fin \ \vee \ s.fin = \ddagger) \vee \\
\quad\ \ \exists(t : t \in T \ \wedge \ t.ini = z : q.t.fin \ \vee \ t.fin = \ddagger)) \\
= \{ \text{ Calculus } \} \\
\quad \forall(S : S \in \mathcal{S} : \exists(s : s \in S \ \wedge \ s.ini = z : q.s.fin \ \vee \ s.fin = \ddagger)) \vee \\
\quad \forall(T : T \in \mathcal{T} : \exists(t : t \in T \ \wedge \ t.ini = z : q.t.fin \ \vee \ s.fin = \ddagger)) \\
= \{ \text{ Definition } wp \ \} \\
\quad wp.\mathcal{S}.q.z \ \vee \ wp.\mathcal{T}.q.z
\end{array}
$$

$$
\begin{array}{l}
\quad wp.\mathcal{S} \ ; \mathcal{T}.q.z \\
= \{ \text{ Definition } ; \ \} \\
\quad wp.\bigwedge(S : S \in \mathcal{S} : \bigvee(s : s \in S : \\
\qquad \bigwedge(T : T \in \mathcal{T} : \bigvee(t : t \in T \ \wedge \ s.fin = t.ini : \{\{\langle s.ini, t.fin\rangle\}\})))).q.z \\
= \{ \text{ Previous two properties, calculus } \} \\
\quad \forall(S : S \in \mathcal{S} : \exists(s : s \in S : \\
\qquad \forall(T : T \in \mathcal{T} : \exists(t : t \in T \ \wedge \ s.fin = t.ini : wp.\{\{\langle s.ini, t.fin\rangle\}\}.q.z)))) \\
= \{ \text{ Definition } wp \ \} \\
\quad \forall(S : S \in \mathcal{S} : \exists(s : s \in S : \forall(T : T \in \mathcal{T} : \\
\qquad \exists(t : t \in T \ \wedge \ s.fin = t.ini : s.ini = z \ \wedge \ (q.t.fin \ \vee \ t.fin = \ddagger))))) \\
= \{ \text{ Calculus } \} \\
\quad \forall(S : S \in \mathcal{S} : \exists(s : s \in S \ \wedge \ s.ini = z : \\
\qquad \forall(T : T \in \mathcal{T} : \exists(t : t \in T \ \wedge \ s.fin = t.ini : q.t.fin \ \vee \ t.fin = \ddagger)))) \\
= \{ \text{ Definition } wp\text{,twice } \} \\
\quad wp.\mathcal{S}.(wp.\mathcal{T}.q).z
\end{array}
$$

2.5 Refinement

We define a refinement relation $\sqsupseteq$ between sets of sets of traces corresponding to the following informal notion: $\mathcal{S} \sqsupseteq \mathcal{T}$ (pronounced "$\mathcal{S}$ refines $\mathcal{T}$") if for every demonic choice for $\mathcal{S}$, $\mathcal{T}$ could have made a demonic choice that is no better. The following calculation shows that this definition of refinement corresponds to the usual definition in terms of wp's. For a set of pairs S we define:

$S.ini = \{x : \exists(y :: \langle x, y\rangle \in S) : x\}$ and
$S \circ x = \{y : \exists(s : s \in S : s = \langle x, y\rangle) : y\}$.
$\circ$ is pronounced "after."

$$
\begin{array}{ll}
& \mathcal{S} \sqsupseteq \mathcal{T} \\
= & \{ \text{ Definition } \sqsupseteq \ \} \\
& \forall(x : x \in \Sigma^+ : \forall(S : S \in \mathcal{S} : \exists(T : T \in \mathcal{T} : S \circ x \supseteq T \circ x))) \\
= & \{ \text{ Healthyness: } \mathcal{S} \neq \emptyset, \forall(S : S \in \mathcal{S} : \langle \ddagger, \ddagger\rangle \in S), s.ini = \ddagger \Rightarrow s.fin = \ddagger \ \ \} \\
& \forall(x : x \in \Sigma : \forall(S : S \in \mathcal{S} : \exists(T : T \in \mathcal{T} : S \circ x \supseteq T \circ x))) \\
= & \{ \text{ Negate } \} \\
& \neg\exists(x :: \exists(S : S \in \mathcal{S} : \forall(T : T \in \mathcal{T} : (T \circ x) \not\subseteq (S \circ x)))) \\
= & \{ \text{ Calculus } \} \\
& \neg\exists(x :: \exists(S : S \in \mathcal{S} : \forall(T : T \in \mathcal{T} : (T \circ x) \cap \overline{(S \circ x)} \neq \emptyset))) \\
= & \{ \text{ Calculus } \} \\
& \neg\exists(x :: \exists(q : q \in \mathcal{P}(\Sigma^+) : \\
& \quad \exists(S : S \in \mathcal{S} : \overline{S \circ x} = q) \ \wedge \ \forall(T : T \in \mathcal{T} : T \circ x \cap q \neq \emptyset))) \\
= & \{ \text{ By mutual implication } \} \\
& \neg\exists(x :: \exists(q : q \in \mathcal{P}(\Sigma^+) : \\
& \quad \exists(S : S \in \mathcal{S} : \overline{S \circ x} \supseteq q) \ \wedge \ \forall(T : T \in \mathcal{T} : T \circ x \cap q \neq \emptyset))) \\
= & \{ \text{ Negate } \} \\
& \forall(x :: \forall(q : q \in \mathcal{P}(\Sigma^+) : \\
& \quad \forall(S : S \in \mathcal{S} : \overline{S \circ x} \not\supseteq q) \ \vee \ \neg\forall(T : T \in \mathcal{T} : T \circ x \cap q \neq \emptyset))) \\
= & \{ \text{ Calculus } \} \\
& \forall(x :: \forall(q : q \in \mathcal{P}(\Sigma^+) : \\
& \quad \forall(S : S \in \mathcal{S} : S \circ x \cap q \neq \emptyset) \Leftarrow \forall(T : T \in \mathcal{T} : T \circ x \cap q \neq \emptyset))) \\
= & \{ \text{ Calculus } \}
\end{array}
$$

$$
\begin{array}{l}
\quad \forall(q : q \in \mathcal{P}(\Sigma^{+}) : \forall(x :: \forall(S : S \in \mathcal{S} : \exists(y : t \in q : \langle x, y\rangle \in S)) \\
\qquad\qquad \Leftarrow \forall(T : T \in \mathcal{T} : \exists(y : y \in q : \langle x, y\rangle \in T)))) \\
= \{ \text{ Calculus } \} \\
\quad \forall(q :: \{x : \forall(S : S \in \mathcal{S} : \exists(s : s \in S : s.ini = x \ \wedge \ q.s.fin)) : x\} \\
\qquad \supseteq \ \{x : \forall(T : T \in \mathcal{T} : \exists(t : t \in T : t.ini = x \ \wedge \ q.t.fin)) : x\}) \\
= \{ \text{ Cases: } q.\ddagger \vee \neg q.\ddagger \ \} \\
\quad \forall(q : q \in \mathcal{P}(\Sigma^{+}) \ \wedge \ q.\ddagger : \\
\qquad\quad \{x : \forall(S : S \in \mathcal{S} : \exists(s : s \in S : s.ini = x \ \wedge \ q.s.fin)) : x\} \\
\qquad \supseteq \ \{x : \forall(T : T \in \mathcal{T} : \exists(t : t \in T : t.ini = x \ \wedge \ q.t.fin)) : x\}) \wedge \\
\quad \forall(q : q \in \mathcal{P}(\Sigma^{+}) \ \wedge \ \neg q.\ddagger : \\
\qquad\quad \{x : \forall(S : S \in \mathcal{S} : \exists(s : s \in S : s.ini = x \ \wedge \ q.s.fin)) : x\} \\
\qquad \supseteq \ \{x : \forall(T : T \in \mathcal{T} : \exists(t : t \in T : t.ini = x \ \wedge \ q.t.fin)) : x\}) \\
= \{ \text{ Calculus } \} \\
\quad \forall(q : q \in \mathcal{P}(\Sigma) : \\
\qquad \{x : \forall(S : S \in \mathcal{S} : \exists(s : s \in S : s.ini = x \ \wedge \ (q.s.fin \ \vee \ s.fin = \ddagger))) : x\} \\
\quad \supseteq \{x : \forall(T : T \in \mathcal{T} : \exists(t : t \in T : t.ini = x \ \wedge \ (q.t.fin \ \vee \ t.fin = \ddagger))) : x\}) \\
\quad \wedge \forall(q : q \in \mathcal{P}(\Sigma) : \\
\qquad \{x : \forall(S : S \in \mathcal{S} : \exists(s : s \in S : s.ini = x \ \wedge \ q.s.fin)) : x\} \\
\quad \supseteq \{x : \forall(T : T \in \mathcal{T} : \exists(t : t \in T : t.ini = x \ \wedge \ q.t.fin)) : x\}) \\
\Rightarrow \{ \text{ Omit second conjunct, definition } wp \ \} \\
\quad \forall(q :: wp.\mathcal{S}.q \Leftarrow wp.\mathcal{T}.q) \\
= \{ \text{ This is the usual definition of refinement with } wp\text{'s. } \} \\
\quad \mathcal{S} \sqsupseteq_{wp} \mathcal{T}
\end{array}
$$

We see that the refinement ordering we have introduced is consistent with *wp* refinement $((\mathcal{S} \sqsupseteq \mathcal{T}) \Rightarrow (\mathcal{S} \sqsupseteq_{wp} \mathcal{T}))$, but we do not have equivalence. The reason is the same as the reason why **abort** and **havoc** and also **magic** and **tmagic** cannot be distinguished by their *wp*'s.

2.6 Summary

In this chapter we have studied a model for sequential programs based on angelic and demonic choice. We have defined a refinement relation and shown that it is consistent with the refinement relation on programs defined by their weakest preconditions. Because the purpose of this chapter was merely to introduce a model of "sets of sets of things" and familiarize the reader with the interpretation in terms of angelic and demonic choice, we stop here. There are a number of issues, however, that we have not dealt with. Perhaps the most important one is that the refinement relation defined here is **not** antisymmetric. The same problem arises in Chapter 4. There we define an equivalence class, and we show that equivalence is preserved by all operators we have introduced.

Chapter 3

A variant of trace theory

3.1 Introduction

In this chapter we develop a version of trace theory [43] that allows both atomic symbols and pair symbols to occur in a trace. Taking sets of such traces as a denotation for processes makes it possible to distinguish between parallel composition and arbitrary interleaving. As an example, consider, in some CSP-like notation, process $(a \| b);\ c$, to which we give denotation $\{abc, bac, (a.b)c\}$ and process $(a;\ b [] b;\ a);\ c$ with denotation $\{abc, bac\}$. (Precise definitions are given in Section 3.) The two processes can be distinguished by synchronizing the a and b actions within the processes. Because we are interested in the synchronization aspect of point-to-point communication, we will assume synchronization is only allowed between pairs of actions.

We shall combine such synchronization with hiding the actions involved in the definition of a **connect** operator. Sometimes we will refer to actions as ports and to pairs of actions as channels.

With the restriction to pairwise synchronization one would expect, the first process to correspond to the process c after a and b are synchronized, whereas the second process would deadlock. Trace theory [43] and CSP [23] ,CCS [33], and ACP [6] deal with hiding rather than connection, and combine parallel composition and synchronization. This simplifies their models, but disallows the introduction of our **connect** operator or other self-synchronization operators [25]. We feel that connecting two ports of one process, in particular when one thinks of processes as circuits, is a reason-able thing to do, and hence we believe the model is worth studying. While this model captures some of the true concurrency aspects of models such as Petri nets [37], it retains the calculational advantages of trace theory. As an added benefit, it turns out that in the model with pairs, processes can be identified with a trace set rather than a trace structure (trace set plus alphabet).

This chapter is organized as follows. After studying some of the properties of projection operators in Section 2, we construct a calculus of processes in Section 3 by defining a number of operators and studying their properties. Because the theory is entirely definitional, the resulting calculus for processes is guaranteed to be consistent. Even though the introduction of pairs in the traces makes the proofs of the algebraic properties more cumbersome, the calculus itself is simple. Proofs of all theorems can be found in Appendix A for section 2 and Appendix B for Section 3. Several of the results and proofs follow those in [43] , but significant differences arise as a result of the inclusion of pairs in the alphabets.

Properties of the operators suggest an interpretation as a concurrent programming language with the synchronization statement as its basic element. In Section 4 we give an example proof for a simple synchronization algorithm in this language. In Section 5 and Appendix C we, identify two subclasses of programs with interesting properties.

3.2 Symbols and traces

A set **Atoms** of uninterpreted symbols is postulated. The set **Pairs** is defined as $\{a, b : a, b \in \textbf{Atoms} : \{a, b\}\}$ and is assumed to be disjoint from the set of atoms. The set **Symbols** is defined as **Atoms** $\cup$ **Pairs**. Elements of **Atoms** will usually be denoted by lower case letters from the beginning of the alphabet. Element of **Pairs** will usually be denoted as a dotted pair of atoms, e.g. $a.b$. Elements of **Symbols** will usually be denoted by lower case letters near the end of the alphabet. Sets of symbols, other than the set **Symbols**, will usually be denoted by upper case letters near the end of the alphabet.

A trace is a finite sequence of symbols. Traces will usually be denoted by the letters r, s, or t. The empty trace is denoted by ϵ. **Traces** is the set of all traces. Other sets of traces will usually be denoted by the letters R, S, or T. Concatenation of traces, and of symbols into traces, is denoted by juxtaposition. The pair constructor . has the highest precedence, so $ab.c = a(b.c)$. It is a symmetric operator; hence, $b.c$ and $c.b$ are the same symbol.

The symbols of trace t, written $\sigma(t)$, is the set of symbols that occur in the trace. Formally,

$$\begin{aligned} \sigma(\epsilon) &= \emptyset \\ \sigma(xt) &= \{x\} \cup \sigma(t) \text{ for all } t \in \textbf{Traces}, x \in \textbf{Symbols} \end{aligned} \tag{3.1}$$

Example: $\sigma(aa.b) = \{a, a.b\}$

The alphabet of trace t, written $\alpha(t)$, is the set of atoms that occur in the trace. Formally,

$$\begin{aligned} \alpha(\epsilon) &= \emptyset \\ \alpha(at) &= \{a\} \cup \alpha(t) \text{ for all } t \in \textbf{Traces}, a \in \textbf{Atoms} \\ \alpha(a.b\ t) &= \{a, b\} \cup \alpha(t) \text{ for all } t \in \textbf{Traces}, a.b \in \textbf{Pairs} \end{aligned} \tag{3.2}$$

Example: $\alpha(aa.b) = \{a, b\}$

The definitions of α and σ are extended to sets of strings and sets of symbols in the usual way.

We define projection of a trace t onto a set of symbols Z, denoted $t \Downarrow Z$, and ejection of Z from t, denoted $t \Uparrow Z$, as follows.

$$\begin{aligned} \epsilon \Downarrow Z &= \epsilon \\ (at) \Downarrow Z &= a(t \Downarrow Z) \text{ for all } t, a \in Z \cap \textbf{Atoms} \\ (at) \Downarrow Z &= t \Downarrow Z \quad \text{ for all } t, a \in \textbf{Atoms} \wedge a \notin Z \\ (xt) \Downarrow Z &= x(t \Downarrow Z) \text{ for all } t, x = a.b, x \in Z \vee (a \in Z \wedge b \in Z) \\ (xt) \Downarrow Z &= a(t \Downarrow Z) \text{ for all } t, x = a.b, x \notin Z \wedge a \in Z \wedge b \notin Z \\ (xt) \Downarrow Z &= t \Downarrow Z \quad \text{ for all } t, x = a.b, x \notin Z \wedge a \notin Z \wedge b \notin Z \end{aligned} \tag{3.3}$$

$$\tag{3.4}$$

$$\begin{array}{rcll}
\epsilon \Uparrow Z &=& \epsilon & \\
(at) \Uparrow Z &=& a(t \Uparrow Z) & \text{for all } t, a \in \textbf{Atoms} \wedge a \notin Z \\
(at) \Uparrow Z &=& t \Uparrow Z & \text{for all } t, a \in \textbf{Atoms} \cap Z \\
(xt) \Uparrow Z &=& x(t \Uparrow Z) & \text{for all } t, x = a.b, x \notin Z \wedge a \notin Z \wedge b \notin Z \\
(xt) \Uparrow Z &=& a(t \Uparrow Z) & \text{for all } t, x = a.b, x \notin Z \wedge a \notin Z \wedge b \in Z \\
(xt) \Uparrow Z &=& t \Uparrow Z & \text{for all } t, x = a.b, x \in Z \vee (a \in Z \wedge b \in Z)
\end{array}$$

Example: $a.b \Downarrow \{a\} = a \qquad a.b \Uparrow \{a\} = b$

Just as the projection of a trace is defined as the concatenation of the projections of the elements, the projection of a set is defined as the set of the projections of the elements in that set.

Calculating with these projection and ejection operators is more cumbersome than calculating with the projection operators of trace theory without pairs. In the remainder of this section we give a list of properties that have been used in proofs of the properties in the following section. Most proofs, given in Appendix A, are straightforward (though often lengthy) and require induction on the length of the traces.

The following three non-theorems, that *are* theorems for trace theory without pairs [43], show that the calculus indeed differs from standard trace theory.

$(t \Downarrow X) \Downarrow Y = t \Downarrow (X \cap Y)$
Counterexample: $t := a.b, X := \{a.b\}, Y := \{b\}$
$(t \Uparrow X) \Uparrow Y = t \Uparrow (X \cup Y)$
Counterexample: $t := a.b, X := \{a\}, Y := \{a.b\}$
$X \subseteq Y \Rightarrow ((t \Uparrow X) \Uparrow Y = t \Uparrow Y)$
Counterexample: $t := a.b, X := \{b\}, Y := \{a.b, b\}$

Fortunately, we can carry out the proofs in the following using these weaker versions of the theorems.

$$(X \supseteq Y) \Rightarrow (t \Downarrow X) \Downarrow Y = t \Downarrow Y \tag{3.5}$$

$$(X \subseteq Y) \Rightarrow (t \Downarrow X) \Downarrow Y = t \Downarrow X \tag{3.6}$$

$$(X \supseteq Y) \Rightarrow (t \Uparrow X) \Uparrow Y = t \Uparrow X \tag{3.7}$$

One more rather specialized theorem is needed to make up for the loss of stronger theorems from trace theory.

$$(\sigma(s) \subseteq Y \ \wedge \ r \Downarrow (X \cup Y) = s) \Rightarrow (r \Downarrow Y = s) \tag{3.8}$$

The following three theorems correspond directly to theorems in standard trace theory. Note, however, that theorem 3.11 mentions σ rather than α.

$$(t \Downarrow X = t) = (t \Uparrow X = \epsilon) \tag{3.9}$$

$$(t \Downarrow X) \Uparrow X = \epsilon = (t \Uparrow X) \Downarrow X \tag{3.10}$$

$$(t \Uparrow \sigma(s) = \epsilon \ \wedge \ t \Downarrow \sigma(s) = s) = (s = t) \tag{3.11}$$

Because synchronization will not be part of a parallel composition operator, we shall require that alphabets of processes composed in parallel are disjoint. Hence we are also interested in properties that involve disjoint alphabets.

$$\alpha(X) \cap \alpha(Y) = \emptyset \Rightarrow (t \Uparrow X) \Uparrow Y = t \Uparrow (X \cup Y) \tag{3.12}$$

$$\alpha(X) \cap \alpha(Y) = \emptyset \Rightarrow t \Downarrow X = (t \Uparrow Y) \Downarrow X \tag{3.13}$$

One might expect a counterpart to theorem 3.12 for projections. The following calculation shows that theorem holds as well. Assuming $\alpha(X) \cap \alpha(Y) = \emptyset$ we have

$$\begin{array}{ll}
 & (t \Downarrow X) \Downarrow Y \\
= & \{ \ 3.13 \ \} \\
 & ((t \Uparrow Y) \Downarrow X) \Downarrow Y \\
= & \{ \ 3.12 \ \} \\
 & (t \Uparrow (X \cup Y)) \Downarrow Y \\
= & \{ \ 3.12 \ \} \\
 & ((t \Uparrow X) \Uparrow Y) \Downarrow Y \\
= & \{ \ 3.10 \ \} \\
 & \epsilon \\
= & \{ \ \text{Definition} \Downarrow \ \} \\
 & t \Downarrow \emptyset \\
= & \{ \ \alpha(X) \cap \alpha(Y) = \emptyset \Rightarrow X \cap Y = \emptyset \ \} \\
 & t \Downarrow (X \cap Y)
\end{array}$$

.

Because the **connect** operator, defined in the next section, takes a set of pairs as its first argument, properties that restrict one set of symbols to a set of pairs are of interest as well.

$$X \subseteq \mathbf{Pairs} \Rightarrow (t \Uparrow X) \Downarrow Y = (t \Uparrow X) \Downarrow (X \cup Y) \tag{3.14}$$

$$Y \subseteq \mathbf{Pairs} \Rightarrow (t \Uparrow X = \epsilon) \ = \ ((t \Uparrow Y) \Uparrow (X - Y) = \epsilon) \tag{3.15}$$

$$\begin{gathered} X \subseteq \mathbf{Pairs} \Rightarrow \\ ((Y \Uparrow X) \Downarrow \alpha(X) = \emptyset) \Rightarrow (t \Uparrow Y) \Uparrow X \ = \ (t \Uparrow X) \Uparrow (Y \Uparrow X) \end{gathered} \tag{3.16}$$

$$\begin{gathered} X \subseteq \mathbf{Pairs} \Rightarrow \\ (t \Uparrow X) \Downarrow ((\sigma(s) \Uparrow X) \cup (Y \Uparrow X)) = s \\ \Rightarrow \exists(r : r \Uparrow X = t \Uparrow X : r \Downarrow (\sigma(s) \cup Y)) = s \end{gathered} \tag{3.17}$$

Finally, we have some properties that hold for disjoint sets of pairs.

$$\begin{gathered} X, Y \subseteq \mathbf{Pairs} \ \wedge \ \alpha(X) \cap \alpha(Y) = \emptyset \Rightarrow \\ (t \Uparrow (X \cup Y)) \Downarrow \alpha(X \cup Y) = \epsilon \\ = ((t \Uparrow (X \cup Y)) \Downarrow \alpha(X) = \epsilon \ \wedge \ (t \Uparrow (X \cup Y) \Downarrow \alpha(Y) = \epsilon)) \end{gathered} \tag{3.18}$$

$$\begin{gathered} X, Y \subseteq \mathbf{Pairs} \ \wedge \ \alpha(X) \cap \alpha(Y) = \emptyset \Rightarrow \\ ((t \Uparrow X) \Downarrow \alpha(X) = \epsilon) = ((t \Uparrow (X \cup Y)) \Downarrow \alpha(X) = \epsilon) \end{gathered} \tag{3.19}$$

3.3 Operators and processes

In this section we construct a calculus for processes by defining operators and listing some of their properties. Even though adding pairs to the traces adds complexity to the proofs of the properties, given in Appendix B, the calculus itself remains quite simple.

We give names to two special tracesets.

$$\mathbf{demon} \ = \emptyset \qquad \mathbf{skip} \ = \{\epsilon\} \tag{3.20}$$

Informally, **skip** corresponds to the process that always terminates, and **demon** corresponds to a process that is deadlocked (in one of its components). Operators $\sqcup$, $\|$, ; , and **connect** are defined as follows.

$$S \sqcup T \;=\; S \cup T \tag{3.21}$$

$$\begin{aligned} S \| T \;=& \{r, s, t : s \in S \;\wedge\; t \in T \;\wedge\; r \Uparrow (\sigma(s) \cup \sigma(t)) \;=\; \epsilon \\ & \wedge\; r \Downarrow \sigma(s) = s \;\wedge\; r \Downarrow \sigma(t) = t \;:\; r\} \end{aligned} \tag{3.22}$$

Example: $\{ab, c\} \| \{d\} = \{abd, ab.d, adb, a.db, dab, cd, c.d, dc\}$

$$S;\; T \;=\; \{s, t : s \in S, t \in T : st\} \tag{3.23}$$

Example: $\{ab, c\};\; \{d\} = \{abd, cd\}$

$$S \;\textbf{connect}\; X = \{s : s \in S \Uparrow X \;\wedge\; s \Downarrow \alpha(X) = \epsilon : s\} \tag{3.24}$$

Example: $\{a.b\ c\}$ **connect** $\{a.b\} = \{c\}$ and $\{a.b\ a\}$ **connect** $\{a.b\} = \{\}$

The operators satisfy the following properties.

$$S \sqcup \textbf{demon} \;= S \tag{3.25}$$

$$S \sqcup S = S \tag{3.26}$$

$$S \sqcup T = T \sqcup S \tag{3.27}$$

$$(R \sqcup S) \sqcup T = R \sqcup (S \sqcup T) \tag{3.28}$$

$$S \| \textbf{skip} \;= S \tag{3.29}$$

$$S \| \textbf{demon} \;= \textbf{demon} \tag{3.30}$$

$$S \| T = T \| S \tag{3.31}$$

$$(R \| S) \| T = R \| (S \| T) \tag{3.32}$$

$$S; \textbf{skip} \;= \textbf{skip}\;; S = S \tag{3.33}$$

$$S; \textbf{demon} \;= \textbf{demon}\;; S = \textbf{demon} \tag{3.34}$$

$$(R; S);\; T = R; (S;\; T) \tag{3.35}$$

$$R; (S \sqcup T) = (R; S) \sqcup (R;\; T) \tag{3.36}$$

In the following four properties X and Y are subsets of **Pairs**.

$$(S \sqcup T)\ \textbf{connect}\ X \ = \ (S\ \textbf{connect}\ X) \sqcup (T\ \textbf{connect}\ X) \tag{3.37}$$

$$\alpha(X) \cap \alpha(Y) = \emptyset \Rightarrow \tag{3.38}$$
$$(S\ \textbf{connect}\ X)\ \textbf{connect}\ Y \ = S\ \textbf{connect}\ (X \cup Y)$$

$$(S;\ T)\ \textbf{connect}\ \ X \ = \ (S\ \textbf{connect}\ \ X)\ ;\ \ (T\ \textbf{connect}\ \ X) \tag{3.39}$$

$$\alpha(X) \cap \alpha(S) = \emptyset \Rightarrow (S \| T)\ \textbf{connect}\ X \ = \ S \| (T\ \textbf{connect}\ X) \tag{3.40}$$

$$\{a, b\} \cap (\alpha(S_0) \cup \alpha(S_1) \cup \alpha(T_0) \cup \alpha(T_1)) = \emptyset \Rightarrow \tag{3.41}$$
$$((S_0;\ a;\ S_1) \| (T_0;\ b;\ T_1))\ \textbf{connect}\ \{a.b\} \ = \ (S_0 \| T_0); (S_1 \| T_1))$$

3.4 Example: N-way synchronization

In this section we use the properties from the previous section to give a proof of a simple deterministic synchronization algorithm.

problem specification

Given a constant $n : n > 1$ and processes $\{i : 0 \leq i < n : A_i\}$ and $\{i : 0 \leq i < n : B_i\}$, give a set of connections X and a set of processes $\{i : 0 \leq i < n : S_i\}$ such that

$$\begin{array}{l} \quad \|(i : 0 \leq i < n : A_i;\ S_i;\ B_i)\ \textbf{connect}\ X \\ = \\ \quad \|(i : 0 \leq i < n : A_i)\ ;\ \ \|(i : 0 \leq i < n : B_i). \end{array}$$

a solution (synchronization over a line)

$X = \{i : 0 < i < n : (x_{2i-2}.x_{2i-1})\} \cup \{i : 0 < i < n : (y_{2i-2}.y_{2i-1})\}$,
$S_{n-1} = x_{2n-3};\ y_{2n-3}$,
$S_i = x_{2i};\ x_{2i-1};\ y_{2i-1};\ y_{2i}$ for $0 < i < n-1$,
$S_0 = x_0;\ y_0$,

where $\forall(i, j : 0 \leq i, j < n : \alpha(S_i) \cap \alpha(A_j) = \emptyset \ \wedge \ \alpha(S_i) \cap \alpha(B_j) = \emptyset)$.

Processes 1 and 0 communicate twice in sequence; hence, one of these communications may be omitted.

proof, by induction on n
Base case ($n = 2$)

$$
\begin{array}{l}
\quad ((A_1;\, x_1;\, y_1;\, B_1)\|(A_0;\, x_0;\, y_0;\, B_0)) \textbf{ connect } \{x_0.x_1, y_0.y_1\} \\
= \{ \text{ Property of } \textbf{connect} \} \\
\quad (((A_1;\, x_1;\, y_1;\, B_1)\|(A_0;\, x_0;\, y_0;\, B_0)) \textbf{ connect } \{x_0.x_1\}) \\
\qquad \textbf{connect } \{y_0.y_1\} \\
= \{ \ (3.41)\ S_0, T_0, S_1, T_1, a, b := A_1, A_0, (y_1;\, B_1), (y_0;\, B_0), x_1, x_0 \ \} \\
\quad ((A_1\|A_0);\, ((y_1;\, B_1)\|(y_0;\, B_0))) \textbf{ connect } \{y_0.y_1\} \\
= \{ \ (\alpha(A_0) \cup \alpha(A_1)) \cap \{y_0, y_1\} = \emptyset, \textbf{ skip} \text{ unit of } ; \ \} \\
\quad (A_1\|A_0);\, (((\textbf{skip};\, y_1;\, B_1)\|(\textbf{skip};\, y_0;\, B_0)) \textbf{ connect } \{y_0.y_1\}) \\
= \{ \ (3.41)\ S_0, T_0, S_1, T_1, a, b := \textbf{skip}, \textbf{skip}, B_1, B_0, y_1, y_0 \ \} \\
\quad (A_1\|A_0);\, ((\textbf{skip}\|\textbf{skip});\, (B_1\|B_0)) \\
= \{ \ (\textbf{skip}\|\textbf{skip}) = \textbf{skip}, \textbf{ skip} \text{ unit of } ; \ \} \\
\quad (A_1\|A_0);\, (B_1\|B_0)
\end{array}
$$

Induction step

$$
\begin{array}{l}
\quad (\quad (A_{n-1};\, x_{2n-3};\, y_{2n-3};\, B_{n-1}) \\
\quad \| \quad \|(i : 0 < i < n-1 : A_i;\, x_{2i};\, x_{2i-1};\, y_{2i-1};\, y_{2i};\, B_i) \\
\quad \| \quad (A_0;\, x_0;\, y_0;\, B_0) \\
\quad) \textbf{ connect } \{i : 0 < i < n : (x_{2i-2}.x_{2i-1})\} \cup \{i : 0 < i < n : (y_{2i-2}.y_{2i-1})\} \\
= \{ \ n > 1, \text{properties of } \textbf{connect} \ \} \\
\quad (\quad ((A_{n-1};\, x_{2n-3};\, y_{2n-3};\, B_{n-1})\| \ (A_{n-2};\, x_{2n-4};\, x_{2n-5};\, y_{2n-5};\, y_{2n-4};\, B_{n-2}) \\
\qquad \textbf{connect } \{x_{2n-4}.x_{2n-3}\}) \\
\quad \| \quad \|(i : 0 < i < n-2 : A_i;\, x_{2i};\, x_{2i-1};\, y_{2i-1};\, y_{2i};\, B_i) \\
\quad \| \quad (A_0;\, x_0;\, y_0;\, B_0) \\
\quad) \textbf{ connect } \{i : 0 < i < n-1 : (x_{2i-2}.x_{2i-1})\} \cup \{i : 0 < i < n : (y_{2i-2}.y_{2i-1})\} \\
= \{ \ (3.41) \ \} \\
\quad (\quad ((A_{n-1}\|A_{n-2});\, ((y_{2n-3};\, B_{n-1})\|(x_{2n-5};\, y_{2n-5};\, y_{2n-4};\, B_{n-2}))\ ) \\
\quad \| \quad \|(i : 0 < i < n-2 : A_i;\, x_{2i};\, x_{2i-1};\, y_{2i-1};\, y_{2i};\, B_i) \\
\quad \| \quad (A_0;\, x_0;\, y_0;\, B_0) \\
\quad) \textbf{ connect } \{i : 0 < i < n-1 : (x_{2i-2}.x_{2i-1})\} \cup \{i : 0 < i < n : (y_{2i-2}.y_{2i-1})\} \\
= \{ \text{ Properties of } \textbf{connect}, \text{ alphabets } \} \\
\quad (\quad (A_{n-1}\|A_{n-2}); \\
\quad (((\textbf{skip} \ ;\, y_{2n-3};\, B_{n-1})\|(x_{2n-5};\, y_{2n-5};\, y_{2n-4};\, B_{n-2})) \textbf{ connect } \{y_{2n-4}.y_{2n-3}\}) \\
\quad \| \quad \|(i : 0 < i < n-2 : A_i;\, x_{2i};\, x_{2i-1};\, y_{2i-1};\, y_{2i};\, B_i) \\
\quad \| \quad (A_0;\, x_0;\, y_0;\, B_0) \\
\quad) \textbf{ connect } \{i : 0 < i < n-1 : (x_{2i-2}.x_{2i-1})\} \cup \{i : 0 < i < n-1 : (y_{2i-2}.y_{2i-1})\} \\
= \{ \ (3.41) \ \}
\end{array}
$$

$$
\begin{array}{ll}
(& ((A_{n-1}\|A_{n-2});\ (\mathbf{skip}\ \|(x_{2n-5};\ y_{2n-5}));\ (B_{n-1}\|B_{n-2})\ ) \\
\| & \|(i : 0 < i < n-2 : A_i;\ x_{2i};\ x_{2i-1};\ y_{2i-1};\ y_{2i};\ B_i) \\
\| & (A_0;\ x_0;\ y_0;\ B_0) \\
) & \mathbf{connect}\ \{i : 0 < i < n-1 : (x_{2i-2}.x_{2i-1})\} \cup \{i : 0 < i < n-1 : (y_{2i-2}.y_{2i-1})\}
\end{array}
$$

$= \{\ \mathbf{skip}\|A = A\ \}$

$$
\begin{array}{ll}
(& ((A_{n-1}\|A_{n-2});\ x_{2n-5};\ y_{2n-5};\ (B_{n-1}\|B_{n-2})\ ) \\
\| & \|(i : 0 < i < n-2 : A_i;\ x_{2i};\ x_{2i-1};\ y_{2i-1};\ y_{2i};\ B_i) \\
\| & (A_0;\ x_0;\ y_0;\ B_0) \\
) & \mathbf{connect}\ \{i : 0 < i < n-1 : (x_{2i-2}.x_{2i-1})\} \cup \{i : 0 < i < n-1 : (y_{2i-2}.y_{2i-1})\}
\end{array}
$$

$= \{$ Induction hypothesis, associativity of $\|$ $\}$

$$\|(i : 0 \le i < n : A_i)\ ;\ \ \|(i : 0 \le i < n : B_i)$$

end of proof

3.5 Two subclasses

In this section we study two classes of programs. We show that each class is closed under all but two of the composition operators we have introduced. The intersection of both classes is closed under all composition operations except $\sqcup$ composition. The motivation for this section is that not all programs we have defined in the previous section can be implemented, or implemented efficiently. Informally, an angelic choice is not (efficiently) implementable if the process cannot decide which alternative to execute by probing [31] the environment. We return to this issue in Section 4.7.

An example of a choice that cannot always be implemented is $\{a, \epsilon\}$. Because the process may be composed sequentially with a process $\{a\}$, we cannot decide safely to implement $\{a, \epsilon\}$ as **if** $probe(a) \rightarrow a \ [\!]\ probe(anything\ but\ a) \rightarrow$ **skip fi**. This implementation would cause the process $(((a \sqcup \mathbf{skip}\ );\ a)\|a')\mathbf{connect}\ \{a.a'\}$ to deadlock on the second a action, whereas a truly angelic choice would have chosen **skip** to avoid deadlock.

A similar problem arises for the process $\{ba, a\}$. Parallel composition with a process $\{a'\}$, followed by connecting $a.a'$, leads to our first problem.

This section discusses a class of processes that can be implemented without restrictions on compositionality or the environment. The second example above leads us to studying the class of "connect compositional" (cc) processes. This class is closed for all composition operators except sequential composition and $\sqcup$ composition. The first example leads to studying the class of "prefix reduced" (pr) processes. This class is closed for all composition operators except **connect** and $\sqcup$ composition. The class of processes that are both prefix reduced and connect compositional is closed under all operations except $\sqcup$ composition. The proofs in this chapter bear some similarity to [40], where classes of delay-insensitive circuits are studied.

Proofs of the theorems were much more difficult than we had expected. Unlike the proofs in the previous section, this does not seem to be solely a consequence of the introduction of pairs in the traces. A small consolation is that similar proofs in [40] are of similar complexity. All proofs are given in detail in Appendix C.

An important practical extension to the classes mentioned here is the use of signaling sets [43]. Signaling sets are sets of port operations. Their use corresponds to a restriction on the use of the connect operator. If we know that a connect operator always takes all or none of the elements from the signaling sets as its argument we can extend the class cc. The new class then corresponds to requiring that the processes with all elements of a signaling set replaced by a representative are in cc.

Definition 3.42 *For any traceset S we define connect compositional, or $cc.S$, as*

$$\begin{gathered}\forall(r, x, y : r \in \mathbf{Traces};\ x, y \in \mathbf{Symbols} : \\ \alpha(x) \cap \alpha(y) = \emptyset \ \wedge\ S \circ rx \neq \emptyset \ \wedge\ S \circ ry \neq \emptyset \Rightarrow \\ S \circ rxy \neq \emptyset \ \wedge\ S \circ ryx \neq \emptyset \ \wedge\ S \circ rxy = S \circ ryx)\end{gathered}$$

Example: $cc.\{abc, bac\} = true,\ cc.\{abc, bad\} = false$

Informally, the class cc contains processes for which, if several actions are simultaneously possible, they can be performed in either order, and the future behavior of the process is independent of the order chosen.

The class of connect compositional processes is closed under parallel composition and connection.

Theorem 3.43 $cc.S \Rightarrow \forall(z : z \in \mathbf{Pairs} : cc.(\mathbf{connect}\ \ \{z\}\ S))$

Theorem 3.44 $cc.S \ \wedge \ cc.T \Rightarrow cc.(S \| T)$

The class of prefix reduced processes is defined as follows.

Definition 3.45 *For any traceset S we define prefix reduced, or $pr.S$, as*

$$\forall(s, t : s, t \in S : \neg(s < t)$$

where $<$ is the prefix ordering on strings.

Example: $pr.\{a, ba, bb\} = true, \ cc.\{a, ab\} = false$

The class of prefix reduced processes is closed under parallel and sequential composition.

Theorem 3.46 $(pr.S \ \wedge \ pr.T) \Rightarrow pr.(S; \ T)$

Theorem 3.47 $(pr.S \ \wedge \ pr.T) \Rightarrow pr.(S \| T)$

The following two theorems show that the class of processes that are both connect compositional and prefix reduced is closed under parallel composition, sequential composition, and connection.

Theorem 3.48 $(cc.S \ \wedge \ pr.S \ \wedge \ cc.T) \Rightarrow cc.(S; \ T)$

Theorem 3.49 $(pr.S \ \wedge \ cc.S) \Rightarrow \forall(z :: pr.(\textbf{connect} \ \ \{z\} \ S))$

In Section 4.7 we discuss how these algebraic properties relate to operational considerations.

3.6 Summary

In this chapter we have shown how traces with pair symbols can be used to model concurrent processes with point-to-point synchronization. We defined several familiar composition operators and a new one in terms of this model. The familiar composition operators have the properties we expect, and the new **connect** operator has satisfying algebraic properties as well. While we feel that the list of properties in Section 3 is what we had hoped for, the road toward them proved to be a lot more tortuous than intended. Undoubtedly related to this, the list of properties in Section 2 seems somewhat haphazard and ripe for improvement.

Chapter 4

Sets of sets of traces

4.1 Angelic and demonic nondeterminism

In the previous chapter a process was specified as a set of traces. Tracesets alone, however, are insufficient to describe both angelic and demonic nondeterminism. In this chapter we extend the model to describe both. The observation that tracesets alone do not suffice is not new, but our extension is new.

Consider two processes. Process A and process B both execute a followed by b or b followed by a, but the choice is angelic in the case of process A and demonic in the case of process B. According to the previous chapter, process A should be specified as $\{ab, ba\}$, and we have no representation for process B; our model has too few degrees of freedom. We add a dimension by going from sets of traces to sets of sets of traces, with the new dimension representing demonic nondeterminism, as was done in Chapter 2. In this model processes A and B are now represented as follows.

- A $\quad ab \sqcup ba \ = \ \{\{ab, ba\}\}$
- B $\quad ab \sqcap ba \ = \ \{\{ab\}, \{ba\}\}$

In this chapter we study sets of sets of traces as a model for a language with synchronization statements as its primitive operation. We specify a refinement ordering for this model in Section 2. The refinement ordering leads us to define classes of equivalent programs. We show that the refinement ordering with these equivalence classes is a partial order. We show that demonic and angelic choice correspond to highest (greatest) lower bound and lowest (least) upper bound with respect to the refinement ordering. The equivalence classes, which we call processes, with demonic and angelic nondeterminism as meet and join, form a complete distributive lattice. Proofs for the theorems in Section 2 are given in Appendix D.

In Section 3 we define sequential, parallel, and connect composition. We give a list of algebraic properties that can be proven from these definitions and the properties of the lattice of processes. Proofs for this Section are given in Appendix E. In Section 4 we introduce two kinds of iteration: angelic iteration and demonic iteration. Proofs for this section are given in Appendix F. In Section 5 we study duals of processes. We prove the properties of duals in Appendix G.

We end with a summary of the main results of this chapter in Section 6 .

4.2 A refinement ordering

Following the suggestions in the introduction we define:

Definition 4.1 A *process* is a set of sets of *traces.*

We will use calligraphic letters such as $\mathcal{R}$, $\mathcal{S}$, and $\mathcal{T}$ to denote processes.

Refinement corresponds to decreasing the amount of demonic nondeterminism, or increasing angelic nondeterminism. This leads us to the following definition of process refinement: $\sqsupseteq$.

Definition 4.2
For $\mathcal{S}, \mathcal{T} \subseteq$ **Tracesets** we define
$\mathcal{S} \sqsupseteq \mathcal{T}$
$=$
$\forall(S : S \in \mathcal{S} : \exists(T : T \in \mathcal{T} : S \supseteq T))$

We give a few examples:

$\{\{a, b\}\} \sqsupseteq \{\{a\}\}$
$\{\{a\}\} \sqsupseteq \{\{a\}, \{b\}\}$
$\{\{c, d\}\} \not\sqsupseteq \{\{a\}, \{b\}\}$
$\{\{aa\}\} \not\sqsupseteq \{\{aa, ba\}, \{aa, ca\}\}$

We have,

Theorem 4.3 $\mathcal{T} \sqsupseteq \mathcal{T}$

Theorem 4.4 $\mathcal{R} \sqsupseteq \mathcal{S} \ \wedge \ \mathcal{S} \sqsupseteq \mathcal{T} \Rightarrow \mathcal{R} \sqsupseteq \mathcal{T}$

therefore the relation $\sqsupseteq$ defines a preorder on **Tracesets**. However, $\sqsupseteq$ is not a partial order, as the following theorem shows.

Theorem 4.5

$$\mathcal{S} \sqsupseteq \mathcal{T} \ \wedge \ \mathcal{T} \sqsupseteq \mathcal{S} =$$

$$\forall(S : S \in \mathcal{S} \ \wedge \ \forall(S0 : S0 \in \mathcal{S} \ \wedge \ S0 \subseteq S : S0 = S) : S \in \mathcal{T}) \wedge$$
$$\forall(T : T \in \mathcal{T} \ \wedge \ \forall(T0 : T0 \in \mathcal{T} \ \wedge \ T0 \subseteq T : T0 = T) : T \in \mathcal{S})$$

We may read the previous theorem as follows. Two processes are equivalent if their minimal elements are equal. This notion of equivalence makes sense if we realize that a demonic choice between two sets, one of which is a subset of the other, always yields the smaller, less angelic, one. We therefore define equivalence between processes as follows.

Definition 4.6 $\mathcal{S} \simeq \mathcal{T} \ = \ \mathcal{S} \sqsupseteq \mathcal{T} \ \wedge \ \mathcal{T} \sqsupseteq \mathcal{S}$

We have,

Theorem 4.7 $\mathcal{S} \simeq \mathcal{S}$

Theorem 4.8 $(\mathcal{S} \simeq \mathcal{T}) = (\mathcal{T} \simeq \mathcal{S})$

Theorem 4.9 $\mathcal{R} \simeq \mathcal{S} \ \ \wedge \ \ \mathcal{S} \simeq \mathcal{T} \Rightarrow \mathcal{R} \simeq \mathcal{T}$

therefore equivalence between processes is indeed an equivalence relation. We also have the following theorem.

Theorem 4.10 $\mathcal{S} \simeq \{S : S \in \mathcal{S} \ \wedge \ \forall(S0 : S0 \in \mathcal{S} \ \wedge \ S0 \subseteq S : S0 = S) : S\}$

Hence a process is equivalent to the set of its minimal elements. The following theorem states that the set of minimal elements uniquely represents an equivalence class.

Theorem 4.11 $(\mathcal{S} \simeq \mathcal{T})$

$$=$$

$$(\{S : S \in \mathcal{S} \ \wedge \ \forall(S0 : S0 \in \mathcal{S} \ \wedge \ S0 \subseteq S : S0 = S) : S\}$$
$$=$$
$$\{T : T \in \mathcal{T} \ \wedge \ \forall(T0 : T0 \in \mathcal{T} \ \wedge \ T0 \subseteq T : T0 = T) : T\})$$

On the equivalence classes, $\sqsupseteq$ defines a partial order. The proof is immediate from the definition of $\simeq$.

We define demonic and angelic composition as follows.

Definition 4.12 $\mathcal{S} \sqcap \mathcal{T} = \mathcal{S} \ \cup \ \mathcal{T}$

Example: $\{\{a, b\}\} \sqcap \{\{c\}\{d\}\} = \{\{a, b\}\{c\}\{d\}\}$

Definition 4.13 $\mathcal{S} \sqcup \mathcal{T} = \{S, T : S \in \mathcal{S}, T \in \mathcal{T} : S \sqcup T\}$

Example: $\{\{a, b\}\} \sqcup \{\{c\}\{d\}\} = \{\{a, b, c\}\{a, b, d\}\}$

The following theorem relates the structure of $\mathcal{S}$ to demonic and angelic nondeterminism.

Theorem 4.14 $\mathcal{S} = \sqcap(S : S \in \mathcal{S} : \sqcup(s : s \in S : \{\{s\}\}))$

The following four theorems show that $\mathcal{S} \sqcap \mathcal{T}$ is a greatest lower bound of $\mathcal{S}$ and $\mathcal{T}$ and $\mathcal{S} \sqcup \mathcal{T}$ is a least upper bound of $\mathcal{S}$ and $\mathcal{T}$.

Theorem 4.15 $\forall(\mathcal{S}, \mathcal{T} :: \mathcal{S} \sqsupseteq \mathcal{S} \sqcap \mathcal{T})$

Theorem 4.16 $\forall(\mathcal{R}, \mathcal{S}, \mathcal{T} : \mathcal{S} \sqsupseteq \mathcal{R} \ \wedge \ \mathcal{T} \sqsupseteq \mathcal{R} \Rightarrow \mathcal{S} \sqcap \mathcal{T} \sqsupseteq \mathcal{R})$

Theorem 4.17 $\forall(\mathcal{S}, \mathcal{T} :: \mathcal{S} \sqcup \mathcal{T} \sqsupseteq \mathcal{S})$

Theorem 4.18 $\forall(\mathcal{R}, \mathcal{S}, \mathcal{T} : \mathcal{R} \sqsupseteq \mathcal{S} \ \wedge \ \mathcal{R} \sqsupseteq \mathcal{T} \Rightarrow \mathcal{R} \sqsupseteq \mathcal{S} \sqcup \mathcal{T})$

Angelic and demonic composition satisfy the following five laws, commonly referred to as idempotence, symmetry, associativity, absorption, and consistency.

Theorem 4.19 $\mathcal{S} \sqcap \mathcal{S} \simeq \mathcal{S} \ \wedge \ \mathcal{S} \sqcup \mathcal{S} \simeq S$

Theorem 4.20 $\mathcal{S} \sqcap \mathcal{T} \simeq \mathcal{T} \sqcap \mathcal{S} \ \wedge \ \mathcal{S} \sqcup \mathcal{T} \simeq \mathcal{T} \sqcup \mathcal{S}$

Theorem 4.21 $(\mathcal{R} \sqcup \mathcal{S}) \sqcup \mathcal{T} \simeq \mathcal{R} \sqcup (\mathcal{S} \sqcup \mathcal{T})$

Theorem 4.22 $\mathcal{S} \sqcup (\mathcal{S} \sqcap \mathcal{T}) \simeq \mathcal{S} \ \wedge \ \mathcal{S} \sqcap (\mathcal{S} \sqcup \mathcal{T}) \simeq \mathcal{S}$

Theorem 4.23 $(\mathcal{S} \simeq \mathcal{S} \sqcap \mathcal{T}) = (\mathcal{T} \sqsupseteq \mathcal{S}) \ \wedge \ (\mathcal{T} \sqsupseteq \mathcal{S}) = (\mathcal{S} \sqcup \mathcal{T} \simeq \mathcal{T})$

From lattice theory [7, 44] we know that any structure with two binary operators that satisfies Theorems 4.19 - 4.22 is a lattice.

Next we show that $\{\}$ is a greatest element and $\{\{\}\}$ is a least element in our lattice.

Theorem 4.24 $\forall(\mathcal{S} :: \{\} \sqsupseteq \mathcal{S})$

Theorem 4.25 $\forall(\mathcal{S} :: \mathcal{S} \sqsupseteq \{\{\}\})$

Hence lowest upper bounds and highest lower bounds can also be defined for empty, and infinite sets, and our lattice is a complete lattice.

