Ready Simulation, Bisimulation, and the Semantics of CCS-Like Languages

Bard Bloom

March 12, 1993

Contents

Part I

Introduction

Ready Simulation, Bisimulation, and the Semantics of CCS-Like Languages

by

Bard Bloom

Submitted to the Department
of Electrical Engineering and Computer Science
on March 12, 1993 in partial fulfillment of the
requirements for the Degree of Doctor of Philosophy in
Computer Science

Abstract

The questions of program comparison — asking when two programs are equal, or when one is a suitable substitute for another — are central in the semantics and verification of programs. It is not obvious what the definitions of comparison should be for parallel programs, even in the relatively simple case of core languages for concurrency, such as the kernel language of Milner's CCS.

We introduce some criteria for judging notions of program comparison. Our basic notion is that of a *congruence*: two programs are equivalent with respect to a language $\mathcal{L}$ and a set of observations $\mathcal{O}$ iff they cannot be distinguished by any observation in $\mathcal{O}$ in any context of $\mathcal{L}$. Bisimulation, the notion of program equivalence ordinarily used with CCS, is finer than CCS congruence: there are two programs which are not bisimilar, but cannot be told apart by CCS contexts.

We explore the possibility of making bisimulation into a congruence. We CCS is defined by a set of *structured operational rules*. We introduce the class of *GSOS rules*, a generalization of the structured operational rules defining CCS. We argue that this class is appropriate, by showing that all GSOS languages have the essential properties of CCS (*e.g.*, a well-defined operational semantics), and that slight extensions of the GSOS rules admit languages which violate these essential properties.

We investigate the notion of congruence with respect to all GSOS languages. This notion, which we call *ready simulation*, has several pleasing mathematical characterizations which closely resemble bisimulation. (Ready simulation is coarser than bisimulation, and so bisimulation cannot be understood as congruence with respect to any CCS-like language.) We consider a variety of kinds of observations and experiments on processes, and ready simulation turns out to be quite robust.

Thesis Supervisor: Albert R. Meyer.
Title: Professor of Computer Science

Chapter 1

Introduction

1.1 Preface

This section gives a very informal introduction to the ideas that lie behind the work of this thesis. It is optional; the reader may skip to Section 1.2 without great loss.

A specification for a program is a more or less formal description of what the program is supposed to do. There are any number of aspects of a program which one may wish to specify: the maximum processing time per input, the number of comments per line of code, the maximum memory requirements, and so on. Ultimately, though, the purpose of writing the program is to do some computational task: predicting weather, running a window system across a network, or whatever. The single most important reason for writing a specification is to state precisely what the computational task is, in a language which allows us to understand whether or not the program does what we want. For the purposes of this discussion, "specifications" are descriptions of the visible behavior of a program, and other requirements (*e.g.*, memory limitations[1] and portability) will appear only tangentially.

There are two dangers in writing specifications: one may specify too little, or one may specify too much. Specifying too little is the more dangerous of the two. A program which is guaranteed not to accidentally launch missiles during weekdays (and has no constraints on its actions on weekends) is rather dangerous. The risks of specifying too much are less destructive, but still potentially annoying.

Consider the following scenario. Sal has hired Pat to write a font editor, and has provided a formal specification S. Pat is an expert programmer and

[1] Under some circumstances, the other requirements are visible; for example, a program demanding too much memory may crash rather than compute properly. We will ignore this complexity, or at best subsume such details under "visible behavior" without further comment.

mathematician, and has written a program P which does **not** meet S.

Sal: I hired you to write a program which meets S. You've come up with this thing, P, which you admit doesn't meet S, and you even seem proud of it. It seems to me that you ought to apologize, or go back and rewrite it.

Pat: *[Starting in the middle with excessive enthusiasm.]* To start with, P is wonderfully efficient. The specification required lots of things that don't matter. For example, it specified that all the integer variables in the program had to be stored as variable-length binary-coded decimal strings. That's pretty silly, because the 1024 by 1024 black-and-white bitmaps are best stored packed into a 128K-byte array, rather than five million bytes in BCD. There were a lot of things in the specification that shouldn't have been there, because they make the program much worse.

Sal: The specification is to make sure that you write a program which does the right thing. I can write an even more efficient program which doesn't meet S — the null program, which never does anything. It's incredibly fast and concise, beautifully written, and the documentation is perfect. It not very good as a font editor though. So why do you think that P is any better than the null program?

Pat: P meets S *as far as anyone using it can tell.* There is a program P' which does meet S, and I can prove that there is no way for a user using P to know that she isn't really using P'.

The notions of *program equivalence* and *program approximation* appear in many contexts in semantics and verification of programs. There are generally several possible definitions of when two pieces of code "do the same thing" or "should be considered as the same," and as we shall see the issues involved are quite subtle. Pat's argument that P meets S as far as anyone can tell raises a crucial point: it is probably the single best definition of code equivalence and approximation:

Two pieces of code should be considered to do the same thing if and only if one can be substituted for the other in any program and no difference in the behavior of the program can be seen. Similarly, One piece of code, P, should be considered an approximation of another, P', *viz.* P is usable in place of P', if P can be put in place of P' in any program and it is impossible to tell that P, rather than P', has been used.

If there is some way to see a difference in the behavior, then the two clearly should be considered different. Conversely, we want to have as much flexibility in meeting specifications as possible, so that we don't accidentally forbid ourselves the use of good programs. So, this informal definition of

code equivalence seems like a good one to use. Similarly, if there is some way to tell that P rather than P' is being used, then P is not a good substitute for P'; and if there is no way, then P should be a good substitute.

Sal: You can't possibly be right. Suppose that I run P and P' under the debugger. I can look at the state of memory, and see that P uses a bitmap where P' uses BCD.

Pat: Of course, and if you look at the running time and space you'll see that they're different as well: my program is smaller and faster. I meant that if you use them in any *legitimate* and *significant* way, you can't tell them apart. Programs that look at the private parts of P — the source and object code, and the private memory space — aren't legitimate; naive users aren't supposed to do such things, and wizards will be able to detect all kinds of things we don't want to specify. I don't consider running time significant for a good reason: we want to allow the fastest and smallest program which does what we want.

The ideas of code equivalence and approximation we have presented is not a single notion. As Sal implies, it depends on two parameters: the kinds of behavior which can be observed, and the ways that code can be used. In a typical setting — my office — I can see what programs display on the screen, and a few similar peripherals (the bell, the printers, and so on); I can combine programs in the various ways that Unix allows (parallel execution connected by pipelines or files). I would like my programs to print the right things on the peripherals; everything that appears on them is important. I prefer that my programs do the right things when run in parallel with other programs, even communicating by pipelines or sockets. I don't want to care about the more intimate details of the system; I take pains to avoid having to single-step programs under the debugger.

The general philosophical or practical problem, then, is to determine what aspects of program behavior are visible, and how programs can be combined. It is rarely clear when one has found the right answer, or even that there is a single right answer. Formal methods can offer some assistance — for example, if several plausible collections of visible behaviors and combinators actually give the *same* notions of code equivalence and approximation, those notions gain in plausibility.

There are several possible endings for the scenario.

1. **Sal:** *[Checks Pat's definitions. Some time passes. Eventually Sal finishes.]*Yes, but that only applies to the programs we have running on the system now. There's a screen examiner under development which you haven't heard about, and it expects the screen to be in variable-length BCD format. If you use any other format, the screen examiner won't work. Your proof is right, but you didn't

expect programs like this. We specified those things because we knew that they would eventually be important.

This possible ending is a common story in semantics. Frequently, two programs are indistinguishable with respect to a language, but turn out to be different with respect to a sensible extension of the language. Adding new operations (or new observations) can increase our *distinguishing power*, our power to tell programs apart. If the operations are plausible, the new distinctions are plausible.

2. **Sal:** *[Sal reads Pat's theorem. Some time passes. Eventually Sal finishes.]* I understand all your mathematics now. Your definition of "legitimate" covers all the programs which don't look into other programs' virtual memory space or code, and I've got to agree that we'd be upset if any of the system programs intended for general users weren't legitimate. (Good for you that we wrote the screen examiner to send messages to the screen driver.) So, it looks like you're right. We ought to have written the specification S to allow programs like P.

   This is also a reasonably common ending in semantics. It is often possible to prove that, whenever a program is correct with respect to a semantical model, then it is correct with respect to all language extensions which can be expressed in that model. For example, if two programs are equal in the standard Scott model for the typed λ-calculus, then they are equal in PCF, and indeed in any programming language for which the Scott model gives an adequate semantics.

3. **Sal:** *[Tosses Pat's theorem into a trashcan without reading it.]* I hired you to write a program matching specification S, and if you don't write one, I will fire you and hire someone else.

   Frequently, issues and concerns outside of pure semantics will influence decisions about the appropriateness of programs. Semantics is not intended to be a direct help in most matters of software engineering, except for providing firm foundations. Semantic methods allow precise definition of programming languages, verification of compilers and other metaprograms. They give some methods for specifying and proving programs, and showing that specifications are satisfiable. They give little guidance on other vital matters; for example, they do not aid in demonstrating that a specification specifies what it is supposed to specify.

1.2 Brief Introduction

There are a large variety of semantics for concurrency, and it is not immediately clear what or how important the distinctions may be. Even when we restrict our attention to languages in the style of Milner's CCS [Mil80] and CSP [Hoa78], and consider only notions of program equivalence, we still find a rather confusing diversity of notions [Abr]. The main purpose of this study is to introduce some methods for comparing and evaluating these notions. We concentrate our attention on CCS-like languages.

Our methods for comparing semantic models are based on the notion of *full abstraction* [Plo77, Abr87]: two programs should be semantically distinct if and only if they behave differently in some context. Frequently, the proposed semantics for a language $\mathcal{L}$ are not fully abstract; however, it is often possible to extend $\mathcal{L}$ in some more or less sensible way to make the semantics fully abstract.

We investigate the possibility of finding fully abstract language extensions. In this investigation, we prove that a certain semantic relation, *ready simulation*, is adequate (and, under suitable conditions, fully abstract) with respect to any sensible extension of CCS. We also show some negative results — that a particular semantics is *not* fully abstract with respect to any sensible extension of $\mathcal{L}$. These results require quantifying over all sensible extensions. The main technical contribution of this work is a method for doing this: we quantify over all languages defined by a suitable *structured operational semantics*[Plo81]. The bulk of this thesis is a study of sensible extensions of CCS. CCS is given by an elegant operational semantics[Mil83]: a collection of rules for deriving the behavior of, say, $P \| Q$ from the behaviors of P and Q. Many other languages in the style of CCS are defined by similar rules.[2]

For our language quantification results, we consider all *GSOS* languages, *viz.* languages defined by suitably generalized rules in the style of CCS. Rather than simply picking an arbitrary generalization, we make an attempt to find a maximal reasonable generalization. We isolate some essential properties of CCS, such as "there is a well-defined operational semantics." We prove that the GSOS languages have all these properties, and give counterexamples showing that minor violations of the GSOS discipline may lead to violations of the essential properties. In fact, most of the languages proposed in the CCS style have either been GSOS languages or violated the essential properties. Finally, the GSOS languages are fairly tractable to analysis. We thus believe that the GSOS languages are an appropriate class for study.

[2] Although we developed this method for CCS, it has proved useful in the investigation of Scott models for typed lambda calculus. In [Blo88a] we show that no PCF-like extension of PCF can be fully abstract for the Scott lattice model.

The theory of GSOS languages leads us to a new notion of program equivalence for CCS-like languages, which we call *ready simulation*; we investigate the theory of ready simulation. Ready simulation is an adequate semantics for all GSOS-definable languages. It is a fully abstract semantics for CCS extended by *process copying* and *controlled communication* primitives. It has several equivalent characterizations, in terms of state correspondences and modal logic, as well as the congruence generated by the extended CCS. It is strictly weaker than bisimulation [Mil88], and strictly finer than most other notions of process equivalence which have been proposed [Abr]. It seems to be a good replacement of (strong) bisimulation in those settings for which bisimulation is useful. It remains to be seen how it extends to other settings.

1.3 Notions of Program Comparison

The notions of program comparison, *viz. program equivalence* and *program approximation*, appear in many contexts in semantics and verification of programs. There are generally several possible definitions of when two pieces of code "do the same thing" or "should be considered as the same," and as we shall see the issues involved are quite subtle.

In general, we want to say that two pieces of code are equivalent if and only if it is impossible to tell them apart, and similarly for approximation. The 'if' direction (formalized later as *adequacy*) is clear; we do not want to identify programs that we can tell apart. The 'only if' direction (formalized as *full abstraction*) is somewhat less important; adequate, non-fully-abstract semantics can be quite useful. An intelligible non-fully-abstract semantics is generally more useful than an incomprehensible fully abstract one — and many fully abstract semantics are significantly more complicated than the adequate semantics for the same languages. (*E.g.*, the fully abstract semantics for PCF presented in [Mil77, Mul85] are much harder to construct and manipulate than the models of [Plo77, Blo88a].) However, all other things being equal, fully abstract semantics have a much closer match between program behavior and mathematical specification, and are therefore preferable.

There are philosophical and practical reasons for desiring full abstraction, or at least investigating whether or not it can be achieved without great sacrifice. The philosophical reasons follow Occam's Razor: distinctions between programs which behave identically are unnecessary, and therefore undesirable. The practical reasons come from the use of semantics in the specification of programs and languages. If one makes the definitions too restrictive, one runs the risk of losing some good implementations of programs and languages [Gut75].

Keeping this in mind, we make the following informal definition:

Informal Definition 1.3.1 *Two pieces of code should be considered to do the same thing if and only if one can be substituted for the other in any program and no difference in the behavior of the program can be seen.*

One piece of code, P, should be considered an approximation of another, P', viz. *P is usable in place of P', if P can be put in place of P' in any program and it is impossible to tell that P, rather than P', has been used.*

1.4 Observations, Languages, and Congruence

Definition 1.3.1 has two implicit parameters, which must be specified to get useful relations. We choose a set of *observations* $\mathcal{O}$ which we may make about programs. Without loss of generality, the observations are yes-or-no questions: "Does the program print a 1 on the screen at any time during its execution?" or "If you press the ENTER key twice, does the program ever halt?" If we want to see, say, what output number a program produces on input 78, we use an infinite family of observations: "Does the program produce output 0 on input 78?", "Does the program produce output 1 on input 78?", and so forth. If P is a program and o an observation, we write P **can-yield** o when P can yield the observation o.[3]

Notice that these observations are in general undecidable, given the program and a computer it will run on. From first principles, we know that anything interesting about programs is undecidable. However, we choose to restrict our attention to relations which are *semidecidable*, *i.e.*, recursively enumerable. If a program yields an observation, we may eventually discover the fact; if it does not, we may wait forever still hoping to observe it. It seems reasonable to require that all of our observations be of things which we may eventually discover. Distinctions based on properties which are not even semidecidable are not appropriate for computational situations. Of course, we may not want to use a kind of observation simply because it is semidecidable; *e.g.*, the observation "`The program has an even number of 'j's`" is decidable, but not generally a good observation.[4]

The other parameter to the congruence is a set of *contexts* $\mathcal{C}$ in which we may put programs. Generally, we will choose a programming language $\mathcal{L}$, and let $\mathcal{C} = \mathcal{C}_{\mathcal{L}}$ be the set of all contexts we can build from $\mathcal{L}$. For our purposes, a context is a piece of text with some *holes* (here written as X's) in it. Running a program P in a context $C[X]$ involves replacing the holes in the context by the text of the program.

[3] Precise definitions are given in later chapters.

[4] Note that in some languages, some version of the text of the program is available for inspection. In C, it is possible to take the address of a function and use it as a pointer to the object code of that function. In many LISPs, *e.g.* the GNU emacs Lisp used to write this thesis, the function `(symbol-function` X`)` returns the code (possibly compiled) of the function X. This function occurs 21 times in the standard code library for emacs, suggesting that it is useful in practice.

For example, let P be the program

```
print "hello, ";
print "world.";
```

and Q be the program

```
print "hello, world.";
```

P and Q, run in isolation, have the same input-output behavior — both print `"hello, world."` and then stop — and so one might want to consider them equivalent. However, if we are working in a programming language with an interleaving parallel construct, and if the `print` statement is atomic, P and Q can be told apart by running them in parallel with `print "real "`. P in this situation might output the string `"hello, real world."` but Q will not.

In general, parallelism (and especially interleaving parallelism) seems to make programs behave *nondeterministically*. The nondeterminism comes, at least initially, from having many possible orders of execution of programs running asynchronously. It is not the backtracking style of nondeterminism found in PROLOG or formal automata; it is merely the admission that a program might (for reasons which are invisible to the programmer and user alike) choose one of several possible next actions.

For this reason, we use the notion "P is capable of yielding o." We will consider the set $\{o : P \textbf{ can-yield } o\}$, and thus implicitly keep track of the observations that P is not capable of yielding. This definition is the formalization of Definition 1.3.1.

Definition 1.4.1 *Given a set $\mathcal{O}$ of observations and a programming language $\mathcal{L}$, the program P is* an approximation of Q with respect to $\mathcal{O}$ and $\mathcal{L}$, $P \sqsubseteq_{\mathcal{O},\mathcal{L}} Q$, *iff for all $o \in \mathcal{O}$ and $C[X]$ over $\mathcal{L}$, we have*

$$C[P] \textbf{ can-yield } o \textit{ implies } C[Q] \textbf{ can-yield } o.$$

Similarly, the programs P and Q are congruent with respect to the language $\mathcal{L}$ and observations $\mathcal{O}$, *written $P \equiv_{\mathcal{O},\mathcal{L}} Q$, iff for all observations $o \in \mathcal{O}$ and all contexts $C[X]$ over $\mathcal{L}$, we have*

$$C[P] \textbf{ can-yield } o \textit{ iff } C[Q] \textbf{ can-yield } o.$$

Note that $P \equiv_{\mathcal{O},\mathcal{L}} Q$ iff $P \sqsubseteq_{\mathcal{O},\mathcal{L}} Q \sqsubseteq_{\mathcal{O},\mathcal{L}} P$.

Different choices of observations and languages yield radically different comparisons. To choose notions of program comparison based on congruence, we must know what language and observations we care about. Even

more confusing is the possibility of language change: tomorrow's dialect of the language may be much more powerful than today's, details of our code which are hidden now may be revealed, and we may have to recode all our programs to compensate for that power and re-hide all the details.

The extreme cases are of observations $\mathcal{O}_0 = \varnothing$, and $\mathcal{O}_\infty$ being the set of observations "The program has P as its code" for all programs P. Neither of these cases is particularly useful for most purposes. $\equiv_{\mathcal{O}_0,\mathcal{L}}$ is the universal relation; $P \equiv_{\mathcal{O}_0,\mathcal{L}} Q$ holds for all P and Q. (Intuitively, if you don't care what a program does, all programs are the same to you.) $\equiv_{\mathcal{O}_\infty,\mathcal{L}}$ is the identity relation; $P \equiv_{\mathcal{O}_\infty,\mathcal{L}} Q$ iff P and Q have the same code.

Notice that in these cases, the language $\mathcal{L}$ does not effect the relation, and that language and observations may be traded off against each other. (The language and observations could be combined into a single set; however, we often vary one and not the other, so we separate them.) In general, the language will effect the relation dramatically; for example, a language $\mathcal{L}_\infty$ with a `symbol-function` operation — *e.g.* `symbol-function`(P) returns the text of P as a string — allows us to observe the text of a program even if $\mathcal{O}$ only includes, say, the ability to observe the string that a program prints when run without input.

In general, $\mathcal{O}$ and $\mathcal{L}$ are taken to be something important in practice. Two common choices are to observe the output of the program, and to observe which specifications the program satisfies. $\mathcal{L}$ is generally the programming language in which the programs are written. Of these, the output is the more fundamental: it is easy to see that a program has produced a particular output, but it is harder to tell that it meets a specification. Still, both kinds of observations are useful.

In many circumstances, including the main topic of this thesis, a congruence will be invariant under a wide range of language and observations; that is, $P \equiv_{\mathcal{O}_1,\mathcal{L}_1} Q$ iff $P \equiv_{\mathcal{O}_2,\mathcal{L}_2} Q$ for all $\langle \mathcal{O}_1, \mathcal{L}_1 \rangle, \langle \mathcal{O}_2, \mathcal{L}_2 \rangle$ in some set $\mathcal{S}$. This suggests that the congruence is a potentially good one. Such congruences can be used without change in many circumstances; *e.g.*, programs which meet their specifications in the 1989 version of the programming language will continue to meet them in the 1993 version, if the designers are willing to stay within the range for which the congruence is appropriate. The congruence we propose in this thesis is good for all languages which are definable by *well-structured* rules in a sense defined in Chapter 3, including most languages in the style of CCS and CSP [Mil80, Hoa85, Bou85, BV89].

1.4.1 CCS and Program Equivalence

CCS [Mil80, Mil83, Mil84, Mil88] is a simple core language for parallelism. The theory of CCS is usually developed with a particular semantics in mind, the elegant *strong bisimulation* (or simply *bisimulation*) semantics of Milner [Mil88, Mil83] and Park [Par81]. The mathematical theory of bisimulation is

quite powerful and appealing. As a case study in our methods of evaluating programming languages and equivalences, we will investigate the appropriateness of bisimulation semantics. We concentrate on equivalence because there is no notion of approximation associated with strong bisimulation.

Recall that congruence is a relative notion, depending on a set of observations and a programming language. We define *CCS congruence* to be congruence with respect to a simple set of observations, the *completed finite traces*, and the language CCS. Completed finite traces give the most information of any of the usual kinds of observations on CCS programs[Abr]. So, CCS congruence is the *finest* kind of congruence relation commonly used with CCS-like languages: that is, if P and Q are CCS congruent, then they are congruent with respect to the other observations and languages commonly used.

One of the crucial theorems about bisimulation semantics is adequacy: if P and Q are considered equal in the bisimulation semantics, then P and Q are CCS congruent, or equivalently the bisimulation semantics are at least as fine as CCS congruence. In practical terms, this means that:

Fact 1.4.2 *If P and Q can be shown to be equal in the bisimulation semantics, then P can be substituted for Q (and vice versa) in any CCS context, and no difference can be seen.*

However, as our discussion suggested, the converse fails; bisimulation is not fully abstract. There are two programs P and Q which bisimulation considers *different*, but which are still CCS congruent. The difference between the two is slight; essentially, they differ in the order of assignment to internal variables (see Figure 1.1). CCS does not provide any way to inspect the internal state of a process, for much the same reason that ALGOL does not provide any way to observe the values of **own** variables of a procedure from the outside; there is no way in CCS to tell P and Q apart.

An example of two processes which are CCS congruent (and indeed ready similar) but not the bisimilar is a *lossy delay link*. A lossy two-stage link repeatedly accepts an input value delays one click, and produces as output either v or a signal x saying that v was lost in transit. We present two ways to specify the lossy delay link in CCS. The first always receives its input correctly, but may lose it during the delay; the second may lose it either initially or during the delay. See Figure 1.1 for an informal version of the code.

For simplicity we assume that 0 is the only possible input value; the example works with any set of inputs. i_0 is the action of accepting 0 as input; d is the delay; o_0 and o_x is the action of producing 0 and x respectively as output.

$$\mathrm{LDL}_1 \quad = \quad i_0(d.o_0.\mathrm{LDL}_1 + d.o_x.\mathrm{LDL}_1)$$

```
LDL1:begin
     repeat
          read v;
          write delay_signal;
          possibly_lose(v);
          write v;
     forever
     end LDL1
```

and

```
LDL2: begin
     repeat
          read v;
          possibly_lose(v);
          write delay_signal;
          possibly_lose(v);
          write v
     forever
     end LDL2
```

where possibly_lose(v) is defined by:

```
     procedure possibly_lose(var v);
     begin
          nondeterministically skip or v := x;
     end
```

Figure 1.1: Informal code of the lossy link

$$\mathrm{LDL}_2 \quad = \quad i_0.d.o_x.\mathrm{LDL}_2 + i_0(d.o_0.\mathrm{LDL}_2 + d.o_x.\mathrm{LDL}_2)$$

If a program is specified up to bisimulation, then a good deal of its internal behavior is fixed — even when this internal behavior cannot be seen externally. At best, this is a nuisance. The programmer must make sure that that the input value has two chances to get lost, and the specifier must write down this kind of tedious detail. The specification may insist on an inefficient and confusing program when an efficient, straightforward, and utterly indistinguishable program could be written.

There are two approaches to mending the failure of full abstraction: we could change bisimulation, or we could change the language (which changes the congruence). Bisimulation, as we shall see, is an invariant notion: it does

not depend on the language or set of observations. Congruence, however, is a relative notion. CCS congruence comes from fixing plausible choices of observations and language; these choices happen to fail to match bisimulation. It might be the case that some other plausible choices would succeed.

Using our techniques of studying CCS-like languages, we investigate and ultimately reject this possibility. We define large classes of observations and languages, and argue formally and informally that they include all the plausible choices. We characterize the strongest congruence generated by these classes, giving it the mathematical formulation of *ready simulation*. That is, if two processes are ready similar, then they agree under all plausible choices of observation and language. It is straightforward to show that ready simulation is weaker than bisimulation, and thus bisimulation cannot be captured by a congruence.

1.5 Limitations, Apologies, and Goals

The arguments about bisimulation and ready simulation in this thesis are incomplete in one major respect. We work mainly with strong bisimulation; most work in bisimulation and CCS uses the related notion of weak bisimulation. Thus, the results in this thesis are not necessarily of great concern to researchers in CCS.

For this reason among others, we do not argue that ready simulation should be the One True Process Equivalence. We do argue that it is as suitable as strong bisimulation for those areas for which strong bisimulation is intended, and that it is on firmer philosophical foundations as indicated in the first section of this introduction. We regard the relation of ready simulation as secondary to the general theoretical framework.

However, the relation of weak bisimulation[5] is of primary concern in the theory of parallel processes. Strong bisimulation is a very fine equivalence; in particular, it pays very close attention to matters of time. Weak bisimulation is an attempt to preserve the ideas and powers of strong bisimulation, yet ignore time as much as possible. Other researchers have invented other variants of bisimulation as well.

We believe that our work is relevant to the bisimulation community, and the semantics community as a whole, for the following reasons:

1. Most variants of bisimulation (of which weak bisimulation is the most important) are attempts to add something — asynchrony, divergence, probability — to bisimulation while preserving as much of its quite

[5]Weak bisimulation is often called "observational equivalence." We avoid this name, as it sounds too much like the notion of equivalence with respect to a set of observations we have introduced informally. Ironically, it seems (*cf.* Chapter 5) that "observational equivalence" cannot be understood as equivalence with respect to any reasonable sort of observations.

attractive structure and theory. Most variants of bisimulation are quite fine equivalence relations. Our canonical example of programs which should be identified but which bisimulation distinguishes are generally distinguished by the variants of bisimulation. It seems quite likely that our *negative* results — those arguing that bisimulation cannot be explained in clean terms — will apply, *mutatis mutandis,* to the variants of bisimulation as well. Preliminary research suggests that this is the case (in fact, there is some hope for the positive results as well); however, much work remains to be done.

2. This work is in part a challenge to the proponents of bisimulation, suggesting grounds on which bisimulation is questionable and urging them to defend it. The challenge is that (1) bisimulation is not quite appropriate philosophically for computer science and (2) some other equivalence may prove to be more appropriate philosophically and somewhat better practically. The possibility of such an improvement is certainly worthy of further investigation.

   Another likely scenario is that bisimulation will be less appropriate philosophically but more useful in practice than other contenders. This, too, will be a good thing to know. Bisimulation could be used as the method of choice when it applies (it is applicable to a wide range of problems), but more general tools are required when it does not apply.

3. The general challenge is not limited to bisimulation; it applies to all the variant notions of process equivalence. In fact, there is a large collection of semantics for CCS-like languages alone [Abr], to say nothing of other sorts of language. Picking the right kind of semantics from this collection is a nontrivial matter; probably different semantics are suited for different tasks. The work in this thesis gives some methods to sort through the collection: some properties (*e.g.*, being a congruence with respect to something pleasant, and having appropriate logical characterizations and decision procedures) which some semantics have and others do not. We have worked out a case study indicating the sort of analysis one might perform to understand a notion of program equivalence.

   The ultimate goal here is a list of semantics and their advantages and disadvantages, together with some methods for choosing a good semantic notion for various tasks and some other methods for creating and evaluating new notions as necessary.

1.6 Related Work

The theory of CCS and bisimulation has been extensively studied in [Bou85, GS85, HM80, Mil80, Mil81, Mil83, Mil84, Tho89]. [Mil88] is perhaps the most recent and detailed survey of the theory of CCS. The connections between logic and bisimulation, in particular Hennessy-Milner logic, are discussed in [HM85, Mil88, GS86] Other research in the comparison of various kinds of semantics for CCS and related languages can be found in [Abr, AV, GV89].

Other researchers have investigated the possibility of understanding bisimulation as a congruence with respect to some extension of CCS. Samson Abramsky [Abr87] gave a language, amounting to the coding of Hennessy-Milner formulas as operations, in which the distinctions made by bisimulation are observable. The language in Chapter 8 is a variation of Abramsky's language. Frits Vaandrager and Jan Friso Groote [GV89] generalize CCS in a somewhat different way than we do, and get similar results but a finer equivalence relation. [dS85] shows that all operations definable by structured rules like those of CCS (a small subset of the GSOS rules) can be defined in a suitably large dialect of CCS/Mieje.

Work using bisimulation methodology to verify programs and protocols includes [Par87b, Par87c, Par87a, ST87, Wal87]. It is often desirable to extend bisimulation to account for various phenomena, such as action priority [CH88] divergence and deadlock [vGW89, Wal88] and probability[LS88].

Work related to CCS but focusing on equivalences other than bisimulation includes [BIM88, BM89, BM90, dNH84, GS87, GV89, Hen83, Hen85, LS88, Phi86, Pnu85, VG88, dNH84]. The theory and practice of CSP is discussed in [Bro83c, Bro83a, Bro83b, BHR84, BHR84, FHLd79, Hoa78, Hoa85, OH86] and elsewhere. Much of this work focuses on the *failures* semantics, which is an elegant fully abstract semantic based on annotated partial traces. [Bro83b] give operational semantics in terms fairly similar to those of CCS.

An area sharing some kinship with the theory of CCS and CSP is that of Process Algebra, exploring equational theories of processes. (Bisimulation classes of CCS processes form models for suitable process algebras.) Process Algebras are somewhat related to this thesis, in that the main concerns are the introduction and examination of various programming constructs. New constructs are introduced equationally, rather than operationally, and it is not always clear that there is an appropriate and straightforward operational semantics. Some work in this area includes [AB84, BB88, BV89, BvG87, BK82, BK84b, BK84a, BK85, BT86, vGR89, vGV87, dBBKM82, dMOZ86, dBZ83].

The field of "true concurrency" works from the hypothesis that there is an important difference between parallelism and interleaving. In CCS and related areas, if a and b are atomic actions, then there is no significant differ-

ence between running a and b in parallel and nondeterministically running a followed by b or b followed by a. (Briefly, the philosophy is that atomic actions are indivisible, and only one of them can be seen at a time; thus anyone looking at the two processes would see an a followed by a b or vice-versa.) True concurrency explores the regions in which this assumption is not true, including the settings of Petri nets, pomsets, and event structures. Work in this area includes [Bou85, BRdS85, BC87, BC89, CH87, CMP87, Gis84, GG89, vGW89, PP88, Vaa89, Vaa89].

In this thesis, we propose a notion of *ready simulation* between processes. We are lead to ready simulation from semantical concerns; *e.g.*, ready simulation semantics are fully abstract with respect to CCS with copying and asymmetric communication primitives. Ready simulation includes a *positive* part (if the smaller process can do something, the larger one can as well) and a negative part (if the smaller process can't do something, the larger one can't either).

Researchers in the verification community have been lead to notions similar to ready simulation for entirely different reasons; *e.g.*, that the related notions give methods of proving that programs meet specifications. These notions are generally instances of *monosimulation*, *viz.* a weaker version of ready simulation using only the positive part of its definition.

Lynch and Tuttle [LT88] introduce *possibilities maps*; Abadi and Lamport [AL88] introduce *refinement maps*; and Alpern and Schneider [AS89] introduce *invariants*[6]. All of these map or relate states in a program to states in its specification. The main question concerning these mappings is whether or not they exist; if they exist, then the program meets its specification. We relate these concepts to this thesis by noting that the property of a possibilities/refinement mapping existing is precisely the relation of monosimulation. If the experience of the verification community is any guide, the general proof methodology of relating states in the program to states in the specification is a powerful one.

The soundness of monosimulation techniques, and therefore the techniques of Lynch-Tuttle and Abadi-Lamport, depend on the collection of operators available for manipulating programs. Lynch and Tuttle define two operations — parallel composition and hiding — on I/O automata; Abadi and Lamport do not consider any particular programming language. Their monosimulation-based techniques are correct for the operations they define, but inadequate for CCS. Monosimulation is incomparable with CCS congruence: there are monosimilar processes which are not congruent, and congruent processes which are not monosimilar.

As a corollary of this thesis, algorithm verifiers who are able to show

[6]Invariants are only one part of Alpern and Schneider's method, and a relatively minor part at that. The discussion in this paragraph refers mainly to the work of the other researchers.

ready simulation rather than simply monosimulation can show a stronger theorem: their algorithm is correct in all CCS-like contexts, rather not merely the fairly sparse languages they ordinarily use. This may be of assistance to the verification of large programs which use the algorithms as subroutines and combine then with CCS or similar combinators. However, as we have said, ready simulation is an appropriate replacement for strong bisimulation rather than weak; it does not adequately take into account hidden moves. Proofs of ready simulation are quite strong — when a process meets its specification up to ready simulation, it takes just the number of hidden steps that the specification does. It is unlikely that many algorithms will meet their specifications up to ready simulation.

Taxonomy of Formal Methods

In this section, we give some idea where the results of this thesis fit in the context of formal methods. We present a brief survey of the problems that formal methods are intended to solve, and the main methods related to the theory in this thesis. The problems include:

Specification: Giving a formal description of what a program is supposed to do.

Algorithm Verification: Algorithm verification involves verifying ingenious programs which are true for subtle reasons (*e.g.*, the network synchronizer verified in [FLS87] and the atomic register algorithms of [Sch88, Blo88b]) This aspect of verification generally requires clever, ad-hoc arguments and methods.

Large-Scale Verification: Verifying possibly immense but more or less straightforward programs. By preference, this would be done mechanically, or at least with mechanical support — and would probably depend on a library of verified algorithms.

Programming: Programmers may use formal techniques for a variety of things while programming. The availability of a formal semantics of a language can be a significant advantage over ambiguous English definitions or definition by compilation; the availability of a formal specification of a program can be a significant help in getting the program right. Also, knowing theorems about the programming language — that, say, C's `for`-loop and `while`-loop are interdefinable — is useful for writing and maintaining programs.

Language Design: Formal methods have proved quite helpful for the design, specification, and even implementation [KH89] of programming languages.

Theory of Programming Languages: Semantical methods allow the comparison of programming languages with each other, as well as investigating programs written in them. It is possible to state theorems implying, for example, that one form of parameter passing more powerful than another.

Negative Results: Formal methods allow the statement, and occasionally the proof, of impossibility and lower bounds. For example, the nonexistence of various Byzantine agreement protocols and the lower bounds on message complexity of function evaluation require formal models. It is also possible to prove negative results within a programming language: *e.g.*, that various kinds of fair merge cannot be constructed from other kinds of fairness in dataflow nets [PS87].

A variety of techniques have been proposed to address these goals. In general, techniques suitable for one goal are not necessarily helpful toward others. Four general schools of techniques stand out as particularly relevant to this thesis. The boundaries between schools are fuzzy, and the schools overlap considerably.

Combinatorial methods have been developed mainly by algorithm verifiers, *e.g.*[LT87, AL88, AS89, FLS87, LT87, HO85, WGS87, Blo88b, Sch88] among others, to allow them to bring the full power of mathematics to bear on their algorithms. The formal model is designed to allow the statement and proof of correctness conditions, giving the prover the ability to observe and take advantage of all aspects of the problem.

The I/O automata of [LT87] and the state machines of [AL88] and [AS89] are good examples of combinatorial methods. Processes are considered as infinite-state automata, with some additional attributes to deal with issues such as fairness. There are few constraints on the language used to describe automata; all of mathematics is available. There are few if any operations on processes: I/O automata have parallel composition and action hiding. Problems are specified by automata; an automaton P meets the specification S if every completed sequence of actions of P is also possible for S. Constraints on the model forbid trivial solutions.

The main application of combinatorial methods is the proof of correctness of algorithms and protocols. Generally these algorithms are true for subtle reasons, and so require subtle arguments to verify in an intelligible manner. One common style of reasoning is to have the state of the automaton be a snapshot of the system (possibly including extra information about the previous history, and occasionally about the future nondeterministic choices the system will make). An arbitrary execution history of the program P is chosen, and shown to be a possible history for the specification

S as well.[7] The role of the formal model is to allow the statement of the correctness condition as a mathematical problem, and then to get out of the way; the verifier will use whatever mathematical tools are applicable to solve the mathematical problem.

Temporal logic [CES83, SPE83, MP83, OL82, Pnu77, NGO85] is concerned with assertions about time: "eventually the program prints a result," "Whenever anyone who wants to do something, they will eventually be able to" and similar constructs. Temporal logics are quite powerful for specifying concurrent programs, in the sense that they allow concise and clean specifications of many kinds of important properties. (They are weaker than combinatorial specifications, in that there are certain properties which can be specified by automata but not by temporal logics. This holds for automata-theoretic reasons; in practice, many intelligible and useful properties seem to be expressible in temporal logic.) Temporal methods are often combined with combinatorial methods [Lam88, OL82] by the addition of predicates telling which statements the programs are executing at the moment.

Combinatorial methods and temporal logic are powerful tools for specification and verification of algorithms. However, they are not intended to serve as semantics for programs in any useful way. In particular, they are not generally *compositional*; in general, reasoning about systems is done considering the system as a whole, rather than by reasoning about its components individually. (The network composition operator of I/O automata is to some extent an exception to this.) Programs are highly structured objects, and this structure has proved to be helpful in designing and writing them. It seems reasonable that analyzing and verifying them could also be helped by working with their structure. Hoare logic [Hoa69], denotational semantics [Sco76], and structured operational semantics [Plo81] are approaches toward exploiting program structure.

Hoare logic[Hoa69], is a kind of logical calculus for manipulating assertions of the form $\{\varphi\}P\{\psi\}$ — that, if a fact φ holds and a program P is executed, then when and if P finishes another fact ψ will hold. For example,

$$\{x = n \wedge y = 0\}\textbf{while } x > 0 \textbf{ do } (x := x - 1;\ y := y + 1)\{x \leq 0 \wedge y = n\}.$$

A Hoare logic for a programming language $\mathcal{L}$ is a collection of inference rules telling how the facts concerning subterms of an expression are to be combined to give the facts true of the whole expression:

$$\frac{\{\varphi\}P\{\psi\}, \{\psi\}Q\{\theta\}}{\{\varphi\}P; Q\{\theta\}} \tag{1.1}$$

Hoare logic was intended to be applicable to sequential programs, and in particular batch programs: the batch program was to be supplied an

[7] There are a number of variations on this method. For example, it is sometimes helpful [LT87] to consider a sequence of specifications $S_1, \ldots, S_n$ in which each specification is an implementation of the next.

input satisfying some conditions, and it would then produce an output satisfying some other conditions. This paradigm does not generally apply to concurrent programs, in which patterns of interaction are most relevant.

For example, the rule (1.1) is not true if a concurrent process is allowed to change x while the **while**-loop is running. Various researchers have adapted Hoare logic to concurrency [Lam80, LS84, OG81, LG81]; for example, if other processes are running concurrently with the **while**-loop, they may be forbidden to change x or y. However, these methods give fairly complicated proof systems and obscure the fundamental questions we are interested in.

The work in this thesis is is best classified as **(mathematical) semantics** or **theory of programming languages**; working with mathematical formulations of the meanings of programs. The denotational semantics of λ-calculus and programming languages based on it [Sco76, Sto77] is the best-known and most successful example of semantics; the field of the semantics of concurrency is hoping to achieve similar successes in some finite amount of time. Programming languages, as well as programs, are the objects of interest; the questions it asks are often more interesting than immediately practical.

Semantic methods can be of some assistance in the practical matters of specification and verification. Semantics provides the model theory to accompany the proof theory of Hoare and temporal logic, proving that the proof rules are sound. Semantics also allows the investigation of related questions; *e.g.*, the question of whether a proof system should have additional rules can be answered (negatively) by a relative completeness theorem. Many language designers [KH89, Arv89] have found semantic methods helpful in the design and specification of programming languages.

By in large, semantical methods are structural: the meaning of a program is generally a combination of the meanings of its subprograms. CCS is an excellent example of this. Programs are built from a rich set of combinators, with an elegant algebraic theory. The hope is that the algebra will assist programmers and verifiers of large programs, offering support and structure to managing large problems. The hope is to manage complexities of scale, allowing the comprehension and verification of large but generally simple programs (probably making use of a library of algorithms verified by other methods.)

Semantics also concerns itself with other questions, which are harder to formulate in the other methods. Denotational semantics was originally intended as a way of formally specifying languages [MS76], in terms of more or less familiar and useful mathematics. Denotational semantics can also guide language design in certain directions, exposing constructs as questionable or incompatible, and occasionally suggesting new constructs. For example, it is impossible to write a procedure taking two call-by-name integer parameters which exchanges their values; this suggests that some other

variable parameter-passing mechanism, such as call-by-reference, might be an appropriate addition (or replacement) to call-by-name in ALGOL-60.[8]

Semantics also allows the study of the theory of programming languages. Computer scientists and programmers spend a good deal of time informally discussing the relative merits of their favorite languages. Denotational methods allow the formalization of these discussions. For example, it is often possible to exhibit a program fragment which can be written in one language but is impossible to simulate in another; *e.g.*, it is straightforward to write a routine which swaps the values of its integer **var** parameters in Pascal, but impossible in ALGOL-60. Conversely, it is possible to show that a language is as powerful as possible within a particular framework: *e.g.*, PCF with parallel conditional and existential is capable of expressing any r.e. function in the Scott semantics [Plo77], and CCS/Mieje is capable of expressing any function definable by rules of a certain form [dS85].

The most interesting results in this thesis concern the metatheory of CCS-like languages. We define a class of languages, the GSOS languages, and argue that it is an appropriate generalization of CCS. It includes CCS and a good deal more (in particular the results of [dS85] do not apply). All GSOS languages have the essential basic properties of CCS; *e.g.*, they have a well-defined operational semantics. Furthermore, any of the obvious extensions of the GSOS format allow languages which violate one or more of these essential properties.

Having defined the GSOS languages, we are in a position to investigate what CCS-like languages can and can't do. We focus on bisimulation, the notion of program equivalence of CCS. Our attempt to understand bisimulation as a congruence leads us to a related notion, which we call ready simulation Ready simulation is an adequate program comparison for all GSOS languages. Investigation of full abstraction suggests that two operations, of *copying* and *controlled communication* be added to CCS; ready simulation is just right for the enhanced CCS, and will continue to be just right for any CCS-like extension of it. In particular, bisimulation is not fully abstract for any CCS-like language.

As a technical exercise, we also investigate some other limitations of CCS-like languages. We show that there are (recursive) automata which cannot be defined in any CCS-like language, and that certain operations (such as sequencing, "do P and then do Q") require some of the powerful features available in the GSOS format to define.

1.7 Overview of the Thesis

1. Introduction and Justifications

[8] Semantics is not the only way to come to this conclusion.

2. Introduction to CCS and Bisimulation

3. Well-Structured Languages. To justify bisimulation in terms of trace congruences, we would have to find some reasonable language $\mathcal{L}$ in which bisimulation was trace congruence. As we are arguing that bisimulation cannot be justified in these terms, we have to quantify over all "reasonable" languages. In this chapter, we present a large class of languages called the GSOS languages, generous enough to include CCS and most of its relatives. We show that any GSOS language possesses the essential mathematical properties of CCS. We consider the obvious extensions of the GSOS languages, and show that each class of extensions includes a language which violates the essential properties we require. We conclude that the class of GSOS languages is an appropriate class of languages which should contain all the "reasonable" ones.

4 Theory of Ready Simulation. This chapter contains the core of the theory of this thesis. We introduce the notion of *ready simulation*, an equivalence between programs, and show that it is a precise characterization of congruence with respect to the class of GSOS languages. The main result is that the following four relations coincide with each other and that none is equal to bisimulation, and that therefore bisimulation cannot be understood as trace congruence with respect to any reasonable language:

1. Ready Simulation: a relation defined in the style of bisimulation.
2. Denial Formula Equivalence: equivalence with respect to a class of modal formulas.
3. Trace congruence with respect to all GSOS languages (and by the thesis of the previous chapter, all reasonable languages.)
4. Trace congruence with respect to a particular extension of CCS.