The next theorem shows that the lattice is distributive.

Theorem 4.26
$\mathcal{R} \sqcup (\mathcal{S} \sqcap \mathcal{T}) = (\mathcal{R} \sqcup \mathcal{S}) \sqcap (\mathcal{R} \sqcup \mathcal{T})$
$\mathcal{R} \sqcap (\mathcal{S} \sqcup \mathcal{T}) = (\mathcal{R} \sqcap \mathcal{S}) \sqcup (\mathcal{R} \sqcap \mathcal{T})$

Intermezzo on failure-set semantics and refinement

In [23], C.A.R. Hoare defines a refinement relation for TCSP (theoretical CSP) based on the model for TCSP introduced in [8]. TCSP is a language similar to and an inspiration for ours. In this intermezzo we point out some of the differences between TCSP and *synchronizing processes*.

Hoare defines an ordering relation as follows,

$$P_0 \sqsupseteq P_1 \ \equiv \ ((A_0 = A_1) \wedge (F_0 \subseteq F_1) \wedge (D_0 \subseteq D_1))$$

where A_x is the alphabet of P_x, F_x its failure set, and D_x the set of its divergences. The failure set of a component is a set of pairs, the first component of which is a prefix of a trace of the component, and the second of which is an action the component is **not** capable of participating in next. Such a definition makes sense only if the set of possible actions in the complete system is known beforehand; hence, the alphabet of possible actions is part of the specification of a process. In order to compose or compare processes, their alphabets need to be identical. Because we consider only finite traces, we do not consider divergences here.

Self-synchronization operators such as **connect** cannot be introduced in TCSP, because the model does not include pair symbols. TCSP therefore introduces two kinds of parallel composition. One, $|||$, is non-synchronizing parallel composition and corresponds to our $\|$. The other ,$\|_A$, specifies an alphabet of symbols on which the traces in the components are to be synchronized. Rather than being a single operator, $\|_A$ is really a class of operators. Concealment of actions in TCSP is accomplished with a *hide* operator. In *synchronizing processes* we have only one (non-synchronizing) parallel composition operator, and synchronization and hiding are combined in the **connect** operator.

TCSP does not have sequential composition, but introduces prefixing; hence, the *synchronizing processes* process a would correspond to the TCSP process $a \rightarrow STOP_A$ where A is the appropriate alphabet. TCSP introduces $\sqcap$, to which our $\sqcap$ corresponds, and $[]$, which is similar to, but not the same as, $\sqcup$.

While $P \sqsupseteq P \sqcap Q$ for the ordering defined in the model for TCSP, $P [] Q \sqsupseteq P$ does not hold in general, as the following calculation shows.

We first quote some definitions from [23] Chapter 3. (Some notation has been modified so as not to confuse the reader.)

$traces(STOP) = \{\epsilon\}$
$traces(c \rightarrow P) = \{\epsilon\} \cup \{t : t \in traces(P) : ct\}$
$traces(P \sqcap Q) = traces(P [] Q) = traces(P) \cup traces(Q)$
$refusals(STOP_A) = \{\epsilon \times \mathcal{P}(A)\}$
$refusals(c \rightarrow P) = \{X :: X \subseteq (\alpha(P) - \{c\})\}$
$refusals(P \sqcap Q) = refusals(P) \cup refusals(Q)$
$refusals(P [] Q) = refusals(P) \cap refusals(Q)$
$failures(P) = \{(s, X) :: s \in traces(P) \wedge X \in refusals(P/s)\}$
where P/s corresponds to P after executing trace s.

Hence if we consider the alphabet $\{a, b\}$ and $P = a \to STOP$ and $Q = b \to STOP$ we have:

$$\begin{aligned}
&\mathit{failures}(a \to STOP)\\
&\quad = \{\epsilon \times \{X :: X \subseteq (\{a, b\} - \{a\})\}, a \times \mathcal{P}(\{a, b\})\}\\
&\quad = \{\epsilon \times \{\emptyset, \{b\}\}, a \times \mathcal{P}(\{a, b\})\}\\
&\text{and similarly}\\
&\mathit{failures}(b \to STOP) = \{\epsilon \times \{\emptyset, \{a\}\}, b \times \mathcal{P}(\{a, b\})\}\\
&\mathit{failures}(P \sqcap Q)\\
&\quad = \{\epsilon \times (\{\emptyset, \{a\}\} \cup \{\emptyset, \{b\}\}), a \times \mathcal{P}(\{a, b\}), b \times \mathcal{P}(\{a, b\})\}\\
&\quad = \{\epsilon \times \{\emptyset, \{a\}, \{b\}\}, a \times \mathcal{P}(\{a, b\}), b \times \mathcal{P}(\{a, b\})\}\\
&\mathit{failures}(P [\!] Q)\\
&\quad = \{\epsilon \times (\{\emptyset, \{a\}\} \cap \{\emptyset, \{b\}\}), a \times \mathcal{P}(\{a, b\}), b \times \mathcal{P}(\{a, b\})\}\\
&\quad = \{\epsilon \times \{\emptyset\}, a \times \mathcal{P}(\{a, b\}), b \times \mathcal{P}(\{a, b\})\}
\end{aligned}$$

Hence we see $P [\!] Q \not\sqsupseteq P$.

The main reason for introducing $\sqsupseteq$ in [23] seems to be that it allows the definition of recursion as a least fixed point. The notion of refinement as being able to replace a specification (or process) by a process that refines it in all contexts is given by a relation **sat**, defined by a set of laws.

In this monograph we have attempted to choose a refinement order that combines TCSP's **sat** and $\sqsupseteq$. This requires $\sqcup$ to be subtly different from TCSP's $[\!]$. Our $\sqcup$ corresponds to angelic choice, whereas TCSP's $[\!]$ corresponds to a choice that is made by probing [31] the environment. TCSP, therefore, has the law $(c \to P)[\!](c \to Q) = c \to (P \sqcap Q)$. Unless P and Q are the same, this is another example where $P' [\!] Q' \not\sqsupseteq P'$.

Choosing $\sqcup$ to represent angelic choice, rather than letting it correspond to some operational notion, has a downside. The choice forces us to study when angelic choice can be (efficiently) implemented, and what restrictions on the environment, if any, are needed. This study is conducted in Section 4.7 and Chapter 5.

End of intermezzo

4.3 Operators and processes

In this section we define operators on the lattice of processes and study their algebraic properties. We give names to five nullary operators.

- **magic** $= \{\}$
- **demon** $= \{\{\}\}$
- **pick** $= \{\{t : t \in \mathbf{Traces} : t\}\}$
- **chaos** $= \{t : t \in \mathbf{Traces} : \{t\}\}$
- **skip** $= \{\{\epsilon\}\}$

We extend the definitions of σ and α to processes.

Definition 4.27 $\sigma(\mathcal{T}) \;=\; \bigcup(S : S \in \mathcal{T} : \sigma(S))$

Example: $\sigma(\{\{aa.b\}\}) \;=\; \{a, a.b\}$

Definition 4.28 $\alpha(\mathcal{T}) \;=\; \bigcup(S : S \in \mathcal{T} : \alpha(S))$

Example: $\alpha(\{\{aa.b\}\}) \;=\; \{a, b\}$

We extend the definitions of sequential composition, alt composition, connect composition, and parallel composition to sets of sets of traces, and we introduce demonic composition. We overload the symbols from Section 3. This is justified by the list of algebraic properties in this section, which closely resembles the list from Chapter 2.

$$\mathcal{S}\|\mathcal{T} = \{S, T : S \in \mathcal{T}, T \in \mathcal{S} : S\|T\} \tag{4.29}$$

$$\mathcal{S};\mathcal{T} \;=\; \sqcap(S : S \in \mathcal{S} : \sqcup(s : s \in S : \sqcap(T : T \in \mathcal{T} : \sqcup(t : t \in T : \{\{st\}\})))) \tag{4.30}$$

$$\mathcal{S}\ \mathbf{connect}\ X = \{S : S \in \mathcal{S} : S\ \mathbf{connect}\ X\} \tag{4.31}$$

The operators we have introduced are monotonic with respect to $\sqsupseteq$.

Theorem 4.32 $\mathcal{S} \sqsupseteq \mathcal{T} \Rightarrow \mathcal{S} \sqcap \mathcal{R} \sqsupseteq \mathcal{T} \sqcap \mathcal{R}$

Theorem 4.33 $\mathcal{S} \sqsupseteq \mathcal{T} \Rightarrow \mathcal{S} \sqcup \mathcal{R} \sqsupseteq \mathcal{T} \sqcup \mathcal{R}$

Theorem 4.34 $\mathcal{S} \sqsupseteq \mathcal{T} \Rightarrow \mathcal{S} \| \mathcal{R} \sqsupseteq \mathcal{T} \| \mathcal{R}$

Theorem 4.35 $\mathcal{R} \sqsupseteq \mathcal{S} \ \wedge \ \mathcal{T} \sqsupseteq \mathcal{U} \Rightarrow \mathcal{R}; \mathcal{T} \sqsupseteq \mathcal{S}; \mathcal{U}$

Theorem 4.36 $\mathcal{S} \sqsupseteq \mathcal{T} \Rightarrow \ \mathcal{S}$ **connect** $X \sqsupseteq \ \mathcal{T}$ **connect** X

The composition operators respect the equivalence relation we have introduced.

Theorem 4.37 $\mathcal{S} \simeq \mathcal{T} \Rightarrow \forall(\mathcal{R} : \mathcal{S} \sqcap \mathcal{R} \simeq \mathcal{T} \sqcap \mathcal{R})$

Theorem 4.38 $\mathcal{S} \simeq \mathcal{T} \Rightarrow \forall(\mathcal{R} : \mathcal{S} \sqcup \mathcal{R} \simeq \mathcal{T} \sqcup \mathcal{R})$

Theorem 4.39 $\mathcal{S} \simeq \mathcal{T} \Rightarrow \forall(\mathcal{R} : \mathcal{S} \| \mathcal{R} \simeq \mathcal{T} \| \mathcal{R})$

Theorem 4.40 $\mathcal{S} \simeq \mathcal{T} \Rightarrow \forall(\mathcal{R} : \mathcal{S}; \mathcal{R} \simeq \mathcal{T}; \mathcal{R})$

Theorem 4.41 $\mathcal{S} \simeq \mathcal{T} \Rightarrow \forall(\mathcal{R} : \mathcal{R}; \mathcal{S} \simeq \mathcal{R}; \mathcal{T})$

Theorem 4.42 $\mathcal{S} \simeq \mathcal{T} \Rightarrow \forall(X : \mathcal{S}$ **connect** $X \simeq \mathcal{T}$ **connect** $X)$

Proof This follows from monotonicity. For example:

$$
\begin{array}{ll}
& \mathcal{S} \simeq \mathcal{T} \Rightarrow \forall(\mathcal{R} : \mathcal{S} \sqcap \mathcal{R} \simeq \mathcal{T} \sqcap \mathcal{R}) \\
\Leftarrow & \{ \text{ Definition } \simeq, \text{ calculus } \} \\
& \mathcal{S} \sqsupseteq \mathcal{T} \Rightarrow \forall(\mathcal{R} : \mathcal{S} \sqcap \mathcal{R} \sqsupseteq \mathcal{T} \sqcap \mathcal{R}) \wedge \\
& \mathcal{T} \sqsupseteq \mathcal{S} \Rightarrow \forall(\mathcal{R} : \mathcal{T} \sqcap \mathcal{R} \sqsupseteq \mathcal{S} \sqcap \mathcal{R}). \\
= & \{ \text{ Monotonicity } \} \\
& \mathit{true}
\end{array}
$$

end of proof

We have the following algebraic properties. Some of these theorems are direct consequences of the lattice algebra, but we repeat those to create an overview of the process algebra. Even though equality holds for many of these theorems, equivalence is the relation we are interested in. Because process equivalence is an equivalence relation, equivalence follows from equality.

$$\textbf{magic} \sqcap \mathcal{T} = \mathcal{T} \tag{4.43}$$

$$\textbf{demon} \sqcap \mathcal{T} \simeq \textbf{demon} \tag{4.44}$$

$$\mathcal{T} \sqcap \mathcal{S} = \mathcal{S} \sqcap \mathcal{T} \tag{4.45}$$

$$(\mathcal{R} \sqcap \mathcal{S}) \sqcap \mathcal{T} = \mathcal{R} \sqcap (\mathcal{S} \sqcap \mathcal{T}) \tag{4.46}$$

$$\textbf{demon} \sqcup \mathcal{S} = \mathcal{S} \tag{4.47}$$

$$\textbf{magic} \sqcup \mathcal{S} = \textbf{magic} \tag{4.48}$$

$$\mathcal{T} \sqcup \mathcal{S} = \mathcal{S} \sqcup \mathcal{T} \tag{4.49}$$

$$(\mathcal{R} \sqcup \mathcal{S}) \sqcup \mathcal{T} = \mathcal{R} \sqcup (\mathcal{S} \sqcup \mathcal{T}) \tag{4.50}$$

$$\textbf{skip} \| \mathcal{S} = \mathcal{S} \tag{4.51}$$

$$\textbf{magic} \| \mathcal{S} = \textbf{magic} \tag{4.52}$$

$$\mathcal{T} \| \mathcal{S} = \mathcal{S} \| \mathcal{T} \tag{4.53}$$

$$(\mathcal{R} \| \mathcal{S}) \| \mathcal{T} = \mathcal{R} \| (\mathcal{S} \| \mathcal{T}) \tag{4.54}$$

$$\textbf{skip} ; \mathcal{S} \simeq \mathcal{S} = \mathcal{S}; \textbf{skip} \tag{4.55}$$

$$\textbf{demon} ; \mathcal{S} = \textbf{demon} \tag{4.56}$$

$$\textbf{magic} ; \mathcal{S} = \textbf{magic} \tag{4.57}$$

$$(\mathcal{R} \sqcup \mathcal{S}); \mathcal{T} = (\mathcal{R}; \mathcal{T}) \sqcup (\mathcal{S}; \mathcal{T}) \tag{4.58}$$

$$(\mathcal{R} \sqcap \mathcal{S}); \mathcal{T} = (\mathcal{R}; \mathcal{T}) \sqcap (\mathcal{S}; \mathcal{T}) \tag{4.59}$$

The following two examples show that ; does not left-distribute over $\sqcap$ and $\sqcup$.

Example

$(\{\{a\}\} \sqcup \{\{b\}\}); (\{\{a\}\} \sqcap \{\{b\}\})$
$=$ { Definition $\sqcup, \sqcap$ }
$\{\{a, b\}\}; (\{\{a\}, \{b\}\})$
$=$ { Definition ; }
$(\{\{a\}\}; \{\{a\}, \{b\}\}) \sqcup (\{\{b\}\}; \{\{a\}, \{b\}\})$
$=$ { Definition ; }
$\{\{aa\}, \{ab\}\} \sqcup \{\{ba\}, \{bb\}\}$
$=$ { Definition $\sqcup$ }
$\{\{aa, ba\}, \{aa, bb\}, \{ab, ba\}, \{ab, bb\}\}$

$$
\begin{aligned}
& ((\{\{a\}\} \sqcup \{\{b\}\}); \{\{a\}\}) \sqcap ((\{\{a\}\} \sqcup \{\{b\}\}); \{\{b\}\}) \\
= & \ \{ \text{ Definition } \sqcup \ \} \\
& (\{\{a, b\}\}; \{\{a\}\}) \sqcap (\{\{a, b\}\}; \{\{b\}\}) \\
= & \ \{ \text{ Definition } ; \ \} \\
& \{\{aa, ba\}\} \sqcap \{\{ab, bb\}\} \\
= & \ \{ \text{ Definition } \sqcap \ \} \\
& \{\{aa, ba\}, \{ab, bb\}\}
\end{aligned}
$$

Example

$$
\begin{aligned}
& (\{\{a\}\} \sqcap \{\{b\}\}); (\{\{a\}\} \sqcup \{\{b\}\}) \\
= & \ \{ \text{ Definition } \sqcup, \sqcap \ \} \\
& \{\{a\}, \{b\}\}; \{\{a, b\}\} \\
= & \ \{ \text{ Definition } ; \ \} \\
& (\{\{a\}\}; \{\{a, b\}\}) \sqcap (\{\{b\}\}; \{\{a, b\}\}) \\
= & \ \{ \text{ Definition } ; \ \} \\
& \{\{aa, ab\}\} \sqcap \{\{ba, bb\}\} \\
= & \ \{ \text{ Definition } \sqcup \ \} \\
& \{\{aa, ab\}, \{ba, bb\}\}
\end{aligned}
$$

$$
\begin{aligned}
& ((\{\{a\}\} \sqcap \{\{b\}\}); \{\{a\}\}) \sqcup ((\{\{a\}\} \sqcap \{\{b\}\}); \{\{b\}\}) \\
= & \ \{ \text{ Definition } \sqcap \ \} \\
& (\{\{a\}, \{b\}\}; \{\{a\}\}) \sqcup (\{\{a\}, \{b\}\}; \{\{b\}\}) \\
= & \ \{ \text{ Definition } ; \ \} \\
& \{\{aa, ba\}\} \sqcup \{\{ab, bb\}\} \\
= & \ \{ \text{ Definition } \sqcap \ \} \\
& \{\{aa, ab\}, \{aa, bb\}, \{ba, ab\}, \{ba, bb\}\}
\end{aligned}
$$

We may understand this informally as follows. Right distribution of semicolon over $\sqcap$ and $\sqcup$ does not change the moment at which the demonic or angelic choices are made, whereas left distribution does. In the first example above we see that parallel composition with the process $\{\{a'a', b'b'\}\}$ and connecting channels $a.a'$ and $b.b'$ can lead to deadlock for the first, but not for the second process.

$$(\mathcal{T}; \mathcal{S}); \mathcal{R} \ = \ \mathcal{T}; (\mathcal{S}; \mathcal{R}) \tag{4.60}$$

$$\alpha(X) \cap \alpha(Y) = \emptyset \Rightarrow (\mathcal{S} \textbf{ connect } X) \textbf{ connect } Y = \mathcal{S} \textbf{ connect } (X \cup Y) \tag{4.61}$$

$$(\mathcal{S}; \mathcal{T}) \textbf{ connect } X = (\mathcal{S} \textbf{ connect } X); (\mathcal{T} \textbf{ connect } X) \tag{4.62}$$

$$(\mathcal{S} \sqcup \mathcal{T}) \textbf{ connect } X = (\mathcal{S} \textbf{ connect } X) \sqcup (\mathcal{T} \textbf{ connect } X) \tag{4.63}$$

$$(\mathcal{S} \sqcap \mathcal{T}) \textbf{ connect } X = (\mathcal{S} \textbf{ connect } X) \sqcap (\mathcal{T} \textbf{ connect } X) \tag{4.64}$$

$$\alpha(X) \cap \sigma(\mathcal{S}) = \emptyset \wedge \alpha(X) \cap \sigma(\mathcal{T}) = \emptyset \Rightarrow \tag{4.65}$$

$$(\mathcal{S} \| \mathcal{T}) \textbf{ connect } X = (\mathcal{S} \textbf{ connect } X) \| (\mathcal{T} \textbf{ connect } X)$$

$$\{a, b\} \cap (\alpha(\mathcal{S}_0) \cup \alpha(\mathcal{S}_1) \cup \alpha(\mathcal{T}_0) \cup \alpha(\mathcal{T}_1)) = \emptyset \Rightarrow \tag{4.66}$$

$$((\mathcal{S}_0;\, a;\, \mathcal{S}_1) \| (\mathcal{T}_0;\, b;\, \mathcal{T}_1)) \textbf{ connect } \{a.b\} = (\mathcal{S}_0 \| \mathcal{T}_0); (\mathcal{S}_1 \| \mathcal{T}_1))$$

4.4 Iteration.

We define two kinds of iteration: angelic ($*$), and demonic ($\dagger$). Angelic and demonic iteration for sequential programs have been studied in [17].

Definition 4.67 For any process $\mathcal{S}$, $\mathcal{S}^*$ is defined as the least solution of

$$\mathcal{S}^* = \textbf{skip} \sqcup (\mathcal{S}; \mathcal{S}^*)$$

Definition 4.68 For any process $\mathcal{S}$, $\mathcal{S}^\dagger$ is defined as the greatest solution of

$$\mathcal{S}^\dagger = \textbf{skip} \sqcap (\mathcal{S}; \mathcal{S}^\dagger)$$

Theorem 4.69 *$\mathcal{S}^*$ and $\mathcal{S}^\dagger$ are well defined.*

Proof From the Knaster-Tarski theorem [7], using monotonicity of $\sqcup$, $\sqcap$, and ; .
end of proof

Examples

$$\{\{a\}\}^* = \{\{n : n \in Nat. \cup \{0\} : a^n\}\}$$

$$
\begin{aligned}
& \{\{\epsilon\}\} \sqcup (\{\{a\}\}; \{\{n : n \in Nat. \cup \{0\} : a^n\}\}) \\
= & \{ \text{ Definition } ; \ \} \\
& \{\{\epsilon\}\} \sqcup \{\{n : n \in Nat. \cup \{0\} : aa^n\}\} \\
= & \{ \text{ Definition } \sqcup \ \} \\
& \{\{\epsilon\} \cup \{n : n \in Nat. \cup \{0\} : aa^n\}\} \\
= & \{ \ \} \\
& \{\{n : n \in Nat. \cup \{0\} : a^n\}\}
\end{aligned}
$$

$$\{\{a\}\}^\dagger = \{n : n \in Nat. \cup \{0\} : \{a^n\}\}$$

$$
\begin{aligned}
& \{\{\epsilon\}\} \sqcap (\{\{a\}\}; \{n : n \in Nat. \cup \{0\} : \{a^n\}\}) \\
= & \{ \text{ Definition } ; \ \} \\
& \{\{\epsilon\}\} \sqcup \{n : n \in Nat. \cup \{0\} : \{aa^n\}\} \\
= & \{ \text{ Definition } \sqcup \ \} \\
& \{\{\epsilon\}\} \cup \{n : n \in Nat. \cup \{0\} : \{aa^n\}\} \\
= & \{ \ \} \\
& \{n : n \in Nat. \cup \{0\} : \{a^n\}\}
\end{aligned}
$$

We have the following monotonicity properties.

Theorem 4.70 $\mathcal{S} \sqsupseteq \mathcal{T} \Rightarrow \mathcal{S}^* \sqsupseteq \mathcal{T}^*$

Theorem 4.71 $\mathcal{S} \sqsupseteq \mathcal{T} \Rightarrow \mathcal{S}^\dagger \sqsupseteq \mathcal{T}^\dagger$

The following theorem is very useful.

Theorem 4.72 $\mathcal{S}^* \| \mathcal{T}^* \sqsupseteq (\mathcal{S} \| \mathcal{T})^*$

4.5 Duals of processes

Definition 4.73 We define **Dual** as follows.

$$\mathbf{Dual}(\mathcal{S}) = \sqcup(S : S \in \mathcal{S} : \sqcap(s : s \in S : \{\{s\}\}))$$

$$\mathbf{Dual}(\mathbf{magic}\ ) \simeq \mathbf{demon} \tag{4.74}$$

$$\mathbf{Dual}(\mathbf{demon}\ ) \simeq \mathbf{magic} \tag{4.75}$$

$$\mathbf{Dual}(\mathbf{pick}\ ) \simeq \mathbf{chaos} \tag{4.76}$$

$$\mathbf{Dual}(\mathbf{chaos}\ ) \simeq \mathbf{pick} \tag{4.77}$$

$$\mathbf{Dual}(\mathbf{skip}\ ) \simeq \mathbf{skip} \tag{4.78}$$

$$\mathbf{Dual}(\{\{s\}\}) \simeq \{\{s\}\} \tag{4.79}$$

The following theorems confirm our choice of name for **Dual**.

Theorem 4.80 $\mathbf{Dual}(\mathcal{S} \sqcap \mathcal{T}) \simeq \mathbf{Dual}(\mathcal{S}) \sqcup \mathbf{Dual}(\mathcal{T})$

Theorem 4.81 $\mathbf{Dual}(\mathcal{S} \sqcup \mathcal{T}) \simeq \mathbf{Dual}(\mathcal{S}) \sqcap \mathbf{Dual}(\mathcal{T})$

Theorem 4.82 $\mathbf{Dual}(\mathbf{Dual}(\mathcal{S})) \simeq \mathcal{S}$

An equivalent definition of **Dual** is

Definition 4.83

$$Dual(\mathcal{S}) \ = \ \{T : \forall(S : S \in \mathcal{S} : \exists(s : s \in S : s \in T)) : T\}$$

In order to prove the equivalence of the two definitions, the following two theorems are useful.

Dual is a Galois connection.

Theorem 4.84 $\mathcal{S} \sqsupseteq Dual(\mathcal{T}) \ = \ \mathcal{T} \sqsupseteq Dual(\mathcal{S})$

Theorem 4.85 $Dual(\mathcal{S}) \sqsupseteq \mathcal{T} \ = \ Dual(\mathcal{T}) \sqsupseteq \mathcal{S}$

The following theorem states that both definitions of dual are in fact equivalent.

Theorem 4.86 $\mathbf{Dual}(\mathcal{S}) \simeq Dual(\mathcal{S})$

The following properties can now easily be proven.

Theorem 4.87 $\mathcal{S} \sqsupseteq \mathcal{T} = \mathbf{Dual}(\mathcal{T}) \sqsupseteq \mathbf{Dual}(\mathcal{S})$

Theorem 4.88 $\mathcal{S} \simeq \mathcal{T} = \mathbf{Dual}(\mathcal{S}) \simeq \mathbf{Dual}(\mathcal{T})$

The following theorems are useful in computing duals of processes.

Theorem 4.89 $\mathbf{Dual}(\mathcal{S}; \mathcal{T}) \simeq \mathbf{Dual}(\mathcal{S}); \mathbf{Dual}(\mathcal{T})$

Theorem 4.90 $\mathbf{Dual}(\mathcal{S}^*) \simeq \mathcal{S}^\dagger$

Theorem 4.91 $\mathbf{Dual}(\mathcal{S}^\dagger) \simeq \mathcal{S}^*$

Duals of processes are useful in that they specify the most demonic environment with which a process can be composed without introducing deadlock. This environment is similar to the *unique maximal environment* of Dill [18]. This fact is stated in the following theorem.

Theorem 4.92 Provided $\alpha(\mathcal{S}) = \sigma(\mathcal{S})$,
$(\mathcal{S} \| \mathbf{Dual}(\mathcal{S}'))$ **connect** $\{x : x \in \alpha(S) : x.x'\} \simeq$ **skip**

where $\mathcal{S}'$ is $\mathcal{S}$ with all actions replaced by their primed counterparts.

Furthermore, the dual is the least environment with this property, as expressed by the following theorem.

Theorem 4.93 Provided $\alpha(\mathcal{S}) = \sigma(\mathcal{S})$,
$\mathcal{T} \sqsubset \mathbf{Dual}(\mathcal{S}') \Rightarrow (\mathcal{S} \| \mathcal{T})$ **connect** $\{x : x \in \alpha(S) : x.x'\} \simeq$ **demon**

4.6 A simple example

In this section we give a simple example to illustrate the use of the machinery introduced in this chapter.

problem specification

Required is

$$
\begin{array}{lll}
 & (& \mathbf{and} \\
 & \| & \\
 & & ((a_0 \| b_0);\ c_0 \\
 & & \sqcap (a_0 \| b_1);\ c_0 \\
 & & \sqcap (a_1 \| b_0);\ c_0 \\
 & & \sqcap (a_1 \| b_1);\ c_1 \\
 & &)^\dagger \\
 &) & \mathbf{connect}\ \{a_0.a_0', a_1.a_1', b_0, b_0, b_1.b_1', c_0.c_0', c_1.c_1'\} \\
\sqsubseteq & & \\
 & \mathbf{skip} &
\end{array}
$$

Give an implementation for **and**
end of problem specification

solution

$$
\begin{array}{l}
\quad \textbf{Dual}(\ ((a_0 \| b_0);\ c_0 \\
\qquad \sqcap (a_0 \| b_1);\ c_0 \\
\qquad \sqcap (a_1 \| b_0);\ c_0 \\
\qquad \sqcap (a_1 \| b_1);\ c_1)^{\dagger}) \\
= \{ \text{ Using the fact that we don't connect } a\text{'s and } b\text{'s } \} \\
\quad \textbf{Dual}(\ ((a_0;\ b_0 \sqcup b_0;\ a_0);\ c_0 \\
\qquad \sqcap (a_0;\ b_1 \sqcup b_1;\ a_0);\ c_0 \\
\qquad \sqcap (a_1;\ b_0 \sqcup b_0;\ a_1);\ c_0 \\
\qquad \sqcap (a_1;\ b_1 \sqcup b_1;\ a_1);\ c_1)^{\dagger}) \\
= \{ \text{ Theorems 4.91,4.81,4.80,4.89 } \} \\
\quad ((a_0;\ b_0 \sqcap b_0;\ a_0);\ c_0 \\
\quad \sqcup (a_0;\ b_1 \sqcap b_1;\ a_0);\ c_0 \\
\quad \sqcup (a_1;\ b_0 \sqcap b_0;\ a_1);\ c_0 \\
\quad \sqcup (a_1;\ b_1 \sqcap b_1;\ a_1);\ c_1)^{*} \\
\sqsubseteq \{ \ \sqcap \text{ is meet in lattice } \} \\
\quad (a_0;\ b_0;\ c_0 \\
\quad \sqcup a_0;\ b_1;\ c_0 \\
\quad \sqcup a_1;\ b_0;\ c_0 \\
\quad \sqcup a_1;\ b_1;\ c_1)^{*} \\
= \{ \text{ left distribution of } \sqcup \text{ over } ; \ \} \\
\quad ((a_0;\ (b_0 \sqcup b_1);\ c_0) \sqcup (a_1;\ (b_0;\ c_0 \sqcup b_1;\ c_1)))
\end{array}
$$

The fact that this is a solution now follows from monotonicity and Theorem 4.92.
end of solution

4.7 Operational considerations

In this section we give an informal operational appreciation of the theory developed in this chapter. The main issue we address is angelic choice versus the [] construct of CSP and the *ALT* construct in programming languages like occam^{TM} [27]. We propose sufficient conditions under which angelic choice can be implemented efficiently. In the next section we follow the discussion up with an example of a process built from several component processes and show that the constructions meet the requirements listed in this section.

Our refinement ordering, and hence our notion of equivalence between processes, is based on the notion of deadlock. One process refines another if it is guaranteed not to deadlock in any environment in which the other process does not deadlock. **demon** , the bottom element of our lattice, corresponds to a process that is deadlocked in one of its components. Demonic nondeterminism allows the process to choose the alternative that causes deadlock. Implementing demonic nondeterminism is not difficult, because either alternative is a refinement, and hence an implementation, of the construct. Angelic nondeterminism requires the process to choose the alternative that avoids deadlock. In general, angelic nondeterminism cannot be implemented, as the event that may cause deadlock can be arbitrarily far in the future. However, if the first actions of the alternatives in an angelic choice are mutually exclusive, then angelic choice can be implemented by probing [31] the environment for its next action. Fully symmetric protocols for communication that allow probing on either side exist [46], but, in order to avoid deadlock, only one of the processes involved in the communication must probe. This leads us to a second restriction on processes that can be adequately described by our algebra: only one side of a synchronization action can be a first action in an alternative of an $\sqcup$ construct. We share this restriction with CSP, because in the failure set model $((a \rightarrow \mathbf{stop}\ [\!]\, b \rightarrow \mathbf{stop}\ )\|(a \rightarrow \mathbf{stop}\ [\!]\, b \rightarrow \mathbf{stop}\ ))/\{a, b\} = \mathbf{stop}$, whereas the ALT construct in occam^{TM} deadlocks when composed with itself.

In our model, parallel composition is defined in terms of angelic nondeterminism, and hence we should expect difficulties when attempting to implement it. Fortunately, the only way in which deadlock can result from replacing sequential with parallel composition (operationally speaking) is if there is a choice that is made on the basis of which sequential component is attempted first. This, after all, is the demonic aspect of parallel composition. Such a choice requires an alternative statement with non-mutually-exclusive guards. Thus, if we guarantee by some external reasoning that alternative statements with non-mutually-exclusive guards do not occur in our programs, then we may indeed regard parallel composition as angelic.

We can weaken the condition of mutually exclusive first actions using the theorems from Chapter 3, Section 5. There a class of programs was identified that includes angelic nondeterminism with first actions that are not mutually exclusive. However, the angelic choice required of processes in this class is implementable, because the processes are capable of doing non-mutually-exclusive actions in any order, and its future behavior may not depend on that order. These restrictions are captured formally in the definition of the class cc in Section 3.5.

A useful notion when reasoning about mutually exclusive first actions are *signaling sets* [43]. A signaling set is a set of mutually exclusive actions. Their typical use is as follows: we prove that a process is implementable under the restriction that certain sets of actions in the environment form a signaling set, and we show that certain sets of outputs form signaling sets as well. The **and** process of the previous section is implementable under the restriction that $a0, a1$ and $b0, b1$ form signaling sets, and it guarantees that $c0, c1$ is a signaling set.

Alternatives with non-mutually-exclusive first communications do occur in practice. They are essential for implementing a (fair) merge between two channels that may be active at the same time. In hardware, they are implemented using arbiters. We conclude that the framework as presented here cannot be used to design concurrent systems that require such a merge or hardware that requires arbiters. Nevertheless, the examples in the next two sections indicate that many interesting programs can be built without them.

In the next chapter we briefly introduce an operational model that uses probes explicitly, and that does not suffer from these restrictions. We indicate there why we believe that models and algebras that conform to that operational model are inherently more complex. Thus, for the design of synchronizing systems with mutually exclusive guards, we believe it is likely that one will resort to a model and process algebra similar to the one presented in this chapter.

4.8 Example: Full adder

4.8.1 Specification

We take the following component as the specification of a (restricted) full adder cell. The example should not be taken too literally; because our language does not allow for probes or expressions in guards, the construction is overly complicated. The reason we give this example anyway is because it illustrates nicely the compositionality of the method and the use of signaling sets.

problem specification

$$
\begin{array}{l}
\mathbf{fa}(a_0, a_1, b_0, b_1, c_0, c_1, d_0, d_1, e_0, e_1) \;=\\
\quad (a_0; (b_0; (c_0; (d_0 \| e_0) \sqcup c_1; (d_1 \| e_0))\\
\qquad \sqcup b_1; (c_0; (d_1 \| e_0) \sqcup c_1; (d_0 \| e_1))\\
\qquad)\\
\quad \sqcup a_1; (b_0; (c_0; (d_1 \| e_0) \sqcup c_1; (d_0 \| e_1))\\
\qquad \sqcup b_1; (c_0; (d_0 \| e_1) \sqcup c_1; (d_1 \| e_1))\\
\quad) \quad)
\end{array}
$$

Relies on: (0) Signaling sets: $\{a_0, a_1\}, \{b_0, b_1\}, \{c_0, c_1\}$.
(1) These signals are not probed in the environment.
Guarantees: (0) Signaling sets: $\{d_0, d_1\}, \{e_0, e_1\}$.
(1) The process does not probe these signals.
end of problem specification

The environment signaling sets are signaling sets that the process may rely on. The process signaling sets are signaling sets that the process must guarantee. We observe that the process is implementable, since angelic choices are made with mutually exclusive first communications that are not probed in the environment.

4.8.2 First decomposition

We define components **split**, **pgen**, and **cgen**.

$\mathbf{split2}(a_0, a_1, a_0', a_1', a_0'', a_1'') =$
$(a_0; (a_0' \| a_0'') \sqcup a_1; (a_1' \| a_1''))$
Relies on: (0) Signaling set: $\{a_0, a_1\}$.
(1) These signals are not probed in the environment.
Guarantees: (0) Signaling sets: $\{a_0', a_1'\}, \{a_0'', a_1''\}$.
(1) The process does not probe these signals.

$$
\begin{aligned}
&\mathbf{pgen}(a_0, a_1, b_0, b_1, c_0, c_1, d_0, d_1) = \\
&\quad (a_0; (b_0; (c_0; d_0 \sqcup c_1; d_1) \\
&\qquad\quad \sqcup b_1; (c_0; d_1 \sqcup c_1; d_0) \\
&\qquad\quad) \\
&\quad \sqcup a_1; (b_0; (c_0; d_1 \sqcup c_1; d_0) \\
&\qquad\quad \sqcup b_1; (c_0; d_0 \sqcup c_1; d_1) \\
&\quad) \quad)
\end{aligned}
$$

Relies on: (0) Signaling sets: $\{a_0, a_1\}, \{b_0, b_1\}, \{c_0, c_1\}$.
(1) These signals are not probed in the environment.
Guarantees: (0) Signaling set: $\{d_0, d_1\}$.
(1) The process does not probe these signals.

$$
\begin{aligned}
&\mathbf{cgen}(a_0, a_1, b_0, b_1, c_0, c_1, e_0, e_1) = \\
&\quad (a_0; (b_0; (c_0; e_0 \sqcup c_1; e_0) \\
&\qquad\quad \sqcup b_1; (c_0; e_0 \sqcup c_1; e_1) \\
&\qquad\quad) \\
&\quad \sqcup a_1; (b_0; (c_0; e_0 \sqcup c_1; e_1) \\
&\qquad\quad \sqcup b_1; (c_0; e_1 \sqcup c_1; e_1) \\
&\quad) \quad)
\end{aligned}
$$

Relies on: (0) Signaling sets: $\{a_0, a_1\}, \{b_0, b_1\}, \{c_0, c_1\}$.
(1) These signals are not probed in the environment.
Guarantees: (0) Signaling set: $\{e_0, e_1\}$.
(1) The process does not probe these signals.

We observe:

$$
\begin{array}{ll}
& (\ \mathbf{split2}(a_0, a_1, a_0', a_1', a_0'', a_0'') \\
& \parallel \mathbf{split2}(b_0, b_1, b_0', b_1', b_0'', b_1'') \\
& \parallel \mathbf{split2}(c_0, c_1, c_0', c_1', c_0'', c_1'') \\
& \parallel \mathbf{pgen}(a_0''', a_1''', b_0''', b_1''', c_0''', c_1''', d_0, d_1) \\
& \parallel \mathbf{cgen}(a_0'''', a_1'''', b_0'''', b_1'''', c_0'''', c_1'''', e_0, e_1) \\
&) \\
\multicolumn{2}{l}{\mathbf{connect}\ \ \{a_0'.a_0''', a_0'', a_0'''', b_0'.b_0''', b_0''.b_0'''', c_0'.c_0''', c_0'', c_0'''',} \\
\multicolumn{2}{l}{\qquad\qquad p_0.p_0', p_1.p_1', c_k.c_k', c_p.c_p', c_g.c_g\}} \\
\sqsupseteq & \\
& \mathbf{fa}(a_0, a_1, b_0, b_1, c_0, c_1, d_0, d_1, e_0, e_1)
\end{array}
$$

We do not give the details of the calculation. Informally, the refinement is valid because parallelism is increased without violating the requirements on the signaling sets. For calculations such as these, where the informal argument is clear, the detailed calculations are best left to a computer.

4.8.3 Carry subcircuits

The following program specifies a component that generates the carry-propagate, carry-generate, and carry-kill signals from the first two bits.

$$
\begin{array}{l}
\mathbf{pgk}(a_0, a_1, b_0, b_1, c_k, c_p, c_g) = \\
\quad (\ a_0; (b_0;\ c_k \sqcup b_1;\ c_p) \\
\quad \sqcup\ a_1; (b_0;\ c_p \sqcup b_1;\ c_g) \\
\quad \sqcup\ b_0; (a_0;\ c_k \sqcup a_1;\ c_p) \\
\quad \sqcup\ b_1; (a_0;\ c_p \sqcup a_1;\ c_g) \\
\quad)
\end{array}
$$

Relies on: (0) Signaling sets: $\{a_0, a_1\}, \{b_0, b_1\}$.
(1) These signals are not probed in the environment.
Guarantees: (0) Signaling set: $\{c_p, c_g, c_k\}$.
(1) The process does not probe these signals.

The component is implementable because, projected onto the signaling sets, it is in the class cc.

The next component we examine is a circuit that generates the carry from the carry-in and the three intermediate signals.

$\mathbf{cckt}(c_0, c_1, c_k, c_p, c_g, e_0, e_1) =$
$(\; c_0; (c_k;\, e_0 \sqcup c_p;\, e_0 \sqcup c_g;\, e_1)$
$\sqcup \; c_1; (c_k;\, e_0 \sqcup c_p;\, e_1 \sqcup c_g;\, e_1)$
$)$

Relies on: (0) Signaling sets: $\{c_0, c_1\}, \{c_p, c_g, c_k\}$.
(1) These signals are not probed in the environment.
Guarantees: (0) Signaling set: $\{e_0, e_1\}$.
(1) The process does not probe these signals.

We observe:

$$
\begin{array}{l}
(\; \mathbf{pgk}(a_0, a_1, b_0, b_1, c_k, c_p, c_g) \\
\| \; \mathbf{cckt}(c_0, c_1, c_k', c_p', c_g', e_0, e_1) \\
) \quad \mathbf{connect} \; \{c_k.c_k', c_p.c_p', c_g.c_g'\} \\
\sqsupseteq \\
\mathbf{cgen}(a_0, a_1, b_0, b_1, c_0, c_1)
\end{array}
$$

Note This subcircuit is more general than needed to meet the original specification, since there it is specified that the bits will arrive in a given order.

Note Another possible implementation of this subcircuit is a circuit that generates the carry as soon as possible, i.e., generating it before the c' communication in the case of c_k or c_g.

4.8.4 Parity subcircuit.

The following program gives an implementation of an exclusive or gate.

$\mathbf{xor}(a_0, a_1, b_0, b_1, p_0, p_1) =$
$(\; a_0; (b_0;\, p_0 \sqcup b_1;\, p_1)$
$\sqcup \; a_1; (b_0;\, p_1 \sqcup b_1;\, p_0)$
$\sqcup \; b_0; (a_0;\, p_1 \sqcup a_1;\, p_0)$
$\sqcup \; b_1; (a_0;\, p_1 \sqcup a_1;\, p_0)$
$)$

Relies on: (0) Signaling sets: $\{a_0, a_1\}, \{b_0, b_1\}$.
(1) These signals are not probed in the environment.
Guarantees: (0) Signaling set: $\{p_0, p_1\}$.

(1) The process does not probe these signals.

The component is implementable because, projected onto the signaling sets, it is in the class cc.

We observe:

$$
\begin{array}{ll}
 & (\ \mathbf{xor}(a_0, a_1, b_0, b_1, p_0, p_1) \\
 & \|\ \mathbf{xor}(c_0, c_1, p_0', p_1', d_0, d_1) \\
 &) \qquad \mathbf{connect}\ \{p_0.p_0', p_1.p_1'\} \\
\sqsupseteq & \\
 & \mathbf{pgen}(a_0, a_1, b_0, b_1, c_0, c_1, d_0, d_1)
\end{array}
$$

4.8.5 Iteration

Using Theorem (4.72) we conclude

$$
\begin{array}{ll}
 & (\ \mathbf{split2}(a_0, a_1, a_0', a_1', a_0'', a_0'') \\
 & \|\ \mathbf{split2}(b_0, b_1, b_0', b_1', b_0'', b_1'') \\
 & \|\ \mathbf{split2}(c_0, c_1, c_0', c_1', c_0'', c_1'') \\
 & \|\ \mathbf{pgen}(a_0''', a_1''', b_0''', b_1''', c_0''', c_1''', d_0, d_1)^* \\
 & \|\ \mathbf{cgen}(a_0'''', a_1'''', b_0'''', b_1'''', c_0'''', c_1'''', e_0, e_1)^* \\
 &) \\
 & \mathbf{connect}\ \ \{a_0'.a_0''', a_0'', a_0'''', b_0'.b_0''', b_0''.b_0'''', c_0'.c_0''', c_0'', c_0'''', \\
 & \qquad p_0.p_0', p_1.p_1', c_k.c_k', c_p.c_p', c_g.c_g\} \\
\sqsupseteq & \\
 & \mathbf{fa}^*
\end{array}
$$

4.9 Example: Memory

In this section we study some memories. We do not give calculations, the purpose of this section is to show that even though our language is very limited, there are interesting programming (circuit-design) examples that fall within its scope.

4.9.1 One-bit memory

We define a one-bit memory thus:

$$\mathbf{mem}(r, d_0, d_1, w_0, w_1) = (\ M_s(x_0', x_1') \| M_i(r, d_0, d_1, w_0, w_1, x_0', x_1')\ )$$
$$\mathbf{connect}\ \{x_0.x_0', x_1.x_1'\}$$

$$M_s(x_0', x_1') = x_0';\ (x_0';\ x_0' \sqcup x_1';\ x_1')^*$$

$$\begin{aligned} & M_i(r, d_0, d_1, w_0, w_1, x_0, x_1) = \\ & \quad (\ x_0;\ (r;\ d_0;\ x_0 \sqcup w_0;\ x_0 \sqcup w_1;\ x_1) \\ & \quad \sqcup\ x_1;\ (r;\ d_1;\ x_1 \sqcup w_0;\ x_0 \sqcup w_1;\ x_1) \\ & \quad)^* \end{aligned}$$

4.9.2 Selector, control, and merge

We define a selector process S for a fixed integer n, $n > 0$:

$$\begin{aligned} & S(ai_0, ai_1, au_0, au_1, ad_0, ad_1, n) = \\ & \quad (\ ai_0;\ (ai_0;\ au_0 \sqcup ai_1;\ au_1)^{n-1} \\ & \quad \sqcup\ ai_1;\ (ai_0;\ ad_0 \sqcup ai_1;\ ad_1)^{n-1} \\ & \quad)^* \end{aligned}$$

The selector takes an n-bit sequence of bits as input and, depending on the value of the first bit, sends the remaining $n-1$ bits on au_0, au_1 or ad_0, ad_1.

We decide to encode the memory action in the last two bits. Control process C translates those last two bits into the appropriate signals for the one-bit memory.

$$\begin{aligned} & C(a_0, a_1, r, w_0, w_1) = \\ & \quad (\ a_0;\ (a_1 \sqcup a_0);\ r \\ & \quad \sqcup\ a_1;\ (a_0;\ w_0 \sqcup a_1;\ w_1) \\ & \quad)^* \end{aligned}$$

The merge process is defined as follows.

$$\begin{aligned} & D(au_0, au_1, ad_0, ad_1, ao_0, ao_1) = \\ & \quad (au_0;\ ao_0 \sqcup au_1;\ ao_1 \sqcup ad_0;\ ao_0 \sqcup ad_1;\ ao_1)^* \end{aligned}$$

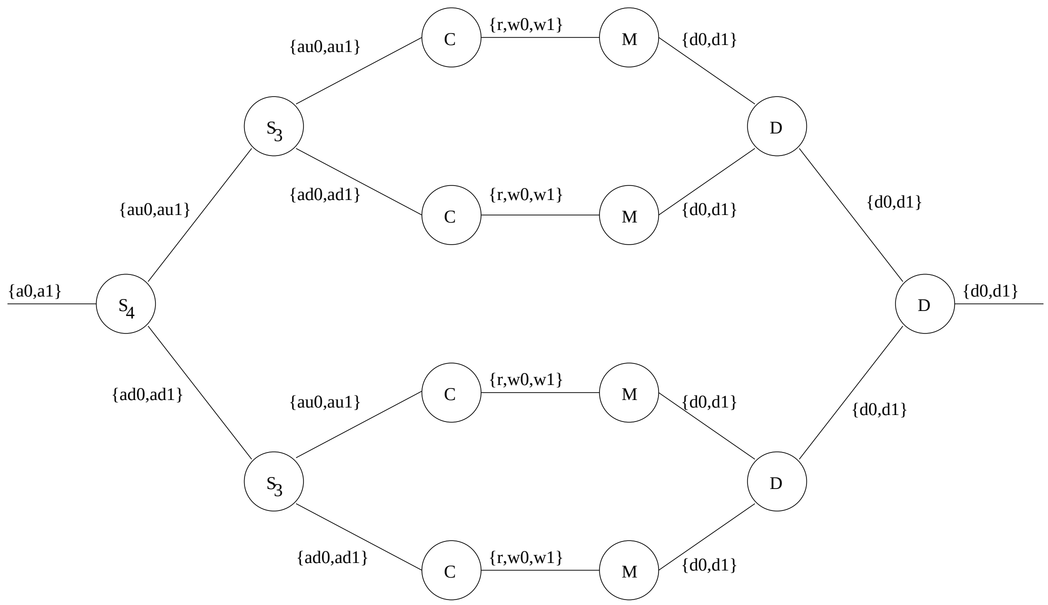


Figure 4.1: Four-bit memory.

4.9.3 Four-bit memory

And example four-bit memory is shown in Figure 4.1.

When we try to argue that this collection of processes is implementable, a problem arises. For all processes except the merge it is easy to argue that they are implementable as a consequence of the signaling-set conventions. However, we cannot guarantee that the guards of the merge processes are mutually exclusive if the environment attempts a sequence of read actions. A possible solution is to synchronize between the last D process and the first S process after every write. This, however, means that the memory can only process one write action at a time. We discuss a less restrictive solution in the next subsection.

4.9.4 Pipelined memory

The solution presented here is inspired by [26]. We modify the processes S and D to S' and D' and introduce a new process B.