5 Experimental Equivalences. In the previous chapters we varied one of the two parameters of the notion of congruence; we kept the observations, the other parameter, fixed. It is difficult to find a good definition of "well-structured observation." We propose a variety of plausible kinds of observation, and show that none of them generates bisimulation as a congruence. In fact, many of them generate ready simulation.

6 Finite and Regular Processes Bisimulation is a useful tool, despite the fact that it is not a natural congruence relation. In general, it is probably better to have a pragmatically useful semantics with weak philosophical justification than a philosophically solid but unusable semantics. We present some excursions in the theory of ready simulation

and of GSOS languages which suggest that ready simulation should have a methodology which is approximately as good as that of bisimulation, and in fact is very similar in flavor. Much of the bisimulation methodology comes form Park's definition of bisimulation; ready simulation has a similar definition, and preliminary indications (*e.g.*, the proofs in this thesis) suggest that it has a similar methodology. In this chapter, we present two main results: ready simulation has a sound and complete axiomatization on a class of *finite* processes, and can be decided in polynomial time on the larger class of *regular* processes. Similar results hold for bisimulation; however, many other variations on bisimulation and congruence are NP-hard or worse.

7 Metatheory of GSOS Languages: We present some further mathematical properties of GSOS languages, characterizing some of the limitations of these languages, and showing how certain of their features influence their expressiveness. These results are of some interest for their own sake; they also illustrate the theoretical power which can be achieved by considering well-structured rules. The main results are:

1. A partial characterization of the processes which cannot be defined by programs in any GSOS language.
2. A proof that distinctions between programs cannot be reduced to observations of a single bit.
3. A proof that certain programming constructs, notably sequencing (`do` P `first, then do` Q) cannot be defined without negative rules. (Negative rules are one of the more surprising and dangerous features of GSOS languages. Languages which do not include them are probably preferable to ones which do.)

8 Global Testing: It is trivial to create a (ridiculous) language in which bisimulation is trace congruence. In this chapter, we extend some work of Samson Abramsky and give a somewhat more reasonable language with this property: a good-faith effort to make bisimulation into a congruence. The language has some peculiar constructs in it, which seem to be impossible to eliminate.

Synchronization Tree Isomorphism	$P \equiv Q$	The finest acceptable process equivalence in this theory. We insist that processes with the same tree be identified, and that all languages respect tree isomorphism. This relation is *too fine*; we want the non-isomorphic processes a and $a + a$ to be identified.
Bisimulation	$P \underline{\leftrightarrow} Q$	Generally accepted in this community to be the *finest* acceptable process equivalence; that is, if P and Q are bisimilar, then P and Q ought to be considered identical. Solves the a vs. $a + a$ problem of tree isomorphism.
Ready Simulation (GSOS Congruence) ($\frac{2}{3}$-bisimulation)	$P \rightleftharpoons Q$	Congruence with respect to all well-structured languages. See Theorem 4.5.1 for a map of equivalent definitions.
Trace Congruence	$P \equiv_{tr}^{\mathcal{L}} Q$	Trace equivalent in all $\mathcal{L}$-contexts. The fineness of this relation depends on $\mathcal{L}$.
Trace Equivalence (Automaton Equivalence)	$P \equiv_{tr} Q$	P and Q have the same finite completed traces. Not an adequate semantics for most languages in the style of CCS.

(Except for trace congruence, which depends on $\mathcal{L}$, these equivalences are in order of fineness.)

Figure 1.2: Main Equivalences Considered In This Thesis

Chapter 2

Preliminaries

2.1 Unavoidable Notation

By in large, we use standard mathematical notation. When there is some ambiguity (*e.g.*, whether $\subset$ means proper or improper set inclusion), definitions are given in the appendix (or, as in this case, unambiguous notation ($\subsetneq$ and $\subseteq$) is used.)

We frequently write $\vec{x}$ for the sequence $x_1, \ldots, x_n$. Generally, the number of elements in the sequence is given by context; when necessary we state it explicitly. We extend relations on items to relations on sequences pointwise; *e.g.*, if $\simeq$ is a relation between x_i's and y_i's, we write $\vec{x} \simeq \vec{y}$ for $\forall i \in [1 \ldots n].\, x_i \simeq y_i$, where $[m \ldots n]$ is the set of integers between m and n inclusive.

A list of notation introduced in this thesis, with page and definition numbers, is provided in Appendix B.

2.2 Introduction to CCS

2.2.1 CCS Terms

We describe the calculus CCS, approximately as presented by Milner in [Mil88].[1] First we have the *actions*, which are uninterpreted symbols. Fix a set A of *names* and a set $\bar{A}$ of *co-names*, with a bijection in each direction $a \mapsto \bar{a}$ such that $\bar{\bar{a}} = a$. Names and co-names are meant to be receives and sends respectively. Fix a distinguished action τ, the *silent action*, not in A or $\bar{A}$. The set Act of actions is $A \cup \bar{A} \cup \{\tau\}$. Fix also a disjoint infinite set of *process variables*. The *CCS terms* are given by the following (infinitary) grammar; P, Q, R, S range over CCS terms.

[1] Most differences are notational: we write "aP" for Milner's "$a.P$" and "fix $\left[X_i; \vec{X} \Leftarrow \vec{P}\right]$" for "$FIX_j\{X_i = P_i : i \in I\}$". We extend the language with $P \parallel Q$, simple interleaving parallelism without communication.

- αP whenever $\alpha \in \mathsf{Act}$. This is intended as a process which performs the action α and then behaves like P; it is called "α-prefixing".
- $\sum_{i \in I} P_i$, where I is an index set of any (finite or infinite) cardinality. This is a process which can choose to behave like any of the P_i; it is called "summation" or "choice."
- $P \mid Q$, intended to represent P and Q running in parallel with possible communication. It is pronounced "P in parallel with Q."
- $P \parallel Q$, running P and Q in parallel without communication. It is pronounced "P interleaving with Q."
- $P \backslash L$, where $L \subseteq A$. This is like P, except that the actions in $L \cup \bar{L}$ are prohibited; it is pronounced "P with L forbidden." We write $P \backslash a$ for $P \backslash \{a\}$.
- $P[f]$ where $f : \mathsf{Act} \to \mathsf{Act}$ satisfies $f(\tau) = \tau$ and $f(\bar{a}) = \overline{f(a)}$. This renames the actions of P via the function f; for example, if P can perform an a, then $P[f]$ can perform an $f(a)$.
- $\mathrm{fix}\left[X_i; \vec{X} \Leftarrow \vec{P}\right]$, where $\vec{X}$ and $\vec{P}$ are possibly infinite vectors of the same length. This behaves like the X_i component of a distinguished solution to the recursive equations $\vec{X} = \vec{P}$; it is called "recursion" or "fixed point."

Some special cases are useful enough to deserve special notation:

$$\begin{aligned} \mathbf{0} &= \sum_{i \in \varnothing} P_i \\ P_1 + P_2 &= \sum_{i \in \{1,2\}} P_i \\ \mathrm{fix}\,[X \Leftarrow P] &= \mathrm{fix}\,[X; X \Leftarrow P] \end{aligned}$$

The usual notions of free and bound variables apply; $\mathrm{fix}\left[X_i; \vec{X} \Leftarrow \vec{P}\right]$ binds each variable in $\vec{X}$ in each P_j. We use $P[\vec{X} := \vec{Q}]$ for simultaneous substitution of Q_j for X_j in P with renaming to avoid capture; see [Bar81] for details. Note that summations and recursions are allowed to be infinite. It is generally preferable for programs — at least programs that we intend to write and run — to be finite terms. (It is often useful to write specifications as, say, infinitary recursions with one variable X_i for each state of the system.)

2.2.2 Operational Semantics

The operational semantics of CCS is given as a *Labeled Transition System* [Kel76]; a structure $\langle S, A, \dot{\rightarrow}\rangle$ where S is a set of *states*, A a set of *actions*, and $\dot{\rightarrow}$ is a multi-subset[2] The phrase $\langle P, a, P'\rangle \in \ \dot{\rightarrow}$ is usually written $P \stackrel{a}{\rightarrow} P'$, and called an *a-transition* of P. It is interpreted as the process P performing an a-action and then behaving like P'.

The labeled transition system of CCS terms has states equal to the closed process terms, actions given by Act, and a transition relation given as the smallest relation $\dot{\rightarrow}$ satisfying the following rules. That is, whenever the relations above the line hold, the relation below it must hold as well.[3] If there are no relations above the line, as for the rule for aP, then the transition always is possible. Note that the derived operation $\mathbf{0}$ has no rules; it is intended to be a *stopped process* unable to do anything.

$$\frac{}{aP \stackrel{a}{\rightarrow} P} \qquad \frac{P_i \stackrel{a}{\rightarrow} Q \quad (i \in I)}{\sum_{i\in I} P_i \stackrel{a}{\rightarrow} Q}$$

$$\frac{P \stackrel{a}{\rightarrow} Q}{P \parallel P' \stackrel{a}{\rightarrow} Q \parallel P'} \qquad \frac{P' \stackrel{a}{\rightarrow} Q'}{P \parallel P' \stackrel{a}{\rightarrow} P \parallel Q'} \tag{2.1}$$

$$\frac{P \stackrel{a}{\rightarrow} Q}{P \mid P' \stackrel{a}{\rightarrow} Q \mid P'} \qquad \frac{P' \stackrel{a}{\rightarrow} Q'}{P \mid P' \stackrel{a}{\rightarrow} P \mid Q'}$$

$$\frac{P \stackrel{a}{\rightarrow} Q, \quad P' \stackrel{\bar{a}}{\rightarrow} Q'}{P \mid Q \stackrel{\tau}{\rightarrow} P' \mid Q'}$$

$$\frac{P \stackrel{a}{\rightarrow} Q, \quad a \notin L \cup \bar{L}}{P\backslash L \stackrel{a}{\rightarrow} Q\backslash L}$$

$$\frac{P \stackrel{a}{\rightarrow} Q}{P[f] \stackrel{f(a)}{\rightarrow} Q[f]}$$

$$\frac{P_j\left[\vec{X_k} := \overrightarrow{\mathrm{fix}\left[X_k; \vec{X} \Leftarrow \vec{P}\right]}\right] \stackrel{a}{\rightarrow} Q}{\mathrm{fix}\left[X_j; \vec{X} \Leftarrow \vec{P}\right] \stackrel{a}{\rightarrow} Q}$$

[2] A multi-subset is much like a subset, except that elements may occur more than once. We use multisets for aesthetic reasons, so that in the next section we may truthfully say that the synchronization tree of $P + Q$ is the synchronization trees of P and Q with the roots identified; in particular, so that a and $a + a$ generate distinct synchronization trees.

[3] The number of times that $P \stackrel{a}{\rightarrow} Q$ — more accurately, the number of times that $\langle P, a, Q\rangle$ occurs in the multiset $\dot{\rightarrow}$ — is defined in the obvious way; *e.g.*, if $P \stackrel{a}{\rightarrow} R$ p times and $Q \stackrel{a}{\rightarrow} R$ q times, then $P + Q \stackrel{a}{\rightarrow} R$ $(p + q)$ times.

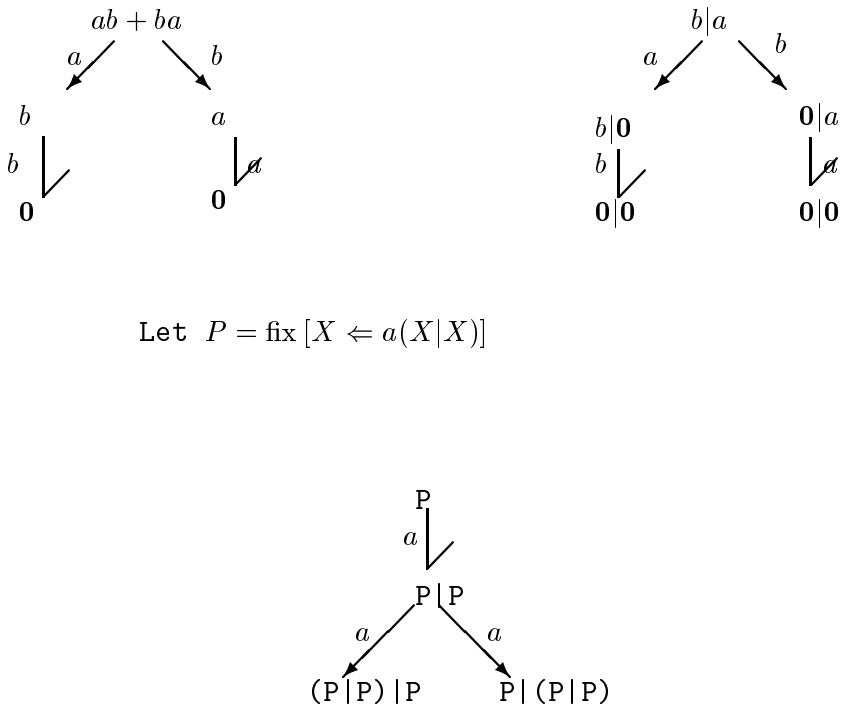


Figure 2.1: Examples of CCS terms and their derivation trees to depth 2

The rules for parallel composition are intended to allow either component to take a step independently, or both to cooperate on a single event (to be interpreted as communication). The restriction operation allows the programmer to force possible communications to occur. We will be doing very little programming in CCS in this thesis, and what we do will not require the full expressive power of CCS. For examples of CCS programming see [Par87b, Par87c, Par87a, ST87, Wal87, Mil80, Mil81, Mil83, Mil84, Mil88].

Some examples of CCS terms and their transitions are given in Figure 2.1. The recursion rule is somewhat elaborate. Little use of recursion is made in this study; the references on CCS may be consulted for details of recursion.

It is worth noting that CCS is a Turing-complete programming language. Given a code n for partial recursive function $f_n : \mathbf{N} \to \mathbf{N}$, there is a CCS program P_n which computes f_n in an appropriate sense: P_n accepts an integer m, suitably coded, as input, and after some computation produces $f_n(m)$, similarly coded, as output; if $f_n(m)$ diverges, P_n runs forever. In general, it is straightforward to build most data structures — integers, stacks, lists, and so on — as CCS processes. This holds even for very restricted finite sub-

languages, using only finite versions of the potentially infinitary constructs and a finite action set.

2.3 Synchronization Trees

As we are studying classes of "CCS-like" languages, we will take a denotational approach to process behavior. CCS process terms are a notation for infinite-state nondeterministic automata. For various technical reasons, it is convenient to unwind these automata completely, giving infinite-state tree-shaped automata, called *synchronization trees*, perhaps inspired by graphs such as those in Figure 2.1.[4] We will often work directly with the trees, rather than terms denoting them; in fact, we will add (constants denoting) some or all synchronization trees to our languages.

Definition 2.3.1 *A* synchronization tree *is a rooted, unordered, finitely branching, possibly countably deep tree with edges labeled by elements of* Act.

As we shall see, synchronization trees give an adequate semantics of CCS; we use the term "process" for synchronization trees. Trees which are like synchronization trees except that they are not finitely branching will be called "infinitely branching synchronization trees." Many concepts and theorems apply to both ordinary and infinitely branching synchronization trees; however, some crucial theorems apply only to finitely branching ones. Unless otherwise stated, all synchronization trees are finitely branching.

Definition 2.3.2 *The tree P' is a* $\left\{\begin{array}{c} \text{child} \\ \text{a-child} \\ \text{descendant} \\ \text{s-descendant} \end{array}\right\}$ *of P if there is* $\left\{\begin{array}{c} \text{an arc} \\ \text{an arc labeled } a \\ \text{a path} \\ \text{a path labeled } s \end{array}\right\}$ *from the root of P to the root of P', where $a \in \mathsf{Act}$ and $s \in \mathsf{Act}^*$. If P' is an a-child (resp. s-descendant) of P, we write $P \xrightarrow{a} P'$ (resp. $P \xrightarrow{s} P'$).*

A set Δ of trees is downward closed *if whenever $P \in \Delta$, all descendants of P are in Δ.*

[4] Trees are used because they simplify certain definitions; in particular, $P + Q$ is a straightforward sum of trees, but has a nontrivial definition as a sum of possibly cyclic automata. Further, our minimal language has notations for all finite trees, but not for all finite automata.

It is clear how to consider a tree as an infinite-state nondeterministic automaton; the root of the tree is the start state of the automaton, and on each clock cycle it selects an edge and performs the action labeling that edge. Conversely, given such an automaton, it can be unwound into a (finitely or countably branching) synchronization tree in an obvious way.

2.4 Tree Isomorphism and Bisimulation

To give a hint of the motivation of bisimulation (and because it is useful in later work) we mention an even finer adequate semantics first.

Definition 2.4.1 *If P and Q are synchronization trees, $P \equiv Q$ if P and Q are isomorphic as unordered edge-labeled trees.*

Figure 2.2: a and $a + a$

Although the synchronization tree semantics $\equiv$ of CCS is adequate and simple to think about, it is not the right semantics. Consider the two processes a and $a + a$ of Figure 2.2. Each can only perform an a-action and then stop; $a+a$ can do that in two ways. It is generally agreed that all ways of performing a-actions and all stopped processes are the same, and so there should be no distinction between a and $a + a$. So, synchronization trees are generally considered to be an overspecification of process behavior, and certain trees must be regarded as equivalent. The question, then, is which trees to identify. In CCS, Milner chose to identify precisely the *bisimilar* trees.

There are several equivalent definitions of bisimulation. The theory of bisimulation has been extensively studied [HM85]; we present only selected highlights. Throughout this thesis, "bisimulation" is Milner's "strong bisimulation."

Definition 2.4.2 *A relation $\underline{\Leftrightarrow}$ between synchronization trees is a* (strong) bisimulation relation *if, whenever $P \underline{\Leftrightarrow} Q$ and $a \in \mathsf{Act}$, then*

- *Whenever* $P \stackrel{a}{\rightarrow} P'$, *then for some* Q', $Q \stackrel{a}{\rightarrow} Q'$ *and* $P' \mathrel{\underline{\Leftrightarrow}} Q'$.
- *Whenever* $Q \stackrel{a}{\rightarrow} Q'$, *then for some* P', $P \stackrel{a}{\rightarrow} P'$ *and* $P' \mathrel{\underline{\Leftrightarrow}} Q'$.

The general idea is that P and Q are bisimilar iff anything that one process can do, the other can do as well and the resulting states are still bisimilar. For example, syntactic equality of program text, synchronization tree isomorphism, and the null relation are all strong bisimulation relations. A somewhat coarser strong bisimulation relation is $\mathrel{\underline{\Leftrightarrow}}_0$, where $t \mathrel{\underline{\Leftrightarrow}}_0 t'$ iff t and t' are isomorphic, or if t and t' are isomorphic to a and $a+a$ respectively. The universal relation $\mathrel{\underline{\Leftrightarrow}}_\infty$, with $P \mathrel{\underline{\Leftrightarrow}}_\infty Q$ for all P and Q, is not a strong bisimulation relation. For example, $aa \mathrel{\underline{\Leftrightarrow}}_\infty \mathbf{0}$ and $aa \stackrel{a}{\rightarrow} a$ but it is not the case that $\mathbf{0} \stackrel{a}{\rightarrow} Q'$ for any Q'.

Definition 2.4.3 *P and Q are* bisimilar, *written $P \mathrel{\underline{\leftrightarrow}} Q$, iff there is a strong bisimulation relation $\mathrel{\underline{\Leftrightarrow}}$ such that $P \mathrel{\underline{\Leftrightarrow}} Q$.*

For example, $a \mathrel{\underline{\leftrightarrow}} a + a$, and indeed $P \mathrel{\underline{\leftrightarrow}} P + P$, where $P + P$ is two copies of P joined at the root. The relation $\mathrel{\underline{\leftrightarrow}}$ is itself a strong bisimulation relation, and in fact the largest one; it is called *(strong) bisimulation*. Bisimulation is an equivalence relation, and it respects all the operations of CCS:

Fact 2.4.4 *If $\vec{P} \mathrel{\underline{\leftrightarrow}} \vec{Q}$ and $C[\vec{X}]$ is a CCS context, then $C[\vec{P}] \mathrel{\underline{\leftrightarrow}} C[\vec{Q}]$.*

Not surprisingly, there are processes which are not bisimilar. For example, $a \mathrel{\underline{\not\leftrightarrow}} b$, for $a \stackrel{a}{\rightarrow} \mathbf{0}$ but there is no B such that $b \stackrel{a}{\rightarrow} B$. Similarly, $a + b \mathrel{\underline{\not\leftrightarrow}} a$, as $a + b \stackrel{b}{\rightarrow} \mathbf{0}$ but $a \stackrel{b}{\not\rightarrow}$.

Lemma 2.4.5 *Let P and Q be processes. Suppose that there is an a-child P' of P such that, for each a-child Q' of Q, $P' \mathrel{\underline{\not\leftrightarrow}} Q'$. Then $P \mathrel{\underline{\not\leftrightarrow}} Q$.*

Proof: Suppose that $P \mathrel{\underline{\leftrightarrow}} Q$, *viz.* $P \mathrel{\underline{\Leftrightarrow}} Q$ for some strong bisimulation relation $\mathrel{\underline{\Leftrightarrow}}$. Then we have $P' \mathrel{\underline{\Leftrightarrow}} Q'$ for some a-child Q' of Q. $\mathrel{\underline{\Leftrightarrow}}$ is a strong bisimulation relation, and so $P' \mathrel{\underline{\Leftrightarrow}} Q'$. However, this implies $P' \mathrel{\underline{\leftrightarrow}} Q'$ contradicting the hypothesis that $P' \mathrel{\underline{\not\leftrightarrow}} Q'$. □

It is now straightforward to build up a collection of similar-looking but non-bisimilar trees. $ab + ac \mathrel{\underline{\not\leftrightarrow}} a(b + c)$, as $ab + ac \stackrel{a}{\rightarrow} b$ and b is not bisimilar to the only a-child $b+c$ of $a(b+c)$. For similar reasons, $a(bc+bd) \mathrel{\underline{\not\leftrightarrow}} ab(c+d)$; it is interesting to note that these processes are CCS/CSP congruent.

The canonical example in this thesis is the pair of processes $P_\star = a(bc + bd)$ and $Q_\star = abc + a(bc+bd) + abd$, shown in Figure 2.3. It will happen that these processes are identified in most of the kinds of semantics we consider, but are not bisimilar. The reasoning is similar to the non-bisimulations shown above; $Q_\star \stackrel{a}{\rightarrow} bc$, and bc is not bisimilar to $P_\star$'s only a-child $bc + bd$.

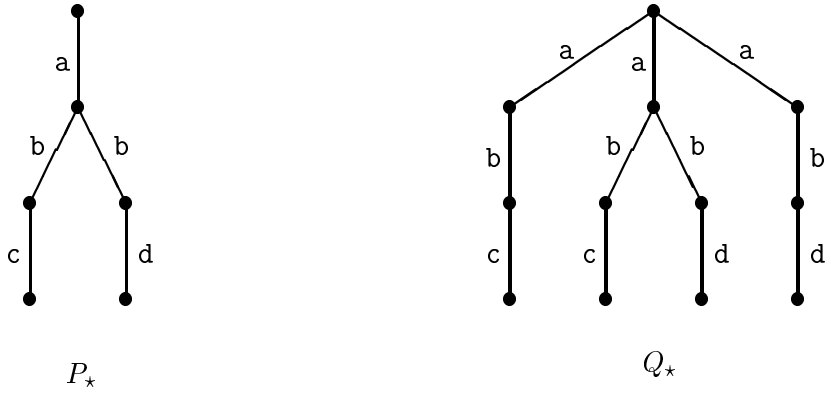


Figure 2.3: $P_\star$ and $Q_\star$: ready similar but not bisimilar

2.4.1 Hennessy-Milner Logic

One of the other characterizations of bisimulation will be particularly important for this study. The following logical characterization holds only for finitely branching trees. It can be extended to trees with larger branching, at the cost of introducing infinitary conjunctions and disjunctions.

Definition 2.4.6 *The* Hennessy-Milner formulas *over* Act *are inductively defined as:*

- tt *and* ff
- $\varphi \wedge \psi$ *and* $\varphi \vee \psi$
- $\langle a \rangle \varphi$ *for each* $a \in \mathsf{Act}$.
- $[a]\varphi$ *for each* $a \in \mathsf{Act}$.

Formulas describe a finite fragment of the behavior of a process. tt is true of every process, and ff of none. $\varphi \wedge \psi$ holds if both φ and ψ hold; $\varphi \vee \psi$ holds if one of the two do. $\langle a \rangle \varphi$ holds of a process if the process can perform an a and afterwards φ will hold; $[a]\varphi$ holds if, no matter how the process performs an a, φ holds.

Definition 2.4.7 *If* φ *is a Hennessy-Milner formula and* P *is a synchronization tree, then the relation of* satisfaction, $P \models \varphi$, *is defined by:*

- $P \models$ tt *always,* $P \models$ ff *never;*
- $P \models (\varphi \wedge \psi)$ *iff* $P \models \varphi$ *and* $P \models \psi$.

- $P \models (\varphi \vee \psi)$ *iff* $P \models \varphi$ *or* $P \models \psi$.
- $P \models \langle a \rangle \varphi$ *iff for some* P', $P \xrightarrow{a} P'$ *and* $P' \models \varphi$.
- $P \models [a]\varphi$ *iff for every* P' *such that* $P \xrightarrow{a} P'$, $P' \models \varphi$.

For example, $P \models \langle a \rangle \mathrm{tt}$ iff P has an a-child. If $\varphi = \langle a \rangle [b] \langle c \rangle \mathrm{tt}$, then φ separates the processes of Figure 2.3: $Q_\star \models \varphi$ but $P_\star \not\models \varphi$. The familiar laws of propositional logic hold for Hennessy-Milner formulas, and so we ambiguously write, *e.g.*, $\varphi_1 \wedge \varphi_2 \wedge \varphi_3$ knowing that the order of parenthesization is irrelevant. Hennessy and Milner, [HM85], show the following soundness and completeness theorem:

Theorem 2.4.8 *If* P_1 *and* P_2 *are finitely-branching synchronization trees, then* $P_1 \underline{\leftrightarrow} P_2$ *iff for each Hennessy-Milner formula* φ, $P_1 \models \varphi$ *iff* $P_2 \models \varphi$.

In particular, bisimulation is fully abstract with respect to observing modal formulas. That is, if we have some way of testing $P \models \varphi$ for all P and φ, then we have a good reason to distinguish between non-bisimilar formulas. However, it is hard to see how to observe modal formulas without observing too much, or to understand them as computational observations. We discuss this further in Chapter 5.

An alternate formulation of Hennessy-Milner Logic is sometimes useful. Consider the formulas generated by the grammar:

$$\varphi ::= \mathrm{tt} \mid \varphi \wedge \varphi \mid \langle a \rangle \varphi \mid \neg \varphi$$

with semantics as before, with the additional clause $P \models \neg\varphi$ iff $P \not\models \varphi$. Hennessy-Milner formulas may be translated into the logic with negation via the equivalence $[a]\varphi \equiv \neg\langle a \rangle \neg\varphi$; formulas with negation may be translated into Hennessy-Milner formulas by this and its dual, the DeMorgan laws, and the laws of double negation.

One consequence of this is that there is no natural unidirectional form of Hennessy-Milner equivalence. That is, suppose that whenever $P \models \varphi$ then $Q \models \varphi$. Then $P \underline{\leftrightarrow} Q$. For, suppose not. Then there is some formula ψ which P and Q disagree on. We must have $Q \models \psi$ and $P \not\models \psi$, or else we would contradict the hypothesis. However, as $P \not\models \psi$, we have $P \models \neg\psi$, and hence $Q \models \neg\psi$ and thus $Q \not\models \psi$, which is a contradiction.

2.4.2 Weak Bisimulation

The equivalence most often used in CCS is *weak bisimulation* or *observational equivalence*.[5] Weak bisimulation is, intuitively, like bisimulation

[5] We prefer the former name, as the latter suggests that it is equivalence with respect to some sort of observation.

except that the special action τ is invisible. The theory of weak bisimulation resembles that of strong bisimulation in many ways, with some notable exceptions.

As we are trying to ignore τ's, we write $P \stackrel{s}{\Rightarrow} Q$ (with $s \in \mathsf{Act}^*$) to mean that P can perform the actions of s (with an arbitrary sprinkling of τ's) and become Q; that is, when $P \stackrel{s'}{\rightarrow} Q$ for some sequence s' such that s is s' with some number of τ's omitted.

Definition 2.4.9 *A relation* $\underline{\Leftrightarrow}$ *is a* weak bisimulation relation *iff, whenever* $P \underline{\Leftrightarrow} Q$*, then*

- *If* $P \stackrel{s}{\Rightarrow} P'$*, then there is some* Q' *such that* $Q \stackrel{s}{\Rightarrow} Q'$ *and* $P' \underline{\Leftrightarrow} Q'$*.*
- *If* $Q \stackrel{s}{\Rightarrow} Q'$*, then there is some* P' *such that* $P \stackrel{s}{\Rightarrow} P'$ *and* $P' \underline{\Leftrightarrow} Q'$*.*

If there is such a $\underline{\Leftrightarrow}$ *with* $P \underline{\Leftrightarrow} Q$*, then* P *and* Q *are* weakly bisimilar*, written* $P \underline{\Leftrightarrow} Q$*.*

Most CCS operations respect weak bisimulation. The exception is $+$, as $a \underline{\Leftrightarrow} \tau a$ but $b + a \not\underline{\Leftrightarrow} b + \tau a$. This can be repaired by considering a slight variant of weak bisimulation, *strong congruence* or *rooted bisimulation*. The resulting theory is described in [Mil88].

Like most researchers in variants of bisimulation, we restrict our attention to the case of strong bisimulation.

1. It seems likely that most results about strong bisimulation will carry over to weak bisimulation mutatis mutandis. Preliminary work on the adaptation of ready simulation to the silent-move case suggests that this intuition holds.

2. The proper definitions are clearer in the case of strong bisimulation. For example, we consider extensions of CCS for which strong bisimulation is a congruence. In the case of weak bisimulation, we clearly cannot consider extensions of CCS which respect weak bisimulation: CCS itself does not. We could consider variants which respect the ad-hoc relation of rooted weak bisimulation, but it is not clear why this and not some further variant is the right one. Similar foundational questions arise in other areas of weak bisimulation; *e.g.*, the treatment of divergence.

3. The theory is much simpler without τ moves.

Part II

CCS-Like Languages and Ready Simulation

Chapter 3

GSOS Languages

3.1 Signatures and Transition Relations

We will be studying the interaction of programming languages and semantics, and in particular we will vary the programming language. We will therefore give general definitions suitable for quantifying over languages. In general, we want to have the most powerful class of languages that can be achieved without losing the essential mathematical and aesthetic properties which characterize CCS. We propose a class of languages, the GSOS languages, and argue that it meets this goal.

A language in the style of CCS is given by a *signature*, *viz.* a set of operation symbols, and an operational semantics over that signature. It is convenient to include a set of synchronization trees in each language.[1] We will include the nodes of the trees themselves as constants in the language.

Definition 3.1.1 *A* signature $\mathcal{F}$ *is a nonempty finite set of* actions $\mathsf{Act}_{\mathcal{F}}$*, a possibly empty downward-closed set* $\Delta_{\mathcal{F}}$ *of synchronization trees over* $\mathsf{Act}_{\mathcal{F}}$*, and a family of disjoint sets* $\mathcal{F}_i$ *for* $i = 0, 1, 2, \ldots$ *such that* $\bigcup_{i \in \mathbf{N}} \mathcal{F}_i$ *is finite. The elements of* $\mathcal{F}_i$ *are the* function symbols of arity i*. We insist that* $\mathbf{0} \in \mathcal{F}_0$*,* $\mathsf{Act} \subseteq \mathcal{F}_1$*, and* $+ \in \mathcal{F}_2$*.*

(Fixed-point operators will not appear in these languages; see Section 3.3.) We fix an infinite set Var of *agent variables*; X, Y, Z range over agent variables. The set of open terms over $\mathcal{F}$, denoted $\mathcal{T}(\mathcal{F})$, is the least set such that

- Each synchronization tree $P \in \Delta_{\mathcal{F}}$ is in $\mathcal{T}(\mathcal{F})$
- Each $X \in \mathsf{Var}$ is in $\mathcal{T}(\mathcal{F})$,

[1] We will frequently want to include *all* synchronization trees. For foundational reasons, we only include only a few representative from each of the 2^{ω} isomorphism classes of synchronization trees, but we ignore this subtlety from now on.

- $f(P_1, \ldots, P_k) \in \mathcal{T}(\mathcal{F})$ whenever $f \in \mathcal{F}_k$ and each $P_i \in \mathcal{T}(\mathcal{F})$.

We write $\mathcal{T}^o(\mathcal{F})$ for the set of closed terms in $\mathcal{T}(\mathcal{F})$, *viz.* those containing no variables.

Aside from the required operations $\mathbf{0}$, $a(\cdot)$, and $+$, CCS has the binary operations $|$ and $\|$, and the unary operations $\backslash L$ for $L \subseteq \mathsf{Act}$ and $[\rho]$ for certain $\rho : \mathsf{Act} \to \mathsf{Act}$. Similar languages such as Mieje, CSP, and ACT use fairly similar sets of operations. Standard operations are written in prefix, infix, and suffix forms following the conventions that have evolved in the field.

The operations are generally given meaning by *structured operational rules*[Plo81]; a language for concurrency in the CCS/CSP style is completely defined by a signature together with a set of structured rules. The operational semantics are given as a labeled transition system on the set of closed terms.

It is difficult to define "structured operational rule" in sufficient generality to cover all the ways used in the literature (*e.g.*, [Blo88a, Mey88, Plo81, Bar81] as well as many places in concurrency theory) and simultaneously avoid pathologies in particular cases. The results in this thesis, among other work, show that the properties of a system defined by structured rules can be quite sensitive to the form of the rules. In general, though, a structured rule has the form

$$\frac{\textit{antecedent}}{\textit{consequent}}$$

where the antecedent and consequent are facts which may have free variables. The intent of the rule is that whenever the antecedent is satisfied by some instantiation of the free variables, then so is the consequent; and conversely that whenever a fact holds, there should be some instantiation of some rule in the language with true antecedent and that fact as consequent. Structured rules, in a variety of guises, are familiar in many areas of mathematics, computer science, and logic.

We will frequently use rule schema in a fairly informal way; *e.g.*, the following scheme describes two rules for each $a \in \mathsf{Act}$.

$$\frac{\textit{antecedent}[a] \quad (a \in \mathsf{Act})}{\textit{consequent-1}, \quad \textit{consequent-2}}$$

To illustrate our structured operational rules, we present the rules for the required operations. It is convenient to have these operations have precisely the same rules in all languages, as we therefore have invariant notations for all finite trees and other concepts. We also describe the effects these rules have on synchronization trees.

- If P is a synchronization tree and P' is an a-child of P then $P \xrightarrow{a} P'$. That is, each synchronization tree P is a process with a synchronization tree isomorphic to P.

- **0** has no rules; it denotes the null tree.
- For each $a \in \mathsf{Act}$, we have the following rule. We could make a distinction between axioms and inference rules; for our purposes, it is simplest to consider axioms such as this as inference rules with an empty set of hypotheses. The synchronization tree of aP is that of P with a new root and an edge labeled a from the new root to the root of P.

$$aX \xrightarrow{a} X \tag{3.1}$$

- For each $a \in \mathsf{Act}$, we have the following two rules. The synchronization tree of $P + Q$ consists of the trees of P and Q with roots identified.

$$\frac{X \xrightarrow{a} Y}{X + X' \xrightarrow{a} Y, \qquad X' + X \xrightarrow{a} Y} \tag{3.2}$$

See Figure 3.1 for pictures of aP and $P + Q$. We write a for the process $a\mathbf{0}$ when no confusion can arise, use infix notation, and omit parentheses following usual mathematical conventions. For example, we mercifully write $a(bc + bd)$ for $a(+(b(c(\mathbf{0})), b(d(\mathbf{0}))))$. It is easy to show that $+$ is commutative and associative in the synchronization-tree semantics — that is, for all synchronization trees P and Q, $P+Q$ and $Q+P$ are isomorphic as synchronization trees — and so we have terms denoting all finite synchronization trees.

Let P be a closed term of $\mathcal{F}$, and $\dot{\to}$ a transition relation over $\mathcal{F}$. The behavior of P under $\dot{\to}$ is a possibly infinite graph edge-labeled by actions, and node-labeled by terms; if $P \xrightarrow{a} P'$ occurs n times in the multiset $\dot{\to}$, then the tree for P has n isomorphic a-children each given by P'. This graph may be unwound to give a possibly infinite tree $[\![P]\!]_{\dot{\to}}$; if it is finitely branching, it is a synchronization tree giving the meaning of P in an obvious sense. Thus, given a transition relation, definitions phrased in terms of synchronization trees may be applied to processes as well; we write *e.g.* $P \underline{\leftrightarrow} Q$ for $[\![P]\!]_{\dot{\to}} \underline{\leftrightarrow} [\![Q]\!]_{\dot{\to}}$.

More generally, given an open term P of variables $X_1, \ldots, X_n$, we may interpret P as an n-ary function on synchronization trees. For example, aX denotes the function of appending a new root and an a-branch to the original root of a tree. Examples of processes and their associated synchronization trees can be found Figure 2.2 and Figure 2.3.

3.1.1 Observations

There is a general notion, or rather a family of notions, of program equality. Programs produce some sort of *observable effects* – printing characters on the screen, flashing lights, crashing computers, and so forth. They also produce some effects which the ordinary user should not be able to observe, such as

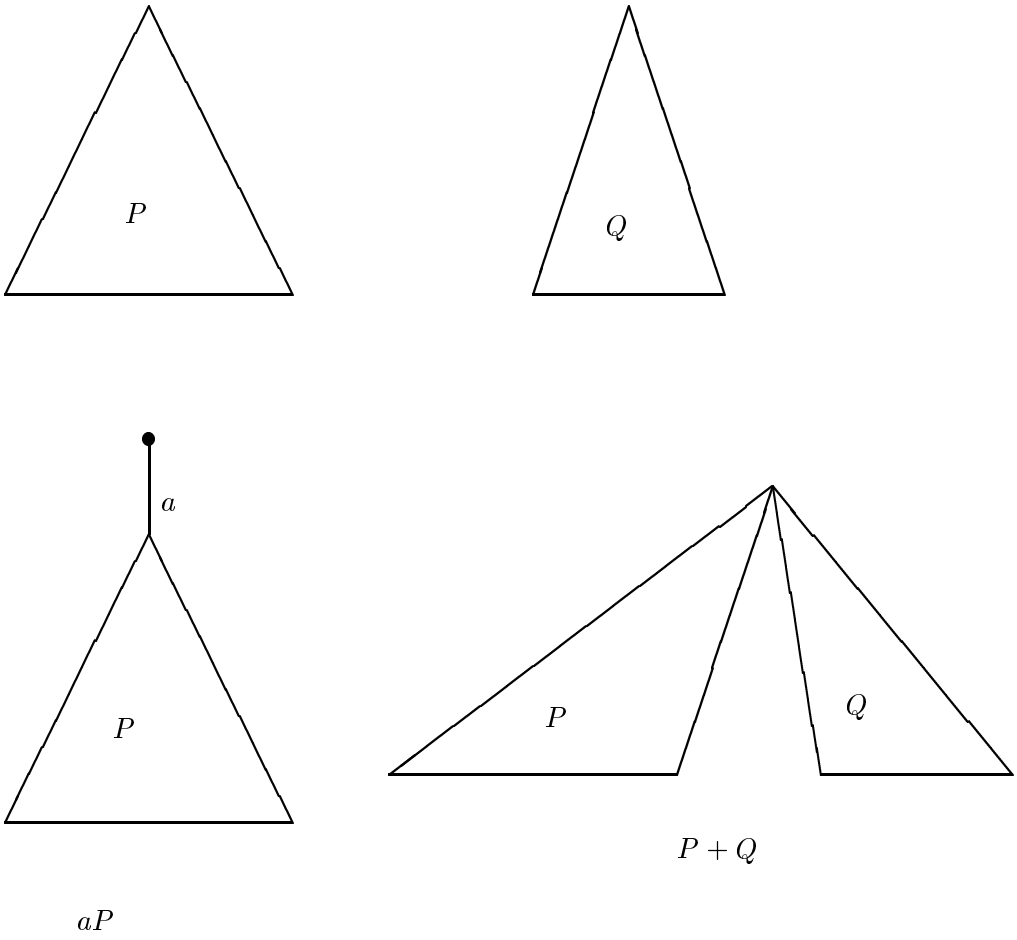


Figure 3.1: Effects of Basic Operations on Trees

setting location `00024D` to `4E`. Two programs should be considered the same if they produce precisely the same observable effects however they are used; that is, if either one can be substituted for the other and no difference can be seen. This general notion, unlike bisimulation, is parameterized by the set of observations possible, and the set of ways that a program can be used.

Hoare's CSP [BHR84, FHLd79, Hoa78] makes distinctions between processes by observing *traces* in contexts over the language CSP. If we imagine these processes as programs printing the actions they take on the screen, then traces are what we can observe: the sequence of actions that the process prints on the screen before it halts. Our programs take one action on each clock tick, so it is possible to tell when one has finished executing. The "ways of using a program" are precisely the CSP contexts. Formally, in a labeled transition system,

Definition 3.1.2 *If $s \in \mathsf{Act}^*$, then $P \xrightarrow{s} P'$ iff there are terms $P_1, \ldots, P_{|s|}$ such that*

$$P \xrightarrow{s_1} P_1 \xrightarrow{s_2} \cdots \xrightarrow{s_{|s|}} P_{|s|} = P'$$

We write $P \xrightarrow{s}$ if for some P' we have $P \xrightarrow{s} P'$; otherwise, $P \xrightarrow{s} \not\rightarrow$. P is stopped *if for every $a \in \mathsf{Act}$, $P \xrightarrow{a} \not\rightarrow$.*

More precisely, CSP observes *finite terminated traces*; It is also possible to use *partial traces*, strings s such that $P \xrightarrow{s}$, or *infinite traces*, infinite strings s such that there exist P_i's such that $P_0 = P$ and $P_i \xrightarrow{s_i} P_{i+1}$ for each i. Partial traces are too weak for our purposes, and infinite traces are too long to observe in any practical sense in finite time. In this thesis, "trace" will always mean "finite terminated trace" unless otherwise specified.

We have used the notation $P \xrightarrow{a} Q$ in two ways for synchronization trees P and Q, in the senses of Definition 2.3.2 and Definition 3.1.2. The ambiguity will cause us no trouble.

Fact 3.1.3 *Our overloading of the $P \xrightarrow{a} Q$ notation is consistent. For example, if P is a synchronization tree, $[\![P]\!]_{\rightarrow}$ is isomorphic to P.*

Definition 3.1.4 *The* trace set *of P,* $\mathrm{tr}(P)$*, is* $\left\{ s \in \mathsf{Act}^* \mid P \xrightarrow{s} P' \text{ and } P' \text{ is stopped} \right\}$

We formalize "using a program" by the notion of a context.

Definition 3.1.5 *A* context of n holes $C[X_1, \ldots, X_n]$ *in a language $\mathcal{L}$ is simply an $\mathcal{L}$-term with free variables at most $\{X_1, \ldots, X_n\}$. The result of substituting P_i for X_i in $C[X_1, \ldots, X_n]$ is written $C[P_1, \ldots, P_n]$.*

There are no variable-binding operations available in our language, so the familiar subtleties of renaming are unnecessary.

Definition 3.1.6 *Let P and Q be synchronization trees.*

1. *P is a* trace approximation *of Q, $P \sqsubseteq_{tr} Q$, iff* $\mathrm{tr}(P) \subseteq \mathrm{tr}(Q)$.

2. *P and Q are* trace equivalent, *$P \equiv_{tr} Q$, iff* $\mathrm{tr}(P) = \mathrm{tr}(Q)$.

3. *P is a* trace approximation with respect to the language $\mathcal{L}$, *$P \sqsubseteq_{tr}^{\mathcal{L}} Q$, iff for all $\mathcal{L}$-contexts $C[X]$ of one free variable, $C[P] \sqsubseteq_{tr} C[Q]$.*

4. *P and Q are* trace congruent with respect to the language $\mathcal{L}$, *$P \equiv_{tr}^{\mathcal{L}} Q$, iff for all $\mathcal{L}$-contexts $C[X]$ of one free variable, $C[P] \equiv_{tr} C[Q]$.*

Notice that $P \equiv_{tr} Q$ iff $P \sqsubseteq_{tr} Q \sqsubseteq_{tr} P$ and $P \equiv_{tr}{}^{\mathcal{L}} Q$ iff $P \sqsubseteq_{tr}^{\mathcal{L}} Q \sqsubseteq_{tr}^{\mathcal{L}} P$.