We move the r/w bit to the front of the packet. A packet now consists of

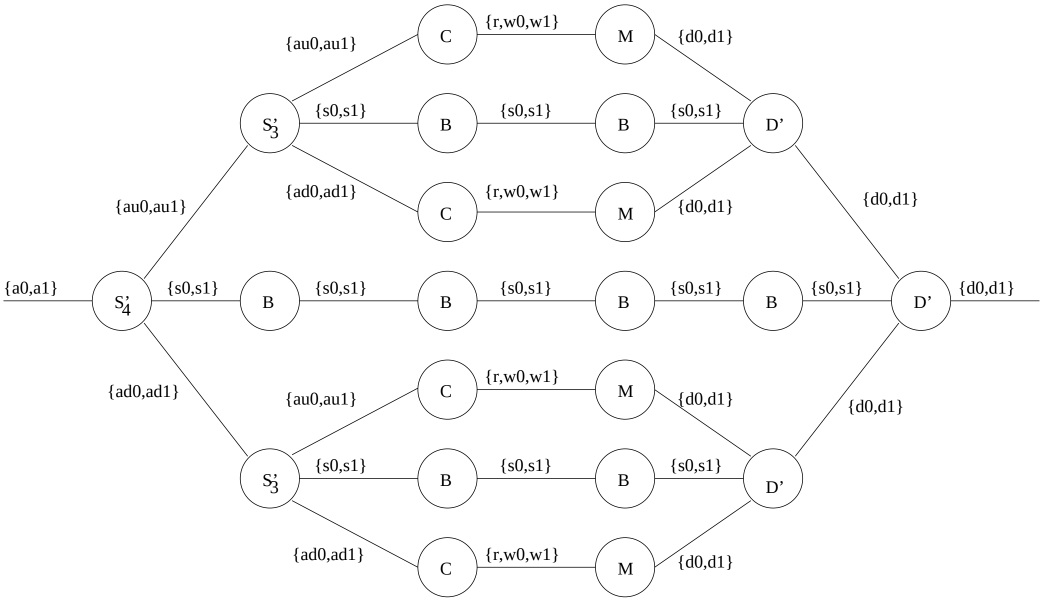


Figure 4.2: Pipelined four-bit memory.

the r/w bit, followed by $n-2$ address bits, followed by the w0/w1 bit, which carries no meaning in the case of a read.

$$
\begin{array}{l}
S'(ai_0,\, ai_1,\, au_0,\, au_1,\, ad_0,\, ad_1,\, s_0,\, s_1,\, n) = \\
\quad (\; ai_0;\, (\; ai_0;\, (au_0 \| s_0);\, (ai_0;\, au_0 \sqcup ai_1;\, au_1)^{n-2} \\
\qquad\quad \sqcup\; ai_1;\, (ad_0 \| s_1);\, (ai_0;\, ad_0 \sqcup ai_1;\, ad_1)^{n-2} \\
\qquad\quad) \\
\quad \sqcup\; ai_1;\, (\; ai_0;\, au_1;\, (ai_0;\, au_0 \sqcup ai_1;\, au_1)^{n-2} \\
\qquad\quad \sqcup\; ai_1;\, ad_1;\, (ai_0;\, ad_0 \sqcup ai_1;\, ad_1)^{n-2} \\
\qquad\quad) \\
\quad)^*
\end{array}
$$

Process S' first receives the r/w bit. In the case of a read the next bit is read to determine whether to route the packet up or down. Next the r/w bit is sent up or down, and the first bit of the address is sent to the last merging process through a series of buffers. Finally the remaining bits are copied up or down.

In the case of a write, the only difference is that no merging information is sent.

$$
\begin{array}{l}
D'(s_0, s_1, au_0, au_1, ad_0, ad_1, ao_0, ao_1) = \\
\quad (\ s_0;\ (au_0;\ ao_0 \sqcup au_1;\ ao_1) \\
\quad \sqcup\ s_1;\ (ad_0;\ ao_0 \sqcup ad_1;\ ao_1) \\
\quad)^*
\end{array}
$$

Process D' differs from process D in that it first receives a bit indicating whether a bit from the up or the down incoming channel is next. It is easy to reason that the order between different read operations is maintained with this construction.

Process B is a buffer.

$$
\begin{array}{l}
B(ai_0, ai_1, ao_0, ao_1) = \\
\quad (ai_0;\ ao_0 \sqcup ai_1;\ ao_1)^*
\end{array}
$$

All processes used in this example are implementable: mutual exclusion of the first communications in alternatives of an angelic choice can easily be shown to be mutually exclusive, as we have done in the previous section. Hence we could use the calculus we have introduced in this chapter to verify whether this memory is a refinement of another, or whether it meets a given specification.

4.10 Example: Mutual exclusion

In this section we study mutual exclusion. The solution requires the use of arbiters, and hence we cannot treat this problem fully within our calculus.

We formulate the problem thus:

Give processes A_0, B_0, A_1, B_1, M and a set of pairs X, such that

$$
\begin{array}{ll}
 & (\ (A_0;\ a;\ B_0)^\dagger \| (A_1;\ b;\ B_1)^\dagger \| (A_2;\ c;\ B_2)^\dagger \| M\ ) \\
 & \quad \mathbf{connect}\ X \\
= & \\
 & (a \sqcap b \sqcap c)^\dagger
\end{array}
.
$$

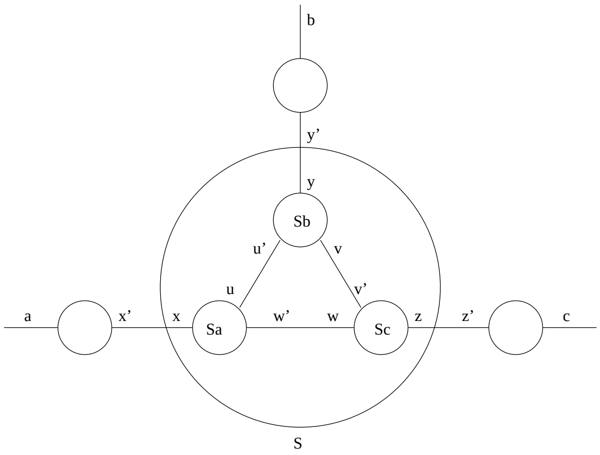


Figure 4.3: Mutual exclusion on a ring.

$$
\begin{array}{ll}
A_0 = x';\ x' & B_0 = x' \\
A_1 = y';\ y' & B_1 = y' \\
A_2 = z';\ z' & B_2 = z'
\end{array}
$$

$$
\begin{aligned}
&M = (S_a \| S_b \| S_c)\textbf{connect}\ \{u.u',\ v.v',\ w.w'\}\\
&S_a = (A \| M_a)\textbf{connect}\ \{a_0.a_0',\ a_1,\ a_1'\}\\
&S_b = (B \| M_b)\textbf{connect}\ \{b_0.b_0',\ b_1.b_1'\}\\
&S_c = (C \| M_c)\textbf{connect}\ \{c_0.c_0',\ c_1.c_1'\}\\
&M_a = a_0';\ (a_0';\ a_0' \sqcup a_1';\ a_1')^*\\
&M_b = b_1';\ (b_0';\ b_0' \sqcup b_1';\ b_1')^*\\
&M_a = c_1';\ (c_0';\ c_0' \sqcup c_1';\ c_1')^*\\
&A = (a_0;\ (u;\ u;\ a_1 \sqcup x;\ x;\ x;\ a_0)\\
&\qquad \sqcup a_1;\ (u;\ w';\ w';\ u;\ a_1 \sqcup x;\ w';\ w';\ x;\ x;\ a_0)\ )^*\\
&B = (b_0;\ (v;\ v;\ b_1 \sqcup y;\ y;\ y;\ b_0)\\
&\qquad \sqcup b_1;\ (v;\ u';\ u';\ v;\ b_1 \sqcup y;\ u';\ u';\ y;\ y;\ b_0)\ )^*\\
&C = (c_0;\ (w;\ w;\ c_1 \sqcup x;\ x;\ x;\ c_0)\\
&\qquad \sqcup c_1;\ (w;\ v';\ v';\ w;\ c_1 \sqcup z;\ v';\ v';\ z;\ z;\ c_0)\ )^*\\
&X = \{x.x',\ y.y',\ z.z'\}
\end{aligned}
$$

The solution is adapted from [30], where it is coined "reflecting privilege." We give an informal discussion only. Each slave process S_i has a one-bit memory M_i associated with it. The one bit memory encodes whether the process holds the token (state 0) or not (state 1). Initially process S_a holds the token. There are four possible actions:

If subprocess A holds the token (a_0) and receives a request from S_b (u), the request is granted immediately (u) and the process loses the token (a_1).

If subprocess A does not hold the token (a_1) and receives a request S_b (u), the token is obtained from S_c (w'; w') and granted (u). The process finally does not hold the token (a_1).

If subprocess A holds the token (a_0) and receives a request from its master (x), the request is granted (x) and the token returned when the master is done (x). The process finally holds the token (a_0).

If subprocess A does not hold the token a_1 and receives a request for the token from its master (x) and it does not hold the token (a_1), the token is obtained (w'; w') and the request is granted (x). The token is returned when the master is done (x), and the process continues to hold the token (a_0).

Observe that even though signaling set $\{a_0, a_1\}$ is probed both in process A and in process M_a, the probes are on every other communication and out of phase. Hence these angelic choices are implementable.

However, the other angelic choices in process A do not have mutually exclusive first communications. Hence an arbiter is required to implement them, and we cannot rely on our calculus to prove the correctness of the solution. Examples such as this one are the reason for studying a different operational model in Chapter 5.

4.11 Summary

In this chapter we have introduced a model for describing processes with angelic and demonic nondeterminism, sequential composition, parallel composition, and **connect** composition. We have introduced a refinement relation, which led to the introduction of equivalence classes. All operators are monotonic with respect to the refinement ordering and preserve process equivalence.

The equivalence classes form a complete lattice, with demonic and angelic nondeterminism as its meet and join. This fact allows us to introduce iteration through a fixed point equation. We have introduced two kinds of iteration: angelic and demonic.

We have studied duals in this lattice and their algebraic properties. Duals of processes have been related to the most demonic environment in which a process will not deadlock.

We have studied conditions under which the constructs we have defined, in particular angelic choice, can be implemented. We have shown that we can specify such conditions in a way that retains compositionality of our theory.

Chapter 5

An operational model for a language with probes

In this chapter we present an operational model that can be used as a basis for an extension of the language discussed in the previous chapters. We clarify the restrictions mentioned in the previous chapter, and we indicate why a language that deals successfully with all the semantic aspects of the model presented in this chapter will have much more complicated algebraic properties.

Operational models abound in the literature. The one we present here distinguishes itself from all others by treating probes explicitly.

5.1 Operational model

Our operational model for processes consists of a set of nodes, a set of labeled directed edges, and two special nodes: the initial node and the final node. Edges are labeled with symbols from a set of actions extended with a special empty (ϵ) action, with a pair of actions, or with a probed action. The main distinction between this model and a transition graph, that it resembles, is that choice is understood to be demonic. If more than one transition is possible from a given node (state), the process makes a demonic choice. An ϵ transition is always possible; other actions are possible if the environment can participate in them.

We represent a process as a four-tuple: set of nodes, edges, initial node,

Figure 5.1: **skip** and **deadlock**.

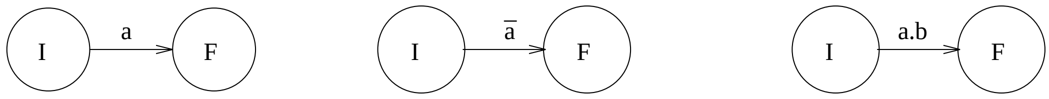


Figure 5.2: a,$\overline{a}$,and $a.b$.

and final node. A process is represented by an upper case letter such as S or T. We define $S = \langle S.nodes, S.edges, S.ini, S.fin \rangle$. **Nodes** is the set of all nodes, **Edges** is the set of all edges, **Atoms** is the set of all actions, **Probes** is the set of all probed actions, and **Pairs** is the set of all pairs of actions. An edge is represented by a triple: initial node, target node, and label.

In Figure 5.1 we have depicted two basic processes, **skip** and **deadlock** . They correspond to the four-tuples

deadlock $= \langle \{I, F\}, \{\}, I, F \rangle$ **skip** $= \langle \{I, F\}, \{\langle I, F, \epsilon \rangle\}, I, F \rangle$

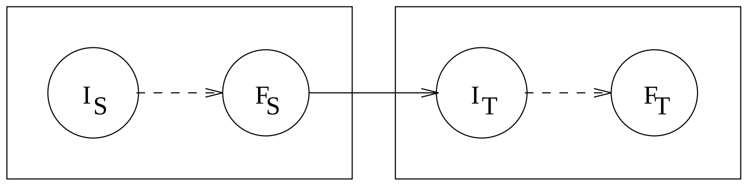
In Figure 5.2 we have depicted three kinds of processes with one labeled transition.

$$a = \langle \{I, F\}, \{\langle I, F, a \rangle\}, I, F \rangle \qquad \overline{a} = \langle \{I, F\}, \{\langle I, F, \overline{a} \rangle\}, I, F \rangle$$
$$a.b = \langle \{I, F\}, \{\langle I, F, a.b \rangle\}, I, F \rangle$$

Sequential composition between processes is depicted in Figure 5.3 and is defined as follows.

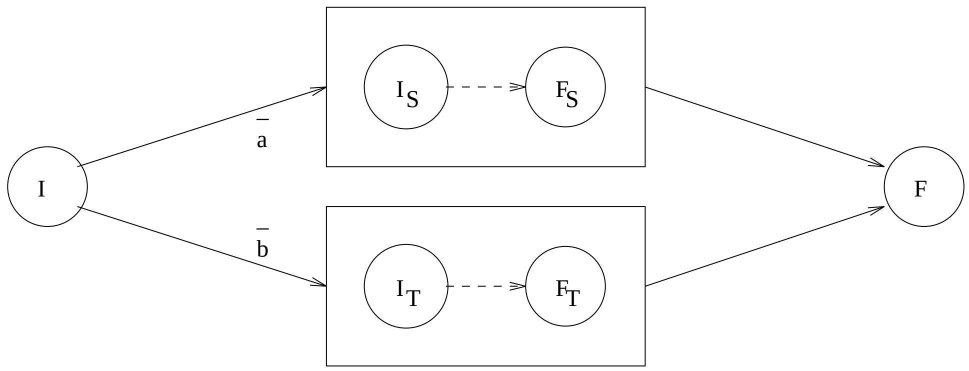
$$S;\ T$$
$$=$$
$$\langle S.nodes \cup T.nodes, S.edges \cup T.edges \cup \langle S.fin,\ T.ini, \epsilon \rangle, S.ini,\ T.fin \rangle$$

Choice composition between processes is depicted in Figure 5.4 and is defined as follows.



$S \; ; \; T$

Figure 5.3: Sequential composition.



$\bar{a} \longrightarrow S \;\; \square \;\; \bar{b} \longrightarrow T$

Figure 5.4: Choice composition.

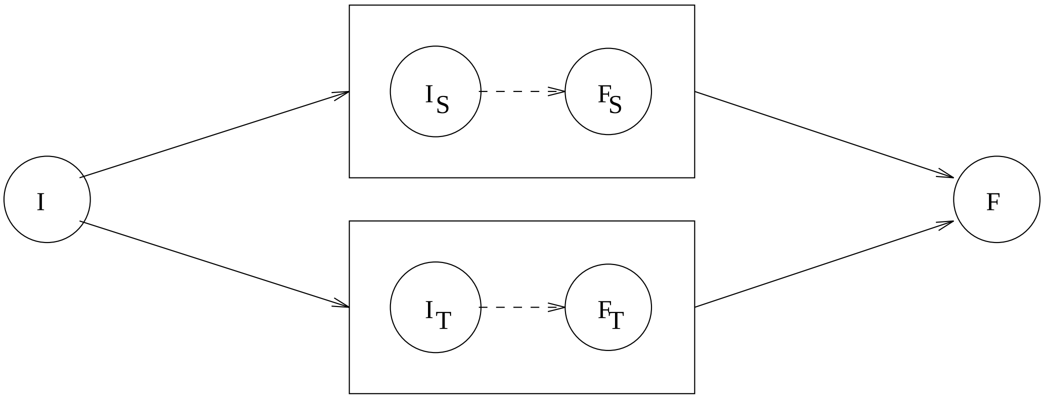


Figure 5.5: Demonic composition.

$$
\begin{array}{ll}
 & \overline{a_0} \rightarrow S_0 \;[]\; .... \;[]\; \overline{a_n} \rightarrow S_n \\
= & \\
 & \langle \{I, F\} \cup \; \cup(i : 0 \leq i \leq n : S_i.nodes) \\
 & \;, \cup(i : 0 \leq i \leq n : \{\langle I, S_i.ini, \overline{a_i}\rangle, \langle S_i.fin, F, \epsilon\rangle\}) \\
 & \quad \cup \; \cup(i : 0 \leq i \leq n : S_i.edges) \\
 & \;, I \\
 & \;, F \\
 & \rangle
\end{array}
$$

Demonic composition between processes is depicted in Figure 5.5 and is defined as follows.

$$
\begin{array}{ll}
 & S_0 \sqcap .... \sqcap S_n \\
= & \\
 & \langle \{I, F\} \cup \; \cup(i : 0 \leq i \leq n : S_i.nodes) \\
 & \;, \cup(i : 0 \leq i \leq n : \{\langle I, S_i.ini, \epsilon\rangle, \langle S_i.fin, F, \epsilon\rangle\}) \\
 & \quad \cup \; \cup(i : 0 \leq i \leq n : S_i.edges) \\
 & \;, I \\
 & \;, F \\
 & \rangle
\end{array}
$$

An example of parallel composition between processes is depicted in Fig-

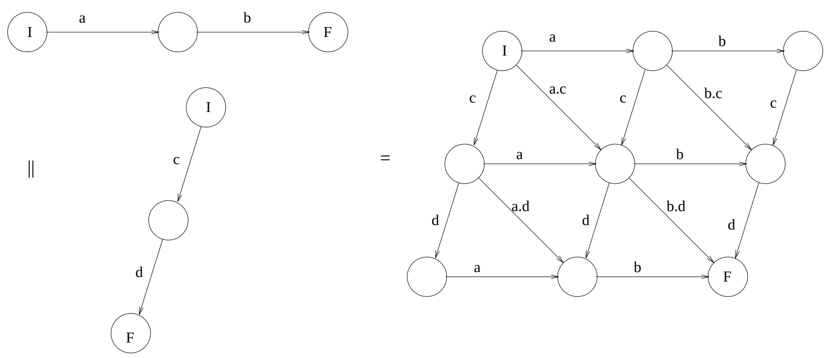


Figure 5.6: Example of parallel composition.

ure 5.6. The figure depicts $(a;\ b)\|(c;\ d)$. Parallel composition is defined as follows. We assume that the nodes in S and T are numbered, and that they are disjoint.

$$
\begin{array}{ll}
 & S\|T \\
= & \\
 & \langle\{i,j : n_i \in S.nodes \ \wedge \ n_j \in T.nodes : n_{i,j}\} \\
 & \ ,\{i,j,k,x : \langle n_i, n_j, x\rangle \in S.edges \ \wedge \ n_k \in T.nodes : \langle n_{i,k}, n_{j,k}, x\rangle\} \\
 & \quad \cup\{i,j,k,x : n_k \in S.nodes \ \wedge \ \langle n_i, n_j, x\rangle \in T.edges : \langle n_{k,i}, n_{k,j}, x\rangle\} \\
 & \quad \cup\{i,j,k,l,x,y : \langle n_i, n_j, x\rangle \in S.edges \ \wedge \ \langle n_k, n_l, y\rangle \in T.edges \\
 & \qquad\qquad \wedge \ x \in \mathbf{Atoms} \ \wedge \ y \in \mathbf{Atoms} : \langle n_{i,k}, n_{j,l}, x.y\rangle\} \\
 & \ ,\{i,j : n_i = S.ini \ \wedge \ n_j = T.ini : n_{i,j}\} \\
 & \ ,\{i,j : n_i = S.fin \ \wedge \ n_j = T.fin : n_{i,j}\} \\
 & \rangle
\end{array}
$$

The explanation for this definition is that it is a direct product (because two processes composed in parallel can progress independently) extended with possible actions in which they can both participate.

An example of connect composition between processes is depicted in Figure 5.7. The figure depicts $((a;\ b)\|(c;\ d))\mathbf{connect}\{a.b\}$. Connect composition is defined as follows.

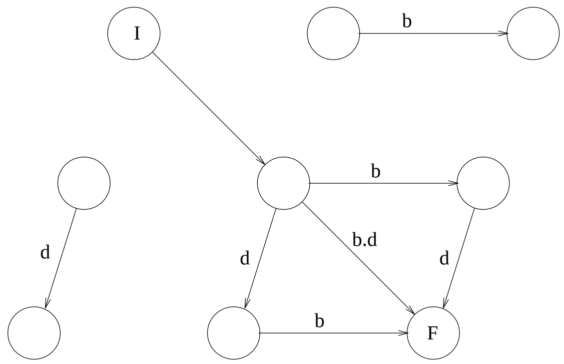


Figure 5.7: Example of connect composition.

$$
\begin{array}{ll}
 & S \textbf{ connect } \{a.b\} \\
= & \\
 & \langle S.nodes \\
 & ,\{n,m,x : \langle n,m,x\rangle \in S.edges \ \wedge \ \alpha(x) \cap \{a,b,\overline{a},\overline{b}\} = \{\} : \langle n,m,x\rangle\} \\
 & \cup\{n,m : \langle n,m,a.b\rangle \in S.edges : \langle n,m,\epsilon\rangle\} \\
 & \cup\{n,m : \langle n,m,\overline{a}\rangle \in S.edges \ \wedge \ \exists(o :: \langle n,o,b\rangle \in S.edges) : \langle n,m,\epsilon\rangle\} \\
 & \cup\{n,m : \langle n,m,\overline{b}\rangle \in S.edges \ \wedge \ \exists(o :: \langle n,o,a\rangle \in S.edges) : \langle n,m,\epsilon\rangle\} \\
 & ,S.ini\ ,S.fin \\
 & \rangle
\end{array}
$$

The explanation for this definition is that when connecting a and b, only pair actions $a.b$ can occur. They are hidden from the environment and therefore replaced by ϵ transitions. Probed transitions can occur if S is in a state in which the corresponding action, i.e., a for $\overline{b}$ and b for $\overline{a}$, is possible.

There are three rules for simplifying graphs.

- Unreachable parts of graphs may be removed.
- A node that has only one outgoing ϵ edge may be removed; its incoming edges are then redirected to the target node of the old outgoing ϵ edge. If the graph is a direct product of two other graphs, these simplifications may be done in the components, and a simplified graph may be

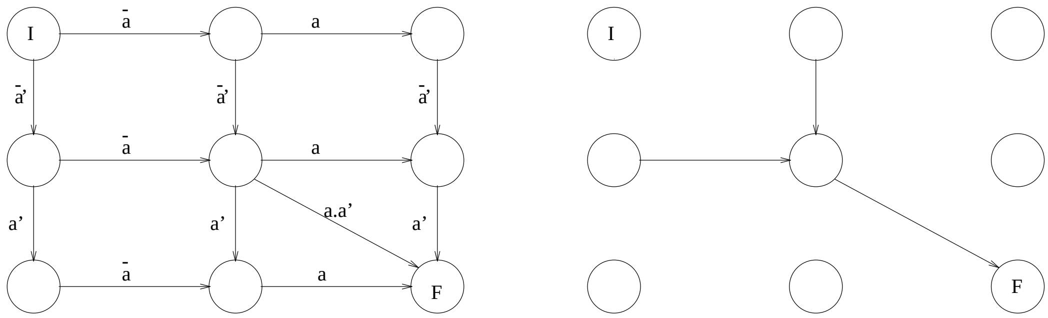


Figure 5.8: First example.

obtained by taking the direct product of the simplified components.

- A graph that has a path along ϵ edges from the initial node to a node with no outgoing edges that is not the final node may be simplified to **deadlock** .

5.2 Some examples

In this section we show some examples in which the operational model presented in the previous section deals successfully with some processes for which our process algebra was insufficient.

Our first example is

$$((\overline{a} \rightarrow a)\|(\overline{a}' \rightarrow a')) \textbf{ connect } \{a.a', b.b'\} \;=\; \textbf{deadlock}.$$

The calculation is represented in Figure 5.8. The first graph represents the process before the **connect** operation; the second graph represents the connected process. Because of our third simplification rule it may be simplified to **deadlock** .

A similar calculation shows

$$\begin{aligned} & ((\overline{a} \rightarrow a \;[]\; \overline{b} \rightarrow b)\|(\overline{a}' \rightarrow a' \;[]\; \overline{b}' \rightarrow b')) \textbf{ connect } \{a.a', b.b'\} \\ = \;& \\ & \textbf{deadlock}. \end{aligned}$$

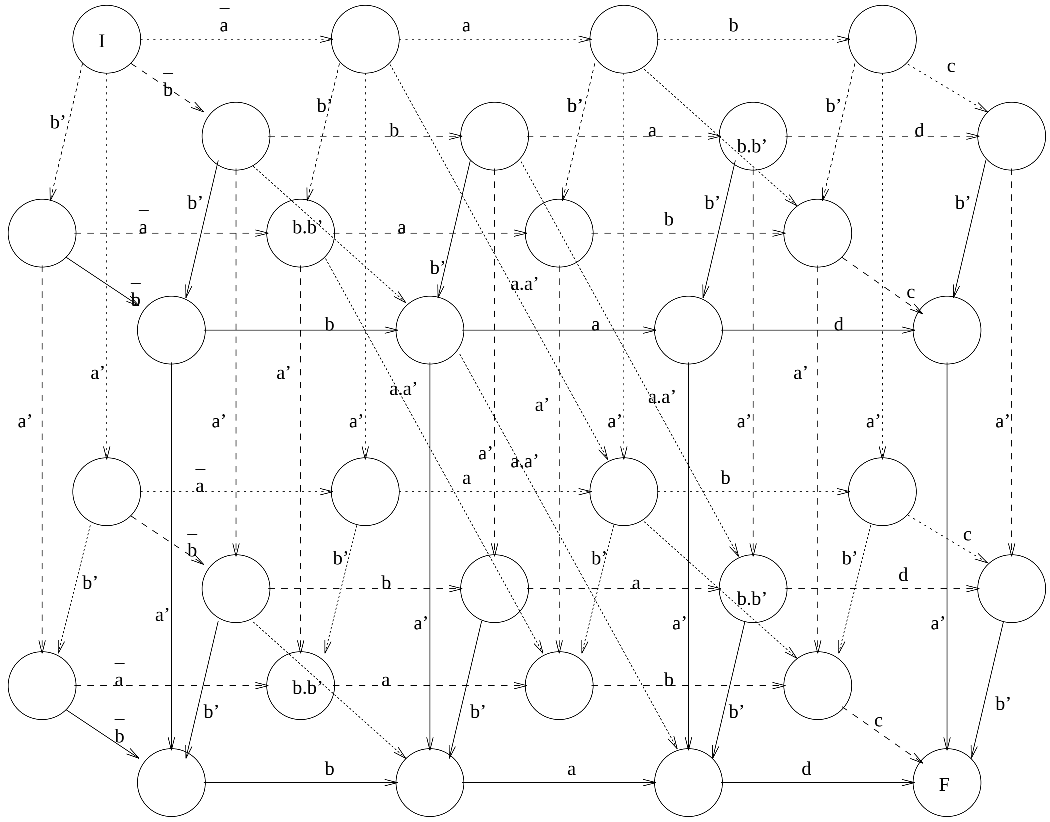


Figure 5.9: Full graph.

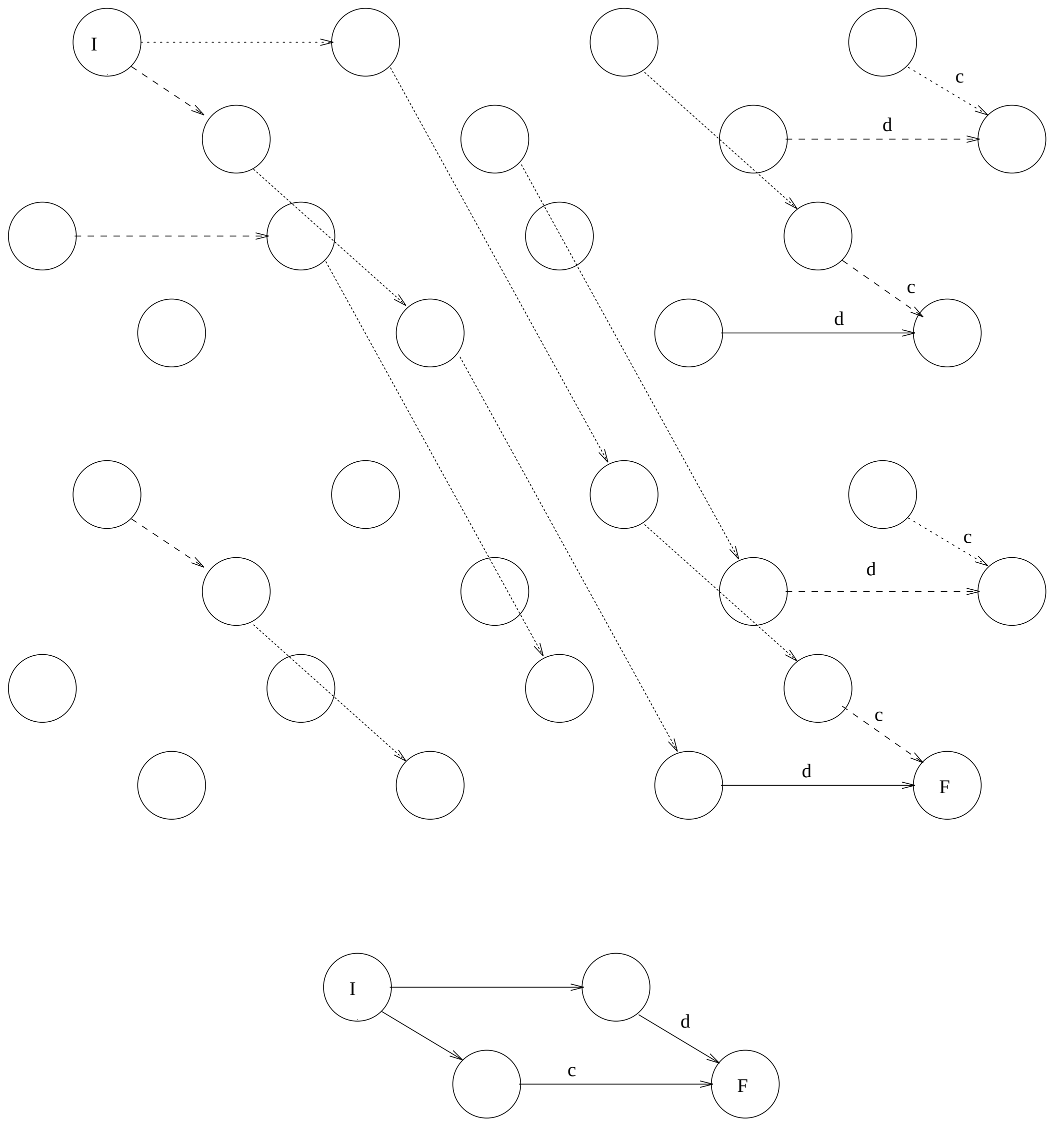


Figure 5.10: Reduced graph.

Our second example is

$$(a' \| b' \| (\overline{a} \rightarrow a;\ b;\ c \ [\!]\ \overline{b} \rightarrow b;\ a;\ d))\ \textbf{connect}\ \ \{a.a', b.b'\} = c \sqcap d$$

The graph for the unconnected process is given in Figure 5.9. A few edges labeled with pairs have been omitted for clarity; they would be removed in the next step anyway. The connect operation and the simplifying rules are applied in Figure 5.10.

A third example is depicted in Figure 5.11. It shows the graph for

$$(a' \| (\overline{a} \rightarrow a;\ b;\ c \ [\!]\ \overline{b} \rightarrow b;\ a;\ d))\ \textbf{connect}\ \ \{a.a'\}$$

.

The resulting graph corresponds to

$$(\epsilon \rightarrow b;\ c \ [\!]\ \overline{b} \rightarrow b;\ d)$$

.

This is *not* equivalent to process $b;\ c$. Composed with the environment b', as in the previous example, the result is $c \sqcap d$, as expected.

5.3 Refinement

A process S refines T if in every environment where T does not deadlock, S also does not deadlock. According to this definition and the examples in the previous section, we see that $a \| b$ does not refine $a;\ b$ because we have

$$\begin{array}{ll} & ((\overline{a} \rightarrow a;\ b;\ c \ [\!]\ \overline{b} \rightarrow b;\ a;\ d) \| a' \| b' \| c') \\ & \quad \textbf{connect}\ \ \{a.a', b.b', c.c', d.d'\} \\ = & \\ & ((c \sqcap d) \| c')\ \textbf{connect}\ \{c.c', d.d'\} \\ = & \\ & \textbf{deadlock} \end{array}$$

whereas

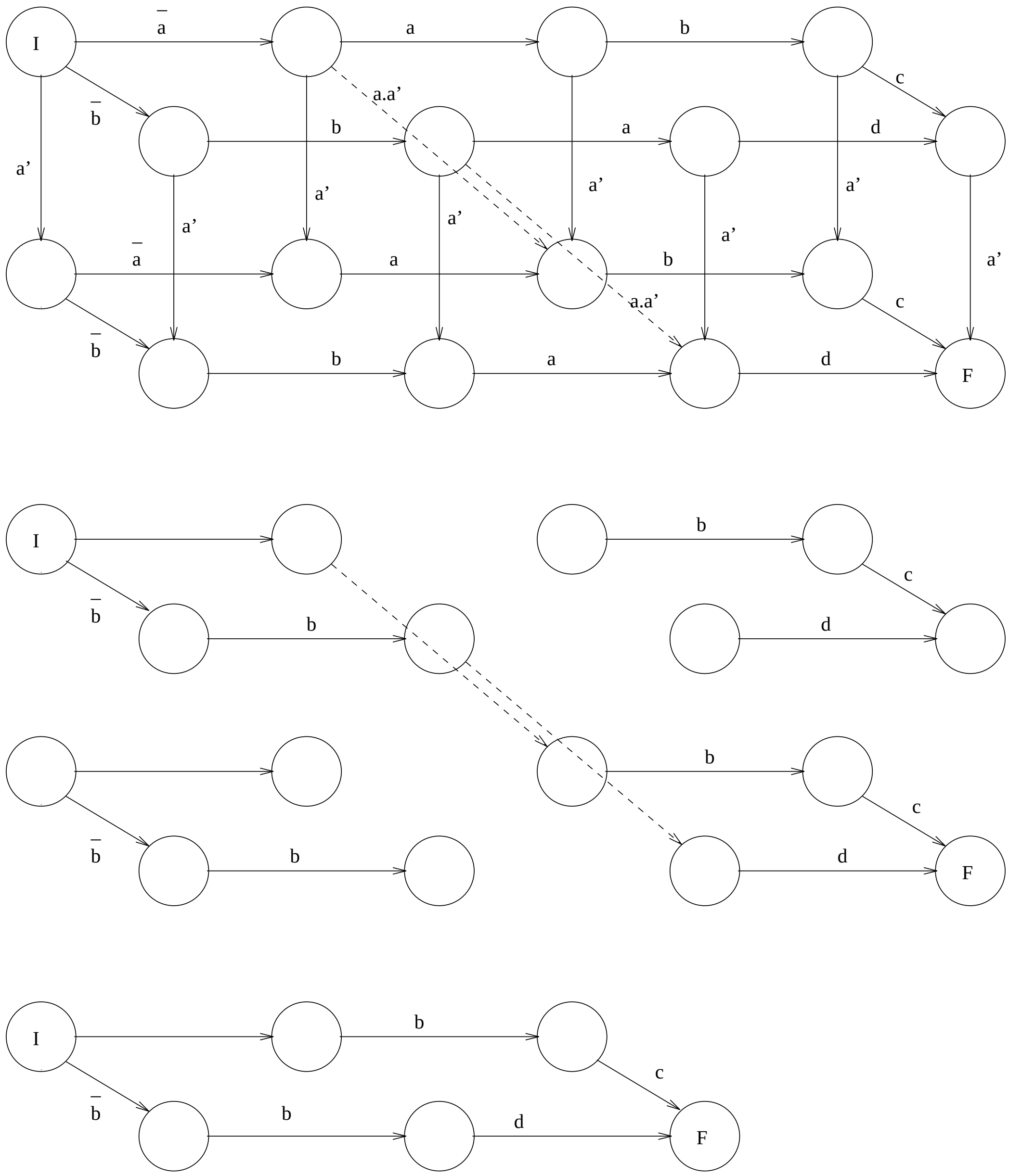


Figure 5.11: Partial connect.

$$
\begin{array}{ll}
 & ((\overline{a} \rightarrow a;\ b;\ c \ [\!]\ \overline{b} \rightarrow b;\ a;\ d) \| (a';\ b') \| c') \\
 & \quad \textbf{connect}\ \ \{a.a', b.b', c.c', d.d'\} \\
= & \\
 & (c \| c')\ \textbf{connect}\ \{c.c', d.d'\} \\
= & \\
 & \textbf{skip}\ .
\end{array}
$$

Also, $(\overline{a} \rightarrow a;\ b \ [\!]\ \overline{b} \rightarrow b;\ a)$ does not refine $a;\ b$ because

$$
\begin{array}{ll}
 & ((\overline{a} \rightarrow a;\ b \ [\!]\ \overline{b} \rightarrow b;\ a) \| (a';\ b'))\ \textbf{connect}\ \{a.a', b.b'\} \\
= & \\
 & \textbf{skip}
\end{array}
$$

whereas

$$
\begin{array}{ll}
 & ((\overline{a} \rightarrow a;\ b \ [\!]\ \overline{b} \rightarrow b;\ a) \\
 & \|(\overline{a} \rightarrow a;\ b \ [\!]\ \overline{b} \rightarrow b;\ a))\ \textbf{connect}\ \{a.a', b.b'\} \\
= & \\
 & \textbf{deadlock}\ .
\end{array}
$$

These and other examples indicate that it is not easy to express parallel composition and ALT composition in terms of angelic and demonic composition. Hence, if we want an algebra that can handle the full operational model with probes, we should expect it to be significantly more complex than the algebra we have discussed in Chapter 4.

5.4 Summary

We have presented a simple operational model in which we can reason about languages with probes. The complications that arise give us confidence that significant gain over the semantics in Chapter 4 cannot be obtained without significant pain. Still, the extension is interesting enough to make it seem worthwhile to try to create a process algebra for it.

Chapter 6

Conclusions and future work

6.1 Conclusions

We have defined a simple language for programming and specifications with synchronization as the basic construct. The language has very pleasing algebraic properties and a model that is easy to understand. A refinement relation forms the core of our approach. All programming constructs we have defined are monotonic with respect to this refinement relation. We believe that concentrating on the refinement relation was a good choice. It has guided us in finding the equivalence classes we identify with processes, it has guided us in choosing the right definitions for the operators in our language, it has ensured compositionality, and it has yielded a refinement calculus.

Including pairsymbols in the traces allows the introduction of a self- synchronizing **connect** operator. This operator corresponds more directly to declarations and rules of scope in a variety of concurrent languages than the more traditional hiding operators.

A big advantage of our approach over that of many others is the degree of compositionality; when constructing a large system we can do so in terms of the specification of the parts, not their implementation.

6.2 Future work

One way of looking at the calculations in Sections 4.7 and 4.8 is to regard them as refinements in a context. We restrict the class of possible environments, and state that one process refines another just when it does so for all environments in the class. Refinement in context seems to be a powerful way to bridle complexity, and deserves further study.

This monograph has dealt only with finite structures. Although one might argue that these should suffice for describing programs that are to execute in finite space and finite time, from a theoretical point of view the extension to infinite constructs is interesting.

Even though the language we have defined in Chapter 4 allows us to compute anything, the language is not very practical. An obvious extension is to allow for expressions in guards, which in turn seems to require the introduction of probes into the language. The operational model of Chapter 5 should help in identifying a subset of programs with probes, that still has elegant algebraic properties.

The next step would be to include state and/or types in the model. A fairly straightforward way of including types is to identify typed variables with signaling sets. This is essentially what has been done in the examples in Chapter 4. Introducing infinite constructs should also allow infinite signaling sets, and thus types with an infinite range. Signaling sets might also form the basis for the treatment of local variables. Still, the problem is by no means solved; it is not yet clear, for instance, how to relate more efficient implementations that require $log(n)$ wires for a variable with n different values in its range to the signaling sets mentioned above. It seems likely that work in data refinement [22, 19] will prove relevant here.

Finally, even though we have written some simple programs to aid in the calculations, an effort should be made to produce software to aid the programmer in the construction of correct concurrent programs. Part of such an effort might be a formalization of the model presented here in a mechanized proof system such as Larch [21] or HOL [20].

Appendix A

Proofs for Section 3.2

(3.5) $\quad (X \supseteq Y) \Rightarrow ((t \Downarrow X) \Downarrow Y \;=\; t \Downarrow Y)$

Proof, by induction on the length of t
Base case, $t = \epsilon$,

$$
\begin{array}{ll}
 & (\epsilon \Downarrow X) \Downarrow Y \\
= & \{ \text{ Definition } \Downarrow \} \\
 & \epsilon \\
= & \{ \text{ Definition } \Downarrow \} \\
 & \epsilon \Downarrow Y
\end{array}
$$

Induction step, cases

$$
\begin{array}{ll}
 & ((at) \Downarrow X) \Downarrow Y = a((t \Downarrow X) \Downarrow Y) \\
= & \{ \text{ Definition } \Downarrow \} \\
 & a \in X \;\wedge\; a \in Y \\
= & \{ \ X \supseteq Y \ \} \\
 & a \in Y \\
= & \{ \text{ Definition } \Downarrow \} \\
 & (at) \Downarrow Y = a(t \Downarrow Y)
\end{array}
$$

$$
\begin{array}{ll}
 & ((at) \Downarrow X) \Downarrow Y = (t \Downarrow X) \Downarrow Y \\
= & \{ \text{ Definition } \Downarrow \} \\
 & a \notin X \;\vee\; a \notin Y \\
= & \{ \ X \supseteq Y \ \}
\end{array}
$$

$$
\begin{aligned}
& a \notin Y \\
= \{ & \text{ Definition } \Downarrow \} \\
& (at) \Downarrow Y = t \Downarrow Y
\end{aligned}
$$

$$
\begin{aligned}
& ((a.b\ t) \Downarrow X) \Downarrow Y = a.b((t \Downarrow X) \Downarrow Y) \\
= \{ & \text{ Definition } \Downarrow \} \\
& (a.b \in X \ \vee \ (a \in X \ \wedge \ b \in X)) \ \wedge \ (a.b \in Y \ \vee \ (a \in Y \ \wedge \ b \in Y)) \\
= \{ & \ X \supseteq Y \ \} \\
& (a.b \in Y \ \vee \ (a \in Y \ \wedge \ b \in Y)) \\
= \{ & \text{ Definition } \Downarrow \} \\
& (a.b\ t) \Downarrow Y = a.b(t \Downarrow Y)
\end{aligned}
$$

$$
\begin{aligned}
& ((a.b\ t) \Downarrow X) \Downarrow Y = a((t \Downarrow X) \Downarrow Y) \\
= \{ & \text{ Definition } \Downarrow \} \\
& ((a.b \in X \ \vee \ (a \in X \ \wedge \ b \in X)) \ \wedge \ (a.b \notin Y \ \wedge \ a \in Y \ \wedge \ b \notin Y)) \vee \\
& ((a.b \notin X \ \wedge \ a \in X \ \wedge \ b \notin X) \ \wedge \ a \in Y) \\
= \{ & \ X \supseteq Y \ \} \\
& (a.b \notin Y \ \wedge \ a \in Y \ \wedge \ b \notin Y) \\
= \{ & \text{ Definition } \Downarrow \} \\
& (a.b\ t) \Downarrow Y = a(t \Downarrow Y)
\end{aligned}
$$

$$
\begin{aligned}
& ((a.b\ t) \Downarrow X) \Downarrow Y = (t \Downarrow X) \Downarrow Y \\
= \{ & \text{ Definition } \Downarrow \} \\
& (a.b \notin X \ \wedge \ a \notin X \ \wedge \ b \notin X) \vee \\
& (a.b \notin X \ \wedge \ a \in X \ \wedge \ b \notin X \ \wedge \ a \notin Y) \vee \\
& (a.b \notin X \ \wedge \ a \notin X \ \wedge \ b \in X \ \wedge \ b \notin Y) \vee \\
& ((a.b \in X \ \vee \ (a \in X \ \wedge \ b \in X)) \ \wedge \ (a.b \notin Y \ \wedge \ a \notin Y \ \wedge \ b \notin Y)) \\
= \{ & \ X \supseteq Y \ \} \\
& (a.b \notin X \ \wedge \ a \notin X \ \wedge \ b \notin X \ \wedge \ a.b \notin Y \ \wedge \ a \notin Y \ \wedge \ b \notin Y) \vee \\
& (a.b \notin X \ \wedge \ a \in X \ \wedge \ b \notin X \ \wedge \ a.b \notin Y \ \wedge \ a \notin Y \ \wedge \ b \notin Y) \vee \\
& (a.b \notin X \ \wedge \ a \notin X \ \wedge \ b \in X) \ \wedge \ a.b \notin Y \ \wedge \ a \notin Y \ \wedge \ b \notin Y) \vee \\
& ((a.b \in X \ \vee \ (a \in X \ \wedge \ b \in X)) \ \wedge \ (a.b \notin Y \ \wedge \ a \notin Y \ \wedge \ b \notin Y)) \\
= \{ & \text{ Calculus } \} \\
& (a.b \notin Y \ \wedge \ a \notin Y \ \wedge \ b \notin Y) \\
= \{ & \text{ Definition } \Downarrow \} \\
& (a.b\ t) \Downarrow Y = t \Downarrow Y
\end{aligned}
$$

which, as can be seen by taking the disjunction of the five conditions in the definition of projection, are the only cases we need to consider.
end of proof

Corollary $(t \Downarrow X) \Downarrow X = (t \Downarrow X)$

$$(3.6) \qquad (X \subseteq Y) \Rightarrow ((t \Downarrow X) \Downarrow Y \;=\; t \Downarrow X)$$

Proof, by induction on the length of t
Base case, $t = \epsilon$,

$$
\begin{aligned}
& (\epsilon \Downarrow X) \Downarrow Y \\
= & \{ \text{ Definition } \Downarrow \} \\
& \epsilon \\
= & \{ \text{ Definition } \Downarrow \} \\
& \epsilon \Downarrow X
\end{aligned}
$$

Induction step, cases

$$
\begin{aligned}
& ((at) \Downarrow X) \Downarrow Y = a((t \Downarrow X) \Downarrow Y) \\
= & \{ \text{ Definition } \Downarrow \} \\
& a \in X \;\wedge\; a \in Y \\
= & \{ X \subseteq Y \} \\
& a \in X \\
= & \{ \text{ Definition } \Downarrow \} \\
& (at) \Downarrow X = a(t \Downarrow X)
\end{aligned}
$$

$$
\begin{aligned}
& ((at) \Downarrow X) \Downarrow Y = (t \Downarrow X) \Downarrow Y \\
= & \{ \text{ Definition } \Downarrow \} \\
& a \notin X \;\vee\; a \notin Y \\
= & \{ X \subseteq Y \} \\
& a \notin X \\
= & \{ \text{ Definition } \Downarrow \} \\
& (at) \Downarrow X = t \Downarrow X
\end{aligned}
$$

$$
\begin{aligned}
& ((a.b\ t) \Downarrow X) \Downarrow Y = a.b((t \Downarrow X) \Downarrow Y) \\
= & \ \{ \text{ Definition } \Downarrow \ \} \\
& (a.b \in X \ \vee \ (a \in X \ \wedge \ b \in X)) \ \wedge \ (a.b \in Y \ \vee \ (a \in Y \ \wedge \ b \in Y)) \\
= & \ \{ \ X \subseteq Y \ \} \\
& (a.b \in X \ \vee \ (a \in X \ \wedge \ b \in X)) \\
= & \ \{ \text{ Definition } \Downarrow \ \} \\
& (a.b\ t) \Downarrow X = a.b(t \Downarrow X)
\end{aligned}
$$

$$
\begin{aligned}
& ((a.b\ t) \Downarrow X) \Downarrow Y = a((t \Downarrow X) \Downarrow Y) \\
= & \ \{ \text{ Definition } \Downarrow \ \} \\
& ((a.b \in X \ \vee \ (a \in X \ \wedge \ b \in X)) \ \wedge \ a.b \notin Y \ \wedge \ a \in Y \ \wedge \ b \notin Y) \vee \\
& (a.b \notin X \ \wedge \ a \in X \ \wedge \ b \notin X \ \wedge \ a \in Y) \\
= & \ \{ \ X \subseteq Y \ \} \\
& (a.b \notin X \ \wedge \ a \in X \ \wedge \ b \notin X) \\
= & \ \{ \text{ Definition } \Downarrow \ \} \\
& (a.b\ t) \Downarrow X = a(t \Downarrow X)
\end{aligned}
$$

$$
\begin{aligned}
& ((a.b\ t) \Downarrow X) \Downarrow Y = (t \Downarrow X) \Downarrow Y \\
= & \ \{ \text{ Definition } \Downarrow \ \} \\
& ((a.b \in X \ \vee \ (a \in X \ \wedge \ b \in X)) \ \wedge \ a.b \notin Y \ \wedge \ a \notin Y \ \wedge \ b \notin Y) \vee \\
& (a.b \notin X \ \wedge \ a \in X \ \wedge \ b \notin X \ \wedge \ a \notin Y) \vee \\
& (a.b \notin X \ \wedge \ a \notin X \ \wedge \ b \in X \ \wedge \ b \notin Y) \vee \\
& (a.b \notin X \ \wedge \ a \notin X \ \wedge \ b \notin X) \\
= & \ \{ \ X \subseteq Y, \text{ hence first three disjuncts are } \mathit{false} \ \} \\
& (a.b \notin X \ \wedge \ a \notin X \ \wedge \ b \notin X) \\
= & \ \{ \text{ Definition } \Downarrow \ \} \\
& (a.b\ t) \Downarrow X = t \Downarrow X
\end{aligned}
$$

which, as can be seen by taking the disjunction of the five conditions in the definition of projection, are the only cases we need to consider.
end of proof

(3.7) $\quad (X \supseteq Y) \Rightarrow ((t \Uparrow X) \Uparrow Y \ = \ t \Uparrow X)$

Proof, by induction on the length of t
Base case, $t = \epsilon$,

$$
\begin{array}{ll}
 & (\epsilon \Uparrow X) \Uparrow Y \\
= & \{ \text{ Definition } \Uparrow \} \\
 & \epsilon \\
= & \{ \text{ Definition } \Uparrow \} \\
 & \epsilon \Uparrow X
\end{array}
$$

Induction step, cases

$$
\begin{array}{ll}
 & ((at) \Uparrow X) \Uparrow Y = a((t \Uparrow X) \Uparrow Y) \\
= & \{ \text{ Definition } \Uparrow \} \\
 & a \notin X \ \wedge \ a \notin Y \\
= & \{ \ X \supseteq Y \ \} \\
 & a \notin X \\
= & \{ \text{ Definition } \Uparrow \} \\
 & (at) \Uparrow X = a(t \Uparrow X)
\end{array}
$$

$$
\begin{array}{ll}
 & ((at) \Uparrow X) \Uparrow Y = (t \Uparrow X) \Uparrow Y \\
= & \{ \text{ Definition } \Uparrow \} \\
 & a \in X \ \vee \ a \in Y \\
= & \{ \ X \supseteq Y \ \} \\
 & a \in X \\
= & \{ \text{ Definition } \Uparrow \} \\
 & (at) \Uparrow X = t \Uparrow X
\end{array}
$$

$$
\begin{array}{ll}
 & ((a.b\ t) \Uparrow X) \Uparrow Y = a.b((t \Uparrow X) \Uparrow Y) \\
= & \{ \text{ Definition } \Uparrow \} \\
 & (a.b \notin X \ \wedge \ a \notin X \ \wedge \ b \notin X) \ \wedge \ (a.b \notin Y \ \wedge \ a \notin Y \ \wedge \ b \notin Y)) \\
= & \{ \ X \supseteq Y \ \} \\
 & (a.b \notin X \ \wedge \ a \notin X \ \wedge \ b \notin X) \\
= & \{ \text{ Definition } \Uparrow \} \\
 & (a.b\ t) \Uparrow X = a.b(t \Uparrow X)
\end{array}
$$

$$
\begin{array}{ll}
 & ((a.b\ t) \Uparrow X) \Uparrow Y = a((t \Uparrow X) \Uparrow Y) \\
= & \{ \text{ Definition } \Uparrow \} \\
 & (a.b \notin X \ \wedge \ a \notin X \ \wedge \ b \notin X \ \wedge \ a.b \notin Y \ \wedge \ a \notin Y \ \wedge \ b \in Y) \vee \\
 & (a.b \notin X \ \wedge \ a \notin X \ \wedge \ b \in X \ \wedge \ a \notin Y) \\
= & \{ \ X \supseteq Y \ \} \\
 & (a.b \notin X \ \wedge \ a \notin X \ \wedge \ b \in X)
\end{array}
$$

$= \{$ Definition $\Uparrow$ $\}$
$(a.b\ t) \Uparrow X = a(t \Uparrow X)$

$((a.b\ t) \Uparrow X) \Uparrow Y = (t \Uparrow X) \Uparrow Y$
$= \{$ Definition $\Uparrow$ $\}$
$(a \in X \ \wedge\ b \in X) \ \vee\ a.b \in X \vee$
$(a.b \notin X \ \wedge\ a \notin X \ \wedge\ b \in X \ \wedge\ a \in Y) \vee$
$(a.b \notin X \ \wedge\ a \in X \ \wedge\ b \notin X \ \wedge\ b \in Y) \vee$
$(a.b \notin X \ \wedge\ a \notin X \ \wedge\ b \notin X \ \wedge\ (a.b \in Y \ \vee\ (a \in Y \ \wedge\ b \in Y)))$
$= \{$ $X \supseteq Y$, hence last three disjuncts are *false* $\}$
$(a.b \in X \ \vee\ (a \in X \ \wedge\ b \in X))$
$= \{$ Definition $\Uparrow$ $\}$
$(a.b\ t) \Uparrow X = t \Uparrow X$

which, as can be seen by taking the disjunction of the five conditions in the definition of projection, are the only cases we need to consider.
end of proof

Corollary $\quad (t \Uparrow X) \Uparrow X = (t \Uparrow X)$

$$(3.8) \qquad \sigma(s) \subseteq Y \ \wedge\ (r \Downarrow (X \cup Y) = s) \Rightarrow (r \Downarrow Y = s)$$

Proof, by induction on the length of r.
Base case $r = \epsilon$, trivial.
Induction step, two cases.