In CSP, then, two programs are distinguished iff there is a good reason to distinguish them – a context $C[X]$ and a string s of actions such that only one of $C[P]$ and $C[Q]$ can perform s and then stop. In fact, there is a fully abstract mathematical semantics of CSP, the *refusals* semantics of [BHR84]; the meaning of a process P is the set of pairs $\langle s, \mathcal{X} \rangle$, such that $P \stackrel{s}{\to} P'$ and $\mathcal{X}$ is any set of actions a such that $P' \stackrel{a}{\not\to}$.

It is well-known that bisimulation is strictly finer than CCS/CSP trace congruence. Logically, the refusals correspond to modal formulas of the form

$$\langle a_1 \rangle \langle a_2 \rangle \cdots \langle a_k \rangle ((\![b_1]\!]\mathrm{ff}) \wedge (\![b_2]\!]\mathrm{ff}) \wedge \cdots \wedge (\![b_l]\!]\mathrm{ff}));$$

that is, two processes are CCS/CSP congruent iff they agree on all such formulas. From this, it is routine to check that $P_\star$ and $Q_\star$ (see Figure 2.3) are CCS/CSP congruent and not bisimilar.

It is not clear why $P_\star$ and $Q_\star$ should be considered different in CCS. In the spirit of [Plo77, Abr87], we investigate the question of what kinds of operations one must add to CCS to make bisimulation be trace congruence.

3.2 Generalizing CCS — A False Start

As we have seen, bisimulation is not trace congruence with respect to CCS. We consider various ways of generalizing CCS, attempting to refine the language's congruence to make it coincide with bisimulation.

There is a straightforward and trivial operation to add to CCS which makes bisimulation precisely equal trace congruence. This operation, called "not-bisim," takes two arguments; it produces a signal if they are not bisimilar, and is stopped if they are bisimilar. This operation is merely intended as an answer to the question "how could one add an operation to CCS making bisimulation into trace congruence?"; even the strongest proponents of bisimulation would not recommend not-bisim as a new primitive of CCS.

$$\frac{X \not\leftrightarrows X'}{\text{not-bisim}(X, X') \stackrel{a}{\to} \mathbf{0}}$$

In this language, if P and Q are not bisimilar, then the context $C[X] =$ not-bisim(X, P) distinguishes them. There are several grounds for criticizing this operation.

- It begs the question. We have explained why we think bisimulation is important by saying that, if we consider it important, then it is important. It would be desirable to explain it in terms of something that looks more fundamental.

- We could explain any other relation between synchronization trees in the same way. For example, a slight variation on that rule would make synchronization tree isomorphism the fundamental relation in the language, distinguishing a from $a + a$.

- The rule is very difficult to apply. In Milner's original SCCS, the question of $P \underline{\leftrightarrow} Q$ is not arithmetic, and probably Σ_1^1 complete. Even for finitely-branching trees, the question $P \underline{\nleftrightarrow} Q$ is r.e.-complete, and so the one-step transition relation $P \stackrel{a}{\rightarrow} Q$ is not decidable. It is semidecidable[2], which is why we phrased the rule in terms of $\underline{\nleftrightarrow}$ rather than $\underline{\leftrightarrow}$.

- The operation not-bisim seems to be useless. At the time of writing, we have found no researchers who exhibit any desire to have not-bisim included in a programming language. The operation is generally and accurately regarded as a theoretician's trick to get full abstraction.

We would like to choose some criterion by which we may judge languages and see if they are reasonable. We will argue that bisimulation is not trace congruence with respect to any "reasonable" language; to make this formal, we will have to have some way of quantifying over all reasonable languages, and hence need some formal characterization of those languages.

One of the elegant features of CCS is the definition by a set of structured operational rules. By contrast, not-bisim is defined by an ad-hoc rule involving the predicate of bisimulation.) From the form of the structured rules, CCS is guaranteed to exhibit several mathematical properties (*e.g.*, that all programs using only guarded recursion are finitely branching, and that the transition relation is decidable.). We choose, therefore, to investigate languages defined by rules which look like the rules of CCS. We will check the soundness of our definition of "looking like the rules of CCS" by making sure that our languages all have the essential properties of CCS. We will try to catch all reasonable languages by taking the largest cleanly-defined class of languages with "well-structured" CCS-like definitions which have these essential properties. The remainder of this chapter is concerned with the discussion of our proposed class of "reasonable" CCS-like process languages.

[2]Strictly, the question $P \underline{\nleftrightarrow} Q$ is r.e. given oracles of the tree constants appearing in P and Q. If there are no non-recursive trees, then the question is r.e.

3.3 The Purpose of Fixed Points

3.3.1 Expressive and Discriminatory Power

There are several kinds of power that a language may have, or more properly several ways of comparing the power of two languages. We are concerned with the *expressive* and *discriminatory* power of languages. Informally, the language L_1 is more expressive than L_2 if everything that can be expressed in L_2, and more, can be expressed in L_1. Similarly, L_1 is more discriminatory than L_2 if some pair of synchronization trees P and Q can be distinguished (*viz.*, used in such a way as to produce observably different results — *e.g.*, different possible traces) in L_1 but not L_2.[3]

In fact, there are two distinct kinds of expressive power. *Process expressive power* gauges the ability of the language to define *processes*. *Operator expressive power* gauges the ability of the language to define operations on synchronization trees as contexts. The two sorts of power are only loosely related. For example, in a minimal calculus with the operations + and prefixing, the definable processes are precisely the finite trees. If we add $\|$, simple interleaving parallel, to the calculus, we cannot define any more processes; if P and Q are finite trees, then $P \| Q$ is also a finite tree. The two calculi have the same process expressive power. We have, however, increased operator expressive power; *e.g.*, the operation "Perform an a-action at some time during, before, or after the execution of X" can be defined as $X \| a$ in the larger calculus, but cannot be defined at all using simply + and prefixing. Most programming language designers are, quite rightly, concerned with issues of expressiveness.

In the bulk of this work, we are not concerned with either form of expressive power per se. We will ignore most issues of process expression, by allowing an arbitrary set of synchronization trees as primitives in our languages (except in Section 7.2.) We will not be as generous with operator expression; we will require that all operations respect the basic mathematical properties of CCS, and be definable in a "well-structured" way.

We are trying to find a language which understands bisimulation, or approaches it as closely as possible. We are therefore interested in the *discriminatory power* of the language: its ability to tell the difference between distinct synchronization trees. Discriminatory power is loosely related to operator expressive power. It is clear that if we add the ability to distinguish a new pair of processes, we have added some previously undefinable operations and thereby increased operator expressiveness. However, it is frequently the case that adding new operations to a language will not increase its discriminatory power. In Section 4.4 we present a language with the

[3]Similar notions apply to other settings which have some well-established space of meanings; *e.g.*, most applied λ-calculi are more expressive than the pure simply-typed λ-calculus.

property that any addition of well-structured operations will not increase its discriminatory power; the addition of, *e.g.*, a sequencing operation will increase its expressive power.

3.3.2 Recursion Considered Harmful (or Inessential)

The fixed-point rule, used without caution, can be dangerous. It is possible to define processes (*e.g.*, $\mathrm{fix}\,[X \Leftarrow a + (X \parallel X)]$) which are *countably branching*; that is, there are a countable set of distinct terms Q_n such that $\mathrm{fix}\,[X \Leftarrow a + (X \parallel X)] \stackrel{a}{\rightarrow} Q_n$.

Infinitely branching trees and so-called "unguarded processes" [Mil83] cause many problems in many aspects of the theory. It seems likely that bisimulation between countably branching trees cannot match trace congruence, for purely recursion-theoretic reasons; we explore this issue in a later paper. Unguarded recursion in Milner's original SCCS makes the transition relation $P \stackrel{a}{\rightarrow} Q$ undecidable; it is an open problem whether or not it is decidable in SCCS with a finite action set. The correspondence between bisimulation and Hennessy-Milner logic becomes harder; Theorem 2.4.8 fails unless infinitary conjunctions and disjunctions are used. Unguarded recursion is also incompatible with negative rules, although this could be construed as a criticism of negative rules. If α is defined by the rule

$$\frac{X \stackrel{a}{\not\rightarrow}}{\alpha(X) \stackrel{a}{\rightarrow} \mathbf{0}}$$

then let $T = \mathrm{fix}\,[X \Leftarrow \alpha(X)]$. Then $T \stackrel{a}{\rightarrow} T'$ iff $\alpha(T) \stackrel{a}{\rightarrow} T'$ iff $T \stackrel{a}{\not\rightarrow}$. Also, the desired operations on processes are not continuous with respect to any useful known topology.

For these reasons and others, restrictions are generally imposed on recursive definitions of processes. In CCS, "guarded" recursion is singled out as attractive, and in CSP and the test-equivalence system of [dNH84], unguarded recursions are treated as though they diverged. The essence of these restrictions is to ensure that definable processes behave like *computable, finitely branching trees*: that there is a program which, given a and P, computes the finite set $\left\{Q \mid P \stackrel{a}{\rightarrow} Q\right\}$.

In the case of guarded recursion, suppose that P and Q are not trace congruent — that is, there is a context $C[X]$ and a string s of actions such that, say, $s \in \mathrm{tr}(C[P]) - \mathrm{tr}(C[Q])$. This context $C[X]$ may involve recursion. However, the guarded fixed point operators appearing in $C[X]$ may be unwound a suitably large but finite number of times and then replaced by $\mathbf{0}$ giving a new context, $C'[X]$, which contains no fixed point operators and also has the property that $s \in \mathrm{tr}(C'[P]) - \mathrm{tr}(C'[Q])$. That is, P and Q can be distinguished by a context not involving recursion at all.

This informal argument can be formalized directly in our class of languages. Furthermore, the basic theorems hold *mutatis mutandis*; the addition of guarded recursion makes no significant changes in the results of this work. The proofs of the theorems are frequently much more complicated: the proofs in this thesis by structural induction must be done by a modified structural induction with special cases for recursion and prefixing. (Unlike many other areas, the recursion isn't much of a special case; guarded recursions, like the other operations, can only perform quite limited computation in a single step. The modifications are simple and uninteresting.) Finally, we will allow operations which generate infinite synchronization trees; in fact, any finite set of guarded recursions may be included in the language as separate operations. For simplicity and without loss of generality, we will not include fixed points in our languages.

3.4 GSOS Rules

We present the general format of *GSOS structured transition rule*, and then argue that this format is a maximal right one. The argument will take the form of some theorems showing that any language defined by these rules has some desirable properties, and an array of counterexamples showing that the obvious classes of extensions do not.

Definition 3.4.1 *A* positive transition formula *is a triple of two terms and an action, written* $T \xrightarrow{a} T'$. *A* negative transition formula *is a pair of a term and an action, written* $T \stackrel{a}{\not\to}$.

Definition 3.4.2 *A* GSOS rule ρ *is a rule of the form:*

$$\frac{\left\{X_i \xrightarrow{a_{ij}} Y_{ij} \mid 1 \le j \le m_i\right\}_{i=1}^{l}, \quad \left\{X_i \stackrel{b_{ik}}{\not\to} \mid 1 \le k \le n_i\right\}_{i=1}^{l}}{\mathrm{op}(X_1, \ldots, X_l) \xrightarrow{c} C[\vec{X}, \vec{Y}]}$$

where all the variables are distinct, $l \ge 0$ *is the arity of* op, $m_i, n_i \ge 0$, *and* $C[\vec{X}, \vec{Y}]$ *is a context with free variables including at most the* X*'s and* Y*'s. (It need not contain all these variables.) The function symbol* op *is the* principal operator *of the rule;* ρ_- *is the set of antecedents, and* ρ_+ *is the consequent.*

A rule is positive if all of its antecedents are positive. Note that every X_i occurring in the antecedent of a GSOS rule must occur as an argument of the principal operator in the consequent, but not every argument of the principal operator need occur in the antecedent. In the future we will not write the ranges of the indices on the rule unless necessary.

Some examples of GSOS rules may be found in Section 2.2.2, as the structured operational semantics of the CCS operations other than recursion

is given as a set of GSOS rules. Some of the other features are shown in the following examples (which are operators contrived to illustrate features rather than provide sensible programming constructs.)

$$\frac{X_1 \stackrel{a}{\to} Y}{\xi(X_1, X_2) \stackrel{b}{\to} aX_2 \parallel \xi(Y \parallel X_1, Y \parallel X_2) + Y}$$

Though the CCS rules use only simple contexts, we allow more elaborate contexts as results.

$$\frac{X_1 \stackrel{a}{\to} Y_1, X_1 \stackrel{b}{\to} Y_2}{\zeta(X_1, X_2) \stackrel{a}{\to} Y_1}$$

(Note that $\zeta(X_1, X_2)$ examines two behaviors of X_1 and none of X_2, and that the resulting context $C[\vec{X}, \vec{Y}] = Y_1$ only involves one of the four process variables.)

$$\frac{X_1 \stackrel{a}{\to} Y, X_1 \stackrel{a}{\not\to}}{\zeta(X_1, X_2) \stackrel{b}{\to} X_2}$$

(This rule cannot be applied, as no process is both able and unable to perform an a-action. It is legitimate, merely useless. A rule is useless in this sense iff it has antecedents $X_i \stackrel{a}{\to} Y_{ij}$ and $X_i \stackrel{a}{\not\to}$. It is not worth the occasional bother to exclude such rules.)

Definition 3.4.3 *A* GSOS rule system $\mathcal{G}$ *over a signature* $\mathcal{F}$ *is a finite set of GSOS rules over the actions and operations in* $\mathcal{F}$*, such that precisely the required rules (3.1) and (3.2) are given for the required operations* $a(\cdot)$ *and* $+$*.*

Now, we must show that each GSOS rule system determines an operational semantics. The operational semantics will be given by a labeled transition system with the closed $\mathcal{F}$-terms as the processes and the actions in Act as the action. The presence of negative rules requires us to do some work to define the transition relation.

Definition 3.4.4 *A* (closed) substitution *is a partial map* σ *from variables to closed terms. We write* $P\sigma$ *for the result of substituting* $\sigma(X)$ *for each* X *occurring in* P*; if* $\sigma(X)$ *is undefined,* $P\sigma$ *is undefined.*

For example, let $\sigma(X) = Q$; then $(aX + bY)\sigma = aQ + bY$. Note that if the free variables in P are $X_1, \ldots, X_n$, then $P\sigma = P[\vec{X} := \vec{Q}]$. All substitutions in this study will be closed.

Definition 3.4.5 *If $\dot{\rightsquigarrow}$ is a transition relation, σ is a substitution, and t is a transition formula, then the predicate $\dot{\rightsquigarrow}, \sigma \models t$ is defined by*

$$\begin{array}{lll} \dot{\rightsquigarrow}, \sigma \models T \xrightarrow{a} T' & \textit{iff} & T\sigma \overset{a}{\rightsquigarrow} T'\sigma \\ \dot{\rightsquigarrow}, \sigma \models T \overset{a}{\not\to} & \textit{iff} & \nexists Q. T\sigma \overset{a}{\rightsquigarrow} Q. \end{array}$$

If $\mathcal{S}$ is a set of transition formulas, $\dot{\rightsquigarrow}, \sigma \models \mathcal{S}$ iff $\forall t \in \mathcal{S}. \dot{\rightsquigarrow}, \sigma \models t$.

For example, let $\dot{\rightsquigarrow}_{CCS}$ be the transition relation of CCS (which we write as $\dot{\to}$ in all sections in which the notation is not ambiguous). Suppose that $\sigma_1(X) = ab + ac$. Then we have $\dot{\rightsquigarrow}_{CCS}, \sigma_1 \models \left\{ X \xrightarrow{a} b, X \xrightarrow{a} c, X \overset{b}{\not\to} \right\}$.

Definition 3.4.6 *If ρ is a GSOS rule, $\dot{\rightsquigarrow}, \sigma \models \rho$ holds iff*

$$\dot{\rightsquigarrow}, \sigma \models \rho_{\text{-}} \textit{ implies } \dot{\rightsquigarrow}, \sigma \models \rho_{\text{-}}$$

For example, $\dot{\rightsquigarrow}_{CCS}, \sigma \models \rho$ for every substitution σ and every CCS rule ρ. However, we also have $\dot{\rightsquigarrow}_{\infty}, \sigma \models \rho$ for every CCS rule as well, where $\dot{\rightsquigarrow}_{\infty}$ is the universal relation: $P \overset{a}{\rightsquigarrow}_{\infty} Q$ for all P, Q, and a.

Definition 3.4.7 *$\dot{\rightsquigarrow}$ is* sound *for $\mathcal{G}$ iff for every rule $\rho \in \mathcal{G}$ and every substitution σ, we have $\dot{\rightsquigarrow}, \sigma \models \rho$.*

In general, many transition relations will be sound for $\mathcal{G}$; for example, $\dot{\rightsquigarrow}_{\infty}$ is sound for every $\mathcal{G}$. One generally takes the *smallest* sound transition relation, showing that there is in fact a smallest one. This is not appropriate for GSOS rules; with negative rules, there may not be a smallest sound transition relation. We delay the example until Section 3.4.1.

However, GSOS languages do define a (unique) operational semantics in an appropriate sense. The point of minimality in the usual case is to ensure that everything which is true is true for some reason, because there is some rule with that fact as consequent and a true antecedent. We make this concern explicit.

Definition 3.4.8 *$\dot{\rightsquigarrow}$ is* witnessing *for $\mathcal{G}$ iff, whenever $P \overset{a}{\rightsquigarrow} P'$ there is a rule $\rho \in \mathcal{G}$ and a substitution σ such that $\dot{\rightsquigarrow}, \sigma \models \rho_{\text{-}}$ and $\rho_{\cdot}\sigma = P \xrightarrow{a} P'$.*

A transition relation is witnessing if, whenever a transition happens, it happens because it was the consequent of (an instantiation of) some rule, and the antecedents of that rule were satisfied. For example, $\dot{\rightsquigarrow}_{CCS}$ is witnessing for CCS. However, $\dot{\rightsquigarrow}_{\infty}$ is not witnessing for CCS. There are no axioms for $\mathbf{0}$, yet $\mathbf{0} \dot{\rightsquigarrow}_{\infty} a$. Soundness and witnessing together select the right transition relation:

Lemma 3.4.9 *There is a unique sound and witnessing transition relation $\dot{\to}_{\mathcal{G}}$ for $\mathcal{G}$.*

Proof: This, like most proofs about GSOS rule systems, is by structural induction on terms. To show existence, define relations $\dot{\rightsquigarrow}$ and $\dot{\not\rightsquigarrow}$ inductively as follows. For each tree constant P and each a-child P' of P, define $P \stackrel{a}{\dot{\rightsquigarrow}} P'$; if there are no such P''s, define $P \stackrel{a}{\dot{\not\rightsquigarrow}}$. For each nullary operator α, $\alpha \stackrel{a}{\dot{\rightsquigarrow}} P$ whenever $\alpha \stackrel{a}{\rightarrow} P$ is an axiom, and $\alpha \stackrel{a}{\dot{\not\rightsquigarrow}}$ if there is no axiom of the form $\alpha \stackrel{a}{\rightarrow} Q$. Define $\mathrm{op}(P_1, \ldots, P_n) \stackrel{a}{\dot{\rightsquigarrow}} Q$ if there is some substitution instance of a rule with its positive antecedents satisfied by $\dot{\rightsquigarrow}$ and its negative antecedents satisfied by $\dot{\not\rightsquigarrow}$, and $\mathrm{op}(P_1, \ldots, P_n) \stackrel{a}{\dot{\rightsquigarrow}} Q$ as conclusion. If there is no such Q, then let $\mathrm{op}(P_1, \ldots, P_n) \stackrel{a}{\dot{\not\rightsquigarrow}}$. It is straightforward to show that $P \stackrel{a}{\dot{\not\rightsquigarrow}}$ iff $\nexists Q.\, P \stackrel{a}{\dot{\rightsquigarrow}} Q$, and using this fact that $\dot{\rightsquigarrow}$ is sound and witnessing for $\mathcal{G}$.

The proof of uniqueness is similar; inductively show that any transition relation sound and witnessing agrees with $\dot{\rightsquigarrow}$. □

We call the unique transition relation the *standard* transition relation, and write it $\dot{\rightarrow}_{\mathcal{G}}$ or simply $\dot{\rightarrow}$. The ambiguity in notation between transition formulas and transitions should cause no difficulty. Notice that P' is an a-child of P precisely if $P \stackrel{a}{\rightarrow} P'$.

An equivalent formulation in terms of proofs is available; due to negative rules, it is necessary to prove both positive and negative transition formulas. A proof of $T \stackrel{a}{\rightarrow} T'$ consists of exhibiting a rule ρ, a substitution σ such that $\rho_{\cdot}\sigma = T \stackrel{a}{\rightarrow} T'$, and a proof of each formula in ρ_{-}. A proof of $T \stackrel{a}{\not\rightarrow}$, where $T = \mathrm{op}(\vec{T})$, is a list of all the rules ρ with consequents of the form $\mathrm{op}(\vec{X}) \stackrel{a}{\rightarrow} T'$ — that is, all the rules which could prove that $T \stackrel{a}{\rightarrow} T'$ — and for each such rule, a refutation of some antecedent. A refutation of a positive antecedent $X_i \stackrel{a_{ij}}{\rightarrow} Y_{ij}$ is a proof that $T_i \stackrel{a_{ij}}{\not\rightarrow}$; a refutation of a negative antecedent $X_i \stackrel{b_{ik}}{\not\rightarrow}$ is a proof that $T_i \stackrel{b_{ik}}{\rightarrow} T''$ for some T''.