$ar \Downarrow (X \cup Y) = s$
$= \{$ Definition $\Downarrow$ $\}$
$(a \Downarrow (X \cup Y))(r \Downarrow (X \cup Y)) = s$
$= \{$ Cases $\}$
$(a \in (X \cup Y) \ \wedge\ a(r \Downarrow (X \cup Y)) = s) \vee$
$(a \notin (X \cup Y) \ \wedge\ r \Downarrow (X \cup Y) = s)$
$= \{$ Decomposition $\}$
$\exists(t : at = s : a \in Y \ \wedge\ (r \Downarrow (X \cup Y)) = t) \vee$
$(a \notin (X \cup Y) \ \wedge\ r \Downarrow (X \cup Y) = s)$
$\Rightarrow \{$ Induction hypothesis, $\sigma(s) \subseteq Y \Rightarrow \sigma(t) \subseteq Y$ $\}$
$\exists(t : at = s : a \in Y \ \wedge\ (r \Downarrow Y) = t) \vee$

$(a \notin Y \ \wedge \ r \Downarrow Y = s)$

$= \{$ Properties $\Uparrow, \Downarrow$ $\}$

$\exists(t : at = s : a \in Y \ \wedge \ (a \Downarrow Y)(r \Downarrow Y) = s) \vee$

$(a \notin Y \ \wedge \ (a \Downarrow Y)(r \Downarrow Y) = s)$

$\Rightarrow \{$ Calculus $\}$

$ar \Downarrow Y = s$

$a.br \Downarrow (X \cup Y) = s$

$= \{$ Definition $\Downarrow$ $\}$

$(a.b \Downarrow (X \cup Y))(r \Downarrow (X \cup Y)) = s$

$= \{$ Cases $\}$

$((a.b \in (X \cup Y) \ \vee \ (a \in (X \cup Y) \ \wedge \ b \in (X \cup Y))) \ \wedge \ a.b(r \Downarrow (X \cup Y)) = s) \vee$

$(a.b \notin (X \cup Y) \ \wedge \ a \in (X \cup Y) \ \wedge \ b \notin (X \cup Y) \ \wedge \ a(r \Downarrow (X \cup Y)) = s) \vee$

$(a.b \notin (X \cup Y) \ \wedge \ a \notin (X \cup Y) \ \wedge \ b \in (X \cup Y) \ \wedge \ b(r \Downarrow (X \cup Y)) = s) \vee$

$(a.b \notin (X \cup Y) \ \wedge \ a \notin (X \cup Y) \ \wedge \ b \notin (X \cup Y) \ \wedge \ r \Downarrow (X \cup Y) = s)$

$\Rightarrow \{ \sigma(s) \subseteq Y$,Calculus $\}$

$\exists(t : a.bt = s : a.b \in Y \ \wedge \ r \Downarrow (X \cup Y) = t) \vee$

$\exists(t : at = s : a.b \notin Y \ \wedge \ a \in Y \ \wedge \ b \notin Y \ \wedge \ r \Downarrow (X \cup Y) = t) \vee$

$\exists(t : bt = s : a.b \notin Y \ \wedge \ a \notin Y \ \wedge \ b \in Y \ \wedge \ r \Downarrow (X \cup Y) = t) \vee$

$(a.b \notin Y \ \wedge \ a \notin Y \ \wedge \ b \notin Y \ \wedge \ r \Downarrow (X \cup Y) = s)$

$\Rightarrow \{$ Ind.hyp. $\sigma(s) \subseteq Y \Rightarrow \sigma(t) \subseteq Y$ $\}$

$\exists(t : a.bt = s : a.b \in Y \ \wedge \ r \Downarrow Y = t) \vee$

$\exists(t : at = s : a.b \notin Y \ \wedge \ a \in Y \ \wedge \ b \notin Y \ \wedge \ r \Downarrow Y = t) \vee$

$\exists(t : bt = s : a.b \notin Y \ \wedge \ a \notin Y \ \wedge \ b \in Y \ \wedge \ r \Downarrow Y = t) \vee$

$(a.b \notin Y \ \wedge \ a \notin Y \ \wedge \ b \notin Y \ \wedge \ r \Downarrow Y = s)$

$= \{$ Calculus $\}$

$\exists(t : a.bt = s : a.b \in Y \ \wedge \ a.br \Downarrow Y = a.bt) \vee$

$\exists(t : at = s : a.b \notin Y \ \wedge \ a \in Y \ \wedge \ b \notin Y \ \wedge \ a.br \Downarrow Y = at) \vee$

$\exists(t : bt = s : a.b \notin Y \ \wedge \ a \notin Y \ \wedge \ b \in Y \ \wedge \ a.br \Downarrow Y = bt) \vee$

$(a.b \notin Y \ \wedge \ a \notin Y \ \wedge \ b \notin Y \ \wedge \ a.br \Downarrow Y = s)$

$\Rightarrow \{$ Calculus $\}$

$a.br \Downarrow Y = s$

end of proof

(3.9) $\quad (t \Downarrow X = t) \ = \ (t \Uparrow X = \epsilon)$

Proof, by induction on the length of t,
Base case, $t = \epsilon$,

$$\begin{array}{ll} & \epsilon \Uparrow X = \epsilon \\ = & \{ \text{ Definition } \Uparrow \} \\ & true \\ = & \{ \text{ Definition } \Downarrow \} \\ & \epsilon \Downarrow X = \epsilon \end{array}$$

Induction step, two cases:

$$\begin{array}{ll} & (at) \Uparrow X = \epsilon \\ = & \{ \text{ Definition } \Uparrow \} \\ & a \in X \ \wedge \ t \Uparrow X = \epsilon \\ = & \{ \text{ Induction step } \} \\ & a \in X \ \wedge \ t \Downarrow X = t \\ = & \{ \text{ Definition } \Downarrow \} \\ & (at) \Downarrow X = at \end{array} \qquad \begin{array}{ll} & (a.b\ t) \Uparrow X = \epsilon \\ = & \{ \text{ Definition } \Uparrow \} \\ & (a.b \in X \ \vee \ (a \in X \ \wedge \ b \in X)) \ \wedge \ t \Uparrow X = \epsilon \\ = & \{ \text{ Induction step } \} \\ & (a.b \in X \ \vee \ (a \in X \ \wedge \ b \in X)) \ \wedge \ t \Downarrow X = t \\ = & \{ \text{ Definition } \Downarrow \} \\ & (a.b\ t) \Downarrow X = a.b\ t \end{array}$$

end of proof

(3.10) $\quad (s \Downarrow X) \Uparrow X = \epsilon$

Proof

$$\begin{array}{ll} & (s \Downarrow X) \Uparrow X = \epsilon \\ = & \{ \text{ Theorem 3.9 } \} \\ & (s \Downarrow X) \Downarrow X = s \Downarrow X \\ = & \{ \text{ Corollary of thm. 3.5 } \} \\ & true \end{array}$$

end of proof

(3.11) $\quad (t \Uparrow \sigma(s) \ = \ \epsilon \ \wedge \ t \Downarrow \sigma(s) \ = \ s) \ = \ (s = t)$

Proof
By induction on the length of trace t.
Base case:

$$
\begin{aligned}
& \epsilon \Uparrow \sigma(s) \ = \ \epsilon \ \wedge \ \epsilon \Downarrow \sigma(s) \ = \ s \\
= \{ & \text{ Definitions } \Downarrow, \Uparrow \ \} \\
& \epsilon \ = \ \epsilon \ \wedge \ \epsilon \ = \ s \\
= \{ & \ \ \} \\
& s \ = \ \epsilon
\end{aligned}
$$

Induction step, $a \in$ **Atoms** :

$$
\begin{aligned}
& (at) \Downarrow \sigma(ys) \ = \ ys \ \wedge \ (at) \Uparrow \sigma(ys) \ = \ \epsilon \\
= \{ & \text{ Definitions } \Downarrow, \Uparrow \ \} \\
& (a \in \sigma(ys) \ \wedge \ a(t \Downarrow \sigma(ys)) \ = \ ys \ \wedge \ t \Uparrow \sigma(ys) \ = \ \epsilon) \vee \\
& (a \notin \sigma(ys) \ \wedge \ t \Downarrow \sigma(ys) \ = \ ys \ \wedge \ a(t \Uparrow \sigma(ys)) \ = \ \epsilon) \\
= \{ & \ a(t \Uparrow \sigma(ys)) \neq \epsilon \ \} \\
& (a \in \sigma(ys) \ \wedge \ a(t \Downarrow \sigma(ys)) \ = \ ys \ \wedge \ t \Uparrow \sigma(ys) \ = \ \epsilon) \\
= \{ & \text{ Case analysis,} a = y \Rightarrow a \in \sigma(ys) \ \} \\
& (a \in \sigma(t) \ \wedge \ y = a \ \wedge \ t \Downarrow \sigma(ys) \ = \ s \ \wedge \ t \Uparrow \sigma(ys) \ = \ \epsilon) \vee \\
& (a \notin \sigma(t) \ \wedge \ y = a \ \wedge \ t \Downarrow \sigma(ys) \ = \ s \ \wedge \ t \Uparrow \sigma(ys) \ = \ \epsilon) \\
= \{ & \ a \in \sigma(t) \Rightarrow a \in \sigma(t \Downarrow \sigma(as)) \\
& \text{hence } a \in \sigma(t) \ \wedge \ t \Downarrow \sigma(as) \ = \ s \Rightarrow a \in \sigma(s) \\
& a \notin \sigma(t) \ \wedge \ t \Downarrow \sigma(as) = s \Rightarrow (t \Downarrow \sigma(as) \ = \ t \Downarrow \sigma(s) \ \wedge \ t \Uparrow \sigma(as) \ = \ t \Uparrow \sigma(s)) \ \} \\
& y = a \ \wedge \ t \Downarrow \sigma(s) \ = \ s \ \wedge \ t \Uparrow \sigma(s) \ = \ \epsilon \\
= \{ & \text{ Induction hypothesis. } \} \\
& t \ = \ s \wedge a \ = \ y \\
= \{ & \ \ \} \\
& at \ = \ ys
\end{aligned}
$$

Induction step, $x \in$ **Pairs**:

$$
\begin{aligned}
& (xt) \Downarrow \sigma(ys) \ = \ ys \ \wedge \ (xt) \Uparrow \sigma(ys) \ = \ \epsilon \ \wedge \ x \in \textbf{Pairs} \\
= \{ & \text{ Definitions } \Downarrow, \Uparrow \ \} \\
& x = a.b \wedge \\
& (x \notin \sigma(ys) \ \wedge \ a \notin \sigma(ys) \ \wedge \ b \notin \sigma(ys) \ \wedge \ t \Downarrow \sigma(ys) = ys \ \wedge \ x(t \Uparrow \sigma(ys)) = \epsilon) \vee \\
& (x \notin \sigma(ys) \ \wedge \ a \in \sigma(ys) \ \wedge \ b \notin \sigma(ys) \ \wedge \ a(t \Downarrow \sigma(ys)) = ys \ \wedge \ b(t \Uparrow \sigma(ys)) = \epsilon) \vee \\
& ((x \in \sigma(ys) \ \vee \ (a \in \sigma(ys) \ \wedge \ b \in \sigma(ys))) \ \wedge \ x(t \Downarrow \sigma(ys)) = ys \ \wedge \ t \Uparrow \sigma(ys) = \epsilon) \\
= \{ & \ x(t \Uparrow \sigma(ys)) \neq \epsilon \ \} \\
& x = a.b \wedge \\
& ((x \in \sigma(ys) \ \vee \ (a \in \sigma(ys) \ \wedge \ b \in \sigma(ys))) \ \wedge \ x(t \Downarrow \sigma(ys)) = ys \ \wedge \ t \Uparrow \sigma(ys) = \epsilon)
\end{aligned}
$$

$= \{$ Case analysis,$x = y \Rightarrow x \in \sigma(ys)$ $\}$
$x = a.b \wedge$
$(x \in \sigma(t) \ \wedge \ x = y \ \wedge \ t \Downarrow \sigma(ys) = s \ \wedge \ t \Uparrow \sigma(ys) = \epsilon) \vee$
$(x \notin \sigma(t) \ \wedge \ x = y \ \wedge \ t \Downarrow \sigma(ys) = s \ \wedge \ t \Uparrow \sigma(ys) = \epsilon)$
$= \{$ $y \in \sigma(t) \Rightarrow y \in \sigma(t \Downarrow \sigma(ys))$
hence $y \in \sigma(t) \ \wedge \ t \Downarrow \sigma(ys) \ = \ s \Rightarrow y \in \sigma(s)$
$y \notin \sigma(t) \Rightarrow (t \Downarrow \sigma(ys) \ = \ t \Downarrow \sigma(s) \ \wedge \ t \Uparrow \sigma(ys) \ = \ t \Uparrow \sigma(s)$ $\}$
$x = a.b \ \wedge \ y = x \ \wedge \ t \Downarrow \sigma(s) = s \ \wedge \ t \Uparrow \sigma(s) = \epsilon$
$= \{$ Induction hypothesis. $\}$
$t \ = \ s \wedge x \ = \ y$
$= \{ \ \}$
$xt \ = \ ys$

end of proof

$$(3.12) \qquad \alpha(X) \cap \alpha(Y) = \emptyset \ \Rightarrow \ (t \Uparrow X) \Uparrow Y \ = \ t \Uparrow (X \cup Y)$$

Proof, by induction on the length of t.

$((at) \Uparrow X) \Uparrow Y \ = \ a((t \Uparrow X) \Uparrow Y)$
$= \{$ Definition $\Uparrow$ $\}$
$a \notin X \ \wedge \ a \notin Y$
$= \{$ Calculus $\}$
$a \notin (X \cup Y)$
$= \{$ Definition $\Uparrow$ $\}$
$(at) \Uparrow (X \cup Y) \ = \ a(t \Uparrow (X \cup Y))$

$((at) \Uparrow X) \Uparrow Y \ = \ (t \Uparrow X) \Uparrow Y$
$= \{$ Definition $\Uparrow$ $\}$
$a \in X \ \vee \ a \in Y$
$= \{$ Calculus $\}$
$a \in (X \cup Y)$
$= \{$ Definition $\Uparrow$ $\}$
$(at) \Uparrow (X \cup Y) \ = \ t \Uparrow (X \cup Y)$

$$
\begin{array}{l}
\quad x = a.b \ \wedge \ ((xt) \Uparrow X) \Uparrow Y \ = \ x((t \Uparrow X) \Uparrow Y) \\
= \{ \text{ “monotonicity” of } \Uparrow \} \\
\quad x = a.b \ \wedge \ (xt) \Uparrow X \ = \ x(t \Uparrow X) \ \wedge \ (x(t \Uparrow X)) \Uparrow Y \ = \ x((t \Uparrow X) \Uparrow Y) \\
= \{ \text{ Definition } \Uparrow \} \\
\quad x = a.b \ \wedge \ x \notin X \ \wedge \ a \notin X \ \wedge \ b \notin X \ \wedge \ x \notin Y \ \wedge \ a \notin Y \ \wedge \ b \notin Y \\
= \{ \text{ Calculus } \} \\
\quad x = a.b \ \wedge \ x \notin (X \cup Y) \ \wedge \ a \notin (X \cup Y) \ \wedge \ b \notin (X \cup Y) \\
= \{ \text{ Definition } \Uparrow \} \\
\quad x = a.b \ \wedge \ (xt) \Uparrow (X \cup Y) \ = \ x(t \Uparrow (X \cup Y))
\end{array}
$$

$$
\begin{array}{l}
\quad x = a.b \ \wedge \ ((xt) \Uparrow X) \Uparrow Y \ = \ a((t \Uparrow X) \Uparrow Y) \\
= \{ \text{ “monotonicity” of } \Uparrow \} \\
\quad x = a.b \ \wedge \ (\ (xt) \Uparrow X \ = \ x(t \Uparrow X) \ \wedge \ (x(t \Uparrow X)) \Uparrow Y \ = \ a((t \Uparrow X) \Uparrow Y)) \\
\qquad\qquad \vee (\ (xt) \Uparrow X \ = \ a(t \Uparrow X) \ \wedge \ (a(t \Uparrow X)) \Uparrow Y \ = \ a((t \Uparrow X) \Uparrow Y)) \\
= \{ \text{ Definition } \Uparrow \} \\
\quad x = a.b \ \wedge \ (\ (\ x \notin X \ \wedge \ a \notin X \ \wedge \ b \notin X \ \wedge \ x \notin Y \ \wedge \ a \notin Y \ \wedge \ b \in Y \ ) \\
\qquad\qquad \vee (\ x \notin X \ \wedge \ a \notin X \ \wedge \ b \in X \ \wedge \ a \notin Y) \ ) \\
= \{ \text{ Calculus } \} \\
\quad x = a.b \ \wedge \ x \notin X \ \wedge \ a \notin X \ \wedge \ a \notin Y \\
\quad\ \ \wedge ((b \notin X \ \wedge \ x \notin Y \ \wedge \ b \in Y) \ \vee \ b \in X) \\
= \{ \ \alpha(X) \cap \alpha(Y) = \emptyset \ \} \\
\quad x = a.b \ \wedge \ x \notin (X \cup Y) \ \wedge \ a \notin (X \cup Y) \ \wedge \ b \in (X \cup Y) \\
= \{ \text{ Definition } \Uparrow \} \\
\quad x = a.b \ \wedge \ (xt) \Uparrow (X \cup Y) \ = \ a(t \Uparrow (X \cup Y))
\end{array}
$$

$$
\begin{array}{l}
\quad x = a.b \ \wedge \ ((xt) \Uparrow X) \Uparrow Y \ = \ (t \Uparrow X) \Uparrow Y \\
= \{ \text{ “monotonicity” of } \Uparrow \} \\
\quad x = a.b \ \wedge \ (\ (\ ((xt) \Uparrow X) \ = \ t \Uparrow X \ \wedge \ (t \Uparrow X) \Uparrow Y \ = \ (t \Uparrow X) \Uparrow Y \ ) \\
\qquad\qquad\quad \vee (\ ((xt) \Uparrow X) \ = \ x(t \Uparrow X) \ \wedge \ (x(t \Uparrow X)) \Uparrow Y \ = \ (t \Uparrow X) \Uparrow Y \ ) \\
\qquad\qquad\quad \vee (\ ((xt) \Uparrow X) \ = \ a(t \Uparrow X) \ \wedge \ (a(t \Uparrow X)) \Uparrow Y \ = \ (t \Uparrow X) \Uparrow Y \ ) \\
\qquad\qquad\quad \vee (\ ((xt) \Uparrow X) \ = \ b(t \Uparrow X) \ \wedge \ (b(t \Uparrow X)) \Uparrow Y \ = \ (t \Uparrow X) \Uparrow Y \ ) \ ) \\
= \{ \text{ Definition } \Uparrow \} \\
\quad x = a.b \ \wedge \ (\ (\ x \in X \ \vee \ (a \in X \ \wedge \ b \in X)) \\
\qquad\qquad\quad \vee (\ x \notin X \ \wedge \ a \notin X \ \wedge \ b \notin X \ \wedge (x \in Y \ \vee \ (a \in Y \ \wedge \ b \in Y))) \\
\qquad\qquad\quad \vee (\ x \notin X \ \wedge \ a \notin X \ \wedge \ b \in X \ \wedge \ a \in Y)
\end{array}
$$

$$
\begin{aligned}
& \qquad\qquad \vee\ (\ x \notin X\ \wedge\ a \in X\ \wedge\ b \notin X\ \wedge\ b \in Y\ )\ ) \\
= & \ \{\ \text{Calculus, using } \alpha(X) \cap \alpha(Y) = \emptyset\ \} \\
& \quad x = a.b\ \wedge\ (\ x \in X\ \vee\ (a \in X\ \wedge\ b \in X) \\
& \qquad\qquad \vee\ \ x \in Y\ \vee\ (a \in Y\ \wedge\ b \in Y) \\
& \qquad\qquad \vee\ (\ b \in X\ \wedge\ a \in Y) \\
& \qquad\qquad \vee\ (\ a \in X\ \wedge\ b \in Y)\ ) \\
= & \ \{\ \text{Calculus, using } \alpha(X) \cap \alpha(Y) = \emptyset\ \} \\
& \quad x = a.b\ \wedge\ (x \in (X \cup Y)\ \vee\ (a \in (X \cup Y)\ \wedge\ b \in (X \cup Y))) \\
= & \ \{\ \text{Definition } \Uparrow\ \} \\
& \quad x = a.b\ \wedge\ (xt) \Uparrow (X \cup Y)\ \ =\ \ t \Uparrow (X \cup Y)
\end{aligned}
$$

The theorem then follows by induction (the base case is trivial).

end of proof

Note $(\{a.b\} \Downarrow \{a,b\}) \Downarrow \{a.b\} = \{a.b\} \neq \{\} = \{a.b\} \Downarrow \{\}$; hence, there is no equivalent theorem for projection.

(3.13) $(\alpha(X) \cap \alpha(Y) = \emptyset) \Rightarrow (s \Downarrow X = (s \Uparrow Y) \Downarrow X)$

Proof

By induction on the length of trace s.

Base case:

$$
\begin{aligned}
& \quad s = \epsilon \\
\Rightarrow & \ \{\ \text{Definition } \Downarrow, \Uparrow\ \} \\
& \quad s \Downarrow X = \epsilon\ \wedge\ s \Uparrow Y = \epsilon \\
= & \ \{\ \text{Calculus}\ \} \\
& \quad s \Downarrow X = (s \Uparrow Y) \Downarrow X
\end{aligned}
$$

Induction step:

$$
\begin{aligned}
& \quad \exists(a, b, t : a, b \in \mathbf{Atoms}, t \in \mathbf{Traces} : s = at\ \vee\ s = a.bt) \\
= & \ \{\ \text{Cases}\ \} \\
& \quad \exists(a, b, t :: (s = at\ \wedge\ a \in X)\ \vee\ (s = at\ \wedge\ a \notin X) \vee \\
& \quad (s = a.bt\ \wedge\ (a.b \in X\ \vee\ (a \in X\ \wedge\ b \in X))) \vee \\
& \quad (s = a.bt\ \wedge\ a.b \notin X\ \wedge\ a \in X\ \wedge\ b \notin X) \vee \\
& \quad (s = a.bt\ \wedge\ a.b \notin X\ \wedge\ a \notin X\ \wedge\ b \in X) \vee
\end{aligned}
$$

$$
\begin{array}{l}
\quad (s = a.bt \ \wedge \ a.b \notin X \ \wedge \ a \notin X \ \wedge \ b \notin X)) \\
\Rightarrow \{ \ \alpha(X) \cap \alpha(Y) = \emptyset \ \} \\
\quad \exists (a, b, t :: \\
\quad (s = at \ \wedge \ a \in X \ \wedge \ a \notin Y) \vee \\
\quad (s = at \ \wedge \ a \notin X) \vee \\
\quad (s = a.bt \ \wedge \ (a.b \in X \ \vee \ (a \in X \ \wedge \ b \in X)) \ \wedge \ a.b \notin Y \ \wedge \ a \notin Y \ \wedge \ b \notin Y) \vee \\
\quad (s = a.bt \ \wedge \ a.b \notin X \ \wedge \ a \in X \ \wedge \ b \notin X \ \wedge \ a.b \notin Y \ \wedge \ a \notin Y) \vee \\
\quad (s = a.bt \ \wedge \ a.b \notin X \ \wedge \ a \notin X \ \wedge \ b \in X \ \wedge \ a.b \notin Y \ \wedge \ b \notin Y) \vee \\
\quad (s = a.bt \ \wedge \ a.b \notin X \ \wedge \ a \notin X \ \wedge \ b \notin X)) \\
= \{ \ \text{Cases} \ \} \\
\quad \exists (a, b, t :: \\
\quad (s = at \ \wedge \ a \in X \ \wedge \ a \notin Y) \vee \\
\quad (s = at \ \wedge \ a \notin X \ \wedge \ a \in Y) \vee \\
\quad (s = at \ \wedge \ a \notin X \ \wedge \ a \notin Y) \vee \\
\quad (s = a.bt \ \wedge \ (a.b \in X \ \vee \ (a \in X \ \wedge \ b \in X)) \ \wedge \ a.b \notin Y \ \wedge \ a \notin Y \ \wedge \ b \notin Y) \vee \\
\quad (s = a.bt \ \wedge \ a.b \notin X \ \wedge \ a \in X \ \wedge \ b \notin X \ \wedge \ a.b \notin Y \ \wedge \ a \notin Y \ \wedge \ b \in Y) \vee \\
\quad (s = a.bt \ \wedge \ a.b \notin X \ \wedge \ a \in X \ \wedge \ b \notin X \ \wedge \ a.b \notin Y \ \wedge \ a \notin Y \ \wedge \ b \notin Y) \vee \\
\quad (s = a.bt \ \wedge \ a.b \notin X \ \wedge \ a \notin X \ \wedge \ b \in X \ \wedge \ a.b \notin Y \ \wedge \ a \in Y \ \wedge \ b \notin Y) \vee \\
\quad (s = a.bt \ \wedge \ a.b \notin X \ \wedge \ a \notin X \ \wedge \ b \in X \ \wedge \ a.b \notin Y \ \wedge \ a \notin Y \ \wedge \ b \notin Y) \vee \\
\quad (s = a.bt \ \wedge \ a.b \notin X \ \wedge \ a \notin X \ \wedge \ b \notin X \ \wedge \ (a.b \in Y \ \vee \ (a \in Y \ \wedge \ b \in Y))) \vee \\
\quad (s = a.bt \ \wedge \ a.b \notin X \ \wedge \ a \notin X \ \wedge \ b \notin X \ \wedge \ a.b \notin Y \ \vee \ a \in Y \ \wedge \ b \notin Y) \vee \\
\quad (s = a.bt \ \wedge \ a.b \notin X \ \wedge \ a \notin X \ \wedge \ b \notin X \ \wedge \ a.b \notin Y \ \vee \ a \notin Y \ \wedge \ b \in Y) \vee \\
\quad (s = a.bt \ \wedge \ a.b \notin X \ \wedge \ a \notin X \ \wedge \ b \notin X \ \wedge \ a.b \notin Y \ \vee \ a \notin Y \ \wedge \ b \notin Y)) \\
= \{ \ \text{Definition} \ \Downarrow, \Uparrow \ \} \\
\quad \exists (a, b, t :: \\
\quad (s = at \ \wedge \ a \in X \ \wedge \ a \notin Y \ \wedge \ s \Downarrow X = a(t \Downarrow X) \ \wedge \ s \Uparrow Y = a(t \Uparrow Y)) \vee \\
\quad (s = at \ \wedge \ a \notin X \ \wedge \ a \in Y \ \wedge \ s \Downarrow X = t \Downarrow X \ \wedge \ s \Uparrow Y = t \Uparrow Y) \vee \\
\quad (s = at \ \wedge \ a \notin X \ \wedge \ a \notin Y \ \wedge \ s \Downarrow X = t \Downarrow X \ \wedge \ s \Uparrow Y = a(t \Uparrow Y)) \vee \\
\quad (s = a.bt \ \wedge \ (a.b \in X \ \vee \ (a \in X \ \wedge \ b \in X)) \ \wedge \ a.b \notin Y \ \wedge \ a \notin Y \ \wedge \ b \notin Y \\
\quad \wedge \, s \Downarrow X = a.b(t \Downarrow X) \ \wedge \ s \Uparrow Y = a.b(t \Uparrow Y)) \vee \\
\quad (s = a.bt \ \wedge \ a.b \notin X \ \wedge \ a \in X \ \wedge \ b \notin X \ \wedge \ a.b \notin Y \ \wedge \ a \notin Y \ \wedge \ b \in Y \\
\quad \wedge \, s \Downarrow X = a(t \Downarrow X) \ \wedge \ s \Uparrow Y = a(t \Uparrow Y)) \vee \\
\quad (s = a.bt \ \wedge \ a.b \notin X \ \wedge \ a \in X \ \wedge \ b \notin X \ \wedge \ a.b \notin Y \ \wedge \ a \notin Y \ \wedge \ b \notin Y \\
\quad \wedge \, s \Downarrow X = a(t \Downarrow X) \ \wedge \ s \Uparrow Y = a.b(t \Uparrow Y)) \vee \\
\quad (s = a.bt \ \wedge \ a.b \notin X \ \wedge \ a \notin X \ \wedge \ b \in X \ \wedge \ a.b \notin Y \ \wedge \ a \in Y \ \wedge \ b \notin Y \\
\quad \wedge \, s \Downarrow X = b(t \Downarrow X) \ \wedge \ s \Uparrow Y = b(t \Uparrow Y)) \vee
\end{array}
$$

$$
\begin{aligned}
& (s = a.bt \ \wedge \ a.b \notin X \ \wedge \ a \notin X \ \wedge \ b \in X \ \wedge \ a.b \notin Y \ \wedge \ a \notin Y \ \wedge \ b \notin Y \\
& \ \wedge s \Downarrow X = b(t \Downarrow X) \ \wedge \ s \Uparrow Y = a.b(t \Uparrow Y)) \vee \\
& (s = a.bt \ \wedge \ a.b \notin X \ \wedge \ a \notin X \ \wedge \ b \notin X \ \wedge \ (a.b \in Y \ \vee \ (a \in Y \ \wedge \ b \in Y)) \\
& \ \wedge s \Downarrow X = t \Downarrow X \ \wedge \ s \Uparrow Y = a.b(t \Uparrow Y)) \vee \\
& (s = a.bt \ \wedge \ a.b \notin X \ \wedge \ a \notin X \ \wedge \ b \notin X \ \wedge \ a.b \notin Y \ \vee \ a \in Y \ \wedge \ b \notin Y \\
& \ \wedge s \Downarrow X = t \Downarrow X \ \wedge \ s \Uparrow Y = b(t \Uparrow Y)) \vee \\
& (s = a.bt \ \wedge \ a.b \notin X \ \wedge \ a \notin X \ \wedge \ b \notin X \ \wedge \ a.b \notin Y \ \vee \ a \notin Y \ \wedge \ b \in Y \\
& \ \wedge s \Downarrow X = t \Downarrow X \ \wedge \ s \Uparrow Y = a(t \Uparrow Y)) \vee \\
& (s = a.bt \ \wedge \ a.b \notin X \ \wedge \ a \notin X \ \wedge \ b \notin X \ \wedge \ a.b \notin Y \ \vee \ a \notin Y \ \wedge \ b \notin Y \\
& \ \wedge s \Downarrow X = t \Downarrow X \ \wedge \ s \Uparrow Y = t \Uparrow Y)) \\
= & \ \{ \text{ Definition } \Downarrow, \Uparrow \ \} \\
& \exists (a, b, t :: \\
& (s = at \ \wedge \ a \in X \ \wedge \ a \notin Y \\
& \ \wedge s \Downarrow X = a(t \Downarrow X) \ \wedge \ (s \Uparrow Y) \Downarrow X = a((t \Uparrow Y) \Downarrow X)) \vee \\
& (s = at \ \wedge \ a \notin X \ \wedge \ a \in Y \ \wedge \ s \Downarrow X = t \Downarrow X \ \wedge \ (s \Uparrow Y) \Downarrow X = (t \Uparrow Y) \Downarrow X) \vee \\
& (s = at \ \wedge \ a \notin X \ \wedge \ a \notin Y \ \wedge \ s \Downarrow X = t \Downarrow X \ \wedge \ (s \Uparrow Y) \Downarrow X = (t \Uparrow Y) \Downarrow X) \vee \\
& (s = a.bt \ \wedge \ (a.b \in X \ \vee \ (a \in X \ \wedge \ b \in X)) \ \wedge \ a.b \notin Y \ \wedge \ a \notin Y \ \wedge \ b \notin Y \\
& \ \wedge s \Downarrow X = a.b(t \Downarrow X) \ \wedge \ (s \Uparrow Y) \Downarrow X = a.b((t \Uparrow Y) \Downarrow X)) \vee \\
& (s = a.bt \ \wedge \ a.b \notin X \ \wedge \ a \in X \ \wedge \ b \notin X \ \wedge \ a.b \notin Y \ \wedge \ a \notin Y \ \wedge \ b \in Y \\
& \ \wedge s \Downarrow X = a(t \Downarrow X) \ \wedge \ (s \Uparrow Y) \Downarrow X = a((t \Uparrow Y) \Downarrow X)) \vee \\
& (s = a.bt \ \wedge \ a.b \notin X \ \wedge \ a \in X \ \wedge \ b \notin X \ \wedge \ a.b \notin Y \ \wedge \ a \notin Y \ \wedge \ b \notin Y \\
& \ \wedge s \Downarrow X = a(t \Downarrow X) \ \wedge \ (s \Uparrow Y) \Downarrow X = a((t \Uparrow Y) \Downarrow X)) \vee \\
& (s = a.bt \ \wedge \ a.b \notin X \ \wedge \ a \notin X \ \wedge \ b \in X \ \wedge \ a.b \notin Y \ \wedge \ a \in Y \ \wedge \ b \notin Y \\
& \ \wedge s \Downarrow X = b(t \Downarrow X) \ \wedge \ (s \Uparrow Y) \Downarrow X = b((t \Uparrow Y) \Downarrow X)) \vee \\
& (s = a.bt \ \wedge \ a.b \notin X \ \wedge \ a \notin X \ \wedge \ b \in X \ \wedge \ a.b \notin Y \ \wedge \ a \notin Y \ \wedge \ b \notin Y \\
& \ \wedge s \Downarrow X = b(t \Downarrow X) \ \wedge \ (s \Uparrow Y) \Downarrow X = b((t \Uparrow Y) \Downarrow X)) \vee \\
& (s = a.bt \ \wedge \ a.b \notin X \ \wedge \ a \notin X \ \wedge \ b \notin X \ \wedge \ (a.b \in Y \ \vee \ (a \in Y \ \wedge \ b \in Y)) \\
& \ \wedge s \Downarrow X = t \Downarrow X \ \wedge \ (s \Uparrow Y) \Downarrow X = (t \Uparrow Y) \Downarrow X) \vee \\
& (s = a.bt \ \wedge \ a.b \notin X \ \wedge \ a \notin X \ \wedge \ b \notin X \ \wedge \ a.b \notin Y \ \vee \ a \in Y \ \wedge \ b \notin Y \\
& \ \wedge s \Downarrow X = t \Downarrow X \ \wedge \ (s \Uparrow Y) \Downarrow X = (t \Uparrow Y) \Downarrow X) \vee \\
& (s = a.bt \ \wedge \ a.b \notin X \ \wedge \ a \notin X \ \wedge \ b \notin X \ \wedge \ a.b \notin Y \ \vee \ a \notin Y \ \wedge \ b \in Y \\
& \ \wedge s \Downarrow X = t \Downarrow X \ \wedge \ (s \Uparrow Y) \Downarrow X = (t \Uparrow Y) \Downarrow X) \vee \\
& (s = a.bt \ \wedge \ a.b \notin X \ \wedge \ a \notin X \ \wedge \ b \notin X \ \wedge \ a.b \notin Y \ \vee \ a \notin Y \ \wedge \ b \notin Y \\
& \ \wedge s \Downarrow X = t \Downarrow X \ \wedge \ (s \Uparrow Y) \Downarrow X = (t \Uparrow Y) \Downarrow X)) \\
\Rightarrow & \ \{ \text{ Ind. hyp. : } t \Downarrow X = (t \Uparrow Y) \Downarrow X \ \} \\
& s \Downarrow X = (s \Uparrow Y) \Downarrow X
\end{aligned}
$$

end of proof

$$(3.14) \qquad X \subseteq \mathbf{Pairs} \Rightarrow ((r \Uparrow X) \Downarrow Y \ = \ (r \Uparrow X) \Downarrow (X \cup Y))$$

Proof

By induction on the length of r.
Base case $r = \epsilon$, trivial.
Induction step

$$\begin{array}{ll}
& (ar \Uparrow X) \Downarrow Y \\
= & \{ \text{ Definition } \Uparrow, X \subseteq \mathbf{Pairs} \} \\
& (a \Downarrow Y)((r \Uparrow X) \Downarrow Y) \\
= & \{ \text{ Definition } \Uparrow, X \subseteq \mathbf{Pairs}, \text{ ind. hyp. } \} \\
& ((a \Uparrow X) \Downarrow (X \cup Y))((r \Uparrow X) \Downarrow (X \cup Y)) \\
= & \{ \text{ properties } \Uparrow, \Downarrow \} \\
& (ar \Uparrow X) \Downarrow (X \cup Y)
\end{array}$$

$$\begin{array}{ll}
& (a.br \Uparrow X) \Downarrow Y = s \\
= & \{ \text{ Definition } \Uparrow, X \subseteq \mathbf{Pairs} \} \\
& (a.b \in X \ \wedge \ (r \Uparrow X) \Downarrow Y = s) \vee \\
& (a.b \notin X \ \wedge \ (a.b \Downarrow Y)((r \Uparrow X) \Downarrow Y) = s) \\
= & \{ \text{ Definition } \Uparrow, \Downarrow, X \subseteq \mathbf{Pairs}, \text{ ind. hyp. } \} \\
& (a.b \in X \ \wedge \ ((a.b \Uparrow X) \Downarrow (X \cup Y))((r \Uparrow X) \Downarrow (X \cup Y)) = s) \vee \\
& (a.b \notin X \ \wedge \ ((a.b \Uparrow X) \Downarrow (X \cup Y))((r \Uparrow X) \Downarrow (X \cup Y)) = s) \\
= & \{ \text{ properties } \Uparrow, \Downarrow \} \\
& (a.br \Uparrow X) \Downarrow (X \cup Y)
\end{array}$$

end of proof

$$(3.15) \qquad Y \subseteq \mathbf{Pairs} \Rightarrow ((t \Uparrow X = \epsilon) \ = \ ((t \Uparrow Y) \Uparrow (X - Y) = \epsilon))$$

Proof

By induction on the length of t.
Base case $t = \epsilon$, trivial.
Induction step

$$
\begin{array}{l}
\quad at \Uparrow X = \epsilon \\
= \{ \text{ Definition } \Uparrow \} \\
\quad a \Uparrow X = \epsilon \ \wedge \ t \Uparrow X = \epsilon \\
= \{ \text{ Definition } \Uparrow \text{, induction hypothesis } \} \\
\quad a \in X \ \wedge \ (t \Uparrow Y) \Uparrow (X - Y) = \epsilon \\
= \{ \ Y \subseteq \mathbf{Pairs} \ \} \\
\quad (at \Uparrow Y) \Uparrow (X - Y) = \epsilon
\end{array}
$$

$$
\begin{array}{l}
\quad a.bt \Uparrow X = \epsilon \\
= \{ \text{ Definition } \Uparrow \} \\
\quad a.b \Uparrow X = \epsilon \ \wedge \ t \Uparrow X = \epsilon \\
= \{ \text{ Definition } \Uparrow \text{, induction hypothesis } \} \\
\quad (a.b \in X \ \vee \ (a \in X \ \wedge \ b \in X)) \ \wedge \ (t \Uparrow Y) \Uparrow (X - Y) = \epsilon \\
= \{ \ Y \subseteq \mathbf{Pairs}\text{, definition } \Uparrow \} \\
\quad (a.bt \Uparrow Y) \Uparrow (X - Y) = \epsilon
\end{array}
$$

end of proof

(3.16) $\quad X \subseteq \mathbf{Pairs} \ \wedge \ (Y \Uparrow X) \Downarrow \alpha(X) = \emptyset \Rightarrow$
$\quad (r \Downarrow Y) \Uparrow X = (r \Uparrow X) \Downarrow (Y \Uparrow X)$

Proof, by induction on $|r|$.
Base case, $r = \epsilon$, trivial.
Induction step, two cases.

$$
\begin{array}{l}
\quad (ar \Downarrow Y) \Uparrow X = t \\
= \{ \text{ Cases, } X \subseteq \mathbf{Pairs} \ \} \\
\quad (a \in Y \ \wedge \ a((r \Downarrow Y) \Uparrow X) = t) \vee \\
\quad (a \notin Y \ \wedge \ (r \Downarrow Y) \Uparrow X = t) \\
= \{ \text{ Ind.hyp. } \} \\
\quad (a \in Y \ \wedge \ a((r \Uparrow X) \Downarrow (Y \Uparrow X)) = t) \vee \\
\quad (a \notin Y \ \wedge \ (r \Uparrow X) \Downarrow (Y \Uparrow X) = t) \\
= \{ \text{ Calculus, } X \subseteq \mathbf{Pairs} \ \} \\
\quad (a \in Y \ \wedge \ (ar \Uparrow X) \Downarrow (Y \Uparrow X) = t) \vee \\
\quad (a \notin Y \ \wedge \ (ar \Uparrow X) \Downarrow (Y \Uparrow X) = t) \\
= \{ \text{ Cases } \} \\
\quad (ar \Uparrow X) \Downarrow (Y \Uparrow X) = t
\end{array}
$$

$(a.br \Downarrow Y) \Uparrow X = t$
$= \{$ Cases, $X \subseteq \mathbf{Pairs}$ $\}$
$((a.b \in Y \ \vee \ (a \in Y \ \wedge \ b \in Y)) \ \wedge \ a.b \in X \ \wedge \ (r \Downarrow Y) \Uparrow X = t) \vee$
$((a.b \in Y \ \vee \ (a \in Y \ \wedge \ b \in Y)) \ \wedge \ a.b \notin X \ \wedge \ a.b((r \Downarrow Y) \Uparrow X) = t) \vee$
$(a.b \notin Y \ \wedge \ a \in Y \ \wedge \ b \notin Y \ \wedge \ a((r \Downarrow Y) \Uparrow X) = t) \vee$
$(a.b \notin Y \ \wedge \ a \notin Y \ \wedge \ b \in Y \ \wedge \ b((r \Downarrow Y) \Uparrow X) = t) \vee$
$(a.b \notin Y \ \wedge \ a \notin Y \ \wedge \ b \notin Y \ \wedge \ (r \Downarrow Y) \Uparrow X = t)$
$= \{$ Ind. hyp., $((Y \Uparrow X) \Downarrow \alpha(X) = \emptyset \ \wedge \ X \subseteq \mathbf{Pairs} \ \wedge \ a \in Y) \Rightarrow a.b \notin X$ $\}$
$((a.b \in Y \ \vee \ (a \in Y \ \wedge \ b \in Y)) \ \wedge \ a.b \in X \ \wedge \ (r \Uparrow X) \Downarrow (Y \Uparrow X) = t) \vee$
$((a.b \in Y \ \vee \ (a \in Y \ \wedge \ b \in Y)) \ \wedge \ a.b \notin X \ \wedge \ a.b((r \Uparrow X) \Downarrow (Y \Uparrow X)) = t) \vee$
$(a.b \notin Y \ \wedge \ a \in Y \ \wedge \ b \notin Y \ \wedge \ a((r \Uparrow X) \Downarrow (Y \Uparrow X)) = t) \vee$
$(a.b \notin Y \ \wedge \ a \notin Y \ \wedge \ b \in Y \ \wedge \ b((r \Uparrow X) \Downarrow (Y \Uparrow X)) = t) \vee$
$(a.b \notin Y \ \wedge \ a \notin Y \ \wedge \ b \notin Y \ \wedge \ (r \Uparrow X) \Downarrow (Y \Uparrow X) = t)$
$= \{$ Calculus, $X \subseteq \mathbf{Pairs}$, $(Y \Uparrow X) \downarrow \alpha(X) = \emptyset$ $\}$
$(a.br \Uparrow X) \Downarrow (Y \Uparrow X) = t$

end of proof

Corollary
$\forall(r, s, X : X \subseteq \mathbf{Pairs} \ \wedge \ r \Downarrow \sigma(s) = s \ \wedge \ (s \Uparrow X) \Downarrow \alpha(X) = \epsilon :$
$(r \Uparrow X) \Downarrow \sigma(s \Uparrow X) = s \Uparrow X)$

Note
The following is *not* a theorem.
$r \Downarrow \sigma(s) = s \Rightarrow \forall(X :: (r \Uparrow X) \Downarrow \sigma(s \Uparrow X) = s \Uparrow X)$
Counterexample: $r := a.b, s := b, X := \{a.b\}$.
end of note

(3.17) $\quad X \subseteq \mathbf{Pairs} \Rightarrow$
$(\ \ ((r \Uparrow X) \Downarrow ((\sigma(s) \Uparrow X) \cup (Y \Uparrow X)) = s \Uparrow X)$
$\Rightarrow$
$\exists(t : t \Uparrow X = r \Uparrow X : t \Downarrow (\sigma(s) \cup Y) = s)$
$)$

Proof, by induction on the length of s.
Two base cases, $|s| = 0$ and $|s| = 1$, first $|s| = 0$,

$$
\begin{array}{ll}
 & (r \Uparrow X) \Downarrow ((\sigma(\epsilon) \Uparrow X) \cup Y \Uparrow X) = \epsilon \Uparrow X \\
= & \{ \ \sigma(\epsilon) = \emptyset, \text{ definition } \Uparrow \ \} \\
 & (r \Uparrow X) \Downarrow (Y \Uparrow X) = \epsilon \\
= & \{ \ \text{Theorem 3.14} \ \} \\
 & (r \Uparrow X) \Downarrow ((Y \Uparrow X) \cup X) = \epsilon \\
= & \{ \ X \subseteq \textbf{Pairs} \Rightarrow ((Y \Uparrow X) \cup X = Y \cup X) \ \} \\
 & (r \Uparrow X) \Downarrow (X \cup Y) = \epsilon \\
= & \{ \ \text{Theorem 3.14} \ \} \\
 & (r \Uparrow X) \Downarrow Y = \epsilon \\
= & \{ \ \sigma(\epsilon) = \emptyset \ \} \\
 & (r \Uparrow X) \Downarrow (\sigma(\epsilon) \cup Y) = \epsilon \\
\Rightarrow & \{ \ \text{Choose } t = r \Uparrow X, \text{ Theorem 3.7} \ \} \\
 & \exists(t : t \Uparrow X = r \Uparrow X : t \Downarrow (\sigma(\epsilon) \cup Y) = \epsilon)
\end{array}
$$

Next, $|s| = 1$, two cases.

$$
\begin{array}{ll}
 & (r \Uparrow X) \Downarrow ((\sigma(a) \Uparrow X) \cup Y \Uparrow X) = a \Uparrow X \\
= & \{ \ X \subseteq \textbf{Pairs}, \text{ definition } \Uparrow \ \} \\
 & (r \Uparrow X) \Downarrow (\sigma(a) \cup Y \Uparrow X) = a \\
= & \{ \ \text{Theorem 3.14} \ \} \\
 & (r \Uparrow X) \Downarrow (\sigma(a) \cup (Y \Uparrow X) \cup X) = a \\
= & \{ \ X \subseteq \textbf{Pairs} \Rightarrow ((Y \Uparrow X) \cup X = Y \cup X) \ \} \\
 & (r \Uparrow X) \Downarrow (\sigma(a) \cup X \cup Y) = a \\
= & \{ \ \text{Theorem 3.14} \ \} \\
 & (r \Uparrow X) \Downarrow (\sigma(a) \cup Y) = a \\
\Rightarrow & \{ \ \text{Choose } t = r \Uparrow X, \text{ Theorem 3.7} \ \} \\
 & \exists(t : t \Uparrow X = r \Uparrow X : t \Downarrow (\sigma(a) \cup Y) = a)
\end{array}
$$

$$
\begin{array}{ll}
 & (r \Uparrow X) \Downarrow ((\sigma(a.b) \Uparrow X) \cup Y \Uparrow X) = a.b \Uparrow X \\
= & \{ \ X \subseteq \textbf{Pairs}, \text{ definition } \Uparrow \ \} \\
 & (a.b \in X \ \wedge \ (r \Uparrow X) \Downarrow (\sigma(\epsilon) \cup Y \Uparrow X) = \epsilon) \vee \\
 & (a.b \notin X \ \wedge \ (r \Uparrow X) \Downarrow (\sigma(a.b) \cup Y \Uparrow X) = a.b) \\
= & \{ \ \text{Theorem 3.14} \ \} \\
 & (a.b \in X \ \wedge \ (r \Uparrow X) \Downarrow (\sigma(\epsilon) \cup Y \Uparrow X \cup X) = \epsilon) \vee \\
 & (a.b \notin X \ \wedge \ (r \Uparrow X) \Downarrow (\sigma(a.b) \cup Y \Uparrow X \cup X) = a.b) \\
= & \{ \ X \subseteq \textbf{Pairs} \Rightarrow ((Y \Uparrow X) \cup X = Y \cup X) \ \} \\
 & (a.b \in X \ \wedge \ (r \Uparrow X) \Downarrow (\sigma(\epsilon) \cup Y \cup X) = \epsilon) \vee
\end{array}
$$

$$
\begin{array}{l}
\quad (a.b \notin X \ \wedge \ (r \Uparrow X) \Downarrow (\sigma(a.b) \cup Y \cup X) = a.b) \\
= \{ \text{ Theorem 3.14}, a.b \in X \Rightarrow X = X \cup \sigma(a.b) \ \} \\
\quad (a.b \in X \ \wedge \ (r \Uparrow X) \Downarrow (\sigma(\epsilon) \cup Y) = \epsilon) \vee \\
\quad (a.b \notin X \ \wedge \ (r \Uparrow X) \Downarrow (\sigma(a.b) \cup Y) = a.b) \\
\Rightarrow \{ \text{ Choose } t = a.b(r \Uparrow X) \text{ for first disjunct,} \\
\quad t = r \Uparrow X \text{ for second,Theorem 3.7 } \} \\
\quad \exists(t : t \Uparrow X = r \Uparrow X : t \Downarrow (\sigma(a.b) \cup Y) = a.b)
\end{array}
$$

Induction step. $|s| \geq 2 \Rightarrow \exists(s_0, s_1 : |s_0| < s \ \wedge \ |s_1| < s : s_0 s_1 = s)$

$$
\begin{array}{l}
\quad (r \Uparrow X) \Downarrow (\sigma(s_0 s_1) \Uparrow X \cup Y \Uparrow X) = (s_0 s_1) \Uparrow X \\
\Rightarrow \{ \text{ Homomorphic properties of } \Uparrow \text{ and } \Downarrow \ \} \\
\quad \exists(r_0, r_1 : (r_0 r_1) \Uparrow X = r \Uparrow X : \\
\qquad ((r_0 \Uparrow X) \Downarrow (\sigma(s_0) \Uparrow X \cup \sigma(s_1) \Uparrow X \cup Y \Uparrow X) = s_0 \Uparrow X) \wedge \\
\qquad ((r_1 \Uparrow X) \Downarrow (\sigma(s_0) \Uparrow X \cup \sigma(s_1) \Uparrow X \cup Y \Uparrow X) = s_1 \Uparrow X)) \\
\Rightarrow \{ \text{ Ind. hyp., twice } \} \\
\quad \exists(r_0, r_1, r_0', r_1' : r_0' \Uparrow X = r_0 \Uparrow X \ \wedge \ r_1' \Uparrow X = r_1 \Uparrow X \ \wedge \ (r_0 r_1) \Uparrow X = r \Uparrow X : \\
\qquad (r_0' \Downarrow (\sigma(s_0) \cup \sigma(s_1) \cup Y) = s_0 \Uparrow X) \wedge \\
\qquad (r_1' \Downarrow (\sigma(s_0) \cup \sigma(s_1) \cup Y) = s_1 \Uparrow X)) \\
\Rightarrow \{ \text{ Choose } t = r_0' r_1' \ \} \\
\quad \exists(t : t \Uparrow X = r \Uparrow X : t \Downarrow (\sigma(s_0 s_1) \cup Y) = s_0 s_1)
\end{array}
$$

end of proof

$$
\begin{array}{l}
(3.18) \qquad \forall(s, X, Y : X, Y \subseteq \textbf{Pairs} \ \wedge \ \alpha(X) \cap \alpha(Y) = \emptyset : \\
\quad (s \Uparrow (X \cup Y) \Downarrow \alpha(X \cup Y) = \epsilon) \\
= \\
\quad (s \Uparrow (X \cup Y) \Downarrow \alpha(X) = \epsilon) \ \wedge \ (s \Uparrow (X \cup Y) \Downarrow \alpha(Y) = \epsilon) \\
)
\end{array}
$$

Proof, by induction on the length of s.

Base case, trivial.

Case $|s| = 1$, two cases.

$$
\begin{array}{l}
\quad a \Uparrow (X \cup Y) \Downarrow \alpha(X \cup Y) = \epsilon \\
= \{ \text{ Definition } \Uparrow, \Downarrow, \ X, Y \subseteq \textbf{Pairs} \ \} \\
\quad a \notin \alpha(X \cup Y)
\end{array}
$$

$= \{$ Definition α $\}$
$a \notin \alpha(X) \ \wedge \ a \notin \alpha(Y)$
$= \{$ Definition $\Uparrow, \Downarrow$, $X, Y \subseteq$ **Pairs** $\}$
$(a \Uparrow (X \cup Y) \Downarrow \alpha(X) = \epsilon) \ \wedge \ (a \Uparrow (X \cup Y) \Downarrow \alpha(Y) = \epsilon)$

$a.b \Uparrow (X \cup Y) \Downarrow \alpha(X \cup Y) = \epsilon$
$= \{$ Definition $\Uparrow, \Downarrow$, $X, Y \subseteq$ **Pairs** $\}$
$a.b \in (X \cup Y) \ \vee \ (a \notin \alpha(X \cup Y) \ \wedge \ b \notin \alpha(X \cup Y))$
$= \{$ Calculus $\}$
$a.b \in (X \cup Y) \ \vee \ (a \notin \alpha(X) \ \wedge \ b \notin \alpha(X) \ \wedge \ a \notin \alpha(Y) \ \wedge \ b \notin \alpha(Y))$
$= \{$ Calculus $\}$
$(a.b \in (X \cup Y) \ \vee \ (a \notin \alpha(X) \ \wedge \ b \notin \alpha(X))) \ \wedge$
$(a.b \in (X \cup Y) \ \vee \ (a \notin \alpha(Y) \ \wedge \ b \notin \alpha(Y)))$
$= \{$ Definition $\Uparrow, \Downarrow$, $X, Y \subseteq$ **Pairs** $\}$
$(a.b \Uparrow (X \cup Y) \Downarrow \alpha(X) = \epsilon) \ \wedge \ (a.b \Uparrow (X \cup Y) \Downarrow \alpha(Y) = \epsilon)$

Induction step.
Immediate from homomorphic properties of $\Downarrow$ and $\Uparrow$.
end of proof

(3.19) $\forall(s, X, Y : X, Y \subseteq \textbf{Pairs} \ \wedge \ \alpha(X) \cap \alpha(Y) = \emptyset :$
$((s \Uparrow X) \Downarrow \alpha(X) = \epsilon) = ((s \Uparrow (X \cup Y)) \Downarrow \alpha(X) = \epsilon)$
$)$

Proof, by induction on the length of s.
Base case, trivial.
Case $|s| = 1$, two cases.

$(a \Uparrow X) \Downarrow \alpha(X) = \epsilon$
$= \{$ $X, Y \subseteq$ **Pairs**, definition $\Uparrow$ $\}$
$(a \Uparrow (X \cup Y)) \Downarrow \alpha(X) = \epsilon$

$(a.b \Uparrow X) \Downarrow \alpha(X) = \epsilon$
$= \{$ Cases $\}$
$a.b \in X \ \vee \ (a \notin \alpha(X) \ \wedge \ b \notin \alpha(X))$
$= \{$ $(a.b \in Y \ \wedge \ \alpha(X) \cap \alpha(Y) = \emptyset) \Rightarrow (a \notin \alpha(X) \ \wedge \ b \notin \alpha(X))$ $\}$
$a.b \in X \ \vee \ a.b \in Y \ \vee \ (a \notin \alpha(X) \ \wedge \ b \notin \alpha(X))$

$= \{$ Calculus $\}$
$\quad a.b \in (X \cup Y) \ \vee \ (a \notin \alpha(X) \ \wedge \ b \notin \alpha(X))$
$= \{$ Definition $\Uparrow, \Downarrow$, $X, Y \subseteq$ **Pairs** $\}$
$\quad (a.b \Uparrow (X \cup Y)) \Downarrow \alpha(X) = \epsilon$

Induction step.
Immediate from homomorphic properties of $\Downarrow$ and $\Uparrow$.
end of proof

Appendix B

Proofs for Section 3.3

Proofs of the following four theorems are immediate from the definition of $\sqcup$.

(3.25) $S \sqcup \textbf{demon} = S$

(3.26) $S \sqcup S = S$

(3.27) $S \sqcup T = T \sqcup S$

(3.28) $(R \sqcup S) \sqcup T = R \sqcup (S \sqcup T)$

(3.29) $\textbf{skip} \parallel T = T \parallel \textbf{skip} = T$

Proof (left unit element)

$$
\begin{aligned}
& \{\epsilon\} \parallel T \\
= & \{ \text{ Definition } \} \\
& \{r, s, t : s \in \{\epsilon\} \ \wedge \ t \in T \ \wedge \ r \Downarrow \sigma(s) = s \ \wedge \ r \Downarrow \sigma(t) = t \\
& \qquad \wedge \ r \Uparrow (\sigma(\epsilon) \cup \sigma(t)) \ = \ \epsilon : r\} \\
= & \{ \ r \Downarrow \sigma(\epsilon) \ = \epsilon \ , \sigma(\epsilon) \ = \ \emptyset \ \} \\
& \{r, t : t \in T \ \wedge \ r \Downarrow \sigma(t) = t \ \wedge \ r \Uparrow (\sigma(t)) \ = \ \epsilon : r\} \\
= & \{ \text{ Theorem 3.11 } \} \\
& \{r, t : t \in T \ \wedge \ r = t : r\} \\
= & \{ \text{ Calculus } \} \\
& T
\end{aligned}
$$

end of proof

(3.30) $\textbf{demon} \parallel T = T \parallel \textbf{demon} = T$

Proof
Immediate from the definition.
end of proof

$$(3.31) \qquad S\|T \;=\; T\|S$$

Proof
Immediate from the symmetry of the definition.
end of proof

(3.32)

$$\alpha(S)\cap\alpha(T)=\emptyset \;\wedge\; \alpha(T)\cap\alpha(R)=\emptyset \;\wedge\; \alpha(R)\cap\alpha(S)=\emptyset$$
$$\Rightarrow$$
$$(R\|S)\|T \;=\; R\|(S\|T)$$

Since this proof, though obvious in retrospect, took me quite a while to find I give it in full detail.

Proof

$$
\begin{array}{l}
\quad R\|(S\|T) \\
= \{ \text{ Definition } \| \; \} \\
\quad R\| \\
\quad \{r0,s,t : s\in S \;\wedge\; t\in T\,\wedge \\
\qquad\qquad r0\Downarrow\sigma(s)=s \;\wedge\; r0\Downarrow\sigma(t)=t \;\wedge\; r0\Uparrow(\sigma(s)\cup\sigma(t)) \;=\; \epsilon : r0\} \\
= \{ \text{ Definition } \| \; \} \\
\quad \{r1,r,t0 : r\in R \;\wedge\; r1\Downarrow\sigma(r)=r\,\wedge \\
\quad\; t0\in\{r0,s,t : s\in S \;\wedge\; t\in T\,\wedge \\
\qquad\qquad r0\Downarrow\sigma(s)=s \;\wedge\; r0\Downarrow\sigma(t)=t \;\wedge\; r0\Uparrow(\sigma(s)\cup\sigma(t)) \;=\; \epsilon : r0\}\,\wedge \\
\quad\; r1\Downarrow\sigma(t0)=t0 \;\wedge\; r1\Uparrow(\sigma(r)\cup\sigma(t0))=\epsilon\,\wedge \\
\quad : \; r1\} \\
= \{ \text{ Calculus } \} \\
\quad \{r1,r,s,t,t0 : r\in R \;\wedge\; s\in S \;\wedge\; t\in T\,\wedge \\
\qquad t0\Downarrow\sigma(s)=s \;\wedge\; t0\Downarrow\sigma(t)=t \;\wedge\; r1\Downarrow\sigma(r)=r \;\wedge\; r1\Downarrow\sigma(t0)=t0\,\wedge \\
\qquad t0\Uparrow(\sigma(s)\cup\sigma(t))=\epsilon \;\wedge\; r1\Uparrow(\sigma(r)\cup\sigma(t0))=\epsilon \\
\quad : \; r1\}
\end{array}
$$

$= \{$ Theorem 3.12 $\}$

$\{r1, r, s, t, t0 : r \in R \ \wedge \ s \in S \ \wedge \ t \in T \ \wedge$
$\quad t0 \Downarrow \sigma(s) = s \ \wedge \ t0 \Downarrow \sigma(t) = t \ \wedge \ r1 \Downarrow \sigma(r) = r \ \wedge \ r1 \Downarrow \sigma(t0) = t0 \ \wedge$
$\quad t0 \Uparrow (\sigma(s) \cup \sigma(t)) = \epsilon \ \wedge \ (r1 \Uparrow \sigma(r)) \Uparrow \sigma(t0) = \epsilon$
$: \ r1\}$

$= \{ \ \alpha(r) \cap \alpha(t0) = \emptyset$, Theorem 3.13 $\}$

$\{r1, r, s, t, t0 : r \in R \ \wedge \ s \in S \ \wedge \ t \in T \ \wedge$
$\quad t0 \Downarrow \sigma(s) = s \ \wedge \ t0 \Downarrow \sigma(t) = t \ \wedge \ r1 \Downarrow \sigma(r) = r \ \wedge \ (r1 \Uparrow \sigma(r)) \Downarrow \sigma(t0) = t0 \ \wedge$
$\quad t0 \Uparrow (\sigma(s) \cup \sigma(t)) = \epsilon \ \wedge \ (r1 \Uparrow \sigma(r)) \Uparrow \sigma(t0) = \epsilon$
$: \ r1\}$

$= \{$ Theorem 3.11:$(s \Uparrow \sigma(t) = \epsilon \ \wedge \ s \Downarrow \sigma(t) = t) = (s = t)$ $\}$

$\{r1, r, s, t, t0 : r \in R \ \wedge \ s \in S \ \wedge \ t \in T \ \wedge \ r1 \Downarrow \sigma(r) = r \ \wedge$
$\quad t0 \Downarrow \sigma(s) = s \ \wedge \ t0 \Downarrow \sigma(t) = t \ \wedge \ r1 \Uparrow \sigma(r) = t0 \ \wedge \ r1 \Downarrow \sigma(t0) = t0 \ \wedge$
$\quad t0 \Uparrow (\sigma(s) \cup \sigma(t)) = \epsilon$
$: \ r1\}$

$= \{$ Theorem 3.9:$(t0 \Uparrow (\sigma(s) \cup \sigma(t)) = \epsilon) = (t0 \Downarrow (\sigma(s) \cup \sigma(t)) = t0)$ $\}$

$\{r1, r, s, t, t0 : r \in R \ \wedge \ s \in S \ \wedge \ t \in T \ \wedge \ r1 \Downarrow \sigma(r) = r \ \wedge$
$\quad t0 \Downarrow \sigma(s) = s \ \wedge \ t0 \Downarrow \sigma(t) = t \ \wedge \ r1 \Uparrow \sigma(r) = t0 \ \wedge \ r1 \Downarrow \sigma(t0) = t0 \ \wedge$
$(r1 \Uparrow \sigma(r)) \Downarrow (\sigma(s) \cup \sigma(t)) = t0 \ \wedge \ t0 \Uparrow (\sigma(s) \cup \sigma(t)) = \epsilon$
$: \ r1\}$

$= \{ \ \alpha(r) \cap \alpha(\sigma(s) \cup \sigma(t)) = \emptyset$, Theorem 3.13 $\}$

$\{r1, r, s, t, t0 : r \in R \ \wedge \ s \in S \ \wedge \ t \in T \ \wedge \ r1 \Downarrow \sigma(r) = r \ \wedge$
$\quad t0 \Downarrow \sigma(s) = s \ \wedge \ t0 \Downarrow \sigma(t) = t \ \wedge \ r1 \Uparrow \sigma(r) = t0 \ \wedge \ r1 \Downarrow \sigma(t0) = t0 \ \wedge$
$r1 \Downarrow (\sigma(s) \cup \sigma(t)) = t0 \ \wedge \ t0 \Uparrow (\sigma(s) \cup \sigma(t)) = \epsilon$
$: \ r1\}$

$= \{$ Use both equalities to eliminate $t0$ $\}$

$\{r1, r, s, t : r \in R \ \wedge \ s \in S \ \wedge \ t \in T \ \wedge \ r1 \Downarrow \sigma(r) = r \ \wedge$
$\quad (r1 \Downarrow (\sigma(s) \cup \sigma(t))) \Downarrow \sigma(s) = s \ \wedge \ (r1 \Downarrow (\sigma(s) \cup \sigma(t))) \Downarrow \sigma(t) = t \ \wedge$
$\quad (r1 \Downarrow (\sigma(s) \cup \sigma(t))) \Uparrow (\sigma(s) \cup \sigma(t)) = \epsilon \ \wedge \ (r1 \Uparrow \sigma(r)) \Uparrow (\sigma(s) \cup \sigma(t)) = \epsilon$
$: \ r1\}$

$= \{$ Theorems 3.5,3.10,3.12 $\}$

$\{r1, r, s, t : r \in R \ \wedge \ s \in S \ \wedge \ t \in T \ \wedge$
$\quad r1 \Downarrow \sigma(s) = s \ \wedge \ r1 \Downarrow \sigma(t) = t \ \wedge \ r1 \Downarrow \sigma(r) = r \ \wedge$
$\quad r1 \Uparrow (\sigma(r) \cup \sigma(s) \cup \sigma(t)) = \epsilon$
$: \ r1\}$

$= \{$ Symmetry $\}$

$(R\|S)\|T$

end of proof

$$(3.33) \qquad S; \mathbf{skip} \ = \ S; \mathbf{skip} \ = \ S$$

Proof

Immediate from the definition.

end of proof

$$(3.34) \qquad \mathbf{demon} \ ; S \ = \ S; \mathbf{demon} \ = \ \mathbf{demon}$$

Proof

Immediate from the definition.

end of proof

$$(3.35) \qquad (R; S); \ T \ = \ R; (S; \ T)$$

Proof

$(R; S); \ T$

$=$ { Definition ; , twice. }

$\{q, t : q \in \{r, s : r \in R, s \in S : rs\}, t \in T : qt\}$

$=$ { }

$\{r, s, t : r \in R, s \in S, t \in T : (rs)t\}$

$=$ { Catenation is associative }

$\{r, s, t : r \in R, s \in S, t \in T : (rs)t\}$

$=$ { Symmetry }

$R; (S; \ T)$

end of proof

$$(3.36) \qquad R; (S \sqcup T) = (R; S) \sqcup (R; T)$$

Proof

$$
\begin{aligned}
& R;(S \sqcup T) \\
= \{ & \text{ Definition } ; , \sqcup . \ \} \\
& \{r, s : r \in R, s \in (S \cup T) : rs\} \\
= \{ & \ \ \} \\
& \{r, s : r \in R, s \in S : rs\} \cup \{r, t : r \in R, t \in T : rt\} \\
= \{ & \text{ Definition } ; , \sqcup . \ \} \\
& (R; S) \sqcup (R; T)
\end{aligned}
$$

end of proof

(3.37) $\quad (S \sqcup T)$ **connect** $X \ = \ (S$ **connect** $X) \sqcup (T$ **connect** $X)$

Proof Immediate from definitions of **connect** and $\sqcup$.
end of proof

(3.38) $\quad X, Y \subseteq$ **Pairs** $\wedge\ \alpha(X) \cap \alpha(Y) = \emptyset \Rightarrow$
$(S$ **connect** $X)$ **connect** $Y \ = \ S$ **connect** $(X \cup Y)$

Proof

$$
\begin{aligned}
& (S \ \mathbf{connect}\ X)\ \mathbf{connect}\, Y \\
= \{ & \text{ Definition } \mathbf{connect} \ \} \\
& \{s : s \in S \Uparrow X \ \wedge\ s \Downarrow \alpha(x) = \epsilon : s\}\ \mathbf{connect}\ Y \\
= \{ & \text{ Definition } \mathbf{connect} \ \} \\
& \{t : t \in \{s : s \in S \Uparrow X \ \wedge\ s \Downarrow \alpha(x) = \epsilon : s\} \Uparrow Y \ \wedge \\
& \quad t \Downarrow \alpha(Y) = \epsilon : t\} \\
= \{ & \text{ Calculus } \} \\
& \{s : s \in S \Uparrow X \ \wedge\ s \Downarrow \alpha(X) = \epsilon\ \wedge \\
& \quad s \Uparrow (Y) \Downarrow \alpha(Y) = \epsilon : s \Uparrow Y\} \\
= \{ & \text{ Calculus } \} \\
& \{s : s \in S \ \wedge\ (s \Uparrow X) \Downarrow \alpha(X) = \epsilon\ \wedge \\
& \quad (s \Uparrow X) \Uparrow (Y) \Downarrow \alpha(Y) = \epsilon : (s \Uparrow X) \Uparrow Y\} \\
= \{ & \text{ Theorems 3.19, 3.18 3.12 } \} \\
& \{s : s \in S \ \wedge\ s \Uparrow (X \cup Y) \Downarrow \alpha(X \cup Y) = \epsilon : s \Uparrow (X \cup Y)\} \\
= \{ & \text{ Calculus } \} \\
& \{s : s \in S \Uparrow (X \cup Y) \ \wedge\ s \Downarrow \alpha(X \cup Y) = \epsilon : s\} \\
= \{ & \text{ Definition } \mathbf{connect} \ \} \\
& S \ \mathbf{connect}\ (X \cup Y)
\end{aligned}
$$

end of proof

(3.39) $\quad (S;\, T)\ \mathbf{connect}\ X \ = \ (S\ \ \mathbf{connect}\ X)\ ;\ (T\ \mathbf{connect}\ X)$

Proof

$$
\begin{array}{l}
\quad (S;\, T)\ \mathbf{connect}\ X \\
= \{\ \text{Definition}\ \ \mathbf{connect}\ \ \} \\
\quad \{r : r \in (S;\, T) \Uparrow X\ \wedge\ r \Downarrow \alpha(X) = \epsilon : r\} \\
= \{\ \text{Definition ;, Calculus}\ \} \\
\quad \{s, t : s \in S\ \wedge\ t \in T\ \wedge\ (st \Uparrow X) \Downarrow \alpha(X) = \epsilon : st \Uparrow X\} \\
= \{\ \text{Definitions}\ \Uparrow, \Downarrow\ \} \\
\quad \{s, t : s \in S\ \wedge\ t \in T\ \wedge\ (s \Uparrow X) \Downarrow \alpha(X) = \epsilon\ \wedge\ (t \Uparrow X) \Downarrow \alpha(X) = \epsilon \\
\quad\ :\ (s \Uparrow X)(t \Uparrow X)\} \\
= \{\ \text{Calculus}\ \} \\
\quad \{s, t : s \in S \Uparrow X\ \wedge\ t \in T \Uparrow X\ \wedge\ s \Downarrow \alpha(X) = \epsilon\ \wedge\ t \Downarrow \alpha(X) = \epsilon : st\} \\
= \{\ \text{Definitions}\ \ \mathbf{connect}\ , ;\ , \text{calculus}\ \} \\
\quad (S\ \mathbf{connect}\ X); (T\ \mathbf{connect}\ X)
\end{array}
$$

end of proof

The following theorem expresses the fact that connection distributes over parallel composition if there is no interaction between the processes.

(3.40) $\quad X \subseteq \mathbf{Pairs}\ \wedge\ \alpha(X) \cap \alpha(S) = \emptyset \Rightarrow$
$(S \| T)\ \mathbf{connect}\ X \ = \ S \| (T\ \mathbf{connect}\ X)$

Proof

$$
\begin{array}{l}
\quad (S \| T)\ \mathbf{connect}\ X \\
= \{\ \text{Definitions}\ \mathbf{connect}\ \text{and}\ \|\ \} \\
\quad ((S \| \{t : t \in T\ \wedge\ (t \Uparrow X) \Downarrow \alpha(X) = \epsilon : t\})\ \mathbf{connect}\ X) \cup \\
\quad ((S \| \{t : t \in T\ \wedge\ (t \Uparrow X) \Downarrow \alpha(X) \neq \epsilon : t\})\ \mathbf{connect}\ X) \\
= \{\ \text{See below}\ \} \\
\quad (S \| \{t : t \in T\ \wedge\ (t \Uparrow X) \Downarrow \alpha(X) = \epsilon : t\})\ \mathbf{connect}\ X \\
= \{\ \text{Definition}\ \mathbf{connect}\ \} \\
\quad \{r : r \in (S \| \{t : t \in T\ \wedge\ (t \Uparrow X) \Downarrow \alpha(X) = \epsilon : t\}) \Uparrow X\ \wedge\ r \Downarrow \alpha(X) = \epsilon : r\} \\
= \{\ \text{Definition}\ \|, \text{calculus}\ \}
\end{array}
$$

$$\{r,s,t : s \in S \ \wedge\ t \in T \ \wedge\ (t \Uparrow X) \Downarrow \alpha(X) = \epsilon \ \wedge\ (r \Uparrow X) \Downarrow \alpha(X) = \epsilon \wedge$$
$$r \Downarrow \sigma(s) = s \ \wedge\ r \Downarrow \sigma(t) = t \ \wedge\ r \Uparrow (\sigma(s) \cup \sigma(t)) = \epsilon$$
$$: r \Uparrow X\}$$

= { Theorems 3.13, 3.15
$(r \Uparrow (\sigma(s) \cup \sigma(t)) = \epsilon \ \wedge\ (t \Uparrow X) \Downarrow \alpha(X) = \epsilon \ \wedge\ \alpha(X) \cap \alpha(s) = \emptyset)$
$\Rightarrow (r \Uparrow X) \Downarrow \alpha(X) = \emptyset$ }

$$\{r,s,t : s \in S \ \wedge\ t \in T \ \wedge\ (t \Uparrow X) \Downarrow \alpha(X) = \epsilon \ \wedge\ (r \Uparrow X) \Downarrow \alpha(X) = \epsilon \wedge$$
$$(r \Uparrow X) \Downarrow \sigma(s) = s \ \wedge\ r \Downarrow \sigma(t) = t \ \wedge\ (r \Uparrow X) \Uparrow ((\sigma(s) \cup \sigma(t)) - X) = \epsilon$$
$$: r \Uparrow X\}$$

= { Corollary of thm. 3.16 }

$$\{r,s,t : s \in S \ \wedge\ t \in T \ \wedge\ (t \Uparrow X) \Downarrow \alpha(X) = \epsilon \ \wedge\ (r \Uparrow X) \Downarrow \sigma(s) = s \wedge$$
$$r \Downarrow \sigma(t) = t \ \wedge\ (r \Uparrow X) \Downarrow \sigma(t \Uparrow X) = t \Uparrow X \ \wedge\ (r \Uparrow X) \Uparrow (\sigma(s) \cup \sigma(t \Uparrow X)) = \epsilon$$
$$: r \Uparrow X\}$$

= { Theorem 3.17, Calculus }

$$\{r,s,t : s \in S \ \wedge\ t \in T \Uparrow X \ \wedge\ t \Downarrow \alpha(X) = \epsilon \wedge$$
$$r \Downarrow \sigma(s) = s \ \wedge\ r \Downarrow \sigma(t) = t \ \wedge\ r \Uparrow (\sigma(s) \cup \sigma(t)) = \epsilon$$
$$: r\}$$

= { Definitions **connect** and $\|$ }
$S \| (T$ **connect** $X)$

The following calculation was postponed in the second step of the proof.

$(S \| \{t : t \in T \ \wedge\ (t \Uparrow X) \Downarrow \alpha(X) \neq \epsilon : t\})$ **connect** X

= { Definitions **connect**, $\|$ }

$$\{r,s,t : s \in S \ \wedge\ t \in T \ \wedge\ (t \Uparrow X) \Downarrow X \neq \epsilon \ \wedge\ r \Uparrow (\sigma(s) \cup \sigma(t)) = \epsilon \wedge$$
$$r \Downarrow \sigma(s) = s \ \wedge\ r \Downarrow \sigma(t) = t \ \wedge\ (r \Uparrow X) \Downarrow \alpha(X) = \epsilon$$
$$: r \Uparrow X\}$$

= { $r \Downarrow \sigma(t) = t \ \wedge\ (t \Uparrow X) \Downarrow X \neq \epsilon \Rightarrow (r \Uparrow X) \Downarrow \alpha(X) \neq \epsilon$ }
$\emptyset$

end of proof

(3.41)
$\forall (S_0, S_1, T_0, T_1, a, b : \{a, b\} \cap (\alpha(S_0) \cup \alpha(S_1) \cup \alpha(T_0) \cup \alpha(T_1)) = \emptyset :$
$((S_0;\ a;\ S_1) \| (T_0;\ b;\ T_1)) = (S_0 \| T_0); (S_1 \| T_1))$ **connect** $\{a.b\}$

Proof

$$
\begin{array}{l}
\quad ((S_0;\ a;\ S_1)\|(T_0;\ b;\ T_1))\ \mathbf{connect}\ \{a.b\} \\
= \{\ \text{Definition}\ ;\ \} \\
\quad (\{s, s_0, s_1 : s_0 \in S_0\ \wedge\ s_1 \in S_1\ \wedge\ s = s_0 a s_1 : s\} \\
\quad \|\{t, t_0, t_1 : t_0 \in T_0\ \wedge\ t_1 \in T_1\ \wedge\ t = t_0 a t_1 : t\}) \\
\quad \mathbf{connect}\ \{a.b\} \\
= \{\ \text{Defintion}\ \|\ \} \\
\quad \{r, s, t, s_0, s_1, t_0, t_1 : s_0 \in S_0\ \wedge\ s_1 \in S_1\ \wedge\ t_0 \in T_0\ \wedge\ t_1 \in T_1 \wedge \\
\qquad s = s_0 a s_1\ \wedge\ t = t_0 b t_1 \wedge \\
\qquad r \Downarrow \sigma(s) = s\ \wedge\ r \Downarrow \sigma(t) = t\ \wedge\ r \Uparrow (\sigma(s) \cup \sigma(t)) = \epsilon \\
\quad\ : r\}\ \mathbf{connect}\ \{a.b\} \\
= \{\ \text{Decomposition}\ \} \\
\quad \{r, s, t, s_0, s_1, t_0, t_1, r_{s0}, r_a, r_{s1}, r_{t0}, r_b, r_{t1} : \\
\qquad s_0 \in S_0\ \wedge\ s_1 \in S_1\ \wedge\ t_0 \in T_0\ \wedge\ t_1 \in T_1 \wedge \\
\qquad s = s_0 a s_1\ \wedge\ t = t_0 b t_1 \wedge \\
\qquad r = r_{s0} r_a r_{s1}\ \wedge\ r_{s0} \Downarrow \sigma(s) = s_0\ \wedge\ r_a \Downarrow \sigma(s) = a\ \wedge\ r_{s1} \Downarrow \sigma(s) = s_1 \wedge \\
\qquad r = r_{t0} r_b r_{t1}\ \wedge\ r_{t0} \Downarrow \sigma(t) = t_0\ \wedge\ r_b \Downarrow \sigma(t) = b\ \wedge\ r_{t1} \Downarrow \sigma(t) = t_1 \wedge \\
\qquad r \Uparrow (\sigma(s) \cup \sigma(t)) = \epsilon \\
\quad\ : r\}\ \mathbf{connect}\ \{a.b\} \\
= \{\ \text{Definition}\ \mathbf{connect}\ \} \\
\quad \{r, s, t, s_0, s_1, t_0, t_1, r_{s0}, r_a, r_{s1}, r_{t0}, r_b, r_{t1} : \\
\qquad s_0 \in S_0\ \wedge\ s_1 \in S_1\ \wedge\ t_0 \in T_0\ \wedge\ t_1 \in T_1 \wedge \\
\qquad s = s_0 a s_1\ \wedge\ t = t_0 b t_1 \wedge \\
\qquad r = r_{s0} r_a r_{s1}\ \wedge\ r_{s0} \Downarrow \sigma(s) = s_0\ \wedge\ r_a \Downarrow \sigma(s) = a\ \wedge\ r_{s1} \Downarrow \sigma(s) = s_1 \wedge \\
\qquad r = r_{t0} r_b r_{t1}\ \wedge\ r_{t0} \Downarrow \sigma(t) = t_0\ \wedge\ r_b \Downarrow \sigma(t) = b\ \wedge\ r_{t1} \Downarrow \sigma(t) = t_1 \wedge \\
\qquad r \Uparrow (\sigma(s) \cup \sigma(t)) = \epsilon\ \wedge\ (r \Uparrow \{a.b\}) \Downarrow \{a, b\} = \epsilon \\
\quad : r \Uparrow \{a.b\}\} \\
= \{\ \text{Reshuffle, homomorphic properties}\ \Uparrow, \Downarrow\ \} \\
\quad \{r, s, t, s_0, s_1, t_0, t_1, r_{s0}, r_a, r_{s1}, r_{t0}, r_b, r_{t1} : \\
\qquad s_0 \in S_0\ \wedge\ s_1 \in S_1\ \wedge\ t_0 \in T_0\ \wedge\ t_1 \in T_1 \wedge \\
\qquad s = s_0 a s_1\ \wedge\ t = t_0 b t_1 \wedge \\
\qquad r = r_{s0} r_a r_{s1}\ \wedge\ r_{s0} \Downarrow \sigma(s) = s_0\ \wedge\ r_{s1} \Downarrow \sigma(s) = s_1 \wedge \\
\qquad r = r_{t0} r_b r_{t1}\ \wedge\ r_{t0} \Downarrow \sigma(t) = t_0\ \wedge\ r_{t1} \Downarrow \sigma(t) = t_1 \wedge \\
\qquad r \Uparrow (\sigma(s) \cup \sigma(t)) = \epsilon\ \wedge\ (r \Uparrow \{a.b\}) \Downarrow \{a, b\} = \epsilon \wedge \\
\qquad r_a \Downarrow \sigma(s) = a\ \wedge\ (r_a \Uparrow \{a.b\}) \Downarrow \{a, b\} = \epsilon\ \wedge\ r_a \Uparrow (\sigma(s) \cup \sigma(t)) = \epsilon \wedge \\
\qquad r_b \Downarrow \sigma(t) = b\ \wedge\ (r_b \Uparrow \{a.b\}) \Downarrow \{a, b\} = \epsilon\ \wedge\ r_b \Uparrow (\sigma(s) \cup \sigma(t)) = \epsilon \\
\quad : r \Uparrow \{a.b\}\}
\end{array}
$$

$= \{ \quad \}$
$$\begin{array}{l}
\{r, s, t, s_0, s_1, t_0, t_1, r_{s0}, r_{s1}, r_{t0}, r_{t1} : \\
\quad s_0 \in S_0 \ \wedge \ s_1 \in S_1 \ \wedge \ t_0 \in T_0 \ \wedge \ t_1 \in T_1 \wedge \\
\quad s = s_0 a s_1 \ \wedge \ t = t_0 b t_1 \wedge \\
\quad r = r_{s0} a.b r_{s1} \ \wedge \ r_{s0} \Downarrow \sigma(s) = s_0 \ \wedge \ r_{s1} \Downarrow \sigma(s) = s_1 \wedge \\
\quad r = r_{t0} a.b r_{t1} \ \wedge \ r_{t0} \Downarrow \sigma(t) = t_0 \ \wedge \ r_{t1} \Downarrow \sigma(t) = t_1 \wedge \\
\quad r \Uparrow (\sigma(s) \cup \sigma(t)) = \epsilon \ \wedge \ (r \Uparrow \{a.b\}) \Downarrow \{a, b\} = \epsilon \wedge \\
: r \Uparrow \{a.b\}\}
\end{array}$$
$= \{ \ (\alpha(S_0) \cup \alpha(S_1) \cup \alpha(T_0) \cup \alpha(T_1)) \cap \{a, b\} = \emptyset \ \}$
$$\begin{array}{l}
\{r, s, t, s_0, s_1, t_0, t_1, r_0, r_1 : \\
\quad s_0 \in S_0 \ \wedge \ s_1 \in S_1 \ \wedge \ t_0 \in T_0 \ \wedge \ t_1 \in T_1 \wedge \\
\quad s = s_0 a s_1 \ \wedge \ t = t_0 b t_1 \ \wedge \ r = r_0 a.b r_1 \wedge \\
\quad r_0 \Downarrow \sigma(s) = s_0 \ \wedge \ r_1 \Downarrow \sigma(s) = s_1 \wedge \\
\quad r_0 \Downarrow \sigma(t) = t_0 \ \wedge \ r_1 \Downarrow \sigma(t) = t_1 \wedge \\
\quad r \Uparrow (\sigma(s) \cup \sigma(t)) = \epsilon \ \wedge \ (r \Uparrow \{a.b\}) \Downarrow \{a, b\} = \epsilon \wedge \\
: r \Uparrow \{a.b\}\}
\end{array}$$
$= \{$ By mutual implication $\}$
$$\begin{array}{l}
\{r, s, t, s_0, s_1, t_0, t_1, r_0, r_1 : \\
\quad s_0 \in S_0 \ \wedge \ s_1 \in S_1 \ \wedge \ t_0 \in T_0 \ \wedge \ t_1 \in T_1 \wedge \\
\quad s = s_0 a s_1 \ \wedge \ t = t_0 b t_1 \ \wedge \ r = r_0 a.b r_1 \wedge \\
\quad r_0 \Downarrow \sigma(s_0) = s_0 \ \wedge \ r_0 \Downarrow \sigma(t_0) = t_0 \ \wedge \ r_0 \Uparrow (\sigma(s0) \cup \sigma(t0)) = \epsilon \wedge \\
\quad r_1 \Downarrow \sigma(s_1) = s_1 \ \wedge \ r_1 \Downarrow \sigma(t_1) = t_1 \ \wedge \ r_1 \Uparrow (\sigma(s1) \cup \sigma(t1)) = \epsilon \wedge \\
\quad (r_0 \Uparrow \{a.b\}) \Downarrow \{a, b\} = \epsilon \ \wedge \ (r_0 \Uparrow \{a.b\}) \Downarrow \{a, b\} = \epsilon \\
: r \Uparrow \{a.b\}\}
\end{array}$$
$= \{$ Definition $\|, ;$, **connect** $\}$
$((S_0 \| T_0); (a \| b); (S_1 \| T_1))$ **connect** $\{a.b\}$
$= \{$ Theorem 3.39 $\}$
$((S_0 \| T_0)$ **connect** $\{a.b\});$
$((a \| b)$ **connect** $\{a.b\});$
$((S_1 \| T_1)$ **connect** $\{a.b\})$
$= \{ \ \{ab, a.b, ba\}$ **connect** $\{a.b\} = \{\epsilon\}, \{\epsilon\}; S = S \ \}$
$(S_0 \| T_0); (S_1 \| T_1)$

end of proof

Appendix C

Proofs for Section 3.4

Lemma C.1

$$cc.S \Rightarrow \forall(s : s \in \textbf{Traces} : cc.(S \circ s))$$

Proof

$$
\begin{aligned}
& \quad cc.S \\
= & \ \{ \text{ Definition } cc \ \} \\
& \quad \forall(r, x, y :: \alpha(x) \cap \alpha(y) = \emptyset \ \wedge \ S \circ rx \neq \emptyset \ \wedge \ S \circ ry \neq \emptyset \Rightarrow \\
& \qquad S \circ rxy \neq \emptyset \ \wedge \ S \circ ryx \neq \emptyset \ \wedge \ S \circ rxy = S \circ ryx) \\
= & \ \{ \text{ Calculus } \} \\
& \quad \forall(r, s, t, x, y : r = st : \\
& \qquad \alpha(x) \cap \alpha(y) = \emptyset \ \wedge \ S \circ stx \neq \emptyset \ \wedge \ S \circ sty \neq \emptyset \Rightarrow \\
& \qquad S \circ stxy \neq \emptyset \ \wedge \ S \circ styx \neq \emptyset \ \wedge \ S \circ stxy = S \circ styx) \\
= & \ \{ \text{ Calculus, property } \circ \ \} \\
& \quad \forall(s, t, x, y :: \\
& \qquad \alpha(x) \cap \alpha(y) = \emptyset \ \wedge \ (S \circ s) \circ tx \neq \emptyset \ \wedge \ (S \circ s) \circ ty \neq \emptyset \Rightarrow \\
& \qquad\quad (S \circ s) \circ txy \neq \emptyset \ \wedge \ (S \circ s) \circ tyx \neq \emptyset \ \wedge \\
& \qquad\quad (S \circ s) \circ txy = (S \circ s) \circ tyx) \\
= & \ \{ \text{ Definition } cc \ \} \\
& \quad \forall(s :: cc.(S \circ s))
\end{aligned}
$$

end of proof

Lemma C.2

$$cc.S \ \wedge \ \alpha(s) \cap \alpha(t) = \emptyset \ \wedge \ S \circ s \neq \emptyset \ \wedge \ S \circ t \neq \emptyset \ \wedge \ s \neq \epsilon \Rightarrow$$

$$S \circ hd.s \circ t \neq \emptyset$$

Proof, by induction on $|t|$, base case $|t| = 0$ trivial

$$\begin{array}{ll}
& s \neq \epsilon \ \wedge \ S \circ s \neq \emptyset \ \wedge \ S \circ t \neq \emptyset \\
= & \{ \text{ Property } \circ, |t| \neq 0 \Rightarrow t \neq \epsilon \ \} \\
& hd.s \neq \epsilon \ \wedge \ S \circ hd.s \circ tl.s \neq \emptyset \ \wedge \ S \circ hd.t \circ tl.t \neq \emptyset \\
= & \{ \ cc.S \ \} \\
& hd.s \neq \epsilon \ \wedge \ S \circ hd.t \circ hd.s \neq \epsilon \ \wedge \\
& \ S \circ hd.s \circ hd.t \neq \epsilon \ \wedge \ S \circ hd.t \circ tl.t \neq \epsilon \\
= & \{ \text{ Induction hypothesis } \} \\
& hd.s \neq \epsilon \ \wedge \ S \circ hd.t \circ hd.s \circ tl.t \neq \epsilon \ \wedge \ S \circ hd.s \circ hd.t \neq \epsilon \\
= & \{ \ cc.S \ \} \\
& S \circ hd.s \circ hd.t \circ tl.t \neq \epsilon \\
= & \{ \text{ Property } \circ \ \} \\
& S \circ hd.s \circ t \neq \emptyset
\end{array}$$

end of proof

Definition C.3 *For any traces s and t we define the traceset $perm(s,t)$ as follows:*

$$\begin{array}{l}
perm(s,t) = s \quad \textbf{if} \quad t = \epsilon \\
perm(s,t) = t \quad \textbf{if} \quad s = \epsilon \\
perm(s,t) = \{ r : r \in perm(tl.s, t) : hd.s \ r \} \\
\qquad\quad \cup \{ r : r \in perm(s, tl.t) : hd.t \ r \} \quad \textbf{otherwise}
\end{array}$$

Lemma C.4

$$cc.S \ \wedge \ \alpha(s) \cap \alpha(t) = \emptyset \ \wedge \ S \circ s \neq \emptyset \ \wedge \ S \circ t \neq \emptyset \Rightarrow$$

$$\forall (u, v : u, v \in perm(s,t) : S \circ u = S \circ v \neq \emptyset)$$

Proof, by induction on $|s| + |t|$, base case $|s| + |t| = 0$ trivial. Also trivial if $s = \epsilon$ or $t = \epsilon$ so assume $s \neq \epsilon$ and $t \neq \epsilon$.

$$
\begin{array}{l}
\quad S \circ s \neq \emptyset \ \wedge \ S \circ t \neq \emptyset \\
\Rightarrow \{ \text{ Lemma C.2, twice } \} \\
\quad S \circ hd.s \circ tl.s \neq \emptyset \ \wedge \ S \circ hd.t \circ tl.t \neq \emptyset \ \wedge \\
\quad S \circ hd.s \circ t \neq \emptyset \quad \wedge \ S \circ hd.t \circ hd.s \neq \emptyset \\
\Rightarrow \{ \text{ Induction hypothesis, twice } \} \\
\quad \forall (u, v : u, v \in perm(tl.s, t) : S \circ hd.s \circ u = S \circ hd.s \circ v \neq \emptyset) \ \wedge \\
\quad \forall (u, v : u, v \in perm(s, tl.t) : S \circ hd.t \circ u = S \circ hd.t \circ v \neq \emptyset) \\
= \{ \ cc.S, \text{ definition } perm \ \} \\
\quad \forall (u, v : u, v \in perm(s, t) : S \circ u = S \circ v \neq \emptyset)
\end{array}
$$

end of proof

Lemma C.5

$$
cc.S \ \wedge \ \alpha(s) \cap \alpha(t) = \emptyset \ \wedge \ S \circ rs \neq \emptyset \ \wedge \ S \circ rt \neq \emptyset \Rightarrow
$$

$$
\forall (u, v : u, v \in perm(s, t) : S \circ ru = S \circ rv \neq \emptyset)
$$

Proof

$$
\begin{array}{l}
\quad cc.S \ \wedge \ \alpha(s) \cap \alpha(t) = \emptyset \ \wedge \ S \circ rs \neq \emptyset \ \wedge \ S \circ rt \neq \emptyset \\
\Rightarrow \{ \text{ Theorem C.1, property } \circ \ \} \\
\quad cc.(S \circ r) \ \wedge \ \alpha(s) \cap \alpha(t) = \emptyset \ \wedge \ (S \circ r) \circ s \neq \emptyset \ \wedge \ (S \circ r) \circ t \neq \emptyset \\
\Rightarrow \{ \text{ Lemma C.4 } \} \\
\quad \forall (u, v : u, v \in perm(s, t) : (S \circ r) \circ u = (S \circ r) \circ v \neq \emptyset) \\
= \{ \ \circ \ \} \\
\quad \forall (u, v : u, v \in perm(s, t) : S \circ ru = S \circ rv \neq \emptyset)
\end{array}
$$

end of proof

Lemma C.6 *For* $z \in$ **Pairs**,

$$
cc.S \Rightarrow (r_0 \ \mathbf{connect} \ \ z = r_1 \ \mathbf{connect} \ \ z \ \wedge \ S \circ r_0 y \neq \emptyset \ \wedge \ S \circ r_1 \neq \emptyset \Rightarrow
$$

$$
S \circ r_1 y \neq \emptyset)
$$

Proof

By induction on $|r_0| + |r_1|$.

Base case: $r_0 = r_1 = \epsilon$, trivial.