3.4.1 Nonexistence of a minimal transition relation.

For a positive GSOS language, it is clear that there always is a minimal sound transition relation, and that it coincides with the standard transition relation. With negative rules, there need be no minimal sound transition relation. Let $\mathcal{G}$ be the GSOS language with the required operations of $+$, $\mathbf{0}$, and prefixing, and $\Box$ and α. $\Box$ has no rules, and thus behaves like $\mathbf{0}$ in the standard transition relation on $\mathcal{G}$; α is defined by

$$\frac{X \stackrel{a}{\not\rightarrow}}{\alpha(X) \stackrel{a}{\rightarrow} \mathbf{0}}$$

Let $\dot{\rightsquigarrow}_1$ be the standard transition relation for $\mathcal{G}$, and $\dot{\rightsquigarrow}_2$ the standard transition relation for $\mathcal{G}' = \mathcal{G} \cup \left\{ \Box \stackrel{a}{\rightarrow} \mathbf{0} \right\}$. We have:

$$\Box \stackrel{a}{\dot{\not\rightsquigarrow}}_1 \qquad \alpha(\Box) \stackrel{a}{\dot{\rightsquigarrow}}_1 \mathbf{0}$$
$$\Box \stackrel{a}{\dot{\rightsquigarrow}}_2 \mathbf{0} \qquad \alpha(\Box) \stackrel{a}{\dot{\not\rightsquigarrow}}_2$$

As $\dot{\rightsquigarrow}_2$ is sound for $\mathcal{G}'$, it is sound for $\mathcal{G}$ as well. However, if $\dot{\rightsquigarrow}_3 \subseteq \dot{\rightsquigarrow}_1 \cap \dot{\rightsquigarrow}_2$, we have both $\Box \overset{a}{\not\dot{\rightarrow}}_3$ and $\alpha(\Box) \overset{a}{\dot{\rightarrow}}_3$, and so $\dot{\rightsquigarrow}_3$ is unsound for $\mathcal{G}$. Therefore there is no minimal sound transition relation for $\mathcal{G}$.

3.5 Why GSOS Rules Are Desirable

In this section, we argue that GSOS rules are appropriate as a generalization of CCS. We give two theorems which, together with Lemma 3.4.9, demonstrate that bisimulation is appropriate in the GSOS setting. Recall that bisimulation is best used with finitely branching trees; we will show that every GSOS language produces only finitely branching trees. We then give an indication of the additional power of GSOS-definable operations by some examples of operations on trees which can be defined in the GSOS setting but not in CCS.

3.5.1 Basic Properties

Theorem 3.5.1 *Let $\mathcal{G}$ be a GSOS rule system. Then the transition relation on $\mathcal{G}$ is computably finitely branching uniformly in the tree constants. That is, there is an algorithm which, given an action a, a term P, and oracles for all the tree constants occurring in P, produces the set of a-children of P; and this set is always finite.*

Proof: The construction in the proof of Lemma 3.4.9 can be adapted to give an algorithm to compute the set of children of P given oracles returning the set of children of the synchronization trees which occur as subterms of P. $\Box$

If one is trying to argue that bisimulation is the right basis for a theory of processes, one should certainly not refine bisimulation. That is, if two programs are bisimilar, there should be no way in the language to tell them apart. This holds in all GSOS rule systems:

Theorem 3.5.2 *Let $\mathcal{G}$ be a GSOS rule system. Then bisimulation is a congruence with respect to the operations in $\mathcal{G}$. That is, if $P \underline{\leftrightarrow} Q$ are synchronization trees and $C[X]$ is a context over $\mathcal{G}$, then $C[P] \underline{\leftrightarrow} C[Q]$.*

Proof: This is a consequence of Theorem 4.5.1, which states that a coarser relation than bisimulation is a congruence. $\Box$

Corollary 3.5.3 *If $P \underline{\leftrightarrow} Q$, then P and Q are trace congruent with respect to $\mathcal{G}$.*

Proof: It is clear that, if $R \underline{\leftrightarrow} S$, then R and S have the same traces. Let $C[\cdot]$ be an arbitrary context; by Theorem 3.5.2, $C[P] \underline{\leftrightarrow} C[Q]$, and hence P and Q have the same traces in $C[\cdot]$. $\Box$

3.5.2 Expressive Power

GSOS rules are quite expressive, as witnessed by the fact that most structured transition rules proposed in the field have been GSOS rules. For example, the rules for CCS are all GSOS rules.

In fact, GSOS rules go beyond the kind of structured rules needed for CCS in two aspects — the use of *negation* and *copying*. Negation allows us to define a pure form of sequential composition: $P;Q$ runs P until it stops, and then runs Q. As an operation on synchronization trees, the $P;Q$ glues a copy of Q at each leaf of P.[4]

$$\frac{X \stackrel{a}{\to} Y}{X;X' \stackrel{a}{\to} Y;X'} \qquad \frac{X' \stackrel{b}{\to} Y',\ \left\{X \stackrel{a}{\not\to} \mid a \in \mathsf{Act}\right\}}{X;X' \stackrel{b}{\to} Y'}$$

Copying allows us, not surprisingly, to make copies of processes. There can be more than one antecedent about the behavior of a single subprocess, and more than one copy of a process in the result in the consequent. For example, the following GSOS rules yield operations which cannot be defined in CCS [dS85].

$$\frac{X \stackrel{a}{\to} Y, X \stackrel{b}{\to} Y'}{\text{a-if-b}(X) \stackrel{a}{\to} \text{a-if-b}(Y)} \qquad \frac{X \stackrel{a}{\to} X'}{\text{double}(X) \stackrel{a}{\to} X' \parallel X'}$$

The operation a-if-b(X) will allow X to exhibit it's a-behavior as long as it also has the possibility of performing b's at each step. The double operator produces two copies of one a-child of X, running in interleaved parallel.

3.6 Obvious Extensions Violate Basic Properties

There are many technical restrictions in our definition of a GSOS rule, and it is natural to ask if they can be relaxed. We indicate how various relaxations may break the key properties of GSOS systems. Note that some systems with non-GSOS rules enjoy the good properties of GSOS systems; however, this is not immediate from the syntactic specifications of these systems. GSOS rules therefore provide a language-design methodology: any language defined purely by GSOS is guaranteed to meet the basic criteria; other

[4] It is possible to define some forms of sequential composition with positive rules. For example, sequencing in CSP runs P until it announces that it has finished by sending a special action, then runs Q. However, processes may finish without announcing that they have finished — called "deadlock" in CSP — or (in general) may announce that they have finished when in fact they are still able to continue; CSP sequencing is not identical with pure sequencing. It is evidently acceptable for programming. Frits Vaandrager has pointed out that in some ways, it is preferable; it allows us to consider *divergence*. However, divergence is not a part of Milner's original notion of strong bisimulation, and we do not consider it.

languages may or may not. Perhaps more importantly, a GSOS language may be extended by GSOS operations and is still guaranteed to behave well; a well-behaved non-GSOS language extended by the same GSOS operations may cease to be well-behaved. The properties which non-GSOS systems most often violate are:

- The guarantee that bisimulation is a congruence. In fact, they typically do not respect synchronization tree isomorphism; *e.g.*, there are two stopped programs which can be distinguished. (Recall that all stopped programs are unable to take any actions, and hence they have the same synchronization tree; in fact, the null tree.)
- The requirement that $\dot{\rightarrow}$ be computably, finitely branching.
- The existence of some transition relation $\dot{\rightarrow}$ agreeing with all the rules.

(i) Many possible extensions of the GSOS format give some kind of pattern-matching ability, which generally prevents bisimulation from being a congruence. For example, the consequent must be of the form $\mathrm{op}(\vec{X}) \stackrel{c}{\rightarrow} C[\vec{X}, \vec{Y}]$. If we allow more structured left-hand sides of the antecedent, we allow a certain kind of pattern matching. Consider a unary operation α defined by the axiom

$$\alpha(\mathbf{0}) \stackrel{a}{\rightarrow} \mathbf{0}$$

Now, $\mathbf{0}$ and $\mathbf{0}+\mathbf{0}$ are bisimilar; indeed, both have the null synchronization tree. However, $\alpha(\mathbf{0}) \stackrel{a}{\rightarrow} \mathbf{0}$ but $\alpha(\mathbf{0}+\mathbf{0})$ is stopped. This gives us a context which distinguishes between two bisimilar terms, showing that bisimulation is not a congruence with respect to this operation. Similar examples apply to other rules which can branch on more than simply the leading operator of the antecedent.

(ii) Similarly, we do not allow the use of variables to do pattern-matching. For example, the following rule has two different variables taking a-steps and then becoming the same variable:

$$\frac{X_1 \stackrel{a}{\rightarrow} Y, \quad X_2 \stackrel{a}{\rightarrow} Y}{\beta(X_1, X_2) \stackrel{a}{\rightarrow} Y}$$

The context $C[Z] = \beta(a\mathbf{0}, aZ)$ distinguishes between the bisimilar processes $\mathbf{0}$ and $\mathbf{0}+\mathbf{0}$. Copying — using a source variable X_i more than once in the antecedent — does not do any harmful pattern-matching and is acceptable. Duplicated source variables in the consequent, however, allow too much pattern-matching to respect bisimulation; *e.g.*, the rule $\delta(X, X) \stackrel{a}{\rightarrow} \mathbf{0}$ causes as much trouble here as its cousin does in the λ-calculus.

(iii) We have insisted that the positive hypotheses be $X \stackrel{a}{\rightarrow} Y$, but the negative ones $X \stackrel{b}{\not\rightarrow}$. There is no point to hypotheses $X \stackrel{a}{\rightarrow}$; simply

use one of the form $X \stackrel{a}{\rightarrow} Y$ and ignore Y. On the other hand, there is a gain of expressive power to negative hypotheses $X \stackrel{b}{\not\rightarrow} Y$. The gain is all in the wrong direction. It is now easy to write some countably-branching processes, but it is hard to see how the power of these rules could be useful.

$$\frac{X \stackrel{b}{\not\rightarrow} Y}{\xi(X) \stackrel{a}{\rightarrow} Y}$$

$\xi(\mathbf{0}) \stackrel{a}{\rightarrow} P$ for every process P.

(iv) We gain expressive power if we allow terms rather than simply variables to appear in antecedents. If the terms appear as targets — $X \stackrel{a}{\rightarrow} (Y + \mathbf{0})$ — we have too much pattern-matching, and examples similar to the previous ones will show that bisimulation need not be a congruence. Terms appearing as the sources cause a more subtle problem: bisimulation will still be a congruence, but a process may have an infinite number of descendants.[5] As an example, consider the nullary operation ω defined by two rules

$$\omega \stackrel{a}{\rightarrow} \mathbf{0} \qquad \frac{\omega \stackrel{a}{\rightarrow} Y}{\omega \stackrel{a}{\rightarrow} aY}$$

Then, ω is not finitely branching, for we have:

$$\begin{array}{lcl} \omega & \stackrel{a}{\rightarrow} & \mathbf{0} \\ \omega & \stackrel{a}{\rightarrow} & a\mathbf{0} \\ \omega & \stackrel{a}{\rightarrow} & aa\mathbf{0} \\ & \vdots & \end{array}$$

This occurs because the operation ω is defined by what amounts to an unguarded recursion: it mentions its own one-step behavior in the definition of its one-step behavior. In some circumstances, this might be a useful sort of operation; *e.g.*, we could define $\omega'(X) \equiv \sum_{i=0}^{\infty} a^i X$, a process which dawdles for a finite but unbounded time before performing X.

We could also repair this flaw by allowing terms in antecedents, but not allow any operation to refer to itself either directly or indirectly. The resulting class of languages guarantee the essential properties of GSOS languages. In fact, they have almost exactly the same properties: every operation definable with terms is definable without them as well. On the other hand, if terms may appear in antecedents, structural induction is no longer a viable proof method.

(v) Another possible extension is allowing multi-step antecedents [GV89]. It is in general inconsistent to have both negative and multi-step

[5] In fact the resulting congruence is coarser than bisimulation; see [GV89] for details.

antecedents. For example, consider the operators $\cdot/a$, α, and π defined as follows:

$$\frac{X \xrightarrow{a} Y_1 \xrightarrow{b} Y_2}{X/a \xrightarrow{b} Y_2}$$

$$\frac{X \stackrel{a}{\not\to}}{\alpha(X) \xrightarrow{a} \mathbf{0}}$$

$$\pi \xrightarrow{a} \alpha(\pi/a) \tag{3.3}$$

It is not hard to show that there is no arrow relation which agrees with these rules. In particular, π/a can move iff it cannot move. This is similar to the fact that unguarded recursion and negation do not mix. The program $\mathrm{fix}\,[x \Leftarrow \alpha(x)]$ is a term involving an unguarded recursion which can take an a-step iff it cannot take an a-step.

It is possible to have some syntactic conditions which guarantee finite branching and existence of a transition relation, but they are necessarily global. The operation π amounts to an unguarded recursion; however, there is nothing about the form of rule (3.3) indicating that it is unguarded. This rule would be perfectly acceptable in a GSOS language, for example; it is turned into an unguarded recursion by the presence of the $\cdot/a$ operator. It is nontrivial to define just what a guarded operator is in the presence of multi-step rules; the operator X/a "unguards" X, and an adequate calculus of guardedness remains to be developed. CSP, which has something like the X/a operator, forbids its use in recursions.

(vi) A system with multi-step positive antecedents alone is quite possible; such systems respect bisimulation, and always have a minimal transition relation[GV89].[6] However, they are not necessarily finitely branching. Consider the operations $/a$ defined above, and κ defined by:

$$\kappa(X) \xrightarrow{a} X + \kappa(X)/a/a$$

Let M_0 be a synchronization tree with an infinite derivation

$$M_0 \xrightarrow{a} M_1 \xrightarrow{a} M_2 \xrightarrow{a} \cdots$$

Let $N = M_0 + \kappa(M_0)/a/a$; so

$$\kappa(M_0) \xrightarrow{a} N.$$

We will show that N has an infinite number of a-children. Clearly $N = M_0 + \kappa(M)/a/a \xrightarrow{a} M_1$.

[6] The transition relation need not be unique – that is, many relations may be sound and witnessing – but we know of no circumstance in which the non-uniqueness causes any difficulties.

Suppose that

$$N \stackrel{a}{\to} M_i.$$

We have

$$\kappa(M_0) \stackrel{a}{\to} N \stackrel{a}{\to} M_i \stackrel{a}{\to} M_{i+1}$$

and so

$$\kappa(M_0)/a/a \stackrel{a}{\to} M_{i+1}.$$

As $N = M_0 + \kappa(M_0)/a/a$, we therefore have

$$N \stackrel{a}{\to} M_{i+1}.$$

In particular, $N \stackrel{a}{\to} M_j$ for each $j \geq 1$. As the M_j are all distinct, N has an infinite number of a-children.

(vii) However, the effects of multi-step antecedents are worse than simply infinite branching. The transition relation $\stackrel{\cdot}{\to}$ ceases to be decidable. It is straightforward to program Turing machines in a suitable GSOS language, so that $M^{(k)} \stackrel{a^{f(k)}b}{\to} \mathbf{0}$ if the k^{th} Turing machine halts on blank tape (where $f(k)$ is approximately the number of steps the Turning machine takes), and $M^{(k)}$ takes a-steps forever otherwise. Then $\kappa(M^{(k)})/a/a \stackrel{b}{\to} \mathbf{0}$ precisely if the k^{th} Turing machine halts on blank tape, which is undecidable. The same example shows that the question $P \stackrel{a}{\to}$ is undecidable.

These examples, and a collection of similar ones, suggest that the GSOS rules are a maximal simply-defined class of rules meet our requirements. Of course, not every system of operations and rules meeting our criteria is in GSOS format. We do not have the largest possible class, as the class of all such languages is larger.

Furthermore, the requirement that the class be "natural" or "simply-defined" is difficult to formalize. In some contexts, GSOS rules are too restrictive; *e.g.*, the $/a$ operation may be desired despite its dangers. We do not argue that GSOS languages are maximal for all purposes; merely that they are a maximal natural class of languages in which it is simple enough to work.

In other contexts, GSOS rules are too generous. There is currently no interpretation of an arbitrary GSOS-defined operation as some kind of unit-time action of parallel processes, in the concurrent version of a SECD machine. This, however, is desirable. We are arguing that bisimulation is not trace congruence with respect to any reasonable operations; we actually show a stronger theorem: that bisimulation is not congruence with respect to some unreasonable operations as well.

Chapter 4

Theory of Ready Simulation

4.1 Overview

In this chapter, we develop the core of the theory of ready simulation and GSOS languages. We present and prove the equivalence of the main definitions of ready simulation, and in particular we show that ready simulation is precisely congruence with respect to all GSOS languages. We also develop a modal logic which matches ready simulation in the same way that Hennessy-Milner logic matches bisimulation. In Section 4.4, we use this logic to build a GSOS language in which ready simulation is precisely congruence. As a corollary of this work, we show that bisimulation is not congruence with respect to any GSOS language.

The three main characterizations presented in this chapter are ready simulation (RS), denial logic (DL), and GSOS congruence (GC). We prove the equivalences in the order

$$\begin{array}{ccc} RS & \Leftrightarrow & DL \\ \Downarrow & \nearrow & \\ GC & & \end{array}$$

It is simpler to talk about synchronization trees (which are absolute) rather than process terms (which change their meaning depending on the language.)

Definition 4.1.1 *Let P and Q be sychronization trees.*

1. *$P \sqsubseteq_{tr}^{GSOS} Q$ iff, for all GSOS languages $\mathcal{G}$ including P and Q as trees, $P \sqsubseteq_{tr}^{\mathcal{G}} Q$.*

2. *$P \equiv_{tr}^{GSOS} Q$, P and Q are* GSOS congruent, *iff for all GSOS languages $\mathcal{G}$ including P and Q as trees, $P \equiv_{tr}^{\mathcal{G}} Q$.*

Two synchronization trees P, Q are GSOS congruent *iff they are congruent with respect to any GSOS language $\mathcal{G}$ such that $P, Q \in \Delta_{\mathcal{G}}$. Two*

processes in the language $\mathcal{G}$ are GSOS congruent iff their synchronization trees with respect to $\mathcal{G}$ are.

We give several equivalent characterizations of GSOS congruence, in terms more like those defining bisimulation.

4.2 Ready Simulation and GSOS Congruence

The following characterization was discovered (after the modal characterization of Section 4.3) by Larsen and Skou [LS88], and independently by R. van Glabbeek.

Definition 4.2.1

1. *A relation $\sqsubseteq'_{\overrightarrow{}}$ between synchronization trees is a* ready simulation *relation iff, whenever $P \sqsubseteq'_{\overrightarrow{}} Q$,*
   - *Whenever $P \xrightarrow{a} P'$ then there is a Q' such that $Q \xrightarrow{a} Q'$ and $P' \sqsubseteq'_{\overrightarrow{}} Q'$.*
   - *Whenever $P \overset{a}{\not\to}$, then $Q \overset{a}{\not\to}$.*
2. *$P \sqsubseteq_{\overrightarrow{}} Q$ if there is some ready simulation relation $\sqsubseteq'_{\overrightarrow{}}$ such that $P \sqsubseteq'_{\overrightarrow{}} Q$.*
3. *$P \rightleftharpoons Q$ iff $P \sqsubseteq_{\overrightarrow{}} Q$ and $Q \sqsubseteq_{\overrightarrow{}} P$. In this case P and Q are said to be* ready similar.

A useful fact follows immediately from the definition. Let the *ready set* of P be defined by

$$\text{readies}(P) = \left\{a : P \xrightarrow{a}\right\}. \tag{4.1}$$

Then $P \sqsubseteq_{\overrightarrow{}} Q$ implies $\text{readies}(P) = \text{readies}(Q)$. In the presence of the first clause in the definition of ready simulation, this is equivalent to the second clause. The name "ready simulation" comes from the use of the set of actions that the process is ready to perform.

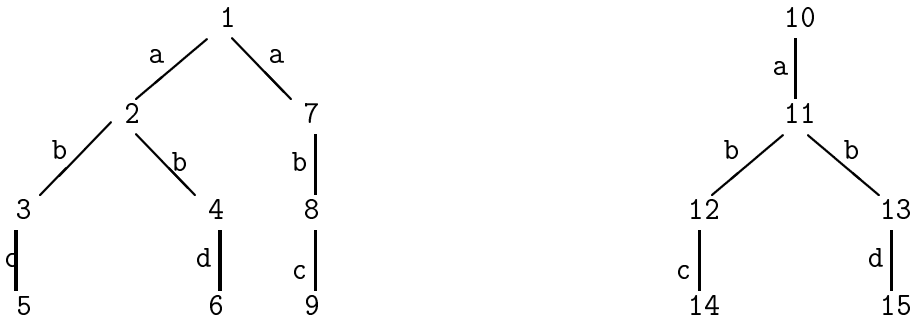
For example, the two processes $a(bc+bd)+abc$ and $a(bc+bd)$ are ready similar; the processes and relations $\sqsubseteq_{\overrightarrow{}_i}$ are shown in Figure 4.1. A similar table shows that $P_\star$ and $Q_\star$ are ready similar.

Lemma 4.2.2 *The relation $\sqsubseteq_{\overrightarrow{}}$ is a ready simulation relation.*

Proof:

Suppose that $P \sqsubseteq_{\overrightarrow{}} Q$, and that $P \xrightarrow{a} P'$. Then there is some ready simulation relation $\sqsubseteq'_{\overrightarrow{}}$ such that $P \sqsubseteq'_{\overrightarrow{}} Q$. By definition there is some Q' such that $Q \xrightarrow{a} Q'$ and $P' \sqsubseteq'_{\overrightarrow{}} Q'$, and hence $P' \sqsubseteq_{\overrightarrow{}} Q'$ as desired. The case of $P \overset{a}{\not\to}$ is equally trivial. □

The main result of this chapter is that $P \rightleftharpoons Q$ iff P and Q are GSOS congruent. Proving this will take the rest of the chapter. Before proving it, we give some examples.



1	$\sqsubseteq_{\rightarrow 1}$	10	10	$\sqsubseteq_{\rightarrow 2}$	1
2	$\sqsubseteq_{\rightarrow 1}$	11	11	$\sqsubseteq_{\rightarrow 2}$	2
3	$\sqsubseteq_{\rightarrow 1}$	12	12	$\sqsubseteq_{\rightarrow 2}$	3
4	$\sqsubseteq_{\rightarrow 1}$	13	13	$\sqsubseteq_{\rightarrow 2}$	4
5	$\sqsubseteq_{\rightarrow 1}$	14	14	$\sqsubseteq_{\rightarrow 2}$	5
6	$\sqsubseteq_{\rightarrow 1}$	15	15	$\sqsubseteq_{\rightarrow 2}$	6
7	$\sqsubseteq_{\rightarrow 1}$	11			
8	$\sqsubseteq_{\rightarrow 1}$	12			
9	$\sqsubseteq_{\rightarrow 1}$	14			

Figure 4.1: Ready Similar Processes

4.2.1 Examples of Ready Simulation

For any process P, we have $P \sqsubseteq_{\rightarrow} P$. Furthermore, for any P and Q we have $aP \sqsubseteq_{\rightarrow} aP + aQ$, using the relation $\sqsubseteq_{\rightarrow}$ itself: the only possible transition of aP is $aP \stackrel{a}{\rightarrow} P$, and $aP + aQ \stackrel{a}{\rightarrow} P$ and $P \sqsubseteq_{\rightarrow} P$ as desired. Also, we have $P_\star \rightleftharpoons Q_\star$; the proof (by exhibiting the two relations) is given Figure 4.1.