Induction step:

$$
\begin{array}{l}
\quad r_0 \textbf{ connect } z = r_1 \textbf{ connect } z \ \wedge\ S \circ r_0 y \neq \emptyset \ \wedge\ S \circ r_1 \neq \emptyset \\
= \{ \text{ Properties of } \textbf{connect} \ \} \\
\quad r_0 \textbf{ connect } z = r_1 \textbf{ connect } z \ \wedge\ S \circ r_0 y \neq \emptyset \ \wedge\ S \circ r_1 \neq \emptyset \ \wedge \\
\quad (hd.r_0 = hd.r_1 \\
\quad\ \vee\ (hd.r_0 \neq z \ \wedge\ \exists(r_1', n :: r_1 = z^n \ hd.r_0 r_1')) \\
\quad\ \vee\ (hd.r_1 \neq z \ \wedge\ \exists(r_0', n :: r_0 = z^n \ hd.r_1 r_0')) \ ) \\
= \{ \text{ Properties of } \textbf{connect} \ , \circ \ \} \\
\quad r_0 \textbf{ connect } z = r_1 \textbf{ connect } z \ \wedge\ S \circ r_0 y \neq \emptyset \ \wedge\ S \circ r_1 \neq \emptyset \ \wedge \\
\quad (\ (hd.r_0 = hd.r_1 \ \wedge\ tl.r_0 \textbf{ connect } z = tl.r_1 \textbf{ connect } z \ \wedge \\
\qquad (S \circ hd.r_0) \circ tl.r_0 y \neq \emptyset \ \wedge\ (S \circ hd.r_0) \circ tl.r_1 \neq \emptyset) \\
\quad\ \vee\ (hd.r_0 \neq z \ \wedge\ \exists(r_1', n :: r_1 = z^n \ hd.r_0 r_1')) \\
\quad\ \vee\ (hd.r_1 \neq z \ \wedge\ \exists(r_0', n :: r_0 = z^n \ hd.r_1 r_0')) \ ) \\
\Rightarrow \{ \text{ Ind.hyp. } \} \\
\quad r_0 \textbf{ connect } z = r_1 \textbf{ connect } z \ \wedge\ S \circ r_0 y \neq \emptyset \ \wedge\ S \circ r_1 \neq \emptyset \ \wedge \\
\quad (\ (hd.r_0 = hd.r_1 \ \wedge\ (S \circ hd.r_0) \circ tl.r_1 y \neq \emptyset) \\
\quad\ \vee\ (hd.r_0 \neq z \ \wedge\ \exists(r_1', n :: r_1 = z^n \ hd.r_0 r_1')) \\
\quad\ \vee\ (hd.r_1 \neq z \ \wedge\ \exists(r_0', n :: r_0 = z^n \ hd.r_1 r_0')) \ ) \\
\Rightarrow \{ \text{ Calculus } \} \\
\quad r_0 \textbf{ connect } z = r_1 \textbf{ connect } z \ \wedge\ S \circ r_0 y \neq \emptyset \ \wedge\ S \circ r_1 \neq \emptyset \ \wedge \\
\quad (\ S \circ r_1 y \neq \emptyset \\
\quad\ \vee\ (hd.r_0 \neq z \ \wedge \\
\qquad \exists(r_1', n :: r_1 = z^n \ hd.r_0 r_1' \ \wedge\ S \circ z^n \ hd.r_0 r_1' \neq \emptyset \\
\qquad\qquad\qquad\qquad \wedge\ S \circ hd.r_0 tl.r_0 y \neq \emptyset)) \\
\quad\ \vee\ (hd.r_1 \neq z \ \wedge\ \exists(r_0', n :: r_0 = z^n \ hd.r_1 r_0')) \ ) \\
\Rightarrow \{ \text{ Lemma C.2 } \} \\
\quad r_0 \textbf{ connect } z = r_1 \textbf{ connect } z \ \wedge\ S \circ r_0 y \neq \emptyset \ \wedge\ S \circ r_1 \neq \emptyset \ \wedge \\
\quad (\ S \circ r_1 y \neq \emptyset \\
\quad\ \vee\ (hd.r_0 \neq z \ \wedge \\
\qquad \exists(r_1', n :: r_1 = z^n \ hd.r_0 r_1' \ \wedge\ S \circ hd.r_0 \ z^n \ r_1' \neq \emptyset \ \wedge\ S \circ hd.r_0 tl.r_0 y \neq \emptyset)) \\
\quad\ \vee\ (hd.r_1 \neq z \ \wedge\ \exists(r_0', n :: r_0 = z^n \ hd.r_1 r_0')) \ ) \\
\Rightarrow \{ \text{ Property } \circ \ \} \\
\quad r_0 \textbf{ connect } z = r_1 \textbf{ connect } z \ \wedge\ S \circ r_0 y \neq \emptyset \ \wedge\ S \circ r_1 \neq \emptyset \ \wedge
\end{array}
$$

$$
\begin{array}{l}
\quad (\ S \circ r_1 y \neq \emptyset \\
\quad\ \vee\ (hd.r_0 \neq z \wedge \\
\qquad \exists (r_1', n :: r_1 = z^n\ hd.r_0 r_1' \ \wedge\ (z^n r')\ \mathbf{connect}\ \ z = tl.r_0\ \mathbf{connect}\ \ z \\
\qquad\ \wedge (S \circ hd.r_0) \circ z^n\ r_1' \neq \emptyset\ \wedge\ (S \circ hd.r_0) \circ tl.r_0 y \neq \emptyset)) \\
\quad\ \vee\ (hd.r_1 \neq z\ \wedge\ \exists (r_0', n :: r_0 = z^n\ hd.r_1 r_0'))\ ) \\
\Rightarrow \{ \text{ Ind. hyp. } \} \\
\quad r_0\ \mathbf{connect}\ \ z = r_1\ \mathbf{connect}\ \ z\ \wedge\ S \circ r_0 y \neq \emptyset\ \wedge\ S \circ r_1 \neq \emptyset \wedge \\
\quad (\ S \circ r_1 y \neq \emptyset \\
\quad\ \vee\ (hd.r_0 \neq z \wedge \\
\qquad \exists (r_1', n :: r_1 = z^n\ hd.r_0 r_1' \\
\qquad\ \wedge (S \circ hd.r_0) \circ z^n\ r_1' y \neq \emptyset\ \wedge\ S \circ z^n \neq \emptyset)) \\
\quad\ \vee\ (hd.r_1 \neq z\ \wedge\ \exists (r_0', n :: r_0 = z^n\ hd.r_1 r_0'))\ ) \\
\Rightarrow \{ \text{ Lemma C.4 } \} \\
\quad r_0\ \mathbf{connect}\ \ z = r_1\ \mathbf{connect}\ \ z\ \wedge\ S \circ r_0 y \neq \emptyset\ \wedge\ S \circ r_1 \neq \emptyset \wedge \\
\quad (\ S \circ r_1 y \neq \emptyset \\
\quad\ \vee\ (hd.r_0 \neq z\ \wedge\ \exists (r_1', n :: r_1 = z^n\ hd.r_0 r_1'\ \wedge\ S \circ z^n hd.r_0 r_1' y \neq \emptyset)) \\
\quad\ \vee\ (hd.r_1 \neq z\ \wedge\ \exists (r_0', n :: r_0 = z^n\ hd.r_1 r_0'))\ ) \\
\Rightarrow \{ \text{ Absorb 2nd alternative, Calculus } \} \\
\quad r_0\ \mathbf{connect}\ \ z = r_1\ \mathbf{connect}\ \ z\ \wedge\ S \circ r_0 y \neq \emptyset\ \wedge\ S \circ r_1 \neq \emptyset \wedge \\
\quad (\ S \circ r_1 y \neq \emptyset \\
\quad\ \vee\ (hd.r_1 \neq z \wedge \\
\qquad \exists (r_0', n :: r_0 = z^n\ hd.r_1 r_0' \wedge \\
\qquad\quad S \circ z^n\ hd.r_1 r_0' y \neq \emptyset\ \wedge\ S \circ hd.r_1 tl.r_1 \neq \emptyset))\ ) \\
\Rightarrow \{ \text{ Lemma C.4 } \} \\
\quad r_0\ \mathbf{connect}\ \ z = r_1\ \mathbf{connect}\ \ z\ \wedge\ S \circ r_0 y \neq \emptyset\ \wedge\ S \circ r_1 \neq \emptyset \wedge \\
\quad (\ S \circ r_1 y \neq \emptyset \\
\quad\ \vee\ (hd.r_1 \neq z \wedge \\
\qquad \exists (r_0', n :: r_0 = z^n\ hd.r_1 r_0' \wedge \\
\qquad\quad S \circ hd.r_1 z^n r_0' y \neq \emptyset\ \wedge\ S \circ hd.r_1 tl.r_1 \neq \emptyset))\ ) \\
\Rightarrow \{ \text{ Properties } \circ, \mathbf{connect}\ \ \} \\
\quad r_0\ \mathbf{connect}\ \ z = r_1\ \mathbf{connect}\ \ z\ \wedge\ S \circ r_0 y \neq \emptyset\ \wedge\ S \circ r_1 \neq \emptyset \wedge \\
\quad (\ S \circ r_1 y \neq \emptyset \\
\quad\ \vee\ (hd.r_1 \neq z \wedge \\
\qquad \exists (r_0', n :: r_0 = z^n\ hd.r_1 r_0'\ \wedge\ z^n r_0'\ \mathbf{connect}\ \ z = tl.r_1\ \mathbf{connect}\ \ z \\
\qquad\quad (S \circ hd.r_1) \circ z^n r_0' y \neq \emptyset\ \wedge\ (S \circ hd.r_1) \circ tl.r_1 \neq \emptyset))\ ) \\
\Rightarrow \{ \text{ Ind. hyp. } \}
\end{array}
$$

r_0 **connect** $z = r_1$ **connect** $z \wedge S \circ r_0 y \neq \emptyset \wedge S \circ r_1 \neq \emptyset \wedge$
$(S \circ r_1 y \neq \emptyset$
$\vee \ (hd.r_1 \neq z \wedge$
$\exists(r_0', n :: (S \circ hd.r_1) \circ tl.r_1 y \neq \emptyset)) \ )$
$\Rightarrow$ { Property $\circ$ }
$S \circ r_1 y \neq \emptyset$

end of proof

Lemma C.7 *For fully compositional* S,$z \in$ **Pairs**,

$$\alpha(x) \cap \alpha(y) = \emptyset \ \wedge \ (S \textbf{ connect } z) \circ rx \neq \emptyset \ \wedge \ (S \textbf{ connect } z) \circ ry \neq \emptyset$$

$$\Rightarrow$$

$$(S \textbf{ connect } z) \circ rxy \neq \emptyset \ \wedge \ (S \textbf{ connect } z) \circ ryx \neq \emptyset$$

Proof

$\alpha(x) \cap \alpha(y) = \emptyset \ \wedge \ (S$ **connect** $z) \circ rx \neq \emptyset \ \wedge \ (S$ **connect** $z) \circ ry \neq \emptyset$
$=$ { Properties **connect** }
$\alpha(x) \cap \alpha(y) = \emptyset \wedge$
$\exists(r_0, r_1 : r_0$ **connect** $z = r1$ **connect** $z = r : S \circ r_0 x \neq \emptyset \ \wedge \ S \circ r_1 y \neq \emptyset)$
$\Rightarrow$ { Lemma C.6 }
$\alpha(x) \cap \alpha(y) = \emptyset \ \wedge \ \exists(r_0 : r_0$ **connect** $z = r : S \circ r_0 x \neq \emptyset \ \wedge \ S \circ r_0 y \neq \emptyset)$
$\Rightarrow$ { $cc.S$ }
$\alpha(x) \cap \alpha(y) = \emptyset \ \wedge \ \exists(r_0 : r_0$ **connect** $z = r : S \circ r_0 xy \neq \emptyset \ \wedge \ S \circ r_0 yx \neq \emptyset)$
$\Rightarrow$ { Properties **connect** }
$(S$ **connect** $z) \circ rxy \neq \emptyset \ \wedge \ (S$ **connect** $z) \circ ryx \neq \emptyset$

end of proof

Lemma C.8 *For fully compositional* S,$z \in$ **Pairs**,
$\alpha(x) \cap \alpha(y) = \emptyset \ \wedge \ (S$ **connect** $z) \circ rx \neq \emptyset \ \wedge \ (S$ **connect** $z) \circ ry \neq \emptyset \ \wedge \ r_0 \in (S$ **connect** $z) \circ rxy$
$\Rightarrow$
$r_0 \in (S$ **connect** $z) \circ ryx$

Proof

$$
\begin{array}{l}
\quad \alpha(x) \cap \alpha(y) = \emptyset \;\wedge\; (S \textbf{ connect } z) \circ rx \neq \emptyset \;\wedge\; (S \textbf{ connect } z) \circ ry \neq \emptyset \;\wedge \\
\quad r_0 \in (S \textbf{ connect } z) \circ rxy \\
= \{ \text{ Properties } \textbf{connect} \; \} \\
\quad \alpha(x) \cap \alpha(y) = \emptyset \;\wedge \\
\quad\; \exists(r', r'', r_0', n : r_0' \textbf{ connect } z = r_0 \;\wedge\; r' \textbf{ connect } z = r'' \textbf{ connect } z = r : \\
\qquad r'xz^n yr_0' \in S \;\wedge\; S \circ r''y \neq \emptyset) \\
\Rightarrow \{ \text{ Theorem C.6 } \} \\
\quad \alpha(x) \cap \alpha(y) = \emptyset \;\wedge \\
\quad\; \exists(r', r_0', n : r_0' \textbf{ connect } z = r_0 \;\wedge\; r' \textbf{ connect } z = r : \\
\qquad r'xz^n yr_0' \in S \;\wedge\; S \circ r'y \neq \emptyset) \\
= \{ \; cc.S \; \} \\
\quad \alpha(x) \cap \alpha(y) = \emptyset \;\wedge \\
\quad\; \exists(r', r_0', n : r_0' \textbf{ connect } z = r_0 \;\wedge\; r' \textbf{ connect } z = r : \\
\qquad r'xz^n yr_0' \in S \;\wedge\; S \circ r'xy \neq \emptyset \;\wedge\; S \circ r'yx \neq \emptyset) \\
= \{ \text{ Theorem C.5,} r'x \text{ for } r \; \} \\
\quad \alpha(x) \cap \alpha(y) = \emptyset \;\wedge \\
\quad\; \exists(r', r_0', n : r_0' \textbf{ connect } z = r_0 \;\wedge\; r' \textbf{ connect } z = r : \\
\qquad r'xz^n yr_0' \in S \;\wedge\; r'xyz^n r_0' \in S \;\wedge\; S \circ r'yx \neq \emptyset) \\
= \{ \; cc.S \; \} \\
\quad \alpha(x) \cap \alpha(y) = \emptyset \;\wedge \\
\quad\; \exists(r', r_0', n : r_0' \textbf{ connect } z = r_0 \;\wedge\; r' \textbf{ connect } z = r : r'yxz^n r_0' \in S) \\
= \{ \text{ Properties } \textbf{connect} \;, \circ \} \\
\quad r_0 \in (S \textbf{ connect } z) \circ ryx
\end{array}
$$

end of proof

With these lemmata the proof of the first theorem is now trivial.

(3.43) $\qquad cc.S \Rightarrow \forall(z : z \in \textbf{Pairs} : cc.(S \textbf{ connect } z))$

Proof

Immediate from previous two lemmata.

end of proof

Lemma C.9

$$cc.S \;\wedge\; cc.T \Rightarrow$$
$$(\alpha(x) \cap \alpha(y) = \emptyset \;\wedge\; (S \| T) \circ rx \neq \emptyset \;\wedge\; (S \| T) \circ ry \neq \emptyset \Rightarrow$$
$$(S \| T) \circ rxy \neq \emptyset \;\wedge\; (S \| T) \circ ryx \neq \emptyset)$$

Proof

$\alpha(x) \cap \alpha(y) = \emptyset \;\wedge\; (S \| T) \circ rx \neq \emptyset \;\wedge\; (S \| T) \circ ry \neq \emptyset$

$=$ { Nonoverlapping alphabets }

$\alpha(x) \cap \alpha(y) = \emptyset$

$\wedge\, S \circ (rx \Downarrow \alpha(S)) \neq \emptyset \;\wedge\; S \circ (ry \Downarrow \alpha(S)) \neq \emptyset$

$\wedge\, T \circ (rx \Downarrow \alpha(T)) \neq \emptyset \;\wedge\; T \circ (ry \Downarrow \alpha(T)) \neq \emptyset$

$=$ { Properties of $\circ$ }

$\alpha(x) \cap \alpha(y) = \emptyset$

$\wedge\, (S \circ (r \Downarrow \alpha(S))) \circ (x \Downarrow \alpha(S)) \neq \emptyset \;\wedge\; (S \circ (r \Downarrow \alpha(S))) \circ (y \Downarrow \alpha(S)) \neq \emptyset$

$\wedge\, (T \circ (r \Downarrow \alpha(T))) \circ (x \Downarrow \alpha(T)) \neq \emptyset \;\wedge\; (T \circ (r \Downarrow \alpha(T))) \circ (y \Downarrow \alpha(T)) \neq \emptyset$

$=$ { Define $S' = S \circ (r \Downarrow \alpha(S))$,$x_S = x \Downarrow \alpha(S)$ idem T', x_T, y_S, y_T }

$\alpha(x) \cap \alpha(y) = \emptyset \;\wedge\; S' \circ x_S \neq \emptyset \;\wedge\; S' \circ y_S \neq \emptyset$

$\wedge\, T' \circ x_T \neq \emptyset \;\wedge\; T' \circ y_T \neq \emptyset$

$=$ { $[0](x_S = x \;\wedge\; x_T = \epsilon \;\wedge\; y_S = y \;\wedge\; y_T = \epsilon) \vee$

$[1](x_S = x \;\wedge\; x_T = \epsilon \;\wedge\; y_S = \epsilon \;\wedge\; y_T = y) \vee$

$[2](x_S = \epsilon \;\wedge\; x_T = x \;\wedge\; y_S = y \;\wedge\; y_T = \epsilon) \vee$

$[3](x_S = \epsilon \;\wedge\; x_T = x \;\wedge\; y_S = \epsilon \;\wedge\; y_T = y) \vee$

$[4](x = a.b \;\wedge\; x_S = a \;\wedge\; x_T = b \;\wedge\; y_S = y \;\wedge\; y_T = \epsilon) \vee$

$[5](x = a.b \;\wedge\; x_S = a \;\wedge\; x_T = b \;\wedge\; y_S = \epsilon \;\wedge\; y_T = y) \vee$

$[6](x_S = x \;\wedge\; x_T = \epsilon \;\wedge\; y = c.d \;\wedge\; y_S = c \;\wedge\; y_T = d) \vee$

$[7](x_S = \epsilon \;\wedge\; x_T = x \;\wedge\; y = c.d \;\wedge\; y_S = c \;\wedge\; y_T = d) \vee$

$[8](x = a.b \;\wedge\; x_S = a \;\wedge\; x_T = b \;\wedge\; y = c.d \;\wedge\; y_S = c \;\wedge\; y_T = d)$ }

$\alpha(x) \cap \alpha(y) = \emptyset \wedge$

$[0](S' \circ x \neq \emptyset \;\wedge\; S' \circ y \neq \emptyset \;\wedge\; T' \neq \emptyset) \vee$

$[1](S' \circ x \neq \emptyset \;\wedge\; S' \neq \emptyset \;\wedge\; T' \neq \emptyset \;\wedge\; T' \circ y \neq \emptyset) \vee$

$[2](S' \neq \emptyset \;\wedge\; S' \circ y \neq \emptyset \;\wedge\; T' \circ x \neq \emptyset \;\wedge\; T' \neq \emptyset) \vee$

$[3](S' \neq \emptyset \;\wedge\; T' \circ x \neq \emptyset \;\wedge\; T' \circ y \neq \emptyset) \vee$

$[4](x = a.b \;\wedge\; S' \circ a \neq \emptyset \;\wedge\; S' \circ y \neq \emptyset \;\wedge\; T' \circ b \neq \emptyset \;\wedge\; T' \neq \emptyset) \vee$

$[5](x = a.b \;\wedge\; S' \circ a \neq \emptyset \;\wedge\; S' \neq \emptyset \;\wedge\; T' \circ b \neq \emptyset \;\wedge\; T' \circ y \neq \emptyset) \vee$

$[6](y = c.d \;\wedge\; S' \circ x \neq \emptyset \;\wedge\; S' \circ c \neq \emptyset \;\wedge\; T' \neq \emptyset \;\wedge\; T' \circ d \neq \emptyset) \vee$

$$
\begin{aligned}
&[7](y = c.d \ \wedge\ S' \neq \emptyset \ \wedge\ S' \circ c \neq \emptyset \ \wedge\ T' \circ x \neq \emptyset \ \wedge\ T' \circ d \neq \emptyset) \vee \\
&[8](x = a.b \ \wedge\ y = c.d \ \wedge\ S' \circ a \neq \emptyset \ \wedge\ S' \circ c \neq \emptyset \\
&\qquad\qquad \wedge\ T' \circ b \neq \emptyset \ \wedge\ T' \circ d \neq \emptyset)
\end{aligned}
$$

We prove the theorem for cases $0, 1, 4, 5,$ and 8. The other cases follow by symmetry.

Proof [0]

$$
\begin{aligned}
&\quad \alpha(x) \cap \alpha(y) = \emptyset \ \wedge\ S' \circ x \neq \emptyset \ \wedge\ S' \circ y \neq \emptyset \ \wedge\ T' \ \neq \emptyset \\
&\Rightarrow \{ \text{ Theorem C.1 } \} \\
&\quad \alpha(x) \cap \alpha(y) = \emptyset \ \wedge\ S' \circ xy \neq \emptyset \ \wedge\ S' \circ yx \neq \emptyset \ \wedge\ T' \neq \emptyset \\
&\Rightarrow \{ \text{ BIG step } \} \\
&\quad (S \| T) \circ rxy \neq \emptyset \ \wedge\ (S \| T) \circ ryx \neq \emptyset
\end{aligned}
$$

Proof [1]

$$
\begin{aligned}
&\quad \alpha(x) \cap \alpha(y) = \emptyset \ \wedge\ S' \circ x \neq \emptyset \ \wedge\ T' \circ y \neq \emptyset \\
&\Rightarrow \{ \text{ BIG step } \} \\
&\quad (S \| T) \circ rxy \neq \emptyset \ \wedge\ (S \| T) \circ ryx \neq \emptyset
\end{aligned}
$$

Proof [4]

$$
\begin{aligned}
&\quad \alpha(x) \cap \alpha(y) = \emptyset \ \wedge\ x = a.b \ \wedge\ S' \circ a \neq \emptyset \ \wedge\ S' \circ y \neq \emptyset \ \wedge\ T' \circ b \neq \emptyset \\
&\Rightarrow \{ \text{ Theorem C.1}, y \neq a \ \ \} \\
&\quad \alpha(x) \cap \alpha(y) = \emptyset \ \wedge\ x = a.b \ \wedge\ S' \circ ay \neq \emptyset \ \wedge\ S' \circ ya \neq \emptyset \ \wedge\ T' \circ b \neq \emptyset \\
&\Rightarrow \{ \text{ BIG step } \} \\
&\quad (S \| T) \circ rxy \neq \emptyset \ \wedge\ (S \| T) \circ ryx \neq \emptyset
\end{aligned}
$$

Proof [5]

$$
\begin{aligned}
&\quad \alpha(x) \cap \alpha(y) = \emptyset \ \wedge\ x = a.b \ \wedge\ S' \circ a \neq \emptyset \ \wedge\ T' \circ b \neq \emptyset \ \wedge\ T' \circ y \neq \emptyset \\
&\Rightarrow \{ \text{ Theorem C.1 } \} \\
&\quad \alpha(x) \cap \alpha(y) = \emptyset \ \wedge\ x = a.b \ \wedge\ S' \circ a \neq \emptyset \ \wedge\ T' \circ by \neq \emptyset \ \wedge\ T' \circ yb \neq \emptyset \\
&\Rightarrow \{ \text{ BIG step } \} \\
&\quad (S \| T) \circ rxy \neq \emptyset \ \wedge\ (S \| T) \circ ryx \neq \emptyset
\end{aligned}
$$

Proof [8]

$$\alpha(x) \cap \alpha(y) = \emptyset \ \wedge \ x = a.b \ \wedge \ y = c.d \ \wedge \ S' \circ a \neq \emptyset \ \wedge \ S' \circ c \neq \emptyset$$
$$\wedge \ T' \circ b \neq \emptyset \ \wedge \ T' \circ d \neq \emptyset$$
$\Rightarrow$ { Theorem C.1,twice }
$$\alpha(x) \cap \alpha(y) = \emptyset \ \wedge \ x = a.b \ \wedge \ y = c.d \ \wedge \ S' \circ ac \neq \emptyset \ \wedge \ S' \circ ca \neq \emptyset$$
$$\wedge \ T' \circ bd \neq \emptyset \ \wedge \ T' \circ db \neq \emptyset$$
$\Rightarrow$ { BIG step }
$$(S \| T) \circ rxy \neq \emptyset \ \wedge \ (S \| T) \circ ryx \neq \emptyset$$

end of proof

Lemma C.10

$$(cc.S \ \wedge \ cc.T) \Rightarrow$$
$$(\alpha(x) \cap \alpha(y) = \emptyset \wedge r_0 \in (S \| T) \circ rxy \wedge (S \| T) \circ ryx \neq \emptyset \Rightarrow r_0 \in (S \| T) \circ ryx)$$

Proof

$$r_0 \in (S \| T) \circ rxy \ \wedge \ (S \| T) \circ ryx \neq \emptyset$$
$\Rightarrow$ { Disjunct alphabets, define $r_S = r \Downarrow \alpha(S)$ etc.. }
$$S \circ r_S x_S y_S \neq \emptyset \ \wedge \ S \circ r_S y_S x_S \neq \emptyset \ \wedge \ T \circ r_T x_T y_T \neq \emptyset \ \wedge \ T \circ r_T y_T x_T \neq \emptyset$$
$$\wedge \ r_0 \in ((S \circ r_S x_S y_S) \| (T \circ r_T x_T y_T))$$
$\Rightarrow$ { Theorem C.1, twice }
$$S \circ r_S x_S y_S \neq \emptyset \ \wedge \ S \circ r_S y_S x_S \neq \emptyset \ \wedge \ T \circ r_T x_T y_T \neq \emptyset \ \wedge \ T \circ r_T y_T x_T \neq \emptyset$$
$$\wedge \ r_0 \in ((S \circ r_S y_S x_S) \| (T \circ r_T y_T x_T))$$
$\Rightarrow$ { Properties $\|$ }
$$r_0 \in (S \| T) \circ ryx$$

end of proof

These lemmata enable us to prove the second theorem.

(3.44) $\quad cc.S \ \wedge \ cc.T \Rightarrow cc.(S \| T)$

Proof

Immediate from previous two lemmata.

end of proof

(3.46) $\quad (pr.S \ \wedge \ pr.T) \Rightarrow pr.(S;\ T)$

Proof by contraposition.

$$
\begin{aligned}
& \neg pr.(S;\,T) \\
= \{ & \text{ Definition } pr \ \} \\
& \exists(s_0,s_1,t_0,t_1 : s_0,s_1 \in S \ \wedge\ t_0,t_1 \in T : s_0t_0 < s_1t_1) \\
= \{ & \text{ Cases } \} \\
& \exists(s_0,s_1,t_0,t_1 : s_0,s_1 \in S \ \wedge\ t_0,t_1 \in T : \\
& \quad (s_0 < s_1 \ \vee\ s_0 = s_1 \ \vee\ s_0 > s_1) \ \wedge\ s_0t_0 < s_1t_1) \\
\Rightarrow \{ & \text{ Definition } pr \ \} \\
& \exists(s_0,s_1,t_0,t_1 : s_0,s_1 \in S \ \wedge\ t_0,t_1 \in T : \neg pr.S \ \vee\ (s_0 = s_1 \ \wedge\ s_0t_0 < s_1t_1)) \\
\Rightarrow \{ & \text{ Definition } pr\text{,calculus } \} \\
& \neg pr.S \ \vee\ \neg pr.T
\end{aligned}
$$

end of proof

(3.47) $\quad (pr.S \ \wedge\ pr.T) \Rightarrow pr.(S\|T)$

Proof by contraposition.

$$
\begin{aligned}
& \neg pr.(S\|T) \\
= \{ & \text{ Definition } pr \ \} \\
& \exists(s_0,s_1,t_0,t_1,r_0,r_1 : s_0,s_1 \in S \ \wedge\ t_0,t_1 \in T : r_0 \in s_0\|t_0 \ \wedge\ r_1 \in s1\|t1 \ \wedge\ r_0 < r_1) \\
= \{ & \text{ Cases } \} \\
& \exists(s_0,s_1,t_0,t_1,r_0,r_1 : s_0,s_1 \in S \ \wedge\ t_0,t_1 \in T : \\
& r_0 \in s_0\|t_0 \ \wedge\ r_1 \in s1\|t1 \ \wedge\ r_0 < r_1 \wedge \\
& (s_0 < s_1 \ \vee\ s_0 > s_1 \ \vee\ s_0 \not<\not> s_1 \ \vee\ s_0 = s_1)) \\
\Rightarrow \{ & \text{ Disjoint alphabets, hence } s_0 \not<\not> s_1 \Rightarrow r_0 \not<\not> r_1 \ \} \\
& \exists(s_0,s_1,t_0,t_1,r_0,r_1 : s_0,s_1 \in S \ \wedge\ t_0,t_1 \in T : \\
& r_0 \in s_0\|t_0 \ \wedge\ r_1 \in s1\|t1 \ \wedge\ r_0 < r_1 \wedge \\
& (\neg pr.S \ \vee\ s_0 = s_1)) \\
\Rightarrow \{ & \text{ Disjoint alphabets, hidden inductive argument } \} \\
& \neg pr.S \ \vee\ \neg pr.T
\end{aligned}
$$

end of proof

Lemma C.11
$cc.S \ \wedge\ pr.S \ \wedge\ cc.T \Rightarrow$
$\quad (x \neq y \ \wedge\ rxr_0 \in (S;\,T) \ \wedge\ ryr_1 \in (S;\,T) \Rightarrow (S;\,T) \circ rxy \neq \emptyset \ \wedge\ (S;\,T) \circ$
$ryx \neq \emptyset)$

Proof

$$
\begin{array}{l}
\quad x \neq y \ \wedge \ rxr_0 \in (S;\ T) \ \wedge \ ryr_1 \in (S;\ T) \\
= \{ \text{ Definition ; } \} \\
\quad x \neq y \ \wedge \ \exists(r_{0S}, r_{0T} : r_{0S} \in S \ \wedge \ r_{0T} \in T : rxr_0 = r_{0S} r_{0T}) \\
\qquad \wedge \ \exists(r_{1S}, r_{1T} : r_{1S} \in S \ \wedge \ r_{1T} \in T : ryr_1 = r_{1S} r_{1T}) \\
= \{ \text{ Calculus,cases } \} \\
\quad x \neq y \wedge \\
\quad \exists(r_{0S}, r_{0T}, r_{1S}, r_{1T} : r_{0S} \in S \ \wedge \ r_{0T} \in T \ \wedge \ r_{1S} \in S \ \wedge \ r_{1T} \in T : \\
\quad\ (r_{0S} < rx \ \vee \ rx \leq r_{0S}) \wedge (r_{1S} < ry \ \vee \ ry \leq r_{1S}) \wedge \\
\quad\ rxr_0 = r_{0S} r_{0T} \ \wedge \ ryr_1 = r_{1S} r_{1T}) \\
= \{ \text{ Calculus } \} \\
\quad x \neq y \wedge \\
\quad \exists(r_{0S}, r_{0T}, r_{1S}, r_{1T} : r_{0S} \in S \ \wedge \ r_{0T} \in T \ \wedge \ r_{1S} \in S \ \wedge \ r_{1T} \in T : \\
\quad\ ((r_{0S} < rx \ \wedge \ r_{1S} < ry) \ \vee \ (r_{0S} < rx \ \wedge \ r_{1S} \geq ry) \vee \\
\quad\ \ (rx \leq r_{0S} \ \wedge \ r_{1S} < ry) \ \vee \ (rx \leq r_{0S} \ \wedge \ ry \leq r_{1S})) \wedge \\
\quad\ rxr_0 = r_{0S} r_{0T} \ \wedge \ ryr_1 = r_{1S} r_{1T}) \\
\Rightarrow \{ \ pr.S \ \} \\
\quad x \neq y \wedge \\
\quad \exists(r_{0S}, r_{0T}, r_{1S}, r_{1T} : r_{0S} \in S \ \wedge \ r_{0T} \in T \ \wedge \ r_{1S} \in S \ \wedge \ r_{1T} \in T : \\
\quad ((r_{0S} < rx \ \wedge \ r_{1S} < ry \ \wedge \ r_{0S} = r_{1S}) \ \vee \ (rx \leq r_{0S} \ \wedge \ ry \leq r_{1S})) \wedge \\
\quad rxr_0 = r_{0S} r_{0T} \ \wedge \ ryr_1 = r_{1S} r_{1T}) \\
\Rightarrow \{ \text{ Calculus } \} \\
\quad \exists(r', r'' : r'r'' = r \ \wedge \ r' \in S : T \circ r''x \neq \emptyset \ \wedge \ T \circ r''y \neq \emptyset) \vee \\
\quad (S \circ rx \neq \emptyset \ \wedge \ S \circ ry \neq \emptyset) \\
\Rightarrow \{ \ cc.S \ \wedge \ cc.T \ \} \\
\quad \exists(r', r'' : r'r'' = r \ \wedge \ r' \in S : T \circ r''xy \neq \emptyset \ \wedge \ T \circ r''yx \neq \emptyset) \vee \\
\quad (S \circ rxy \neq \emptyset \ \wedge \ S \circ ryx \neq \emptyset) \\
\Rightarrow \{ \ ; \ \} \\
\quad (S;\ T) \circ rxy \neq \emptyset \ \wedge \ (S;\ T) \circ ryx \neq \emptyset
\end{array}
$$

end of proof

Lemma C.12
$cc.S \ \wedge \ pr.S \ \wedge \ cc.T \Rightarrow$
$(x \neq y \ \wedge \ rxyr_0 \in (S;\ T) \ \wedge \ ryxr_1 \in (S;\ T) \Rightarrow ryxr_0 \in (S;\ T) \ \wedge \ rxyr_1 \in (S;\ T))$

Proof

$$
\begin{array}{l}
\quad x \neq y \ \wedge \ rxyr_0 \in (S;\ T) \ \wedge \ ryxr_1 \in (S;\ T) \\
= \{ \text{ Cases } \} \\
\quad x \neq y \wedge \\
\quad \exists (r_{0S}, r_{0T}, r_{1S}, r_{1T} : r_{0S} \in S \ \wedge \ r_{0T} \in T \ \wedge \ r_{1S} \in S \ \wedge \ r_{1T} \in T : \\
\quad\ \ (r_{0S} < rx \ \vee \ r_{0S} = rx \ \vee \ r_{0S} > rx) \wedge \\
\quad\ \ (r_{1S} < ry \ \vee \ r_{1S} = ry \ \vee \ r_{0S} > ry) \wedge \\
\quad\ \ rxyr_0 = r_{0S} r_{0T} \ \wedge \ ryxr_1 = r_{1S} r_{1T}) \\
= \{ \text{ Calculus } \} \\
\quad x \neq y \wedge \\
\quad \exists (r_{0S}, r_{0T}, r_{1S}, r_{1T} : r_{0S} \in S \ \wedge \ r_{0T} \in T \ \wedge \ r_{1S} \in S \ \wedge \ r_{1T} \in T : \\
\quad\ \ ((r_{0S} < rx \ \wedge \ r_{1S} < ry) \vee \\
\quad\ \ \ (r_{0S} < rx \ \wedge \ r_{1S} = ry) \vee \\
\quad\ \ \ (r_{0S} < rx \ \wedge \ r_{1S} > ry) \vee \\
\quad\ \ \ (r_{0S} = rx \ \wedge \ r_{1S} < ry) \vee \\
\quad\ \ \ (r_{0S} = rx \ \wedge \ r_{1S} = ry) \vee \\
\quad\ \ \ (r_{0S} = rx \ \wedge \ r_{1S} > ry) \vee \\
\quad\ \ \ (r_{0S} > rx \ \wedge \ r_{1S} < ry) \vee \\
\quad\ \ \ (r_{0S} > rx \ \wedge \ r_{1S} = ry) \vee \\
\quad\ \ \ (r_{0S} > rx \ \wedge \ r_{1S} > ry)) \wedge \\
\quad\ \ rxyr_0 = r_{0S} r_{0T} \ \wedge \ ryxr_1 = r_{1S} r_{1T}) \\
= \{ \ pr.S \ \} \\
\quad x \neq y \wedge \\
\quad \exists (r_{0S}, r_{0T}, r_{1S}, r_{1T} : r_{0S} \in S \ \wedge \ r_{0T} \in T \ \wedge \ r_{1S} \in S \ \wedge \ r_{1T} \in T : \\
\quad\ \ ((r_{0S} < rx \ \wedge \ r_{1S} < ry \ \wedge \ r_{0S} = r_{1S}) \vee \\
\quad\ \ \ (r_{0S} = rx \ \wedge \ r_{1S} = ry) \vee \\
\quad\ \ \ (r_{0S} = rx \ \wedge \ r_{1S} > ry) \vee \\
\quad\ \ \ (r_{0S} > rx \ \wedge \ r_{1S} = ry) \vee \\
\quad\ \ \ (r_{0S} > rx \ \wedge \ r_{1S} > ry)) \wedge \\
\quad\ \ rxyr_0 = r_{0S} r_{0T} \ \wedge \ ryxr_1 = r_{1S} r_{1T}) \\
= \{ \ pr.S \ \wedge \ cc.S \ \} \\
\quad x \neq y \wedge \\
\quad \exists (r_{0S}, r_{0T}, r_{1S}, r_{1T} : r_{0S} \in S \ \wedge \ r_{0T} \in T \ \wedge \ r_{1S} \in S \ \wedge \ r_{1T} \in T : \\
\quad\ \ ((r_{0S} < rx \ \wedge \ r_{1S} < ry \ \wedge \ r_{0S} = r_{1S}) \vee \\
\quad\ \ \ (r_{0S} > rx \ \wedge \ r_{1S} > ry)) \wedge \\
\quad\ \ rxyr_0 = r_{0S} r_{0T} \ \wedge \ ryxr_1 = r_{1S} r_{1T})
\end{array}
$$

$= \{\ \ \}$
$\quad x \neq y \wedge$
$\quad (\exists(r', r'' : r'r'' = r : r' \in S \ \wedge\ r''xyr_0 \in T \ \wedge\ r''yxr_1 \in T)$
$\quad\ \exists(r'_0, r''_0, r'_1, r''_1 : r'_0 r''_0 = r_0 \ \wedge\ r'_1 r''_1 = r_1 :$
$\quad\quad rxyr'_0 \in S \ \wedge\ r''_0 \in T \ \wedge\ ryxr'_1 \in S \ \wedge\ r''_1 \in T))$
$= \{\ cc.S \ \wedge\ cc.T\ \}$
$\quad (\exists(r', r'' : r'r'' = r : r' \in S \ \wedge\ r''yxr_0 \in T \ \wedge\ r''xyr_1 \in T)$
$\quad\ \exists(r'_0, r''_0, r'_1, r''_1 : r'_0 r''_0 = r_0 \ \wedge\ r'_1 r''_1 = r_1 :$
$\quad\quad ryxr'_0 \in S \ \wedge\ r''_0 \in T \ \wedge\ rxyr'_1 \in S \ \wedge\ r''_1 \in T))$
$= \{\ ;\ \}$
$\quad rxyr_1 \in (S;\ T) \ \wedge\ ryxr_0 \in (S;\ T)$

end of proof

$$(3.48) \qquad (cc.S \ \wedge\ pr.S \ \wedge\ cc.T) \Rightarrow cc.(S;\ T)$$

Proof
Immediate from previous two lemmata.
end of proof

$$(3.49) \qquad (pr.S \ \wedge\ cc.S) \Rightarrow \forall(z :: pr.(S \ \mathbf{connect}\ \ z))$$

Proof by contraposition.
By induction on $|s|$.
Base case $|s| = 0$, hence $s = \epsilon$

$\quad rs \in S \ \wedge\ rt \in S \ \wedge\ rs \neq rt \ \wedge\ (rs\ \mathbf{connect}\ \ z) < (rt\ \mathbf{connect}\ \ z) \ \wedge\ s = \epsilon$
$\Rightarrow \{$ Substitute ϵ for s $\}$
$\quad r \in S \ \wedge\ rt \in S \ \wedge\ r \neq rt$
$\Rightarrow \{$ Calculus $\}$
$\quad r \in S \ \wedge\ rt \in S \ \wedge\ r < rt$
$\Rightarrow \{$ Definition pr $\}$
$\quad \neg pr.S$

Induction step

$$
\begin{array}{l}
\quad rs \in S \ \wedge \ rt \in S \ \wedge \ rs \neq rt \ \wedge \ (rs \ \mathbf{connect} \ \ z) < (rt \ \mathbf{connect} \ \ z) \\
= \{ \text{ cases } \} \\
\quad rs \in S \ \wedge \ rt \in S \ \wedge \ rs \neq rt \ \wedge \ (rs \ \mathbf{connect} \ \ z) < (rt \ \mathbf{connect} \ \ z) \\
\quad \wedge ((hd.s = z \ \wedge \ hd.t = z) \ \vee \ (hd.s = z \ \wedge \ hd.t \neq z) \ \vee \ (hd.s \neq z \ \wedge \ hd.t = z)) \\
= \{ \text{ cases } \} \\
\quad rs \in S \ \wedge \ rt \in S \ \wedge \ rs \neq rt \ \wedge \ (rs \ \mathbf{connect} \ \ z) < (rt \ \mathbf{connect} \ \ z) \wedge \\
\quad (\exists (r', s', t' :: s = zs' \ \wedge \ t = zt' \ \wedge \ r' = rz : \\
\qquad r's' \in S \ \wedge \ r't' \in S \ \wedge \ r's' \neq r't' \ \wedge \ (r's' \ \mathbf{connect} \ \ z) < (r't' \ \mathbf{connect} \ \ z)) \vee \\
\quad \ \exists (n, s', t' :: s = z^n hd.t \ s' \ \wedge \ t = hd.t \ t') \vee \\
\quad \ \exists (n, s', t' :: s = hd.s \ s' \ \wedge \ t = z^n hd.s \ t')) \\
\Rightarrow \{ \text{ Ind. hyp. first case, lemma C.4 other two } \} \\
\quad rs \in S \ \wedge \ rt \in S \ \wedge \ rs \neq rt \ \wedge \ (rs \ \mathbf{connect} \ \ z) < (rt \ \mathbf{connect} \ \ z) \wedge \\
\quad (\neg pr.S \vee \\
\quad \ \exists (n, s', t' :: s = z^n hd.t \ s' \ \wedge \ t = hd.t \ t' \ \wedge \ r \ hd.t \ z^n s' \in S) \vee \\
\quad \ \exists (n, s', t' :: s = hd.s \ s' \ \wedge \ t = z^n hd.s \ t' \ \wedge \ r \ hd.s \ z^n t' \in S)) \\
\Rightarrow \{ \ \ \} \\
\quad \neg pr.S \vee \\
\quad \ \exists (n, r', s'', t' :: \\
\qquad s = hd.t \ s'' \ \wedge \ t = hd.t \ t' \ \wedge \ r' = r \ hd.t \ \wedge \ r's'' \in S \ \wedge \ r't' \in S \ \wedge \\
\qquad s'' \neq t' \ \wedge \ (r's'' \ \mathbf{connect} \ \ z) < (r't' \ \mathbf{connect} \ \ z)) \vee \\
\quad \ \exists (n, r', s', t'' :: \\
\qquad s = hd.s \ s' \ \wedge \ t = hd.t \ t'' \ \wedge \ r' = r \ hd.t \ \wedge \ r's' \in S \ \wedge \ r't'' \in S \ \wedge \\
\qquad s' \neq t'' \ \wedge \ (r's' \ \mathbf{connect} \ \ z) < (r't'' \ \mathbf{connect} \ \ z)) \\
= \{ \text{ Ind.hyp. } \} \\
\quad \neg pr.S
\end{array}
$$

end of proof

Appendix D

Proofs for Section 4.2

(4.3) $\quad \mathcal{S} \sqsupseteq \mathcal{S}$

Proof

$$
\begin{array}{l}
\quad \mathcal{S} \sqsupseteq \mathcal{S} \\
= \{ \text{ Definition } \} \\
\quad \forall(S : S \in \mathcal{S} : \exists(T : T \in \mathcal{S} : S \supseteq T)) \\
= \{ \text{ Choose } T \text{ equal to } S \ \} \\
\quad \mathit{true}
\end{array}
$$

end of proof

(4.4) $\quad \mathcal{R} \sqsupseteq \mathcal{S} \ \wedge \ \mathcal{S} \sqsupseteq \mathcal{T} \Rightarrow \mathcal{R} \sqsupseteq \mathcal{T}$

Proof

$$
\begin{array}{l}
\quad \mathcal{R} \sqsupseteq \mathcal{S} \ \wedge \ \mathcal{S} \sqsupseteq \mathcal{T} \\
= \{ \text{ Definition } \sqsupseteq \ \} \\
\quad \forall(R : R \in \mathcal{R} : \exists(S : S \in \mathcal{S} : R \supseteq S)) \wedge \\
\quad \forall(S : S \in \mathcal{S} : \exists(T : T \in \mathcal{T} : S \supseteq T)) \\
= \{ \text{ Calculus } \} \\
\quad \forall(R : R \in \mathcal{R} : \exists(S, T : S \in \mathcal{S} \ \wedge \ T \in \mathcal{T} : R \supseteq S \ \wedge \ S \supseteq T)) \\
\Rightarrow \{ \text{ Calculus, transitivity set inclusion } \} \\
\quad \forall(R : R \in \mathcal{R} : \exists(T : T \in \mathcal{T} : R \supseteq T)) \\
= \{ \text{ Definition } \sqsupseteq \ \} \\
\quad \mathcal{R} \sqsupseteq \mathcal{T}
\end{array}
$$

end of proof

$$
\begin{array}{l}
(4.5) \qquad \mathcal{S} \sqsupseteq \mathcal{T} \ \wedge\ \mathcal{T} \sqsupseteq \mathcal{S} = \\
\forall(S : S \in \mathcal{S} \ \wedge\ \forall(S0 : S0 \in \mathcal{S} \ \wedge\ S0 \subseteq S : S0 = S) : S \in \mathcal{T}) \wedge \\
\forall(T : T \in \mathcal{T} \ \wedge\ \forall(T0 : T0 \in \mathcal{T} \ \wedge\ T0 \subseteq T : T0 = T) : T \in \mathcal{S})
\end{array}
$$

Proof

$$
\begin{array}{ll}
& \mathcal{S} \sqsupseteq \mathcal{T} \ \wedge\ \mathcal{T} \sqsupseteq \mathcal{S} \\
= & \{ \text{ Definition } \} \\
& \forall(S : S \in \mathcal{S} : \exists(T : T \in \mathcal{T} : S \supseteq T)) \wedge \\
& \forall(T : T \in \mathcal{T} : \exists(S : S \in \mathcal{S} : T \supseteq S)) \\
= & \{ \text{ Duplicate, add conjunct equivalent to } \mathit{true}\text{, then split. } \} \\
& \forall(S : S \in \mathcal{S} \ \wedge\ \forall(S0 : S0 \in \mathcal{S} \ \wedge\ S0 \subseteq S : S0 = S) : \exists(T : T \in \mathcal{T} : S \supseteq T)) \wedge \\
& \forall(S : S \in \mathcal{S} \ \wedge\ \neg\forall(S0 : S0 \in \mathcal{S} \ \wedge\ S0 \subseteq S : S0 = S) : \exists(T : T \in \mathcal{T} : S \supseteq T)) \wedge \\
& \forall(T : T \in \mathcal{T} \ \wedge\ \forall(T0 : T0 \in \mathcal{T} \ \wedge\ T0 \subseteq T : T0 = T) : \exists(S : S \in \mathcal{S} : T \supseteq S)) \wedge \\
& \forall(T : T \in \mathcal{T} \ \wedge\ \neg\forall(T0 : T0 \in \mathcal{T} \ \wedge\ T0 \subseteq T : T0 = T) : \exists(S : S \in \mathcal{S} : T \supseteq S)) \wedge \\
& \forall(S : S \in \mathcal{S} : \exists(T : T \in \mathcal{T} : S \supseteq T)) \wedge \\
& \forall(T : T \in \mathcal{T} : \exists(S : S \in \mathcal{S} : T \supseteq S)) \\
= & \{ \text{ Calculus } \} \\
& \forall(S : S \in \mathcal{S} \ \wedge\ \forall(S0 : S0 \in \mathcal{S} \ \wedge\ S0 \subseteq S : S0 = S) \\
& \quad : \exists(T, S0 : T \in \mathcal{T} \ \wedge\ S0 \in \mathcal{S} : S \supseteq T \ \wedge\ T \supseteq S0)) \wedge \\
& \forall(S : S \in \mathcal{S} \ \wedge\ \exists(S0 : S0 \in \mathcal{S} \ \wedge\ S0 \subseteq S : S0 \neq S) : \exists(T : T \in \mathcal{T} : S \supseteq T)) \wedge \\
& \forall(T : T \in \mathcal{T} \ \wedge\ \forall(T0 : T0 \in \mathcal{T} \ \wedge\ T0 \subseteq T : T0 = T) \\
& \quad : \exists(S, T0 : S \in \mathcal{S} \ \wedge\ T \in \mathcal{T} : T \supseteq S)) \wedge \\
& \forall(T : T \in \mathcal{T} \ \wedge\ \exists(T0 : T0 \in \mathcal{T} \ \wedge\ T0 \subseteq T : T0 \neq T) : \exists(S : S \in \mathcal{S} : T \supseteq S)) \\
= & \{ \text{ Assume if there exists one, there exists a smallest. } \} \\
& \forall(S : S \in \mathcal{S} \ \wedge\ \forall(S0 : S0 \in \mathcal{S} \ \wedge\ S0 \subseteq S : S0 = S) : S \in \mathcal{T}) \wedge \\
& \forall(S : S \in \mathcal{S} \ \wedge\ \exists(S0 : S0 \in \mathcal{S} \ \wedge\ S0 \subseteq S\ \wedge \\
& \qquad \forall(S1 : S1 \in \mathcal{S} \ \wedge\ S1 \subseteq S0 : S1 = S0) : S0 \neq S) : \exists(T : T \in \mathcal{T} : S \supseteq T)) \wedge \\
& \forall(T : T \in \mathcal{T} \ \wedge\ \forall(T0 : T0 \in \mathcal{T} \ \wedge\ T0 \subseteq T : T0 = T) : T \in \mathcal{S}) \wedge \\
& \forall(T : T \in \mathcal{T} \ \wedge\ \exists(T0 : T0 \in \mathcal{T} \ \wedge\ T0 \subseteq T\ \wedge \\
& \qquad \forall(T1 : T1 \in \mathcal{T} \ \wedge\ T1 \subseteq T0 : T1 = T0) : T0 \neq T) : \exists(S : S \in \mathcal{S} : T \supseteq S)) \\
= & \{ \text{ Calculus } \} \\
& \forall(S : S \in \mathcal{S} \ \wedge\ \forall(S0 : S0 \in \mathcal{S} \ \wedge\ S0 \subseteq S : S0 = S) : S \in \mathcal{T}) \wedge \\
& \forall(S : S \in \mathcal{S} \ \wedge\ \exists(S0 : S0 \in \mathcal{S} \ \wedge\ S0 \subseteq S \ \wedge\ S0 \in \mathcal{T} : S1 = S0) : S0 \neq S) \\
& \quad : \exists(T : T \in \mathcal{T} : S \supseteq T)) \wedge
\end{array}
$$

$$\forall(T : T \in \mathcal{T} \;\wedge\; \forall(T0 : T0 \in \mathcal{T} \;\wedge\; T0 \subseteq T : T0 = T) : T \in \mathcal{S}) \wedge$$
$$\forall(T : T \in \mathcal{T} \;\wedge\; \exists(T0 : T0 \in \mathcal{T} \;\wedge\; T0 \subseteq T \;\wedge\; T0 \in \mathcal{S} : T1 = T0) : T0 \neq T)$$
$$: \exists(S : S \in \mathcal{S} : T \supseteq S))$$
$= \{$ Calculus $\}$
$$\forall(S : S \in \mathcal{S} \;\wedge\; \forall(S0 : S0 \in \mathcal{S} \;\wedge\; S0 \subseteq S : S0 = S) : S \in \mathcal{T}) \wedge$$
$$\forall(T : T \in \mathcal{T} \;\wedge\; \forall(T0 : T0 \in \mathcal{T} \;\wedge\; T0 \subseteq T : T0 = T) : T \in \mathcal{S})$$

end of proof

(4.7) $\quad \mathcal{S} \simeq \mathcal{S}$

Proof

$\mathcal{S} \simeq \mathcal{S}$
$= \{$ Definition $\simeq$ $\}$
$\mathcal{S} \sqsupseteq \mathcal{S} \;\wedge\; \mathcal{S} \sqsupseteq \mathcal{S}$
$= \{$ Reflexivity of $\sqsupseteq$ $\}$
$true$

end of proof

(4.8) $\quad (\mathcal{S} \simeq \mathcal{T}) = (\mathcal{T} \simeq \mathcal{S})$

Proof

$\mathcal{S} \simeq \mathcal{T}$
$= \{$ Definition $\simeq$ $\}$
$\mathcal{S} \sqsupseteq \mathcal{T} \;\wedge\; \mathcal{T} \sqsupseteq \mathcal{S}$
$= \{$ Symmetry of $\wedge$, definition $\simeq$ $\}$
$\mathcal{T} \simeq \mathcal{S}$

end of proof

(4.9) $\quad \mathcal{R} \simeq \mathcal{S} \;\wedge\; \mathcal{S} \simeq \mathcal{T} \Rightarrow \mathcal{R} \simeq \mathcal{T}$

Proof

$$
\begin{array}{ll}
& \mathcal{R} \simeq \mathcal{S} \ \wedge \ \mathcal{S} \simeq \mathcal{T} \\
= & \{ \text{ Definition } \simeq \} \\
& \mathcal{R} \sqsupseteq \mathcal{S} \ \wedge \ \mathcal{S} \sqsupseteq \mathcal{T} \ \wedge \mathcal{T} \sqsupseteq \mathcal{S} \ \wedge \ \mathcal{S} \sqsupseteq \mathcal{R} \\
\Rightarrow & \{ \text{ Transitivity } \sqsupseteq \} \\
& \mathcal{R} \sqsupseteq \mathcal{T} \ \wedge \ \mathcal{T} \sqsupseteq \mathcal{R} \\
= & \{ \text{ Definition } \simeq \} \\
& \mathcal{R} \simeq \mathcal{T}
\end{array}
$$

end of proof

$$
\begin{array}{l}
(4.10) \qquad \qquad \mathcal{S} \\
\simeq \\
\qquad \{S : S \in \mathcal{S} \ \wedge \ \forall(S0 : S0 \in \mathcal{S} \ \wedge \ S0 \subseteq S : S0 = S) : S\}
\end{array}
$$

Proof

$$
\begin{array}{ll}
& \neg(\mathcal{S} \sqsupseteq \{S : S \in \mathcal{S} \ \wedge \ \forall(S0 : S0 \in \mathcal{S} \ \wedge \ S0 \subseteq S : S0 = S) : S\}) \\
= & \{ \text{ Definition } \sqsupseteq \} \\
& \neg(\forall(S1 : S1 \in \mathcal{S} : \\
& \qquad \exists(S2 : S2 \in \{S : S \in \mathcal{S} \ \wedge \ \forall(S0 : S0 \in \mathcal{S} \ \wedge \ S0 \subseteq S : S0 = S) : S\} : \\
& \qquad \qquad S1 \supseteq S2))) \\
= & \{ \text{ Calculus } \} \\
& \neg(\forall(S1 : S1 \in \mathcal{S} : \\
& \qquad \exists(S2 : S2 \in \mathcal{S} \ \wedge \ \forall(S0 : S0 \in \mathcal{S} \ \wedge \ S0 \subseteq S2 : S0 = S2) : S1 \supseteq S2))) \\
= & \{ \text{ Shunting } \} \\
& \neg(\forall(S1 : S1 \in \mathcal{S} : \\
& \qquad \exists(S2 : S2 \in \mathcal{S} : S1 \supseteq S2 \ \wedge \ \forall(S0 : S0 \in \mathcal{S} \ \wedge \ S0 \subseteq S2 : S0 = S2)))) \\
= & \{ \text{ Calculus } \} \\
& \exists(S1 : S1 \in \mathcal{S} : \\
& \qquad \forall(S2 : S2 \in \mathcal{S} : \neg(S1 \supseteq S2) \ \vee \ \exists(S0 : S0 \in \mathcal{S} \ \wedge \ S0 \subseteq S2 : S0 \neq S2))) \\
= & \{ \text{ Second term independent of } S1 \ \} \\
& \exists(S1 : S1 \in \mathcal{S} : \forall(S2 : S2 \in \mathcal{S} : \neg(S1 \supseteq S2))) \vee \\
& \exists(S1 : S1 \in \mathcal{S} : \forall(S2 : S2 \in \mathcal{S} : \exists(S0 : S0 \in \mathcal{S} \ \wedge \ S0 \subseteq S2 : S0 \neq S2))) \\
= & \{ \text{ Calculus, first disjunct is } \mathit{false}\text{; shunting } \} \\
& \mathcal{S} = \{\} \ \vee \ \forall(S2 : S2 \in \mathcal{S} : \exists(S0 : S0 \in \mathcal{S} : S0 \subseteq S2 \ \wedge \ S0 \neq S2)) \\
= & \{ \text{ Calculus } \} \\
& \mathcal{S} = \{\} \ \vee \ \forall(S2 : S2 \in \mathcal{S} : \exists(S0 : S0 \in \mathcal{S} : S0 \subset S2))
\end{array}
$$

$= \{ \; \}$
$\quad false$

Also,

$$
\begin{array}{l}
\quad \{S : S \in \mathcal{S} \;\wedge\; \forall(S0 : S0 \in S \;\wedge\; S0 \subseteq S : S0 = S) : S\} \sqsupseteq \mathcal{S} \\
= \{ \text{ Definition } \sqsupseteq \} \\
\quad \forall(S0 : S0 \in \{S : S \in \mathcal{S} \;\wedge\; \forall(S0 : S0 \in S \;\wedge\; S0 \subseteq S : S0 = S) : S\} : \\
\qquad \exists(S1 : S1 \in \mathcal{S} : S0 \subseteq S1)) \\
\Leftarrow \{ \text{ Calculus } \} \\
\quad \forall(S0 : S0 \in \{S : S \in \mathcal{S} : S\} : \exists(S1 : S1 \in \mathcal{S} : S0 \subseteq S1)) \\
= \{ \text{ Calculus, definition } \sqsupseteq \} \\
\quad \mathcal{S} \sqsupseteq \mathcal{S} \\
= \{ \text{ Theorem (4.3) } \} \\
\quad true
\end{array}
$$

end of proof

$$
\begin{array}{ll}
(4.11) & (\mathcal{S} \simeq \mathcal{T}) \\
= & \\
& (\{S : S \in \mathcal{S} \;\wedge\; \forall(S0 : S0 \in \mathcal{S} \;\wedge\; S0 \subseteq S : S0 = S) : S\} \\
& = \\
& \{T : T \in \mathcal{T} \;\wedge\; \forall(T0 : T0 \in \mathcal{T} \;\wedge\; T0 \subseteq T : T0 = T) : T\})
\end{array}
$$

Proof

We first prove $\Rightarrow$:

$$
\begin{array}{l}
\quad S1 \in \{S : S \in \mathcal{S} \;\wedge\; \forall(S0 : S0 \in \mathcal{S} \;\wedge\; S0 \subseteq S : S0 = S) : S\} \;\wedge\; \mathcal{S} \simeq \mathcal{T} \\
= \{ \text{ Calculus, definition } \sqsupseteq \} \\
\quad S1 \in \mathcal{S} \;\wedge\; \forall(S0 : S0 \in \mathcal{S} \;\wedge\; S0 \subseteq S1 : S0 = S1) \wedge \\
\quad \forall(S2 : S2 \in \mathcal{S} : \exists(T2 : T2 \in \mathcal{T} : S2 \supseteq T2)) \wedge \\
\quad \forall(T3 : T3 \in \mathcal{T} : \exists(S3 : S3 \in \mathcal{S} : T3 \supseteq S3)) \\
\Rightarrow \{ \text{ Choose } S1 \text{ for } S2 \} \\
\quad S1 \in \mathcal{S} \;\wedge\; \forall(S0 : S0 \in \mathcal{S} \;\wedge\; S0 \subseteq S1 : S0 = S1) \wedge \\
\quad \exists(T2 : T2 \in \mathcal{T} : S1 \supseteq T2) \wedge \\
\quad \forall(T3 : T3 \in \mathcal{T} : \exists(S3 : S3 \in \mathcal{S} : T3 \supseteq S3)) \\
= \{ \text{ If there exists one, there exists a smallest } \}
\end{array}
$$

$$
\begin{aligned}
& S1 \in \mathcal{S} \ \wedge \ \forall(S0 : S0 \in \mathcal{S} \ \wedge \ S0 \subseteq S1 : S0 = S1) \wedge \\
& \exists(T2 : T2 \in \mathcal{T} \ \wedge \ \forall(T0 : T0 \in \mathcal{T} : T0 \subseteq T2 : T0 = T2) : S1 \supseteq T2) \wedge \\
& \forall(T3 : T3 \in \mathcal{T} : \exists(S3 : S3 \in \mathcal{S} : T3 \supseteq S3)) \\
\Rightarrow & \{ \text{ Use last conjunct, choose } T2 \text{ for } T3 \ \} \\
& S1 \in \mathcal{S} \ \wedge \ \forall(S0 : S0 \in \mathcal{S} \ \wedge \ S0 \subseteq S1 : S0 = S1) \wedge \\
& \exists(T2 : T2 \in \mathcal{T} \ \wedge \ \forall(T0 : T0 \in \mathcal{T} : T0 \subseteq T2 : T0 = T2) : \\
& \qquad\qquad S1 \supseteq T2 \ \wedge \ \exists(S3 : S3 \in \mathcal{S} : T2 \supseteq S3)) \\
= & \{ \text{ Calculus } \} \\
& S1 \in \mathcal{S} \ \wedge \ \forall(S0 : S0 \in \mathcal{S} \ \wedge \ S0 \subseteq S1 : S0 = S1) \wedge \\
& \exists(T2, S3 : T2 \in \mathcal{T} \ \wedge \ \forall(T0 : T0 \in \mathcal{T} : T0 \subseteq T2 : T0 = T2) : \\
& \qquad\qquad S1 \supseteq T2 \ \wedge \ T2 \supseteq S3) \\
\Rightarrow & \{ \text{ Use second conjunct } \} \\
& S1 \in \mathcal{S} \wedge \\
& \exists(T2, S3 : T2 \in \mathcal{T} \ \wedge \ \forall(T0 : T0 \in \mathcal{T} : T0 \subseteq T2 : T0 = T2) : \\
& \qquad\qquad S1 \supseteq T2 \ \wedge \ T2 \supseteq S1) \\
= & \{ \text{ Set inclusion is antisymmetric } \} \\
& S1 \in \mathcal{S} \wedge \\
& \exists(T2 : T2 \in \mathcal{T} \ \wedge \ \forall(T0 : T0 \in \mathcal{T} : T0 \subseteq T2 : T0 = T2) : S1 = T2) \\
\Rightarrow & \{ \text{ Calculus } \} \\
& S1 \in \{T2 : T2 \in \mathcal{T} \ \wedge \ \forall(T0 : T0 \in \mathcal{T} : T0 \subseteq T2 : T0 = T2) : T2\}
\end{aligned}
$$

The other half of the proof for $\Rightarrow$ follows by symmetry.
Proof of $\Leftarrow$:

$$
\begin{aligned}
& \mathcal{S} \sqsupseteq \mathcal{T} \\
= & \{ \text{ Definition } \sqsupseteq \ \} \\
& \forall(S : S \in \mathcal{S} : \exists(T : T \in \mathcal{T} : S \supseteq T)) \\
\Leftarrow & \{ \text{ Calculus } \} \\
& \forall(S : S \in \mathcal{S} : \\
& \quad \exists(T : T \in \mathcal{T} \ \wedge \ \forall(T0 : T0 \in \mathcal{T} \ \wedge \ T0 \subseteq T : T0 = T) : S \supseteq T)) \\
\Leftarrow & \{ \text{ Calculus } \} \\
& \forall(S : S \in \mathcal{S} \ \wedge \ \forall(S0 : S0 \in \mathcal{S} \ \wedge \ S0 \subseteq S : S0 = S) : \\
& \quad \exists(T : T \in \mathcal{T} \ \wedge \ \forall(T0 : T0 \in \mathcal{T} \ \wedge \ T0 \subseteq T : T0 = T) : S \supseteq T)) \\
\Leftarrow & \{ \text{ Calculus } \} \\
& \{S : S \in \mathcal{S} \ \wedge \ \forall(S0 : S0 \in \mathcal{S} \ \wedge \ S0 \subseteq S : S0 = S) : S\} \\
& = \ \{T : T \in \mathcal{T} \ \wedge \ \forall(T0 : T0 \in \mathcal{T} \ \wedge \ T0 \subseteq T : T0 = T) : T\}
\end{aligned}
$$

$\mathcal{T} \sqsupseteq \mathcal{S}$ follows by symmetry.

end of proof

$$(4.14) \qquad \mathcal{S} = \sqcap(S : S \in \mathcal{S} : \sqcup(s : s \in S : \{\{s\}\}))$$

Proof

$$
\begin{array}{l}
\quad \sqcap(S : S \in \mathcal{S} : \sqcup(s : s \in S : \{\{s\}\})) \\
= \{ \text{ Definition } \sqcup \ \} \\
\quad \sqcap(S : S \in \mathcal{S} : \{\cup(s : s \in S : \{s\})\}) \\
= \{ \text{ Calculus } \} \\
\quad \sqcap(S : S \in \mathcal{S} : \{S\}) \\
= \{ \text{ Definition } \sqcap \ \} \\
\quad \cup(S : S \in \mathcal{S} : \{S\}) \\
= \{ \text{ Calculus } \} \\
\quad \mathcal{S}
\end{array}
$$

end of proof

$$(4.15) \qquad \forall(\mathcal{S}, \mathcal{T} :: \mathcal{S} \sqsupseteq \mathcal{S} \sqcap \mathcal{T})$$

Proof

$$
\begin{array}{l}
\quad \mathcal{S} \sqsupseteq \mathcal{S} \sqcap \mathcal{T} \\
= \{ \text{ Definition } \sqsupseteq, \sqcap \ \} \\
\quad \forall(S : S \in \mathcal{S} : \exists(T : T \in \mathcal{S} \cup \mathcal{T} : S \supseteq T)) \\
\Leftarrow \{ \text{ Calculus } \} \\
\quad \forall(S : S \in \mathcal{S} : \exists(T : T \in \mathcal{S} \cup \mathcal{T} : S \supseteq T)) \\
\Leftarrow \{ \text{ Choose } S \text{ for } T \ \} \\
\quad \mathit{true}
\end{array}
$$

end of proof

$$(4.16) \qquad \forall(\mathcal{R}, \mathcal{S}, \mathcal{T} : \mathcal{S} \sqsupseteq \mathcal{R} \ \wedge \ \mathcal{T} \sqsupseteq \mathcal{R} \Rightarrow \mathcal{S} \sqcap \mathcal{T} \sqsupseteq \mathcal{R})$$

Proof

$$
\begin{array}{ll}
& \mathcal{S} \sqsupseteq \mathcal{R} \ \wedge \ \mathcal{T} \sqsupseteq \mathcal{R} \\
= & \{ \text{ Definition } \sqsupseteq \} \\
& \forall(S : S \in \mathcal{S} : \exists(R : R \in \mathcal{R} : S \supseteq R)) \wedge \\
& \forall(T : T \in \mathcal{T} : \exists(R : R \in \mathcal{R} : T \supseteq R)) \\
= & \{ \text{ Calculus } \} \\
& \forall(S : S \in \mathcal{S} \cup \mathcal{T} : \exists(R : R \in \mathcal{R} : S \supseteq R)) \\
= & \{ \text{ Definition } \sqcap \} \\
& \forall(S : S \in \mathcal{S} \sqcap \mathcal{T} : \exists(R : R \in \mathcal{R} : S \supseteq R)) \\
= & \{ \text{ Definition } \sqsupseteq \} \\
& (\mathcal{S} \sqcap \mathcal{T}) \sqsupseteq \mathcal{R}
\end{array}
$$

end of proof

(4.17) $\qquad \forall(\mathcal{S}, \mathcal{T} :: \mathcal{S} \sqcup \mathcal{T} \sqsupseteq \mathcal{S})$

Proof

$$
\begin{array}{ll}
& \mathcal{S} \sqcup \mathcal{T} \sqsupseteq \mathcal{S} \\
= & \{ \text{ Definition } \sqsupseteq, \sqcap \} \\
& \forall(S : S \in \{S', T' : S' \in \mathcal{S} \ \wedge \ T' \in \mathcal{T} : S' \cup T'\} : \\
& \qquad \exists(S'' : S'' \in \mathcal{S} : S \supseteq S'')) \\
= & \{ \text{ Calculus } \} \\
& \forall(S, T : S \in \mathcal{S} \ \wedge \ T \in \mathcal{T} : \exists(S'' : S'' \in (\mathcal{S} \cup \mathcal{T} : S \cup T \supseteq S'')) \\
\Leftarrow & \{ \text{ Choose } S \text{ for } S'' \} \\
& \mathit{true}
\end{array}
$$

end of proof

(4.18) $\qquad \forall(\mathcal{R}, \mathcal{S}, \mathcal{T} : \mathcal{R} \sqsupseteq \mathcal{S} \ \wedge \ \mathcal{R} \sqsupseteq \mathcal{T} \Rightarrow \mathcal{R} \sqsupseteq \mathcal{S} \sqcup \mathcal{T})$

Proof

$$
\begin{array}{ll}
& \mathcal{R} \sqsupseteq \mathcal{S} \ \wedge \ \mathcal{R} \sqsupseteq \mathcal{T} \\
= & \{ \text{ Definition } \sqsupseteq, \text{Calculus } \} \\
& \forall(R : R \in \mathcal{R} : \exists(S : S \in \mathcal{S} : R \supseteq S)) \wedge \\
& \forall(R : R \in \mathcal{R} : \exists(T : T \in \mathcal{T} : R \supseteq T)) \\
\Rightarrow & \{ \text{ Calculus } \}
\end{array}
$$

$$\forall(R : R \in \mathcal{R} : \exists(S, T : S \in \mathcal{S} \ \wedge \ T \in \mathcal{T} : R \supseteq S \ \wedge \ R \supseteq T))$$
$= \{$ Calculus $\}$
$$\forall(R : R \in \mathcal{R} : \exists(S, T : S \in \mathcal{S} \ \wedge \ T \in \mathcal{T} : R \supseteq S \cup T))$$
$= \{$ Calculus $\}$
$$\forall(R : R \in \mathcal{R} : \exists(S' : S' \in \{S, T : S \in \mathcal{S} \ \wedge \ T \in \mathcal{T} : S \cup T\} : R \supseteq S'))$$
$= \{$ Definition $\sqsupseteq, \sqcup$ $\}$
$$\mathcal{R} \sqsupseteq (\mathcal{S} \sqcup \mathcal{T})$$

end of proof

(4.19) $\quad \mathcal{S} \sqcap \mathcal{S} \simeq \mathcal{S} \wedge \mathcal{S} \sqcup \mathcal{S} \simeq S$

Proof

$$\mathcal{S} \sqcap \mathcal{S}$$
$= \{$ Definition $\sqcap$ $\}$
$$\mathcal{S} \cup \mathcal{S}$$
$= \{$ Calculus $\}$
$$\mathcal{S}$$

$$\mathcal{S} \sqcup \mathcal{S}$$
$= \{$ Definition $\sqcup$ $\}$
$$\{S, T : S \in \mathcal{S} \ \wedge \ T \in \mathcal{S} : S \cup T\}$$
$\simeq \{$ $S \neq T \Rightarrow S \cup T \supseteq S$, theorem (4.10) $\}$
$$\{S : S \in \mathcal{S} : S\}$$
$= \{$ Calculus $\}$
$$\mathcal{S}$$

end of proof

(4.20) $\quad \mathcal{S} \sqcap \mathcal{T} \simeq \mathcal{T} \sqcap \mathcal{S} \ \wedge \ \mathcal{S} \sqcup \mathcal{T} \simeq \mathcal{T} \sqcup \mathcal{S}$

Proof

$$
\begin{array}{ll}
\quad \mathcal{S} \sqcap \mathcal{T} & \quad \mathcal{S} \sqcup \mathcal{T} \\
= \{ \text{ Definition } \sqcap \} & = \{ \text{ Definition } \sqcup \} \\
\quad \mathcal{S} \cup \mathcal{T} & \quad \{S, T : S \in \mathcal{S} \ \wedge \ T \in \mathcal{T} : S \cup T\} \\
= \{ \text{ Symmetry set union } \} & = \{ \text{ Symmetry set union, quantification } \} \\
\quad \mathcal{T} \cup \mathcal{S} & \quad \{T, S : T \in \mathcal{T} \ \wedge \ S \in \mathcal{S} : T \cup S\} \\
= \{ \text{ Definition } \sqcap \} & = \{ \text{ Definition } \sqcup \} \\
\quad \mathcal{T} \sqcap \mathcal{S} & \quad \mathcal{T} \sqcup \mathcal{S}
\end{array}
$$

end of proof

$$(4.21) \qquad (\mathcal{S} \sqcup \mathcal{T}) \sqcup \mathcal{R} \simeq \mathcal{S} \sqcup (\mathcal{T} \sqcup \mathcal{R})$$

Proof

$$
\begin{array}{l}
\quad (\mathcal{S} \sqcup \mathcal{T}) \sqcup \mathcal{R} \\
= \{ \text{ Definition } \sqcup \text{, twice } \} \\
\quad \{S, R : S \in \{S', T : S' \in \mathcal{S} \ \wedge \ T \in \mathcal{T} : S' \cup T\} \ \wedge \ R \in \mathcal{R} : S \cup R\} \\
= \{ \text{ Calculus, renaming dummy } S' \text{ to } S \ \} \\
\quad \{S, T, R : S \in \mathcal{S} \ \wedge \ T \in \mathcal{T} \ \wedge \ R \in \mathcal{R} : S \cup T \cup R\} \\
= \{ \text{ Symmetry, associativity of set union } \} \\
\quad \mathcal{S} \sqcup (\mathcal{T} \sqcup \mathcal{R})
\end{array}
$$

end of proof

$$(4.22) \qquad \mathcal{S} \sqcup (\mathcal{S} \sqcap \mathcal{T}) \simeq \mathcal{S} \wedge \mathcal{S} \sqcap (\mathcal{S} \sqcup \mathcal{T}) \simeq \mathcal{S}$$

Proof

$\mathcal{S} \sqcup (\mathcal{S} \sqcap \mathcal{T}) \sqsupseteq S$ follows from theorem (4.17).
$\mathcal{S} \sqsupseteq \mathcal{S} \sqcap (\mathcal{S} \sqcup \mathcal{T})$ follows from theorem (4.15).
We prove the two other directions.

$$
\begin{array}{l}
\quad \mathcal{S} \sqsupseteq \mathcal{S} \sqcup (\mathcal{S} \sqcap \mathcal{T}) \\
= \{ \text{ Definition } \sqsupseteq, \sqcup, \sqcap \ \} \\
\quad \forall (S : S \in \mathcal{S} : \\
\qquad \exists (T : T \in \{S0, T0 : S0 \in \mathcal{S} \ \wedge \ T0 \in (\mathcal{S} \cup \mathcal{T}) : S0 \cup T0\} : S \supseteq T)) \\
= \{ \text{ Calculus } \}
\end{array}
$$

$$
\begin{array}{l}
\quad \forall(S : S \in \mathcal{S} : \\
\quad\;\; \exists(S0, T0 : S0 \in \mathcal{S} \;\wedge\; (T0 \in \mathcal{S} \;\vee\; T0 \in \mathcal{T}) : S \supseteq (S0 \cup T0))) \\
\Leftarrow \{ \text{ Choose } S0, T0 \text{ equal to } S \text{ } \} \\
\quad \forall(S : S \in \mathcal{S} : S \supseteq (S \cup S)) \\
= \{ \text{ Calculus } \} \\
\quad \mathit{true}
\end{array}
$$

$$
\begin{array}{l}
\quad \mathcal{S} \sqcap (\mathcal{S} \sqcup \mathcal{T}) \sqsupseteq \mathcal{S} \\
= \{ \text{ Definition } \sqsupseteq, \sqcup, \sqcap \text{ } \} \\
\quad \forall(S : S \in \mathcal{S} \cup \{S0, T0 : S0 \in S \;\wedge\; T0 \in T : S0 \cup T0\} : \\
\quad\qquad \exists(S1 : S1 \in \mathcal{S} : S \supseteq S1)) \\
= \{ \text{ Calculus } \} \\
\quad \forall(S : S \in \mathcal{S} : \exists(S1 : S1 \in \mathcal{S} : S \supseteq S1)) \wedge \\
\quad \forall(S : S \in \{S0, T0 : S0 \in S \;\wedge\; T0 \in T : S0 \cup T0\} : \\
\quad\qquad \exists(S1 : S1 \in \mathcal{S} : S \supseteq S1)) \\
\Leftarrow \{ \text{ Choose } S1 \text{ equal to } S \text{ in first conjunct } \} \\
\quad \forall(S : S \in \mathcal{S} : S \supseteq S) \wedge \\
\quad \forall(S0, T0 : S0 \in S \;\wedge\; T0 \in T : \exists(S1 : S1 \in \mathcal{S} : (S0 \cup T0) \supseteq S1)) \\
\Leftarrow \{ \text{ Choose } S1 \text{ equal to } S0 \text{ } \} \\
\quad \mathit{true}
\end{array}
$$

end of proof

$$(4.23) \qquad (\mathcal{S} \simeq \mathcal{S} \sqcap \mathcal{T}) = (\mathcal{T} \sqsupseteq \mathcal{S}) = (\mathcal{S} \sqcup \mathcal{T} = \mathcal{T})$$

Proof

$$
\begin{array}{l}
\quad \mathcal{S} \simeq \mathcal{S} \sqcap \mathcal{T} \\
= \{ \text{ Definition } \simeq \text{ } \} \\
\quad (\mathcal{S} \sqsupseteq \mathcal{S} \sqcap \mathcal{T}) \;\wedge\; (\mathcal{S} \sqcap \mathcal{T} \sqsupseteq \mathcal{S}) \\
= \{ \text{ Theorem (4.15), definition } \sqsupseteq, \sqcap \text{ } \} \\
\quad \mathit{true} \;\wedge\; \forall(S : S \in \mathcal{S} \cup \mathcal{T} : \exists(S0 : S0 \in \mathcal{S} : S \supseteq S0)) \\
= \{ \text{ Calculus } \} \\
\quad \forall(S : S \in \mathcal{S} : \exists(S0 : S0 \in \mathcal{S} : S \supseteq S0)) \wedge \\
\quad \forall(T : T \in \mathcal{T} : \exists(S0 : S0 \in \mathcal{S} : T \supseteq S0)) \\
= \{ \text{ First conjunct } \mathit{true}, \text{ definition } \sqsupseteq \text{ } \} \\
\quad \mathcal{T} \sqsupseteq \mathcal{S}
\end{array}
$$

$$\begin{array}{ll}
& \mathcal{S} \sqcup \mathcal{T} \simeq \mathcal{T} \\
= & \{ \text{ Definition } \simeq \} \\
& (\mathcal{S} \sqcup \mathcal{T} \sqsupseteq \mathcal{T}) \wedge (\mathcal{T} \sqsupseteq \mathcal{S} \sqcup \mathcal{T}) \\
= & \{ \text{ Theorem (4.17), definition } \sqsupseteq, \sqcup \} \\
& \mathit{true} \wedge \forall(T : T \in \mathcal{T} : \exists(S0, T0 : S0 \in \mathcal{S} \wedge T0 \in \mathcal{T} : T \supseteq S0 \cup T0)) \\
= & \{ \text{ Choose } T \text{ for } T0 \text{, calculus } \} \\
& \forall(T : T \in \mathcal{T} : \exists(S0 : S0 \in \mathcal{S} : T \supseteq S0)) \\
= & \{ \text{ Definition } \sqsupseteq \} \\
& \mathcal{T} \sqsupseteq \mathcal{S}
\end{array}$$

end of proof

(4.24) $\forall(\mathcal{S} :: \{\} \sqsupseteq \mathcal{S})$

Proof

$$\begin{array}{ll}
& \{\} \sqsupseteq \mathcal{S} \\
= & \{ \text{ Definition } \sqsupseteq \} \\
& \forall(T : T \in \{\} : \exists(S : S \in \mathcal{S} : T \supseteq S)) \\
= & \{ \text{ Empty range } \} \\
& \mathit{true}
\end{array}$$

end of proof

(4.25) $\forall(\mathcal{S} :: \mathcal{S} \sqsupseteq \{\{\}\})$

Proof

$$\begin{array}{ll}
& \mathcal{S} \sqsupseteq \{\{\}\} \\
= & \{ \text{ Definition } \sqsupseteq \} \\
& \forall(S : S \in \mathcal{S} : \exists(T : T \in \{\{\}\} : S \supseteq T)) \\
= & \{ \text{ Calculus } \} \\
& \forall(S : S \in \mathcal{S} : S \supseteq \{\}) \\
= & \{ \text{ Calculus } \} \\
& \mathit{true}
\end{array}$$

end of proof

(4.26) Proof of distributivity :

Proof

$$
\begin{array}{l}
\quad \mathcal{S} \sqcap (\mathcal{T} \sqcup \mathcal{R}) \\
= \{ \text{ Definition } \sqcap \ \} \\
\quad \mathcal{S} \cup (\mathcal{T} \sqcup \mathcal{R}) \\
\simeq \{ \ \mathcal{S} \sqcup \mathcal{S} \simeq \mathcal{S}, \mathcal{S} \sqcap (\mathcal{S} \sqcup \mathcal{T}) \simeq \mathcal{S} \ \} \\
\quad (\mathcal{S} \sqcup \mathcal{S}) \cup (\mathcal{S} \sqcup \mathcal{T}) \cup (\mathcal{S} \sqcup \mathcal{R}) \cup (\mathcal{T} \sqcup \mathcal{R}) \\
= \{ \text{ Definition } \sqcup \ \} \\
\quad \{T, R : T \in \mathcal{S} \ \wedge \ R \in \mathcal{S} : T \cup R\} \cup \{T, R : T \in \mathcal{T} \ \wedge \ R \in \mathcal{S} : T \cup R\} \cup \\
\quad \{T, R : T \in \mathcal{S} \ \wedge \ R \in \mathcal{R} : T \cup R\} \cup \{T, R : T \in \mathcal{T} \ \wedge \ R \in \mathcal{R} : T \cup R\} \\
= \{ \text{ Calculus } \} \\
\quad \{T, R : T \in \mathcal{S} \cup \mathcal{T} \ \wedge \ R \in \mathcal{S} \cup \mathcal{R} : T \cup R\} \\
= \{ \text{ Definition } \sqcup, \sqcap \ \} \\
\quad (\mathcal{S} \sqcap \mathcal{T}) \sqcup (\mathcal{S} \sqcap \mathcal{R})
\end{array}
$$

$$
\begin{array}{l}
\quad \mathcal{S} \sqcup (\mathcal{T} \sqcap \mathcal{R}) \\
= \{ \text{ Definition } \sqcap \ \} \\
\quad \mathcal{S} \sqcup (\mathcal{T} \cup \mathcal{R}) \\
= \{ \text{ Definition } \sqcup \ \} \\
\quad \{S, T : S \in \mathcal{S} \ \wedge \ T \in \mathcal{S} \cup \mathcal{R} : S \cup T\} \\
= \{ \text{ Calculus } \} \\
\quad \{S, T : S \in \mathcal{S} \ \wedge \ T \in \mathcal{T} : S \cup T\} \cup \{S, R : S \in \mathcal{S} \ \wedge \ R \in \mathcal{R} : S \cup R\} \\
= \{ \text{ Definition } \sqcup, \sqcap \ \} \\
\quad (\mathcal{S} \sqcup \mathcal{T}) \sqcap (\mathcal{S} \sqcup \mathcal{R})
\end{array}
$$

end of proof

Appendix E

Proofs for Section 4.3

(4.32) $\mathcal{S} \sqsupseteq \mathcal{T} \Rightarrow \mathcal{S} \sqcap \mathcal{R} \sqsupseteq \mathcal{T} \sqcap \mathcal{R}$

Proof

$$
\begin{array}{l}
\quad (\mathcal{S} \sqcap \mathcal{R}) \sqsupseteq (\mathcal{T} \sqcap \mathcal{R}) \\
= \{ \text{ Definition } \sqsupseteq \} \\
\quad \forall(S : S \in (\mathcal{S} \sqcap \mathcal{R}) : \exists(T : T \in (\mathcal{T} \sqcap \mathcal{R}) : S \sqsupseteq T)) \\
= \{ \text{ Definition } \sqcap \} \\
\quad \forall(S : S \in (\mathcal{S} \cup \mathcal{R}) : \exists(T : T \in (\mathcal{T} \cup \mathcal{R}) : S \sqsupseteq T)) \\
\Leftarrow \{ \text{ Calculus } \} \\
\quad \forall(S : S \in \mathcal{S} : \exists(T : T \in \mathcal{T} : S \supseteq \mathcal{T})) \\
= \{ \text{ Definition } \sqsupseteq \} \\
\quad \mathcal{S} \sqsupseteq \mathcal{T}
\end{array}
$$

end of proof

(4.33) $\mathcal{S} \sqsupseteq \mathcal{T} \Rightarrow \mathcal{S} \sqcup \mathcal{R} \sqsupseteq \mathcal{T} \sqcup \mathcal{R}$

Proof

$$
\begin{array}{l}
\quad \mathcal{S} \sqcup \mathcal{R} \sqsupseteq \mathcal{T} \sqcup \mathcal{R} \\
= \{ \text{ Definitions } \sqsupseteq, \sqcup \} \\
\quad \forall(S : S \in \{S', R : S' \in \mathcal{S} \ \wedge \ R \in \mathcal{R} : S' \cup R\} : \\
\qquad \exists(T : T \in \{T', R : T' \in \mathcal{T} \ \wedge \ R \in \mathcal{R} : T' \cup R\} : S \supseteq T)) \\
= \{ \text{ Calculus } \}
\end{array}
$$

$$\forall(S, R : S \in \mathcal{S} \ \wedge \ R \in \mathcal{R} :$$
$$\quad \exists(T, R' : T \in \mathcal{T} \ \wedge \ R' \in \mathcal{R}' : S \cup R \supseteq T \cup R'))$$
$\Leftarrow$ { Calculus }
$$\forall(S : S \in \mathcal{S} : \exists(T : T \in \mathcal{T} : S \supseteq \mathcal{T}))$$
$=$ { Definition $\sqsupseteq$ }
$$\mathcal{S} \sqsupseteq \mathcal{T}$$

end of proof

(4.34) $\quad \mathcal{S} \sqsupseteq \mathcal{T} \Rightarrow \mathcal{S} \| \mathcal{R} \sqsupseteq \mathcal{T} \| \mathcal{R}$

Proof

$$\mathcal{S} \| \mathcal{R} \sqsupseteq \mathcal{T} \| \mathcal{R}$$
$=$ { Definitions $\sqsupseteq, \|$ }
$$\forall(S : S \in \{S', R : S' \in \mathcal{S} \ \wedge \ R \in \mathcal{R} : S' \| R\} :$$
$$\quad \exists(T : T \in \{T', R : T' \in \mathcal{T} \ \wedge \ R \in \mathcal{R} : T' \| R\} : S \supseteq T))$$
$=$ { Calculus }
$$\forall(S, R : S \in \mathcal{S} \ \wedge \ R \in \mathcal{R} : \exists(T, R' : T \in \mathcal{T} \ \wedge \ R' \in \mathcal{R}' : S \| R \supseteq T \| R'))$$
$\Leftarrow$ { Monotonicity $\|$ on tracesets }
$$\forall(S : S \in \mathcal{S} : \exists(T : T \in \mathcal{T} : S \supseteq \mathcal{T}))$$
$=$ { Definition $\sqsupseteq$ }
$$\mathcal{S} \sqsupseteq \mathcal{T}$$

end of proof

(4.35) $\quad \mathcal{R} \sqsupseteq \mathcal{S} \ \wedge \ \mathcal{T} \sqsupseteq \mathcal{U} \Rightarrow \mathcal{R}; \mathcal{T} \sqsupseteq \mathcal{S}; \mathcal{U}$

Proof

$$\mathcal{R}; \mathcal{T} \sqsupseteq \mathcal{S}; \mathcal{U}$$
$=$ { Definition $\sqsupseteq$ }
$$\forall(R : R \in \mathcal{R}; \mathcal{T} : \exists(S : S \in \mathcal{S}; \mathcal{U} : R \supseteq S))$$
$=$ { Definition ; }
$$\forall(R : R \in \sqcap(R' : R' \in \mathcal{R} : \sqcup(r : r \in R' : \sqcap(T \in \mathcal{T} : \sqcup(t \in T : \{\{rt\}\})))) :$$
$$\quad \exists(S : S \in \sqcap(S' : S' \in \mathcal{S} : \sqcup(s : s \in S : \sqcap(U \in \mathcal{U} : \sqcup(u \in U : \{\{su\}\}))))))$$
$\Leftarrow$ { }

$\quad \forall(R : R \in \mathcal{R} : \exists(S : S \in \mathcal{S} : R \supseteq S)) \wedge$
$\quad \forall(T : T \in \mathcal{T} : \exists(T : T \in \mathcal{T} : T \supseteq U))$
$=$ { Definition $\sqsupseteq$ }
$\quad \mathcal{R} \sqsupseteq \mathcal{S} \wedge \mathcal{T} \sqsupseteq \mathcal{U}$

end of proof

(4.36) $\quad \mathcal{S} \sqsupseteq \mathcal{T} \Rightarrow \mathcal{S}$ **connect** $X \sqsupseteq \mathcal{T}$ **connect** X

Proof

$\quad \mathcal{S}$ **connect** $X \sqsupseteq \mathcal{T}$ **connect** X
$=$ { Definition $\sqsupseteq$ }
$\quad \forall(S : S \in \mathcal{S}$ **connect** $X : \exists(T : T \in \mathcal{T}$ **connect** $X : S \supseteq T))$
$\Leftarrow$ { Monotonicity **connect** on tracesets }
$\quad \forall(S : S \in \mathcal{S} : \exists(T : T \in \mathcal{T} : S \supseteq T))$
$=$ { Definition $\sqsupseteq$ }
$\quad \mathcal{S} \sqsupseteq \mathcal{T}$

end of proof

(4.43) $\quad$ **magic** $\sqcap \mathcal{T} = \mathcal{T}$

Proof

$\quad$ **magic** $\sqcap \mathcal{T}$
$=$ { Definitions **magic** $, \sqcap$ }
$\quad \{\} \cup \mathcal{T}$
$=$ { Calculus }
$\quad \mathcal{T}$

end of proof

(4.44) $\quad$ **demon** $\sqcap \mathcal{T} \simeq$ **demon**

Proof

$$\begin{array}{ll} & \textbf{demon} \sqcap \mathcal{T} \\ = & \{ \text{ Definition } \sqcap \ \} \\ & \{\{\}\} \cup \mathcal{T} \\ \simeq & \{ \text{ Process equiv. to set of minimal elements, thm. (4.10) } \} \\ & \{\{\}\} \end{array}$$

end of proof

(4.45) $\quad \mathcal{T} \sqcap \mathcal{S} \ = \ \mathcal{S} \sqcap \mathcal{T}$

Proof Set union is commutative. **end of proof**

(4.46) $\quad (\mathcal{R} \sqcap \mathcal{S}) \sqcap \mathcal{T} \ = \ \mathcal{R} \sqcap (\mathcal{S} \sqcap \mathcal{T})$

Proof Set union is associative. **end of proof**

(4.47) $\quad \textbf{demon} \sqcup \mathcal{S} = \mathcal{S}$

Proof

$$\begin{array}{ll} & \textbf{demon} \sqcup \mathcal{S} \\ = & \{ \text{ Definitions } \sqcup, \textbf{demon} \ \} \\ & \{T, S : T \in \{\{\}\} \wedge S \in \mathcal{S} : S \cup T\} \\ = & \{ \text{ Calculus } \} \\ & \{S : S \in \mathcal{S} : S \cup \{\}\} \\ = & \{ \text{ Calculus } \} \\ & \{S : S \in \mathcal{S} : S\} \\ = & \{ \text{ Calculus } \} \\ & \mathcal{S} \end{array}$$

end of proof

(4.48) $\quad \textbf{magic} \sqcup \mathcal{S} \ = \ \textbf{magic}$

Proof

$$
\begin{array}{l}
\quad \textbf{magic} \sqcup \mathcal{S} \\
= \{ \text{ Definition } \sqcup, \textbf{magic} \ \} \\
\quad \{T, S : T \in \{\} \ \wedge \ S \in \mathcal{S} : T \cup S\} \\
= \{ \text{ Calculus } \} \\
\quad \{\} \\
= \{ \text{ Definition } \textbf{magic} \ \} \\
\quad \textbf{magic}
\end{array}
$$

end of proof

(4.49) $\quad \mathcal{T} \sqcup \mathcal{S} \ = \ \mathcal{S} \sqcup \mathcal{T}$

Proof

Immediate from the symmetry of the definition and commutativity of set union.

end of proof

(4.50) $\quad (\mathcal{R} \sqcup \mathcal{S}) \sqcup \mathcal{T} \ = \ \mathcal{R} \sqcup (\mathcal{S} \sqcup \mathcal{T})$

Proof

$$
\begin{array}{l}
\quad (\mathcal{R} \sqcup \mathcal{S}) \sqcup \mathcal{T} \\
= \{ \text{ Definition } \sqcup \text{, twice. } \} \\
\quad \{S, T : S \in \{R, S' : R \in \mathcal{R} \ \wedge \ S' \in \mathcal{S} : R \cup S'\} \ \wedge \ T \in \mathcal{T} : S \cup T\} \\
= \{ \text{ Calculus, renaming dummy } S' \text{ to } S \ \} \\
\quad \{R, S, T : R \in \mathcal{R} \ \wedge \ S \in \mathcal{S} \ \wedge \ T \in \mathcal{T} : R \cup S \cup T\} \\
= \{ \text{ Symmetry, associativity of set union. } \} \\
\quad \mathcal{R} \sqcup (\mathcal{S} \sqcup \mathcal{T})
\end{array}
$$

end of proof

(4.51) $\quad \textbf{skip} \, \| \mathcal{S} \simeq S \simeq S \| \textbf{skip}$

Proof (left unit element)

$$
\begin{aligned}
& \{\{\epsilon\}\}\|\mathcal{T} \\
= \{ & \text{ Definition } \} \\
& \{S, T : S \in \{\{\epsilon\}\}, T \in \mathcal{T} : S\|T\} \\
= \{ & \text{ Calculus } \} \\
& \{T : T \in \mathcal{T} : \{\epsilon\}\|T\ \} \\
= \{ & \text{ Theorem 3.29 } \} \\
& \{T : T \in \mathcal{T} : T\} \\
= \{ & \text{ Calculus } \} \\
& \mathcal{T}
\end{aligned}
$$

end of proof

(4.52) $\quad \textbf{magic}\ \|\mathcal{S} \ = \ \textbf{magic}$

Proof (left zero element)

$$
\begin{aligned}
& \textbf{magic}\ \|\mathcal{T} \\
= \{ & \text{ Definitions } \textbf{magic}\ , \| \ \} \\
& \{S, T : S \in \{\} \ \wedge \ T \in \mathcal{T} : S\|T\} \\
= \{ & \text{ Calculus } \} \\
& \{\} \\
= \{ & \text{ Definition } \textbf{magic} \ \ \} \\
& \textbf{magic}
\end{aligned}
$$

end of proof

(4.53) $\quad \mathcal{T}\|\mathcal{S} \ = \ \mathcal{S}\|\mathcal{T}$

Proof
Immediate from Definition 4.29 and theorem 3.31.
end of proof

(4.54) $\quad (\mathcal{R}\|\mathcal{S})\|\mathcal{T} \ = \ \mathcal{R}\|(\mathcal{S}\|\mathcal{T})$

Proof

$$
\begin{array}{ll}
 & (\mathcal{R}\|\mathcal{S})\|\mathcal{T} \\
= & \{ \text{ Definition } \} \\
 & \{R, S : R \in \mathcal{R}, S \in \mathcal{S} : R\|S\}\|T \\
= & \{ \text{ Definition } \} \\
 & \{P, T : P \in \{R, S : R \in \mathcal{R}, S \in \mathcal{S} : R\|S\}, T \in \mathcal{T} : P\|T\} \\
= & \{ \text{ Calculus } \} \\
 & \{R, S, T : R \in \mathcal{R}, S \in \mathcal{S}, T \in \mathcal{T} : (R\|S)\|T\} \\
= & \{ \text{ Symmetry, Theorem 3.32 } \} \\
 & \mathcal{R}\|(\mathcal{S}\|\mathcal{T})
\end{array}
$$

end of proof

(4.55) $\quad \textbf{skip}\ ; \mathcal{S} \simeq S \ = \ S; \textbf{skip}$

Proof (left unit element)

$$
\begin{array}{ll}
 & \textbf{skip}\ ; \mathcal{T} \\
= & \{ \text{ Definitions } \textbf{skip}\ ,;\ \} \\
 & \sqcap(S : S \in \{\{\epsilon\}\} : \sqcup(s : s \in S : \\
 & \qquad \sqcap(T : T \in \mathcal{T} : \sqcup(t : t \in T : \{\{st\}\})))) \\
= & \{ \text{ Calculus } \} \\
 & \sqcap(T : T \in \mathcal{T} : \sqcup(t : t \in T : \{\{\epsilon t\}\}))) \\
= & \{ \epsilon t = t \ \} \\
 & \sqcap(T : T \in \mathcal{T} : \sqcup(t : t \in T : \{\{t\}\}))) \\
= & \{ \text{ Theorem (4.14) } \} \\
 & \mathcal{T}
\end{array}
$$

end of proof

(4.56) $\quad \textbf{demon}\ ; \mathcal{S} \ = \ \textbf{demon}$

Proof (left zero element)

$$
\begin{array}{ll}
 & \textbf{demon}\ ; \mathcal{T} \\
= & \{ \text{ Definitions } \textbf{demon}\ ,;\ \} \\
 & \sqcap(S : S \in \{\{\ \}\} : \sqcup(s : s \in S : \\
 & \qquad \sqcap(T : T \in \mathcal{T} : \sqcup(t : t \in T : \{\{st\}\}))))
\end{array}
$$

$= \{$ Calculus $\}$
$\sqcup(s : s \in \{\} : \sqcap(T : T \in \mathcal{T} : \sqcup(t : t \in T : \{\{st\}\})))$
$= \{ \sqcup$ over empty range is **demon** $\}$
demon

end of proof

$$(4.57) \qquad \textbf{magic} \,; \mathcal{S} \;=\; \textbf{magic}$$

Proof (left zero element)

$\textbf{magic} \,; \mathcal{T}$
$= \{$ Definitions **magic** ,; $\}$
$\sqcap(S : S \in \{\} : \sqcup(s : s \in S :$
$\quad \sqcap(T : T \in \mathcal{T} : \sqcup(t : t \in T : \{\{st\}\}))))$
$= \{ \sqcap$ over empty range is **magic** $\}$
magic

end of proof

Lemma E.1 $\qquad \{\{s\}\}; \mathcal{T} = \sqcap(T : T \in \mathcal{T} : \sqcup(t : t \in T : \{\{st\}\}))$

Proof

$\{\{s\}\}; \mathcal{T}$
$= \{$ Definition ; $\}$
$\sqcap(S : S \in \{\{s\}\} : \sqcup(r : r \in S : \sqcap(T : T \in \mathcal{T} : \sqcup(t : t \in T : \{\{rt\}\}))))$
$= \{$ Calculus $\}$
$\sqcap(T : T \in \mathcal{T} : \sqcup(t : t \in T : \{\{st\}\}))$

end of proof

$$(4.58) \qquad (\mathcal{R} \sqcup \mathcal{S}); \mathcal{T} \;=\; (\mathcal{R}; \mathcal{T}) \sqcup (\mathcal{S}; \mathcal{T})$$