Theorem 4.2.3 *Bisimulation is a strict refinement of ready simulation. That is, $P \underline{\leftrightarrow} Q$ implies $P \rightleftharpoons Q$ but not conversely.*

Proof: The first clauses in the definition of bisimulation and ready simulation are the same. Suppose that $P \underline{\leftrightarrow} Q$ and $P \stackrel{a}{\not\rightarrow}$. If $Q \stackrel{a}{\rightarrow} Q'$ for some Q', then by the definition of bisimulation we have $P \stackrel{a}{\rightarrow} P'$ for some $P' \underline{\leftrightarrow} Q'$; in particular, we cannot have $P \stackrel{a}{\not\rightarrow}$. Thus, $Q \stackrel{a}{\not\rightarrow}$ as desired. This shows that bisimulation is a ready simulation relation, and hence that $P \underline{\leftrightarrow} Q$ implies $P \rightleftharpoons Q$. As we have seen, the processes $P_\star$ and $Q_\star$ are ready similar but not bisimilar. □

As a final example, the lossy delay links of Section 1.4.1 are ready similar but not bisimilar.

4.2.2 Ready Simulation Implies GSOS Congruence

In this section, we show the following theorem:

Theorem 4.2.4 *Let P and Q be synchronization trees.*

1. *If $P \sqsubseteq_{\supset} Q$ then $P \sqsubseteq_{tr}^{GSOS} Q$.*
2. *If $P \rightleftharpoons Q$ then $P \equiv_{tr}^{GSOS} Q$.*

The proof of (1) occupies the rest of the section; (2) follows immediately from (1). Remember that the operations $\mathrm{op}(\vec{X})$ were defined by rules of the form

$$\frac{X_i \stackrel{a_{ij}}{\rightarrow} Y_{ij}, \qquad X_i \stackrel{b_{ik}}{\not\rightarrow}}{\mathrm{op}(\vec{X}) \stackrel{c}{\rightarrow} C[\vec{X}, \vec{Y}]}$$

In fact, the same is true of each context $D[\vec{X}]$ as well as the simple contexts $\mathrm{op}(\vec{X})$. That is, the behavior of $D[\vec{X}]$ can be completely captured by a set of derived rules of the form:

$$\frac{X_i \stackrel{a_{ij}}{\rightarrow} Y_{ij}, \qquad X_i \stackrel{b_{ik}}{\not\rightarrow}}{D[\vec{X}] \stackrel{c}{\rightarrow} C[\vec{X}, \vec{Y}]}$$

We will call these constructs "ruloids" rather than "rules" because they are not the rules used to define the language and because they violate our definition of a GSOS rule (the source of the consequent has the wrong form).

Definition 4.2.5 *A set of ruloids R is* specifically witnessing *for a context $D[\vec{X}]$ and action c iff all the consequents of ruloids in R are of the form $D[\vec{X}] \stackrel{c}{\rightarrow} C[\vec{X}, \vec{Y}]$, and whenever $D[\vec{P}] \stackrel{c}{\rightarrow} Q$, there is a ruloid $\rho \in R$ and substitution σ such that $\rho_{\cdot}\sigma = D[\vec{P}] \stackrel{c}{\rightarrow} Q$ and $\stackrel{\cdot}{\rightarrow}, \sigma \models \rho_{-}$.*

Theorem 4.2.6 *Let $\mathcal{G}$ be a GSOS language. For each $\mathcal{G}$-context $D[\vec{X}]$ and action c, there exists a finite set $R_{D,c}$ of ruloids of the form*

$$\frac{X_i \stackrel{a_{ij}}{\rightarrow} Y_{ij}, \qquad X_i \stackrel{b_{ik}}{\not\rightarrow}}{D[\vec{X}] \stackrel{c}{\rightarrow} C[\vec{X}, \vec{Y}]}$$

such that rules $R_{D,c}$ are sound and specifically witnessing for $D[\vec{X}]$.

Proof:

The proof is by induction on the structure of contexts. The construction is straightforward, but the details are tedious. We describe the construction for the context $D[Z_1, Z_2] = \alpha(Z_1 \star Z_2)$, where α and $\star$ are contrived to illustrate the difficult cases of the construction.

$$\frac{X_1 \stackrel{b}{\to} Y_1, \quad X_2 \stackrel{c}{\to} Y_2}{X_1 \star X_2 \stackrel{a}{\to} (Y_1 + Y_2)}, \quad \frac{X_2 \stackrel{e}{\not\to}}{X_1 \star X_2 \stackrel{a}{\to} f X_1}$$

$$\frac{X \stackrel{a}{\not\to}}{\alpha(X) \stackrel{a}{\to} g}, \quad \frac{X \stackrel{a}{\to} Y_1, \quad X \stackrel{a}{\to} Y_2}{\alpha(X) \stackrel{a}{\to} Y_1 \parallel Y_2}$$

The ruloids for $Z_1 \star Z_2$ are simply those for $\star$. For $D[Z_1, Z_2]$, we must consider each rule giving behavior to $\alpha(\cdot)$. The first rule has a single negative hypothesis, $X \stackrel{a}{\not\to}$. This is satisfied iff there is no rule allowing X to take an a-step; that is, if some hypothesis fails on each ruloid which could potentially allow $X = Z_1 \star Z_2$ to take an a-step. In this case, there are two such rules. The first rule is inapplicable precisely if $X_1 \stackrel{b}{\not\to}$ or $X_2 \stackrel{c}{\not\to}$; the second rule is inapplicable precisely if $X_2 \stackrel{e}{\to}$.

We will have one ruloid for each way that *all* the rules giving $Z_1 \star Z_2$ an a-action can fail. Notice that we must invent variables T_1 and T_2, so that we can express the condition that the second rule is inapplicable.[1] So, we start our list of ruloids for $D[Z_1, Z_2]$ with:

$$\frac{Z_1 \stackrel{b}{\not\to}, Z_2 \stackrel{e}{\to} T_1}{\alpha(Z_1 \star Z_2) \stackrel{a}{\to} g}, \quad \frac{Z_2 \stackrel{c}{\not\to}, Z_2 \stackrel{e}{\to} T_2}{\alpha(Z_1 \star Z_2) \stackrel{a}{\to} g}$$

The second rule defining α has two hypotheses. Each of these hypotheses can be fulfilled in two ways, and so we get four more ruloids. Following the algorithm we are sketching, we write the hypotheses in the order of the ruloids they come from, and do not remove redundant duplicates.

$$\frac{Z_1 \stackrel{b}{\to} T_3, \ Z_2 \stackrel{c}{\to} T_4, \ Z_1 \stackrel{b}{\to} T_5, \ Z_2 \stackrel{c}{\to} T_6}{\alpha(Z_1 \star Z_2) \stackrel{a}{\to} (T_3 + T_4) \parallel (T_5 + T_6)}, \quad \frac{Z_1 \stackrel{b}{\to} T_7, \ Z_2 \stackrel{c}{\to} T_8, \ Z_2 \stackrel{e}{\not\to}}{\alpha(Z_1 \star Z_2) \stackrel{a}{\to} (T_7 + T_8) \parallel (f Z_1)}$$

$$\frac{Z_2 \stackrel{e}{\not\to}, \ Z_1 \stackrel{b}{\to} T_9, \ Z_2 \stackrel{c}{\to} T_{10}}{\alpha(Z_1 \star Z_2) \stackrel{a}{\to} (f Z_1) \parallel (T_9 + T_{10})}, \quad \frac{Z_2 \stackrel{e}{\not\to}, \ Z_2 \stackrel{e}{\not\to}}{\alpha(Z_1 \star Z_2) \stackrel{a}{\to} (f Z_1) \parallel (f Z_1)}$$

The algorithm and its correctness should be clear from this example. □

Definition 4.2.7 *The* ruloid set *of a GSOS language $\mathcal{G}$ is the union of the sets $R_{D,c}$ given by Theorem 4.2.6.*

Clearly, $D[\vec{P}] \stackrel{c}{\to} P'$ iff there is a ruloid in the ruloid set of $\mathcal{G}$ with consequent of the form $D[\vec{X}] \stackrel{a}{\to} C[\vec{X}, \vec{Y}]$ witnessing this transition. Now it is fairly straightforward to prove Theorem 4.2.4.

[1] Naturally, this variable must not conflict with any other variable used in the ruloid.

Lemma 4.2.8 *Let $\mathcal{G}$ be a GSOS language, and $P, Q \in \Delta_G$. If $P \sqsubseteq_{\supset} Q$, then $C[P] \sqsubseteq_{\supset} C[Q]$ for each $\mathcal{G}$-context $C[X]$.*

Proof:

To show that $C[P] \sqsubseteq_{\supset} C[Q]$, it suffices to exhibit a ready simulation relation $\sqsubseteq_{\supset}^{\star}$ such that $C[P] \sqsubseteq_{\supset}^{\star} C[Q]$. The obvious candidate is the congruence extension of $\sqsubseteq_{\supset}$ itself, defined by $D[\vec{R}] \sqsubseteq_{\supset}^{\star} D[\vec{S}]$ whenever $\vec{R}$ and $\vec{S}$ are vectors of trees of the right length such that $R_i \sqsubseteq_{\supset} S_i$ for each i. It remains to show that $\sqsubseteq_{\supset}^{\star}$ is a ready simulation relation.

Suppose that $D[\vec{R}] \xrightarrow{c} R'$ for some R'. By Theorem 4.2.6, this is true precisely if there is some ruloid ρ in the ruloid set of $\mathcal{G}$

$$\frac{X_i \xrightarrow{a_{ij}} Y_{ij}, \qquad X_i \xrightarrow{b_{ik}}\!\!\!\!\!\!\rightarrow}{D[\vec{X}] \xrightarrow{c} C[\vec{X}, \vec{Y}]}$$

and trees R_{ij} such that $R_i \xrightarrow{a_{ij}} R'_{ij}$, $R_i \xrightarrow{b_{ik}}\!\!\!\!\!\!\rightarrow$, and $R' = C[\vec{R}, \vec{R'}]$.

As each $R_i \sqsubseteq_{\supset} S_i$, we know that (1) there are S'_{ij} such that $S_i \xrightarrow{a_{ij}} S'_{ij}$ and $R'_{ij} \sqsubseteq_{\supset} S'_{ij}$ for each i, j and (2) $S_i \xrightarrow{b_{ik}}\!\!\!\!\!\!\rightarrow$ for each i, k. So, by ρ, we know that $D[\vec{S}] \xrightarrow{c} C[\vec{S}, \vec{S'}] = S'$. By definition of $\sqsubseteq_{\supset}^{\star}$, we know that

$$R' = C[\vec{R}, \vec{R'}] \;\sqsubseteq_{\supset}^{\star}\; C[\vec{S}, \vec{S'}] = S'$$

and so we have verified the first half of the definition of a ready simulation relation.

The second half is similar; if $D[\vec{R}]$ is unable to take a c-step, then some hypothesis of each ruloid which could allow it to take a c-step must fail. From the fact that $R_i \sqsubseteq_{\supset} S_i$, we discover that the corresponding hypothesis of each ruloid fails for $D[\vec{S}]$ as well, and so $D[\vec{S}] \xrightarrow{c}\!\!\!\!\!\!\rightarrow$ as desired. □

This completes the proof of Lemma 4.2.8. To finish Theorem 4.2.4, we must show:

Lemma 4.2.9 *If $P \sqsubseteq_{\supset} Q$ are synchronization trees, then $P \sqsubseteq_{tr} Q$.*

Proof:

Suppose that $P \sqsubseteq_{\supset} Q$ and $P \xrightarrow{s} P'$ with P' stopped. There is a sequence of processes $P = P_0, P_1, \ldots, P_n = P'$ such that

$$P_0 \xrightarrow{s_1} P_1 \xrightarrow{s_2} \cdots \xrightarrow{s_n} P_n \text{ stopped}$$

By definition, we have $P \sqsubseteq_{\supset} Q$. From the definition of $\sqsubseteq_{\supset}$, we know that there are processes $Q = Q_0, Q_1, \ldots, Q_n$ such that $Q_i \xrightarrow{s_{i+1}} Q_{i+1}$ and $P_i \sqsubseteq_{\supset} Q_i$ for each i. We have

$$Q_0 \xrightarrow{s_1} Q_1 \xrightarrow{s_2} \cdots \xrightarrow{s_n} Q_n.$$

It remains to show that Q_n is stopped. We have $P_n \sqsubseteq_{\supseteq} Q_n$, and so readies$(P_n)$ = readies(Q_n). However, P_n is stopped, and so readies$(P) = \varnothing$. Therefore Q_n is stopped, and so s is a completed trace of Q as desired. □

Theorem 4.2.4 now follows routinely. Suppose that $P \sqsubseteq_{\supseteq} Q$, and $C[X]$ is a context in a GSOS language $\mathcal{G}$. We have $C[P] \sqsubseteq_{\supseteq} C[Q]$ by Lemma 4.2.8, and then $C[P] \sqsubseteq_{tr} C[Q]$ by Lemma 4.2.9. Hence $P \sqsubseteq_{tr}^{\mathcal{G}} Q$. As this holds for all $\mathcal{G}$, we have $P \sqsubseteq_{tr}^{GSOS} Q$.

The following fact will be used later:

Proposition 4.2.10 *If $\mathcal{G}$ is a positive GSOS language, then the ruloid set of $\mathcal{G}$ consists entirely of positive ruloids.*

Proof: Inspection of the proof of Theorem 4.2.6 shows that negative ruloids are only introduced to handle negative rules. □

4.3 A Modal Characterization of Ready Simulation

Recall from Theorem 2.4.8 that bisimulation of finitely branching processes coincides with equivalence with respect to Hennessy-Milner formulas. A similar fact holds for ready simulation. The modal logic is useful for some purposes; *e.g.*, it characterizes the properties preserved by ready simulation. Also, the modal characterization is mathematically useful; in Section 4.4, we use the modal characterization to show that ready simulation is precisely GSOS congruence.

The class of *denial formulas* is

$$\varphi ::== \mathrm{tt} \mid \mathrm{ff} \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \langle a \rangle \varphi \mid \text{Can't}(a) \tag{4.2}$$

The notion of satisfaction is the same for HML formulas and denial formulas, Definition 2.4.6, with the additional clause $P \models \text{Can't}(a)$ iff $P \stackrel{a}{\not\rightarrow}$. Notice that Can't$(a)$ is equivalent to a restricted use of the $[a]$ modality, *viz.* $[a]\mathrm{ff}$, and we do not allow the full use of this modality. Denial logic is not closed under negation, either syntactically or semantically; for example, the formula $\langle a \rangle \langle a \rangle \mathrm{tt}$ has neither kind of negatation.

Definition 4.3.1 *$P \leq_{DL} Q$ iff for all denial formulas φ, $P \models \varphi$ implies $Q \models \varphi$.*
$P \equiv_{DL} Q$ iff $P \leq_{DL} Q$ and $Q \leq_{DL} P$.

Equivalently, $P \equiv_{DL} Q$ iff for all denial formulas φ, $P \models \varphi$ iff $Q \models \varphi$. This theorem was first stated but not proved in [LS88].

Theorem 4.3.2 *If P and Q are finitely-branching synchronization trees, then*

- $P \sqsubseteq_{\rightarrow} Q$ *iff* $P \leq_{DL} Q$.
- $P \rightleftharpoons Q$ *iff* $P \equiv_{DL} Q$.

Proof: The second half follows from the first. Suppose that $P \sqsubseteq_{\rightarrow} Q$. We show that $P \models \varphi$ implies $Q \models \varphi$ by induction on φ simultaneously for all P and Q.

1. tt and ff are trivial.
2. Suppose $P \models \varphi \wedge \psi$. Then $P \models \varphi$ and $P \models \psi$, and by induction we have $Q \models \varphi$ and $Q \models \psi$ and hence $Q \models \varphi \wedge \psi$ as desired. Disjunctions are similar.
3. Suppose that $P \models \langle a \rangle \varphi$. Then there is a P' such that $P \stackrel{a}{\rightarrow} P'$ and $P' \models \varphi$. As $P \sqsubseteq_{\rightarrow} Q$, there is a Q' such that $P' \sqsubseteq_{\rightarrow} Q'$ and $Q \stackrel{a}{\rightarrow} Q'$. By induction, $Q' \models \varphi$; hence $Q \models \langle a \rangle \varphi$.
4. Suppose that $P \models \text{Can't}(a)$. Then $P \stackrel{a}{\not\rightarrow}$, and so $Q \stackrel{a}{\not\rightarrow}$; which is to say $Q \models \text{Can't}(a)$.

To prove the converse, we show that $\leq_{DL}$ is a ready simulation relation. Suppose that $P \leq_{DL} Q$.

- Suppose that $P \stackrel{a}{\rightarrow} P'$. We must show that there is some Q' such that $Q \stackrel{a}{\rightarrow} Q'$ and $P' \leq_{DL} Q'$. Suppose for contradiction that there is no a-child Q' of Q such that $P' \leq_{DL} Q'$. Q has a finite number of a-children, $Q_1, \ldots, Q_n$. For each child Q_i, there is a formula ψ_i such that $P' \models \psi_i$ but $Q' \not\models \psi_i$. Let $\psi = \psi_1 \wedge \cdots \wedge \psi_n$; if there are no children, then let $\psi = \text{tt}$. Then $P' \models \psi$ and so $P \models \langle a \rangle \psi$. However, $Q \not\models \langle a \rangle \psi$, which violates the assumption that $P \leq_{DL} Q$
- Suppose that $P \stackrel{a}{\not\rightarrow}$. Then $P \models \text{Can't}(a)$, and so $Q \models \text{Can't}(a)$. This is equivalent to $Q \stackrel{a}{\not\rightarrow}$ as desired.

We have shown that $\leq_{DL}$ is a ready simulation relation, and so $P \leq_{DL} Q$ implies $P \sqsubseteq_{\rightarrow} Q$.

□

In fact, the full syntax of denial formulas is not required; disjunctions and ff are not necessary. In particular, the formulas $\langle a \rangle(\varphi \vee \psi)$ and $(\langle a \rangle \varphi) \vee (\langle a \rangle \psi)$ are logically equivalent. The *essential denial formulas* are given by the syntax:

$$\varphi ::== \text{tt} \mid \varphi \wedge \varphi \mid \langle a \rangle \varphi \mid \text{Can't}(a).$$

Lemma 4.3.3 *$P =_{DL} Q$ iff P and Q agree on all essential denial formulas.*

Proof: Use the fact that $\langle a \rangle(\varphi \vee \psi)$ and $(\langle a \rangle \varphi) \vee (\langle a \rangle \psi)$ are logically equivalent, and the other rules of modal logic. □

4.4 Ready Simulation Can Be Traced

In this section we introduce an extension CCSSS of CCS whose congruence is just ready simulation. We add two operations. $\Diamond\!\!\Diamond P$ is a copying operator: when P signals that it wants to fork, $\Diamond\!\!\Diamond P$ forks. $S \triangleright P$ is a sort of controlled communication: S runs alone, except that it occasionally allows P the ability to take a step and communicate with it.

Using these operations, we will code denial formulas into contexts and traces, and so understand ready simulation in CCSSS. $C_\varphi[P]$ tests the process P to see if it satisfies φ, producing a characteristic kind of trace if it does and not if it does not.

Formally, we fix several distinct actions. We will use o as a sort of "visible silent action;" processes will emit o's while they are operating. The actions c_1 and c_2 are used by processes to signal to the $\Diamond\!\!\Diamond$ operator that they wish to fork. In $S \triangleright P$, S uses the d action to signal that it wishes to communicate with P. There is an auxiliary operator $\triangleright\!\!\!\triangleright$ used by $\triangleright$.

$\Diamond\!\!\Diamond(P)$ usually does just what P does. However, when P signals that it wants to be forked (by the c_1 and c_2 actions), $\Diamond\!\!\Diamond(P)$ forks it.

$$\frac{X \stackrel{a}{\to} X' \quad (a \notin \{c_1, c_2\})}{\Diamond\!\!\Diamond X \stackrel{a}{\to} \Diamond\!\!\Diamond X'} \qquad \frac{X \stackrel{c_1}{\to} X_1, X \stackrel{c_2}{\to} X_2}{\Diamond\!\!\Diamond X \stackrel{a}{\to} (\Diamond\!\!\Diamond X_1) \parallel (\Diamond\!\!\Diamond X_2)}$$

$S \triangleright P$ usually does just what S does; P is frozen. However, when S signals that it wants to communicate with P (by performing a d-step), $S \triangleright P$ unfreezes P and lets it take a step in cooperation with S. This operation needs a bit of control state, telling whether or not P is frozen; we use the $\triangleright$ operator when P is frozen, and the $\triangleright\!\!\!\triangleright$ operator when P is unfrozen.

$$\frac{S \stackrel{a}{\to} S' \quad (a \neq d)}{S \triangleright P \stackrel{a}{\to} S' \triangleright P} \qquad \frac{S \stackrel{d}{\to} S'}{S \triangleright P \stackrel{o}{\to} S' \triangleright\!\!\!\triangleright P}$$

The operation $\triangleright\!\!\!\triangleright$ does one step of communication and then behaves like $\triangleright$.

$$\frac{S \stackrel{a}{\to} S', \quad P \stackrel{a}{\to} P'}{S \triangleright\!\!\!\triangleright P \stackrel{o}{\to} S' \triangleright P'}$$

We now define the coding of formulas. To make strings of actions easier to read, we write prefixing with a dot: "$d.a.t.S$" instead of "$datS$." Fix two actions t and f, distinct from the previously-mentioned actions, which we use for partial success and failure.

$$\begin{aligned} S_{tt} &= \mathbf{0} \\ S_{\mathrm{can't}(a)} &= d.a.f \\ S_{\varphi \wedge \psi} &= c_1 S_\varphi + c_2 S_\psi \\ S_{\langle a \rangle \varphi} &= d.a.t.S_\varphi \end{aligned}$$

The context $C_\varphi[X]$ is defined to be $\Diamond\!\!\!\!\Diamond (S_\varphi \triangleright X)$. $C_\varphi[P]$ will compute, emitting o's while it is working. Each time it processes an $\langle a \rangle$ correctly, it will emit a t. Each time it fails to perform a Can't(a) correctly, it will emit an f. We will show that $P \models \varphi$ iff $C_\varphi[P]$ produces a trace with enough t's and no f's.