Proof

$$
\begin{array}{l}
\quad (\mathcal{R} \sqcup \mathcal{S}); \mathcal{T} \\
= \{ \text{ Definition } \sqcup, ; \ \} \\
\quad \sqcap(R0 : R0 \in \{R, S : R \in \mathcal{R} \wedge S \in \mathcal{S} : R \cup S\} : \sqcup(r : r \in R0 : \{\{r\}\}; \mathcal{T})) \\
= \{ \text{ Calculus } \} \\
\quad \sqcap(R, S : R \in \mathcal{R} \wedge S \in \mathcal{S} : \sqcup(r : r \in R \cup S : \{\{r\}\}; \mathcal{T})) \\
= \{ \text{ Definition } ; \ \} \\
\quad \sqcap(R, S : R \in \mathcal{R} \wedge S \in \mathcal{S} : \sqcup(r : r \in R : \{\{r\}\}; \mathcal{T}) \ \sqcup \ \sqcup(r : r \in S : \{\{r\}\}; \mathcal{T})) \\
= \{ \text{ Definition } ; \ \} \\
\quad \sqcap(R, S : R \in \mathcal{R} \wedge S \in \mathcal{S} : \{R\}; \mathcal{T} \ \sqcup \ \{S\}; \mathcal{T}) \\
= \{ \text{ Distributivity } \} \\
\quad \sqcap(R : R \in \mathcal{R} : \{R\}; \mathcal{T}) \ \sqcup \ \sqcap(S : S \in \mathcal{S} : \{S\}; \mathcal{T}) \\
= \{ \text{ Definition } ; \ \} \\
\quad (\mathcal{R}; \mathcal{T}) \sqcup (\mathcal{S}; \mathcal{T})
\end{array}
$$

end of proof

$$(4.59) \qquad (\mathcal{R} \sqcap \mathcal{S}); \mathcal{T} \ = \ (\mathcal{R}; \mathcal{T}) \sqcap (\mathcal{S}; \mathcal{T})$$

Proof

$$
\begin{array}{l}
\quad (\mathcal{R} \sqcap \mathcal{S}); \mathcal{T} \\
= \{ \text{ Definition } ; , \sqcap, \text{ lemma (E.1) } \} \\
\quad \sqcap(R : R \in \mathcal{R} \cup \mathcal{S} : \sqcup(r : r \in R : \{\{r\}\}; \mathcal{T})) \\
= \{ \text{ Calculus } \} \\
\qquad \sqcap(R : R \in \mathcal{R} : \sqcup(r : r \in R : \{\{r\}\}; \mathcal{T})) \\
\quad \sqcap \ \sqcap(S : S \in \mathcal{S} : \sqcup(s : s \in S : \{\{s\}\}; \mathcal{T})) \\
= \{ \text{ Definition } ; , \sqcap \ \} \\
\quad (\mathcal{R}; \mathcal{T}) \sqcap (\mathcal{S}; \mathcal{T})
\end{array}
$$

end of proof

$$(4.60) \qquad (\mathcal{T}; \mathcal{S}); \mathcal{R} \ = \ \mathcal{T}; (\mathcal{S}; \mathcal{R})$$

Proof

$\mathcal{R}; (\mathcal{S}; \mathcal{T})$
$= \{$ Definition ; , lemma (E.1) $\}$
$\sqcap(R : R \in \mathcal{R} : \sqcup(r : r \in R : \{\{r\}\}; (\mathcal{S}; \mathcal{T})))$
$= \{$ Definition ; $\}$
$\sqcap(R : R \in \mathcal{R} : \sqcup(r : r \in R : \{\{r\}\}; \sqcap(S : S \in \mathcal{S} : \sqcup(s : s \in S : \{\{s\}\}; \mathcal{T}))))$
$= \{$ Definition ; , lemma (E.1) $\}$
$\sqcap(R : R \in \mathcal{R} : \sqcup(r : r \in R : \sqcap(S : S \in \mathcal{S} : \sqcup(s : s \in S : \{\{r\}\}; (\{\{s\}\}; \mathcal{T})))))$
$= \{$ Definition ; , associativity of concatenation $\}$
$\sqcap(R : R \in \mathcal{R} : \sqcup(r : r \in R : \sqcap(S : S \in \mathcal{S} : \sqcup(s : s \in S : (\{\{r\}\}; \{\{s\}\}); \mathcal{T}))))$
$= \{$ Left distribution of ; over $\sqcup$ $\}$
$\sqcap(R : R \in \mathcal{R} : \sqcup(r : r \in R : \sqcap(S : S \in \mathcal{S} : \sqcup(s : s \in S : (\{\{r\}\}; \{\{s\}\})); \mathcal{T})))$
$= \{$ Definition ; $\}$
$\sqcap(R : R \in \mathcal{R} : \sqcup(r : r \in R : \sqcap(S : S \in \mathcal{S} : (\{\{r\}\}; \{S\}); \mathcal{T})))$
$= \{$ Left distribution of ; over $\sqcap$ $\}$
$\sqcap(R : R \in \mathcal{R} : \sqcup(r : r \in R : \sqcap(S : S \in \mathcal{S} : (\{\{r\}\}; \{S\})); \mathcal{T}))$
$= \{$ Definition ; $\}$
$\sqcap(R : R \in \mathcal{R} : \sqcup(r : r \in R : (\{\{r\}\}; \mathcal{S}); \mathcal{T}))$
$= \{$ Left distribution of ; over $\sqcup$ $\}$
$\sqcap(R : R \in \mathcal{R} : \sqcup(r : r \in R : (\{\{r\}\}; \mathcal{S})); \mathcal{T})$
$= \{$ Definition ; $\}$
$\sqcap(R : R \in \mathcal{R} : (\{R\}; \mathcal{S}); \mathcal{T})$
$= \{$ Left distribution of ; over $\sqcap$ $\}$
$\sqcap(R : R \in \mathcal{R} : (\{R\}; \mathcal{S})); \mathcal{T}$
$= \{$ Definition ; $\}$
$(\mathcal{R}; \mathcal{S}); \mathcal{T}$

end of proof

$$(4.61) \qquad \alpha(X) \cap \alpha(Y) = \emptyset \Rightarrow (\mathcal{S} \textbf{ connect } X) \textbf{ connect } Y \;=\; \mathcal{S} \textbf{ connect } (X \cup Y)$$

Proof

$(\mathcal{S} \textbf{ connect } X) \textbf{ connect } Y$
$= \{$ Definition **connect** $\}$
$\{S : S \in \mathcal{S} : S \textbf{ connect } X\} \textbf{ connect } Y$
$= \{$ Definition **connect** $\}$

$$
\begin{array}{ll}
& \{S' : S' \in \{S : S \in \mathcal{S} : S \textbf{ connect } X\} : S' \textbf{ connect } Y\} \\
= & \{ \text{ Calculus } \} \\
& \{S : S \in \mathcal{S} : (S \textbf{ connect } X) \textbf{ connect } Y\} \\
= & \{ \text{ Theorem 3.38 } \} \\
& \{S : S \in \mathcal{S} : S \textbf{ connect } (X \cup Y)\} \\
= & \{ \text{ Definition } \textbf{ connect } \} \\
& \mathcal{S} \textbf{ connect } (X \cup Y)
\end{array}
$$

end of proof

$$(4.62) \qquad (\mathcal{S}; \mathcal{T}) \textbf{ connect } X \;=\; (\mathcal{S} \textbf{ connect } X); (\mathcal{T} \textbf{ connect } X)$$

Proof

$$
\begin{array}{ll}
& (\mathcal{S}; \mathcal{T}) \textbf{ connect } X \\
= & \{ \text{ Definition } \textbf{ connect } \} \\
& \{R : R \in (\mathcal{S}; \mathcal{T}) : R \textbf{ connect } X\} \\
= & \{ \text{ Definition } ; \} \\
& \{R : R \in \{S, T : S \in \mathcal{S} \;\wedge\; T \in \mathcal{T} : S;\, T\} : R \textbf{ connect } X\} \\
= & \{ \text{ Calculus } \} \\
& \{S, T : S \in \mathcal{S} \;\wedge\; T \in \mathcal{T} : (S;\, T) \textbf{ connect } X\} \\
= & \{ \text{ Theorem 3.39 } \} \\
& \{S, T : S \in \mathcal{S} \;\wedge\; T \in \mathcal{T} : (S \textbf{ connect } X); (T \textbf{ connect } X)\} \\
= & \{ \text{ Definition } \textbf{ connect } \text{, twice } \} \\
& (\mathcal{S} \textbf{ connect } X); (\mathcal{T} \textbf{ connect } X)
\end{array}
$$

end of proof

$$(4.63) \qquad (\mathcal{S} \sqcup \mathcal{T}) \textbf{ connect } X \;=\; (\mathcal{S} \textbf{ connect } X); (\mathcal{T} \textbf{ connect } X)$$

Proof

$$
\begin{array}{ll}
& (\mathcal{S} \sqcup \mathcal{T}) \textbf{ connect } X \\
= & \{ \text{ Definition } \textbf{ connect } \} \\
& \{R : R \in (\mathcal{S} \sqcup \mathcal{T}) : R \textbf{ connect } X\} \\
= & \{ \text{ Definition } \sqcup \} \\
& \{R : R \in \{S, T : S \in \mathcal{S} \;\wedge\; T \in \mathcal{T} : S \cup T\} : R \textbf{ connect } X\}
\end{array}
$$

$= \{$ Calculus $\}$
$\{S, T : S \in \mathcal{S} \ \wedge \ T \in \mathcal{T} : (S \cup T) \textbf{ connect } X\}$
$= \{$ Theorem 3.37 $\}$
$\{S, T : S \in \mathcal{S} \ \wedge \ T \in \mathcal{T} : (S \textbf{ connect } X) \cup (T \textbf{ connect } X)\}$
$= \{$ Two steps $\}$
$(\mathcal{S} \textbf{ connect } X) \sqcup (\mathcal{T} \textbf{ connect } X)$

end of proof

$$(4.64) \qquad (\mathcal{S} \sqcap \mathcal{T}) \textbf{ connect } X \ = \ (\mathcal{S} \textbf{ connect } X) \sqcap (\mathcal{T} \textbf{ connect } X)$$

Proof

$(\mathcal{S} \sqcap \mathcal{T}) \textbf{ connect } X$
$= \{$ Definition **connect** $\}$
$\{R : R \in (\mathcal{S} \cup \mathcal{T}) : R \textbf{ connect } X\}$
$= \{$ Definition $\sqcap$ $\}$
$\{R : R \in \mathcal{S} : R \textbf{ connect } X\} \cup \{R : R \in \mathcal{T} : R \textbf{ connect } X\}$
$= \{$ Definition $\sqcap$ $\}$
$(\mathcal{S} \textbf{ connect } X) \sqcap (\mathcal{T} \textbf{ connect } X)$

end of proof

$$(4.65) \qquad \alpha(X) \cap \sigma(\mathcal{S}) = \emptyset \ \wedge \ \alpha(X) \cap \sigma(\mathcal{T}) = \emptyset \Rightarrow$$
$$(\mathcal{S} \| \mathcal{T}) \textbf{ connect } X \ = \ (\mathcal{S} \textbf{ connect } X) \| (\mathcal{T} \textbf{ connect } X)$$

Proof

$(\mathcal{S} \| \mathcal{T}) \textbf{ connect } X$
$= \{$ Definition **connect** $\}$
$\{R : R \in (\mathcal{S} \| \mathcal{T}) : R \textbf{ connect } X\}$
$= \{$ Definition $\|$ $\}$
$\{R : R \in \{S, T : S \in \mathcal{S} \ \wedge \ T \in \mathcal{T} : S \| T\} : R \textbf{ connect } X\}$
$= \{$ Calculus $\}$
$\{S, T : S \in \mathcal{S} \ \wedge \ T \in \mathcal{T} : (S \| T) \textbf{ connect } X\}$
$= \{$ Theorem 3.40 $\}$
$\{S, T : S \in \mathcal{S} \ \wedge \ T \in \mathcal{T} : (S \textbf{ connect } X) \| (T \textbf{ connect } X)\}$
$= \{$ Two steps $\}$
$(\mathcal{S} \textbf{ connect } X) \| (\mathcal{T} \textbf{ connect } X)$

end of proof

(4.66) $\{a, b\} \cap (\alpha(\mathcal{S}_0) \cup \alpha(\mathcal{S}_1) \cup \alpha(\mathcal{T}_0) \cup \alpha(\mathcal{T}_1)) = \emptyset \Rightarrow$
$((\mathcal{S}_0;\ a;\ \mathcal{S}_1) \| (\mathcal{T}_0;\ b;\ \mathcal{T}_1))\ \textbf{connect}\ \ \{a.b\}\ =\ (\mathcal{S}_0 \| \mathcal{T}_0); (\mathcal{S}_1 \| \mathcal{T}_1))$

Proof

We assume $\{a, b\} \cap (\alpha(\mathcal{S}_0) \cup \alpha(\mathcal{S}_1) \cup \alpha(\mathcal{T}_0) \cup \alpha(\mathcal{T}_1)) = \emptyset$

$((\mathcal{S}_0;\ a;\ \mathcal{S}_1) \| (\mathcal{T}_0;\ b;\ \mathcal{T}_1))\ \textbf{connect}\ \ \{a.b\}$
$=$ { Definitions $\|$ and ; }
$\sqcap(S_0, S_1, T_0, T_1 : S_0 \in \mathcal{S}_0\ \wedge\ S_1 \in \mathcal{S}_1\ \wedge\ T_0 \in \mathcal{T}_0\ \wedge\ T_1 \in \mathcal{T}_1 :$
$((S_0;\ a;\ S_1) \| (T_0;\ b;\ T_1)))\ \textbf{connect}\ \ \{a.b\}$
$=$ { Theorem 4.64 }
$\sqcap(S_0, S_1, T_0, T_1 : S_0 \in \mathcal{S}_0\ \wedge\ S_1 \in \mathcal{S}_1\ \wedge\ T_0 \in \mathcal{T}_0\ \wedge\ T_1 \in \mathcal{T}_1 :$
$((S_0;\ a;\ S_1) \| (T_0;\ b;\ T_1))\ \textbf{connect}\ \ \{a.b\})$
$=$ { Theorem 3.39 }
$\sqcap(S_0, S_1, T_0, T_1 : S_0 \in \mathcal{S}_0\ \wedge\ S_1 \in \mathcal{S}_1\ \wedge\ T_0 \in \mathcal{T}_0\ \wedge\ T_1 \in \mathcal{T}_1 :$
$((S_0 \| T_0); (S_1 \| T_1)))$
$=$ { Definitions $\|$ and ; }
$(\mathcal{S}_0 \| \mathcal{T}_0); (\mathcal{S}_1 \| \mathcal{T}_1)$

end of proof

Appendix F

Proofs for Section 4.4

(4.70) $\mathcal{S} \sqsupseteq \mathcal{T} \Rightarrow \mathcal{S}^* \sqsupseteq \mathcal{T}^*$

Proof

$\mathcal{S}^*$
$= \{$ Definition * $\}$
$\{\{\epsilon\}\}\ \mathbf{alt}(\mathcal{S}; \mathcal{S}^*)$
$\sqsupseteq \{$ Monotonicity of $\sqcup$ and ; ,$\mathcal{S} \sqsupseteq \mathcal{T}$ $\}$
$\{\{\epsilon\}\}\ \mathbf{alt}(\mathcal{T}; \mathcal{S}^*)$

hence,

true
$= \{$ Previous proof $\}$
$\mathcal{S}^* \sqsupseteq \{\{\epsilon\}\}\ \mathbf{alt}(\mathcal{T}; \mathcal{S}^*)$
$= \{$ Calculus $\}$
$\mathcal{S}^* \in \{\mathcal{R} :: \mathcal{R} \sqsupseteq \{\{\epsilon\}\}\ \mathbf{alt}(\mathcal{T}; \mathcal{R})\}$
$= \{$ Lattice properties $\}$
$\mathcal{S}^* \sqsupseteq \sqcap(\mathcal{R} : \mathcal{R} \sqsupseteq \{\{\epsilon\}\}\ \mathbf{alt}(\mathcal{T}; \mathcal{R}) : \mathcal{R})$
$= \{$ Knaster-Tarski $\}$
$\mathcal{S}^* \sqsupseteq \mu(\mathcal{R} : \mathcal{R} = \{\{\epsilon\}\}\ \mathbf{alt}(\mathcal{T}; \mathcal{R}) : \mathcal{R})$
$= \{$ Definition * $\}$
$\mathcal{S}^* \sqsupseteq \mathcal{T}^*$

end of proof

(4.71) $\mathcal{S} \sqsupseteq \mathcal{T} \Rightarrow \mathcal{S}^{\dagger} \sqsupseteq \mathcal{T}^{\dagger}$

Proof Similar to previous proof. **end of proof**

(4.72) $\mathcal{S}^{*} \| \mathcal{T}^{*} \sqsupseteq (\mathcal{S} \| \mathcal{T})^{*}$

Proof

$$
\begin{array}{ll}
& \mathcal{S}^{*} \| \mathcal{T}^{*} \\
= & \{ \text{ Property of } ^{*} \} \\
& (\{\{\epsilon\}\} \sqcup \mathcal{S}; \mathcal{S}^{*}) \| (\{\{\epsilon\}\} \sqcup \mathcal{T}; \mathcal{T}^{*}) \\
= & \{ \text{ Definition } \sqcup \ \} \\
& \{S0, S1 : S0 \in \{\{\epsilon\}\} \ \wedge \ S1 \in (\mathcal{S}; \mathcal{S}^{*}) : S0 \cup S1\} \| \\
& \{T0, T1 : T0 \in \{\{\epsilon\}\} \ \wedge \ T1 \in (\mathcal{T}; \mathcal{T}^{*}) : T0 \cup T1\} \\
= & \{ \text{ Calculus } \} \\
& \{S1 : S1 \in (\mathcal{S}; \mathcal{S}^{*}) : \{\epsilon\} \cup S1\} \| \\
& \{T1 : T1 \in (\mathcal{T}; \mathcal{T}^{*}) : \{\epsilon\} \cup T1\} \\
= & \{ \text{ Definition } ; \ \} \\
& \{S2, S3 : S2 \in \mathcal{S} \ \wedge \ S3 \in \mathcal{S}^{*} : \{\epsilon\} \cup S2; S3\} \| \\
& \{T2, T3 : T2 \in \mathcal{T} \ \wedge \ T3 \in \mathcal{T}^{*} : \{\epsilon\} \cup T2; T3\} \\
= & \{ \text{ Definition } \| \ \} \\
& \{S2, S3, T2, T3 : S2 \in \mathcal{S} \ \wedge \ S3 \in \mathcal{S}^{*} \ \wedge \ T2 \in \mathcal{T} \ \wedge \ T3 \in \mathcal{T}^{*} : \\
& \qquad (\{\epsilon\} \cup S2; S3) \| (\{\epsilon\} \cup T2; T3)\} \\
\sqsupseteq & \{ \text{ Lemma } \} \\
& \{S2, S3, T2, T3 : S2 \in \mathcal{S} \ \wedge \ S3 \in \mathcal{S}^{*} \ \wedge \ T2 \in \mathcal{T} \ \wedge \ T3 \in \mathcal{T}^{*} : \\
& \qquad \{\epsilon\} \cup (S2 \| T2); (S3 \| T3)\} \\
= & \{ \text{ Definitions } \sqcup \text{ and } ; \ \} \\
& \{\{\epsilon\}\} \sqcup (\mathcal{S} \| \mathcal{T}); (\mathcal{S}^{*} \| \mathcal{T}^{*})
\end{array}
$$

hence

$$
\begin{array}{ll}
& true \\
\Rightarrow & \{ \text{ Previous proof } \} \\
& (\mathcal{S}^{*} \| \mathcal{T}^{*}) \in \{\mathcal{R} :: \mathcal{R} \sqsupseteq \{\{\epsilon\}\} \sqcup ((\mathcal{S} \| \mathcal{T}); \mathcal{R})\} \\
\Rightarrow & \{ \text{ Lattice properties } \} \\
& (\mathcal{S}^{*} \| \mathcal{T}^{*}) \sqsupseteq \ \sqcap(\mathcal{R} : \mathcal{R} \sqsupseteq \{\{\epsilon\}\} \sqcup ((\mathcal{S} \| \mathcal{T}); \mathcal{R}) : \mathcal{R}) \\
= & \{ \text{ Knaster-Tarski } \}
\end{array}
$$

$$(\mathcal{S}^*\|\mathcal{T}^*) \sqsupseteq \ \sqcap(\mathcal{R} : \mathcal{R} = \{\{\epsilon\}\} \sqcup ((\mathcal{S}\|\mathcal{T}); \mathcal{R}) : \mathcal{R})$$

= { Definition * }

$$(\mathcal{S}^*\|\mathcal{T}^*) \sqsupseteq (\mathcal{S}\|\mathcal{T})^*$$

end of proof

Appendix G

Proofs for Section 4.5

(4.74) $\quad$ **Dual(magic) $\simeq$ demon**

Proof

$\quad$ **Dual(magic)**
$=$ { Definition **Dual**, **magic** }
$\quad \sqcup(S : S \in \{\} : \sqcap(s : s \in S : \{\{s\}\}))$
$=$ { $\sqcup$ over empty set is **demon** }
$\quad$ **demon**

end of proof

(4.75) $\quad$ **Dual(demon) $\simeq$ magic**

Proof

$\quad$ **Dual(demon)**
$=$ { Definition **Dual**, **demon** }
$\quad \sqcup(S : S \in \{\{\}\} : \sqcap(s : s \in S : \{\{s\}\}))$
$=$ { Calculus, $\sqcap$ over empty set is **magic** }
$\quad$ **magic**

end of proof

(4.76) $\quad$ **Dual(pick) $\simeq$ chaos**

Proof

$\quad\ \mathbf{Dual}(\mathbf{pick}\ )$
$= \{$ Definition $\mathbf{Dual}, \mathbf{pick}\ \ \}$
$\quad\ \sqcup(S : S \in \{\{t : t \in \mathbf{Traces} : t\}\} : \sqcap(s : s \in S : \{\{s\}\}))$
$= \{$ Calculus $\}$
$\quad\ \sqcap(s : s \in \mathbf{Traces} : \{\{s\}\})$
$= \{$ Definition $\mathbf{chaos}\ \ \}$
$\quad\ \mathbf{chaos}$

end of proof

$$(4.77) \qquad \mathbf{Dual}(\mathbf{chaos}\ ) \simeq \mathbf{pick}$$

Proof

$\quad\ \mathbf{Dual}(\mathbf{chaos}\ )$
$= \{$ Definition $\mathbf{Dual}, \mathbf{chaos}\ \ \}$
$\quad\ \sqcup(S : S \in \{t : t \in \mathbf{Traces} : \{t\}\} : \sqcap(s : s \in S : \{\{s\}\}))$
$= \{$ Calculus $\}$
$\quad\ \sqcup(t : t \in \mathbf{Traces} : \{\{t\}\})$
$= \{$ Definition $\mathbf{pick}\ \ \}$
$\quad\ \mathbf{pick}$

end of proof

$$(4.78) \qquad \mathbf{Dual}(\mathbf{skip}\ ) \simeq \mathbf{skip}$$

Proof See next proof. **end of proof**

$$(4.79) \qquad \mathbf{Dual}(\{\{s\}\}) \simeq \{\{s\}\}$$

Proof

$\quad\ \mathbf{Dual}(\{\{s\}\})$
$= \{$ Definition $\mathbf{Dual}\ \}$
$\quad\ \sqcup(S : S \in \{\{s\}\} : \sqcap(s' : s' \in S : \{\{s'\}\}))$
$= \{$ Calculus $\}$
$\quad\ \{\{s\}\}$

end of proof

$$\text{(4.80)} \qquad \mathbf{Dual}(\mathcal{S} \sqcap \mathcal{T}) \simeq \mathbf{Dual}(\mathcal{S}) \sqcup \mathbf{Dual}(\mathcal{T})$$

Proof

$\quad \mathbf{Dual}(\mathcal{S} \sqcap \mathcal{T})$
$= \{$ Definition $\mathbf{Dual}, \sqcap$ $\}$
$\quad \sqcup(S : S \in (\mathcal{S} \cup \mathcal{T}) : \sqcap(s : s \in S : \{\{s\}\}))$
$= \{$ Calculus $\}$
$\quad \sqcup(S : S \in \mathcal{S} : \sqcap(s : s \in S : \{\{s\}\}))\sqcup$
$\quad \sqcup(S : S \in \mathcal{T} : \sqcap(s : s \in S : \{\{s\}\}))$
$= \{$ Definition $\mathbf{Dual}$ $\}$
$\quad (\mathbf{Dual}(\mathcal{S})) \sqcup (\mathbf{Dual}(\mathcal{T}))$

end of proof

$$\text{(4.81)} \qquad \mathbf{Dual}(\mathcal{S} \sqcup \mathcal{T}) \simeq \mathbf{Dual}(\mathcal{S}) \sqcap \mathbf{Dual}(\mathcal{T})$$

Proof

$\quad \mathbf{Dual}(\mathcal{S} \sqcup \mathcal{T})$
$= \{$ Definition $\mathbf{Dual}, \sqcup$ $\}$
$\quad \sqcup(S : S \in \{S', T : S' \in \mathcal{S} \ \wedge \ T \in \mathcal{T} : S' \cup T\} : \sqcap(s : s \in S : \{\{s\}\}))$
$= \{$ Calculus $\}$
$\quad \sqcup(S, T : S \in \mathcal{S} \ \wedge \ T \in \mathcal{T} : \sqcap(s : s \in (S \cup T) : \{\{s\}\}))$
$= \{$ Calculus $\}$
$\quad \sqcup(S, T : S \in \mathcal{S} \ \wedge \ T \in \mathcal{T} : \sqcap(s : s \in S : \{\{s\}\}) \ \sqcap \ \sqcap(t : t \in T : \{\{t\}\}))$
$= \{$ Calculus (Generalized De Morgan) $\}$
$\quad \sqcup(S : S \in \mathcal{S} : \sqcap(s : s \in S : \{\{s\}\})) \ \sqcap \ \sqcup(T : T \in \mathcal{T} : \sqcap(t : t \in T : \{\{t\}\}))$
$= \{$ Definition $\mathbf{Dual}$ $\}$
$\quad (\mathbf{Dual}(\mathcal{S})) \sqcap (\mathbf{Dual}(\mathcal{T}))$

end of proof

$$\text{(4.82)} \qquad \mathbf{Dual}(\mathbf{Dual}(\mathcal{S})) \simeq \mathcal{S}$$

Proof

$$
\begin{aligned}
& \quad \mathbf{Dual}(\mathbf{Dual}(\mathcal{S})) \\
= & \ \{ \text{ Theorem (4.14) } \} \\
& \quad \mathbf{Dual}(\mathbf{Dual}(\sqcap(S : S \in \mathcal{S} : \sqcup(s : s \in S : \{\{s\}\})))) \\
\simeq & \ \{ \text{ Theorem (4.80) } \} \\
& \quad \mathbf{Dual}(\sqcap(S : S \in \mathcal{S} : \mathbf{Dual}(\sqcup(s : s \in S : \{\{s\}\})))) \\
\simeq & \ \{ \text{ Theorem (4.81) } \} \\
& \quad \mathbf{Dual}(\sqcup(S : S \in \mathcal{S} : \sqcap(s : s \in S : \mathbf{Dual}(\{\{s\}\})))) \\
= & \ \{ \text{ Theoren (4.79) } \} \\
& \quad \mathbf{Dual}(\sqcup(S : S \in \mathcal{S} : \sqcap(s : s \in S : \{\{s\}\}))) \\
\simeq & \ \{ \text{ Theorem (4.81) } \} \\
& \quad \sqcap(S : S \in \mathcal{S} : \mathbf{Dual}(\sqcap(s : s \in S : \{\{s\}\}))) \\
\simeq & \ \{ \text{ Theorem (4.80) } \} \\
& \quad \sqcap(S : S \in \mathcal{S} : \sqcup(s : s \in S : \mathbf{Dual}(\{\{s\}\}))) \\
\simeq & \ \{ \text{ Theorem (4.79) } \} \\
& \quad \sqcap(S : S \in \mathcal{S} : \sqcup(s : s \in S : \{\{s\}\})) \\
\simeq & \ \{ \text{ Theorem (4.14) } \} \\
& \quad \mathcal{S}
\end{aligned}
$$

end of proof

$$(4.84) \qquad \mathcal{S} \sqsupseteq Dual(\mathcal{T}) = \mathcal{T} \sqsupseteq Dual(\mathcal{S})$$

Proof

$$
\begin{aligned}
& \quad \mathcal{S} \sqsupseteq Dual(\mathcal{T}) \\
= & \ \{ \text{ Definition } \sqsupseteq \} \\
& \quad \forall(S : S \in \mathcal{S} : \exists(T : T \in Dual(\mathcal{T}) : S \supseteq T)) \\
= & \ \{ \text{ Definition } Dual \} \\
& \quad \forall(S : S \in \mathcal{S} : \exists(T : \forall(T' : T' \in \mathcal{T} : \exists(t : t \in T' : t \in T)) : S \supseteq T)) \\
= & \ \{ \text{ Calculus } \} \\
& \quad \forall(S : S \in \mathcal{S} : \exists(T : \forall(T' : T' \in \mathcal{T} : \exists(t : t \in T' : t \in T)) \ \wedge \ S \supseteq T : true)) \\
= & \ \{ \text{ Ping pong argument } \} \\
& \quad \forall(S : S \in \mathcal{S} : \exists(T : \forall(T' : T' \in \mathcal{T} : \exists(t : t \in T' : t \in S)) : true)) \\
= & \ \{ \text{ Calculus } \} \\
& \quad \forall(S : S \in \mathcal{S} : \forall(T' : T' \in \mathcal{T} : \exists(t : t \in T' : t \in S))) \\
= & \ \{ \text{ Calculus } \} \\
& \quad \forall(S, T' : S \in \mathcal{S} \ \wedge \ T' \in \mathcal{T} : \exists(t :: t \in T' \ \wedge \ t \in S))
\end{aligned}
$$

$= \{$ Symmetry $\}$
$\quad \mathcal{T} \sqsupseteq Dual(\mathcal{S})$

end of proof

(4.85) $\qquad Dual(\mathcal{S}) \sqsupseteq \mathcal{T} = Dual(\mathcal{T}) \sqsupseteq \mathcal{S}$

Proof

$\quad Dual(\mathcal{S}) \sqsupseteq \mathcal{T}$
$= \{$ Definition $\sqsupseteq$ $\}$
$\quad \forall(S : S \in Dual(\mathcal{S}) : \exists(T : T \in \mathcal{T} : S \supseteq T))$
$= \{$ Big step $\}$
$\quad \forall(S : S \in Dual(\mathcal{S}) : \forall(T' : \forall(T : T \in T' : \exists(t : t \in T' : t \in T))$
$\qquad : \exists(t : t \in T' : t \in S)))$
$= \{$ Definition $Dual$ $\}$
$\quad \forall(S, T' : S \in Dual(\mathcal{S}) \ \wedge \ T' \in Dual(\mathcal{T}) : \exists(t :: t \in T' \ \wedge \ t \in S))$
$= \{$ Symmetry $\}$
$\quad Dual(\mathcal{T}) \sqsupseteq \mathcal{S}$

end of proof

(4.86) $\qquad \mathbf{Dual}\mathcal{S} \simeq Dual\mathcal{S}$

Proof

The following two properties are direct consequences of the fact that $Dual$ is a Galois connection.

$\quad Dual(\mathcal{S} \sqcap \mathcal{T}) \simeq Dual(\mathcal{S}) \sqcup Dual(\mathcal{T})$
$\quad Dual(\mathcal{S} \sqcup \mathcal{T}) \simeq Dual(\mathcal{S}) \sqcap Dual(\mathcal{T})$

We also have

$\quad Dual(\{\{s\}\})$
$= \{$ Definition $Dual$ $\}$
$\quad \{T : \forall(S : S \in \{\{s\}\} : \exists(t : t \in S : t \in T)) : T\}$
$= \{$ Calculus $\}$
$\quad \{T : \exists(t : t \in T : t \in \{\{s\}\}) : T\}$
$\simeq \{$ Theorem (4.10) $\}$
$\quad \{\{s\}\}$

These three properties define $Dual$ up to process equivalence, as the following calculation shows.

$$\begin{array}{ll} & Dual(\mathcal{S}) \\ = & \{ \text{ Theorem (4.14) } \} \\ & Dual(\sqcap(S : S \in \mathcal{S} : \sqcup(s : s \in S : \{\{s\}\}))) \\ \simeq & \{ \text{ Use the three properties } \} \\ & \sqcup(S : S \in \mathcal{S} : \sqcap(s : s \in S : \{\{s\}\})) \\ = & \{ \text{ Definition } \mathbf{Dual} \} \\ & \mathbf{Dual}(\mathcal{S}) \end{array}$$

end of proof

$$(4.87) \qquad \mathcal{S} \sqsupseteq \mathcal{T} = \mathbf{Dual}(\mathcal{T}) \sqsupseteq \mathbf{Dual}(\mathcal{S})$$

Proof
Take $\mathbf{Dual}(\mathcal{S})$ for $\mathcal{S}$ in Theorem 4.84 and use Theorem 4.82.
end of proof

$$(4.88) \qquad \mathcal{S} \simeq \mathcal{T} = \mathbf{Dual}(\mathcal{S}) \simeq \mathbf{Dual}(\mathcal{T})$$

Proof Immediate from above. **end of proof**

$$(4.89) \qquad \mathbf{Dual}(\mathcal{S}; \mathcal{T}) \simeq \mathbf{Dual}(\mathcal{S}); \mathbf{Dual}(\mathcal{T})$$

Proof

$$\begin{array}{ll} & \mathbf{Dual}(\mathcal{S}; \mathcal{T}) \\ = & \{ \text{ Definition } ; \} \\ & \mathbf{Dual}(\sqcap(S : S \in \mathcal{S} : \sqcup(s : s \in S : \sqcap(T : T \in \mathcal{T} : \sqcup(t : t \in T : \{\{st\}\}))))) \\ = & \{ \mathbf{Dual} \text{ over } \sqcap, \sqcup \} \\ & \sqcup(S : S \in \mathcal{S} : \sqcap(s : s \in S : \sqcup(T : T \in \mathcal{T} : \sqcap(t : t \in T : \mathbf{Dual}(\{\{st\}\}))))) \\ = & \{ Dual\{\{s\}\} = \{\{s\}\} \} \\ & \sqcup(S : S \in \mathcal{S} : \sqcap(s : s \in S : \sqcup(T : T \in \mathcal{T} : \sqcap(t : t \in T : \{\{st\}\})))) \\ = & \{ \text{ Left distribution of } \sqcap, \sqcup \text{ over } ; \} \\ & \sqcup(S : S \in \mathcal{S} : \sqcap(s : s \in S : \{\{s\}\}; \sqcup(T : T \in \mathcal{T} : \sqcap(t : t \in T : \{\{t\}\})))) \end{array}$$

$= \{$ Definition **Dual** $\}$
$\quad \sqcup(S : S \in \mathcal{S} : \sqcap(s : s \in S : \{\{s\}\}; \mathbf{Dual}(\mathcal{T})))$
$= \{$ Left distribution of $\sqcap, \sqcup$ over ; $\}$
$\quad \sqcup(S : S \in \mathcal{S} : \sqcap(s : s \in S : \{\{s\}\})); \mathbf{Dual}(\mathcal{T})$
$= \{$ Definition **Dual** $\}$
$\quad \mathbf{Dual}(\mathcal{S}); \mathbf{Dual}(\mathcal{T})$

end of proof

(4.90) $\quad \mathbf{Dual}(\mathcal{S}^*) \simeq \mathcal{S}^\dagger$

Proof

$\quad \mathbf{Dual}(\mathcal{S}^*)$
$= \{$ Definition S^* $\}$
$\quad \mathbf{Dual}(\mu(\mathcal{T} : \mathcal{T} \simeq \mathbf{skip} \sqcup \mathcal{S}; \mathcal{T} : \mathcal{T}))$
$= \{$ $\sqcup$ and ; are monotonic hence fixed points form complete lattice $\}$
$\quad \mathbf{Dual}(\sqcap(\mathcal{T} : \mathcal{T} \simeq \mathbf{skip} \sqcup \mathcal{S}; \mathcal{T} : \mathcal{T}))$
$= \{$ **Dual** over $\sqcap$ $\}$
$\quad \sqcup(\mathcal{T} : \mathcal{T} \simeq \mathbf{skip} \sqcup \mathcal{S}; \mathcal{T} : \mathbf{Dual}(\mathcal{T}))$
$= \{$ Calculus $\}$
$\quad \sqcup(\mathcal{T} : \mathcal{T} \simeq \mathbf{skip} \sqcup \mathcal{S}; \mathcal{T} : \mathbf{Dual}(\mathbf{skip} \sqcup \mathcal{S}; \mathcal{T}))$
$= \{$ **Dual** over $\sqcup, ;$. $\mathbf{Dual}(\mathbf{skip}) = \mathbf{skip}$ $\}$
$\quad \sqcup(\mathcal{T} : \mathcal{T} \simeq \mathbf{skip} \sqcup \mathcal{S}; \mathcal{T} : \mathbf{skip} \sqcap \mathbf{Dual}(\mathcal{S}); \mathbf{Dual}(\mathcal{T}))$
$= \{$ Theorem (4.88) $\}$
$\quad \sqcup(\mathcal{T} : \mathbf{Dual}\mathcal{T} \simeq \mathbf{skip} \sqcap \mathbf{Dual}(\mathcal{S}); \mathbf{Dual}(\mathcal{T}) : \mathbf{skip} \sqcap \mathbf{Dual}(\mathcal{S}); \mathbf{Dual}(\mathcal{T}))$
$= \{$ Calculus $\}$
$\quad \sqcup(\mathcal{T} : \mathcal{T} \simeq \mathbf{skip} \sqcap \mathbf{Dual}(\mathcal{S}); \mathcal{T} : \mathcal{T})$
$= \{$ *DEM* and ; are monotonic hence fixed points form complete lattice $\}$
$\quad \nu(\mathcal{T} : \mathcal{T} \simeq \mathbf{skip} \sqcap \mathbf{Dual}(\mathcal{S}); \mathcal{T} : \mathcal{T})$
$= \{$ Definition $\mathcal{S}^\dagger$ $\}$
$\quad \mathcal{S}^\dagger$

end of proof

(4.91) $\quad \mathbf{Dual}(\mathcal{S}^\dagger) \simeq \mathcal{S}^*$

Proof Immediate from previous theorem and Theorem 4.82. **end of proof**

(4.92) $\alpha(\mathcal{S}) = \sigma(\mathcal{S}) \Rightarrow$
$(\mathcal{S} \| \mathbf{Dual}(\mathcal{S}'))$ **connect** $\{x : x \in \alpha(S) : x.x'\} \simeq$ **skip**
where $\mathcal{S}'$ is $\mathcal{S}$ with all actions replaced by their primed counterparts.

Proof

$$
\begin{array}{l}
\quad \mathcal{S} \| \mathbf{Dual}(\mathcal{S}')\mathbf{connect}\ \{x : x \in \alpha(S) : x.x'\} \\
= \{\ \text{Definition } \mathbf{Dual}\ \} \\
\quad \mathcal{S} \| \{S' : \forall(S'' : S' \in \mathcal{S} : \exists(s : s \in S'' : S \in S')) : S'\} \\
\quad\ \mathbf{connect}\ \{x : x \in \alpha(S) : x.x'\} \\
= \{\ \text{Definition } \|\ \} \\
\quad \{S, T : S \in \mathcal{S}\ \wedge\ T \in \{S' : \forall(S'' : S' \in \mathcal{S} : \exists(s : s \in S'' : S \in S')) : S'\} : S \| T\} \\
\quad\ \mathbf{connect}\ \{x : x \in \alpha(S) : x.x'\} \\
= \{\ \text{Calculus}\ \} \\
\quad \{S, S' : S \in \mathcal{S}\ \wedge\ \forall(S'' : S' \in \mathcal{S} : \exists(s : s \in S'' : S \in S')) : S \| S'\} \\
\quad\ \mathbf{connect}\ \{x : x \in \alpha(S) : x.x'\} \\
= \{\ \text{Property of } \mathbf{connect}\ \} \\
\quad \{S, S' : S \in \mathcal{S}\ \wedge\ \forall(S'' : S' \in \mathcal{S} : \exists(s : s \in S'' : S \in S')) : \\
\quad (S \| S')\ \mathbf{connect}\ \{x : x \in \alpha(S) : x.x'\}\} \\
= \{\ \text{Definition } \sqcap, \|\ \} \\
\quad \sqcap(S, S' : S \in \mathcal{S}\ \wedge\ \forall(S'' : S' \in \mathcal{S} : \exists(s : s \in S'' : S \in S')) : \\
\quad\ \sqcup(s, s' : s \in S\ \wedge\ s' \in S' : \{\{s\}\} \| \{\{s'\}\})\ \mathbf{connect}\ \{x : x \in \alpha(S) : x.x'\}) \\
= \{\ \text{Property } \mathbf{connect}\ \} \\
\quad \sqcap(S, S' : S \in \mathcal{S}\ \wedge\ \forall(S'' : S' \in \mathcal{S} : \exists(s : s \in S'' : S \in S')) : \\
\quad\ \sqcup(s, s' : s \in S\ \wedge\ s' \in S' : (\{\{s\}\} \| \{\{s'\}\})\ \mathbf{connect}\ \{x : x \in \alpha(S) : x.x'\})) \\
= \{\ \text{Properties of } \mathbf{connect}\ ,\ \alpha(\mathcal{S}) = \sigma(\mathcal{S})\ \} \\
\quad \sqcap(S, S' : S \in \mathcal{S}\ \wedge\ \forall(S'' : S' \in \mathcal{S} : \exists(s : s \in S'' : S \in S')) : \\
\quad\ \sqcup(s, s' : s \in S\ \wedge\ s' \in S'\ \wedge\ s' = \mathit{primed}(s) : \mathbf{skip}\ )) \\
= \{\ \text{Calculus}\ \} \\
\quad \mathbf{skip}
\end{array}
$$

end of proof

(4.93) $\alpha(\mathcal{S}) = \sigma(\mathcal{S}) \Rightarrow$
$\mathcal{T} \sqsubset \mathbf{Dual}(\mathcal{S}') \Rightarrow (\mathcal{S} \| \mathcal{T})$ **connect** $\{x : x \in \alpha(S) : x.x'\} \simeq$ **demon**

Proof

We have

$\mathcal{T} \sqsubset \mathbf{Dual}(\mathcal{S}') \Rightarrow \exists (T, S : T \in \mathcal{T} \ \wedge \ \mathcal{S} \in \mathcal{S}' : \forall (\sqcup : \sqcup \in \mathcal{T} : \sqcup \notin \mathcal{S}))$

The theorem then follows from a calculation similar to the one for the previous theorem, using the fact that **demon** is the unit of $\sqcup$ and the zero of $\sqcap$.

end of proof

Bibliography

[1] S. Agerholm. Mechanizing program verification in HOL. Master's thesis, Aarhus University, 1992.

[2] R.-J.R. Back and J. von Wright. A lattice-theoretical basis for a specification language. In J.L.A. van de Snepscheut, editor, *Proceedings of the first Mathematics of Program Construction Conference*, pages 139–156, 1989. LNCS 375.

[3] R.-J.R. Back and J. von Wright. Duality in specification languages: A lattice-theoretical approach. *Acta Informatica*, 27:583–625, 1990.

[4] R.-J.R. Back and J. von Wright. Combining, angels, demons, and miracles in program specifications. *Theoretical Computer Science*, 100(2), 1992.

[5] R.J.R. Back. *On the Correctness of Refinement Steps in Program Development*. Ph.D. thesis, University of Helsinki, 1978. Report A-1978-4.

[6] J.A. Bergstra and J.W. Klop. Process algebra for synchronous communication. *Information and Control*, 60(1-3):109–137, 1984.

[7] G. Birkhoff. *Lattice Theory*. Colloquium Publications, Volume 25. American Mathematical Society, 1967.

[8] S.D. Brookes, C.A.R. Hoare, and A.W. Roscoe. A theory of communicating sequential processes. *J. ACM*, 31(3):560–599, 1984.

[9] S.M. Burns and A.J. Martin. Syntax-directed translation of concurrent programs into self-timed circuits. In *Proceedings of the Fifth MIT Conference on Advanced Research in VLSI*, pages 35–40. MIT press, 1988.

[10] K.M. Chandy. Concurrent program archetypes. 1994. Keynote address, Scalable Parallel Libraries Conference.

[11] K.M. Chandy and J. Misra. *Parallel Program Design, a foundation.* Addison-Wesley, 1988.

[12] E.W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[13] E.W. Dijkstra. On the unification of three calculi. In *Marktoberdorf Proceedings*, 1992.

[14] E.W. Dijkstra and C.S. Scholten. *Predicate Calculus and Program Semantics.* Springer-Verlag, 1990.

[15] E.W. Dijkstra and A.J.M. van Gasteren. A simple fixpoint argument without the restriction to continuity. *Acta Informatica*, 23:1–7, 1986.

[16] R.M. Dijkstra. Relational calculus and relational program semantics, 1992. master's thesis, Univ. of Aachen.

[17] R.M. Dijkstra. DUALITY: a simple formalism for the analysis of UNITY. Technical report, Department of Mathematics and Computing Science, University of Groningen, 1994. CS-R9404.

[18] D.L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits.* Ph.D. thesis, Computer Science Department, Carnegie Mellon University, 1988. CMU-CS-88-119.

[19] P. Gardiner and C. Morgan. Data Refinement of Predicate Transformers. *Theoretical Computer Science*, 87(1), 1991.

[20] M. Gordon. HOL: A Machine Oriented Formulation of Higher-Order Logic. Technical Report 68, Computer Laboratory, University of Cambridge, England, 1985. revised version.

[21] J. V. Guttag and J. J. Horning. *Larch: Languages and Tools for Formal Specification.* Springer-Verlag, New York, New York, 1993.

[22] C.A.R. Hoare. Proof of correctness and data representations. *Acta Informatica*, 1:271–281, 1972.

[23] C.A.R. Hoare. *Communicating Sequential Processes*. Series in Computer Science (C.A.R. Hoare, ed.). Prentice-Hall International, 1985.

[24] H.P. Hofstee. Distributing a class of sequential programs. *Science of Computer Programming*, 22:45–65, 1994.

[25] L. Jategaonkar and A.R. Meyer. Self-synchronization of concurrent processes (preliminary report). 1993. Proceedings 1993 LICS conference.

[26] A. Lines. CS185a project specification. 1993. private communication.

[27] INMOS Ltd. occam$^{\mathrm{TM}}$ *Programming Manual*. Prentice Hall International, 1984.

[28] J.J. Lukkien. An operational semantics for the guarded command language. In R.S.Bird, C.C. Morgan, and J.C.P.Woodcock, editors, *Mathematics of Program Construction*, number 669 in Lecture Notes in Computer Science, pages 233–249. Springer-Verlag, 1993.

[29] A.J. Martin. An axiomatic definition of synchronization primitives. *Acta Informatica*, 16:219–235, 1981.

[30] A.J. Martin. Distributed mutual exclusion on a ring of processes. *Science of Computer Programming*, 5:265–276, 1985.

[31] A.J. Martin. The probe: an addition to communication primitives. *Information Processing Letters*, 20:125–130, 1985. and (21)107.

[32] A.J. Martin. Compiling communicating processes into delay-insensitive VLSI circuits. *Distributed Computing*, 1(4), 1986.

[33] R. Milner. *Communication and Concurrency*. Series in Computer Science (C.A.R. Hoare, ed.). Prentice-Hall International, 1989.

[34] C. Morgan. The specification statement. *ACM TOPLAS*, 10(3):403–419, 1988.

[35] C. Morgan. *Programming from Specifications*. Series in Computer Science (C.A.R. Hoare, ed.). Prentice-Hall International, 1990.

[36] G. Nelson. A Generalization of Dijkstra's Calculus. *ACM Transactions on Programming Languages and Systems*, 11(4):517–561, 1989.

[37] C.A. Petri. *Communikation mit Automaten.* Schriften des Institutes für Instrumentelle Mathematik, Bonn, 1962. In German.

[38] C.L. Seitz. System timing. In C.A. Mead and L.A. Conway, editors, *Introduction to VLSI Systems*, chapter 7. Addison Wesley, 1980.

[39] J. Staunstrup and M.R. Greenstreet. Synchronized transitions. In J. Staunstrup, editor, *Formal Methods for VLSI Design*, pages 71–128. North-Holland/Elsevier, 1990.

[40] J.T. Udding. *Classification and Composition of Delay-Insensitive Circuits.* Ph.D. thesis, Technische Hogeschool Eindhoven, 1984.

[41] C.H. van Berkel, J. Kessels, M. Rocken, R.W.J.J. Saeijs, and F. Schalij. The VLSI-programming language tangram and its translation into handshake circuits. In *Proceedings of the 1991 European Design Automation Conference*, pages 384–389. IEEE Computer Society, Los Alamitos, California, 1991.

[42] A. van de Mortel-Fronczak. *Models of Trace Theory Systems.* Ph.D. thesis, Technische Universiteit Eindhoven, 1993.

[43] J.L.A. van de Snepscheut. *Trace Theory and VLSI Design*, volume 200 of *Lecture Notes in Computer Science.* Springer-Verlag, 1985.

[44] J.L.A. van de Snepscheut. JAN 180. On Lattice Theory and Program Semantics. Technical Report CS 93-19, California Institute of Technology, 1993.

[45] J.L.A. van de Snepscheut. JAN 187. Mechanized Support for Stepwise Refinement. Technical Report CS-TR-94-01, California Institute of Technology, 1994.

[46] J.L.A. van de Snepscheut and J.T. Udding. An alternative implementation of communication primitives. *Information Processing Letters*, 23:231–238, 1986.

[47] R. van Glabbeek and U. Goltz. Equivalence notions for concurrent systems and refinement of actions. *LNCS*, 379:237–248, 1989. Proceedings of MFCS '89.

[48] J. von Wright. *A Lattice-theoretical Basis for Program Refinement.* Ph.D. thesis, Åbo Akademi, 1990.