For example,

$$\begin{aligned}
C_{\langle a\rangle tt \wedge \langle b\rangle tt}[a+b] &= \Diamond\!\!\!\!\Diamond ((c_1.d.a.t + c_2.d.b.t) \triangleright (a+b)) \\
&\xrightarrow{o} \Diamond\!\!\!\!\Diamond (d.a.t \triangleright (a+b)) \parallel \Diamond\!\!\!\!\Diamond (d.b.t \triangleright (a+b)) \\
&\xrightarrow{o} \Diamond\!\!\!\!\Diamond (a.t \triangleright\!\!\!\triangleright (a+b)) \parallel \Diamond\!\!\!\!\Diamond (d.b.t \triangleright (a+b)) \\
&\xrightarrow{o} \Diamond\!\!\!\!\Diamond (t \triangleright \mathbf{0}) \parallel \Diamond\!\!\!\!\Diamond (d.b.t \triangleright (a+b)) \\
&\xrightarrow{t} \Diamond\!\!\!\!\Diamond (\mathbf{0} \triangleright \mathbf{0}) \parallel \Diamond\!\!\!\!\Diamond (d.b.t. \triangleright (a+b)) \\
&\xrightarrow{o} \Diamond\!\!\!\!\Diamond (\mathbf{0} \triangleright \mathbf{0}) \parallel \Diamond\!\!\!\!\Diamond (b.t \triangleright\!\!\!\triangleright (a+b)) \\
&\xrightarrow{o} \Diamond\!\!\!\!\Diamond (\mathbf{0} \triangleright \mathbf{0}) \parallel \Diamond\!\!\!\!\Diamond (t \triangleright \mathbf{0}) \\
&\xrightarrow{t} \Diamond\!\!\!\!\Diamond (\mathbf{0} \triangleright \mathbf{0}) \parallel \Diamond\!\!\!\!\Diamond (\mathbf{0} \triangleright \mathbf{0})
\end{aligned}$$

To illustrate how the testing for Can't(a) works, consider:

$$\begin{aligned}
C_{\mathrm{can't}(a)}[(a+b)] &= \Diamond\!\!\!\!\Diamond (d.a.f \triangleright (a+b)) \\
&\xrightarrow{o} \Diamond\!\!\!\!\Diamond (a.f \triangleright\!\!\!\triangleright (a+b)) \\
&\xrightarrow{o} \Diamond\!\!\!\!\Diamond (f \triangleright \mathbf{0}) \\
&\xrightarrow{f} \Diamond\!\!\!\!\Diamond (\mathbf{0} \triangleright \mathbf{0})
\end{aligned}$$

So, the only trace of $C_{\mathrm{can't}(a)}[a+b]$ contains an f. However, the only computation of

$$C_{\mathrm{can't}(a)}[b] = \Diamond\!\!\!\!\Diamond (d.a.f \triangleright b) \xrightarrow{o} \Diamond\!\!\!\!\Diamond (a.f \triangleright\!\!\!\triangleright b)$$

gets stuck after performing an o.

Define $\lfloor \varphi \rfloor$ to be the number of $\langle a \rangle$'s occurring in φ; that is:

$$\begin{aligned}
\lfloor \mathrm{tt} \rfloor = \lfloor \mathrm{Can't}(a) \rfloor &= 0 \\
\lfloor \varphi \wedge \psi \rfloor &= \lfloor \varphi \rfloor + \lfloor \psi \rfloor \\
\lfloor \langle a \rangle \varphi \rfloor &= 1 + \lfloor \varphi \rfloor
\end{aligned}$$

We say that a trace s is φ-happy if it contains exactly $\lfloor \varphi \rfloor$ t's and no f's. A trace is φ-sad if it it contains fewer than $\lfloor \varphi \rfloor$ t's, or at least one f.

Lemma 4.4.1 *If $P \models \varphi$ then $C_\varphi[P]$ has a φ-happy trace. If $P \not\models \varphi$, then all traces of $C_\varphi[P]$ are φ-sad.*

Proof:

We induct on φ, with the hypothesis given by the lemma and that no trace of $C_\varphi[P]$ has more than $\lfloor \varphi \rfloor$ t's.

$\varphi = \mathrm{tt}$: $C_{tt}[P]$ is stopped for all P, as desired.

$\varphi = \psi \wedge \theta$: $C_{\psi\wedge\theta}[P] = \langle\!\langle\, ((c_1 S_\psi + c_2 S_\theta) \triangleright P)$. As $((c_1 S_\psi + c_2 S_\theta) \triangleright P)$ can make both c_1 and c_2 transitions, the $\langle\!\langle$ forks the process:

$$C_{\psi\wedge\theta}[P] = \langle\!\langle\, ((c_1 S_\psi + c_2 S_\theta) \triangleright P) \xrightarrow{o} \langle\!\langle\, (S_\psi \triangleright P) \| \langle\!\langle\, (S_\theta \triangleright P) = C_\psi[P] \| C_\theta[P]$$

The induction hypothesis follows from the ordinary properties of sequences and interleaving.

$\varphi = \mathrm{Can't}(a)$

$$C_{\mathrm{can't}(a)}[P] = \langle\!\langle\, (d.a.f \triangleright P) \xrightarrow{o} \langle\!\langle\, (a.f \triangleright P)$$

If $P \overset{a}{\not\to}$, then $\langle\!\langle\, (a.f \triangleright P)$ cannot move and the trace is simply the φ-happy trace o.

If $P \xrightarrow{a} P'$, then

$$\langle\!\langle\, (a.f \triangleright P) \xrightarrow{o} \langle\!\langle\, (f \triangleright P') \xrightarrow{f} \langle\!\langle\, (\mathbf{0} \triangleright P')$$

In this case, the trace of $C_{\mathrm{can't}(a)}[P]$ is oof, which is φ-sad.

$\varphi = \langle a \rangle \psi$

$$C_{\langle a\rangle\psi}[P] \xrightarrow{o} \langle\!\langle\, (a.t.S_\psi \triangleright P)$$

Consider any P' such that $P \xrightarrow{a} P'$. (If there are no such P''s, then the process is stuck and the trace is φ-sad as required.)

$$\langle\!\langle\, (a.t.S_\psi \triangleright P) \xrightarrow{o} \langle\!\langle\, (t.S_\psi \triangleright P') \xrightarrow{t} \langle\!\langle\, (S_\psi \triangleright P') = C_\psi[P']$$

If $P \models \varphi$, then there is a P' such that $P \xrightarrow{a} P' \models \psi$. By the induction hypothesis $C_\psi[P']$ has a ψ-happy trace, and so we have found a φ-happy trace of $C_\varphi[P]$.

If $P \not\models \varphi$, then $P' \not\models \psi$, and so every trace of $C_\psi[P]$ is ψ-sad; thus every trace of $C_\varphi[P]$ that goes through $C_\psi[P']$ is φ-sad. Every such trace must go through some $C_\psi[P]$, and so every trace of $C_\varphi[P]$ must be φ-sad.

The requirement that no trace have more than $\lfloor \varphi \rfloor$ t's is routine.

□

4.4.1 Partial Traces

Although we have concentrated on finite completed traces, similar theory applies to the case of finite partial traces. For finite completed traces, we do not need negative rules; there is enough negation expressed in the statement that s is a completed trace of P. For partial traces, negative rules are essential for the following theorem. [Blo89] We use the sequencing operator $P;Q$, and add one more action x to mark the end of a trace.

Theorem 4.4.2 *$P \rightleftharpoons Q$ iff P and Q are partial trace congruent with respect to all GSOS languages.*

Proof: Clearly if $P \rightleftharpoons Q$ then P and Q have the same set of partial traces. If $C[X]$ is a GSOS context, then $C[P] \rightleftharpoons C[Q]$; hence P and Q have the same partial traces in all GSOS contexts.

Conversely, let CCSSSS be CCSSS with sequencing. Let $C_\varphi[X]$ be as in Lemma 4.4.1, and let $C'_\varphi[X] = C_\varphi[X];x$ where x is a new action. Then $C'_\varphi[P]$ has a partial trace containing $\lfloor\varphi\rfloor$ t's and no f's ending in an x. As before, non-ready-similar processes can be distinguished by the presence of such partial traces. □

4.5 Summary of Ready Simulation

Combining the results of the previous sections, we obtain the following set of equivalent characterizations of GSOS congruence.

Theorem 4.5.1 *The following are equivalent:*

1. *$P \sqsubseteq\!\!\!\!_{\rightarrow}\ Q$. (State-correspondance definition)*
2. *$P \sqsubseteq_{tr}^{GSOS} Q$. (Approximation in all GSOS languages.)*
3. *$P \sqsubseteq_{tr}^{\mathcal{L}} Q$ for all positive GSOS languages $\mathcal{L}$*
4. *$P \sqsubseteq Q$ with respect to partial traces and all GSOS languages*
5. *$P \leq_{DL} Q$. (Approximation with respect to all denial formulas)*
6. *$P \models \varphi$ implies $Q \models \varphi$ for all essential denial formulas φ.*
7. *$P \sqsubseteq_{tr}^{CCSSS} Q$. (Trace approximation in CCSSS)*

The implications have been shown in the order

$$1 \Rightarrow 2 \Rightarrow 3 \Rightarrow 7$$
$$1 \Rightarrow 4$$
$$1 \Leftrightarrow 5 \Leftrightarrow 6$$
$$7 \Rightarrow 6$$
$$4 \Leftrightarrow 6$$

Corollary 4.5.2 *Bisimulation is a strict refinement of ready simulation, and hence of GSOS congruence. In particular, the processes $P_\star$ and $Q_\star$ are trace congruent with respect to every GSOS language, although they are bisimilar.*

There are a few other definitions of ready simulation, but they are of less interest. For example, it is possible to define the n^{th} approximant to ready simulation in the way that [Mil80, Mil81] defined the n^{th} approximant to bisimulation; predictably, if $P \sqsubseteq_{\supset n} Q$ for all n, then $P \sqsubseteq_{\supset} Q$.

Chapter 5

Experimental Equivalences

5.1 Experiments on Machines

The notion of congruence depends on two parameters: the programming language and the set of observations. In the preceding work, we have varied the language over all GSOS languages, but the observations have remained fixed: we have always observed finite completed traces. In this chapter, we will consider other kinds of observations. We have a conceptual framework for evaluating experiments; however, we have no metatheory of experiments comparable with the metatheory of GSOS languages. The results in this chapter are suggestive rather than definitive.

We draw inspiration from the most common physical analogy for CCS processes. A process is thought of as a black box, with one button for each action that it can potentially take.[1] The experimenter presses buttons on the box. If the process can actually take that action, the machine will change state; if it cannot, the button cannot be pressed. Two processes are identified iff the experimenter cannot tell them apart, no matter how they are used. The basic experiment on a process P is to place it in some context $C[P]$, and then observe the outcomes of experiments on $C[P]$.

Although the original definition of bisimulation was not given in terms of button-pushing experiments on black boxes, Milner does offer a form of justification in these terms[Mil81]. In these experiments, the experimenter is given the ability to perform repeated subexperiments from any state, allowing the exploration of the alternatives available in a given state. This may be phrased in several ways; for example, one might permit the experimenter to *save* states and later *restore* the process to any saved state. Abramsky [Abr, AV] uses the metaphor of allowing the experimenter to *undo* steps. An alternative and more tangible formulation is that the experimenter is equipped with a *duplicator*, allowing the creation of identical copies of the

[1] In the asynchronous case, it is necessary to add a "running light," which is lit while the machine is making internal transitions.

process in any state. The experimenter may perform experiments on the copies, and combine the results.

In general, an experiment on P should consist of placing P in a context, $C[P]$, and performing experiments on $C[P]$. However, it will turn out that the use of contexts expressible in CCS — or indeed in any GSOS language — does not change the stronger forms of experimental equivalences which we consider, and so it suffices to simply perform experiments on the processes in isolation.

How then does Milner characterize bisimulation by experiments? The experimenter must be able to do more than simply make duplicates of a process; he must be able to make enough duplicates to know that he has explored all possible alternative behaviors. Milner uses the metaphor of "weather conditions," which determine which nondeterministic choice the process will make. We take the equivalent metaphor of a *global-testing duplicator*, which makes copies of the process displaying all of its alternatives. This allows the experimenter to be sure that he has in fact explored all of its possible behaviors. There is a notion of global-testing experiment (which we call *modal* experiments) which precisely captures bisimulation[Mil81].

However, the restriction on experiments raises an uncomfortable point. The restriction amounts to denying the experimenter the ability to *count* the number of processes that the global-testing duplicator has produced, or equivalently the number of kinds of weather available. The duplicates or weathers are available for performing tests on, and in general the experimenter must test each of them; the physical justification for forbidding counting them is less than totally clear. However, allowing the experimenter access to this information allows him to make more distinctions than we want.

We will explore several kinds of experiments, trying to make them as powerful as bisimulation. We introduce the *duplicator* and *global-testing duplicator* scenarios, and some variations; we show the following:

- Duplicator equivalence coincides with ready simulation.
- A restricted form of global-testing duplicator equivalence coincides with bisimulation.
- Global-testing duplicator equivalence coincides with synchronization tree isomorphism.

Let us consider the simplest kind of black box which we have described. It has buttons labeled with actions, and no other controls. The buttons serve as sensors, in a weak sense. If P can perform a, then it is possible to press the a-button, but then the machine will change state. If P cannot perform a, then the a-button is locked; the experimenter can press the button, discover that the machine cannot perform an a, and then continue experimenting on P itself.

It is informative to formalize this notion of experiment. An elementary experiment is simply a sequence of buttons to push, and for each an indicator of whether the button should be enabled or disabled (*viz.*, whether or not the process can perform the action given by that button). It may be represented as a sequence of pairs of actions and `no`'s or `yes`'s, looking something like "$\langle a, \texttt{no}\rangle\langle b, \texttt{yes}\rangle\langle c, \texttt{no}\rangle$". To do this, the experimenter presses the a, b, and c buttons in that order; if only the second can be pressed, the experiment succeeds. Note that P may be able to both succeed and fail on the same experiment, depending on which nondeterministic choices the process makes; for example, $b + bc$ can succeed and can fail on $\langle a, \texttt{no}\rangle\langle b, \texttt{yes}\rangle\langle c, \texttt{no}\rangle$. Formally, the process P can succeed on the experiment σ iff there is a sequence $P_1, \ldots, P_n$, where $n = |\sigma|$, such that $P = P_1$ and for each $i = 1, \ldots, n$,

- If the i^{th} component of σ is $\langle a_i, \texttt{yes}\rangle$, then $P_i \stackrel{a_i}{\rightarrow} P_{i+1}$.
- If the i^{th} component of σ is $\langle a_i, \texttt{no},\rangle$, then $P_i \stackrel{a_i}{\not\rightarrow}$ and $P_i = P_{i+1}$.

The formalization of failure is similar: P can fail on σ iff there is an integer $k < |\sigma|$ such that P succeeds on the first $k-1$ elements of σ, and fails on the k^{th}; *viz.*, if $\sigma_k = \langle a_k, \texttt{yes}\rangle$ then $P_k \stackrel{a_k}{\not\rightarrow}$ and if $\sigma_k = \langle a_k, \texttt{no}\rangle$ then $P_k \stackrel{a_k}{\rightarrow}$. Note that this is not equivalent to the claim that "P can fail on σ iff P cannot succeed on σ," which is not true.

There are more elaborate experiments one could perform, but they can all be understood as sets of elementary experiments. For example, the experiment "Press the a, b, c, and d buttons in that order, and succeed if two of them can be pressed" can succeed iff one of the obvious set of six elementary experiments can. Other kinds of experiments involve decisions of what to do next based on partial results: "Press the a button. If it can be pressed, press b and succeed if it is enabled. Otherwise, press c and succeed if it is disabled." This experiment can be passed iff either $\langle a, \texttt{yes}\rangle\langle b, \texttt{yes}\rangle$ or $\langle a, \texttt{no}\rangle\langle c, \texttt{no}\rangle$ can. We do not lose information by working only with elementary experiments, if we choose the right set of elementary experiments.

Definition 5.1.1 *Two processes are* equivalent *with respect to a class of experiments iff they can succeed on precisely the same experiments of that class. P* approximates *Q with respect to a class of experiments iff whenever P can succeed on an experiment, so can Q.*

In this section, we pursue the metaphor of performing experiments on machines. We will consider a process as a machine or black box, with a few controls and indicators, and try to understand what a determined experimenter could observe about such a machine.

A number of variants of these simple button-pushing experiments are possible, varying the machine's behavior when a disabled button is pushed.

Perhaps the most detailed kind of experiment involving simply button-pushing is a *lighted-button experiment*. In this scenario, the black box resembles certain soft-drink machines; its buttons have lights inside them, and the light on the a-button is lit when that button is disabled. In other words, the experimenter can see at every stage which actions are possible and which are not, without changing the state of the machine. The basic experiment on such a machine is a sequence $S_0 a_1 S_1 \ldots a_n S_n$ of alternating sets of actions and actions; it succeeds when S_0 is the set of initially disabled actions, S_1 is the set of disabled actions after a_1 is pressed, and so on. It is easy to explain to programmers why they should care about distinctions observed in lighted-button terms; the programmers are familiar with such machines in the real world, and frequently subsist on food from them.

5.2 Duplicator Equivalence

The previous experiments have involved fairly simple use of processes. Finer equivalences can be obtained by equipping the experimenter with a *duplicator*, allowing him to copy the machine at any time, and perform independent experiments on the copies. (Equivalently, we allow him to save and restore states of the machine. These are the only permissible operations on states — in particular he cannot determine if two states are equal. If the machine is a program running on a computer, saving states is relatively plausible. In many systems, it would be possible to implement a save/restore state function, vaguely related to the `roll-in` and `roll-out` functions of some LISPs, and the `fork()` system call, without excessive difficulty.) The typical sort of *duplicator*-experiment looks something like the following:

1. Press the a-button on P, and call the resulting machine P_a. Fail if the a-button cannot be pushed.

2. Make two copies of P_a, call them P_{a1} and P_{a2}. This step cannot fail.

3. Press the b-button and then the c-button of P_{a1}. Fail if either cannot be pushed.

4. Press the b-button and then the c-button of P_{a2}. Fail if either the b-button cannot be pushed, or the c-button can be pushed.

5. Succeed if none of the previous steps have failed.

The process $a(bc + bd)$ can pass this test, but the process $abc + abd$ cannot. Note that these two processes are lighted-button equivalent. It is possible to simulate a lighted-button experiment with a duplicator, so we have gained expressive power. Duplicator experiments give us a familiar process equivalence:

Theorem 5.2.1 *Approximation and equivalence with respect to duplicator experiments is precisely ready simulation approximation and equivalence.*

Recall that a duplicator experiment gives the experimenter the ability to press buttons on boxes, and make copies of a box and perform different experiments on each. Experiments are thus defined to be

- succeed and fail, which always succeed and fail respectively;
- push a then E, for pushing the a-button and then performing the experiment E; this succeeds if a can be pushed and E then succeeds, and fails if a cannot be pushed or E fails.
- copy$(E\beta E')$, for making two copies of the box, performing E on one, E' on the other, and combining the results of the experiments by the binary Boolean function $\beta : \mathbf{B}^2 \to \mathbf{B}$.

Note that we allow arbitrary Boolean functions β on copying; in particular, we allow negation. For simplicity we do not allow arbitrary Boolean functions on button-pushes; the Boolean functions on copying experiments are sufficiently powerful to encode all the ways of interpreting results of push a then E. We could give a more powerful form of the push a then E experiment giving an experiment E' to perform if a cannot be pushed, but this can be simulated with suitable use of copying.

We define not E to be the experiment copy(E xor succeed), where xor is exclusive-or; P can succeed on not E iff it can fail on E and vice versa. Similarly, we define E and $E' =$ copy$(E \wedge E')$. In some loose sense, not E is a negation, push a then E is a possibility modality, and EandE' is conjunction. It looks as if we could code Hennessy-Milner (in the form with negation) logic as duplicator experiments, defining E_φ inductively in the obvious way:

$$\begin{aligned} E_{tt} &= \text{succeed} \\ E_{\varphi\wedge\psi} &= E_\varphi \text{ and } E_\psi \\ E_{\langle a\rangle\varphi} &= \text{push } a \text{ then } E_\varphi \\ E_{\neg\varphi} &= \text{not } E_\varphi \end{aligned}$$

However, this is deceptive; it is not the case that $P \models \varphi$ iff P can succeed on the experiment E_φ. For example, let $P = ab + ac$ and $\varphi = [a]\langle b\rangle\mathrm{tt} = \neg\langle a\rangle\neg\langle b\rangle\mathrm{tt}$. E_φ first pushes a (succeeding if it cannot be pushed), then b, *failing* if it cannot be pushed and succeeding otherwise. Notice that $P \not\models \varphi$, but $P \xrightarrow{a} b$ is a successful run of the experiment. This kind of experiment does not understand necessity very well; it treats it almost like a possibility modality (succeeding rather than failing if the button can't be pushed).

We formalize the passing and failing of experiments, $P \mathrel{\Delta} E$ and $P \mathrel{\nabla} E$, as follows:

Definition 5.2.2 • $P \mathbin{\Delta\!\!\!\!\!+} \mathsf{succeed}$ *and* $P \mathbin{\nabla\!\!\!\!\!+} \mathsf{fail}$ *are always true;* $P \mathbin{\Delta\!\!\!\!\!+} \mathsf{fail}$ *and* $P \mathbin{\nabla\!\!\!\!\!+} \mathsf{succeed}$ *are always false.*

- $P \mathbin{\Delta\!\!\!\!\!+} \mathsf{push}\ a\ \mathsf{then}\ E$ *is true iff there is some* P' *such that* $P \xrightarrow{a} P$ *and* $P' \mathbin{\Delta\!\!\!\!\!+} E$.
  $P \mathbin{\nabla\!\!\!\!\!+} \mathsf{push}\ a\ \mathsf{then}\ E$ *is true if either* $P \xrightarrow{a}$, *or there is some* P' *such that* $P \xrightarrow{a} P'$ *and* $P' \mathbin{\nabla\!\!\!\!\!+} E'$.

- $P \mathbin{\Delta\!\!\!\!\!+} \mathsf{copy}(E_1 \beta E_2)$ *is true if there are truth values* b_1 *and* b_2 *such that* $\beta(b_1, b_2) = \mathrm{tt}$ *and* $\left\{ \begin{array}{ll} \text{if } b_i = \mathrm{tt} & \text{then } P \mathbin{\Delta\!\!\!\!\!+} E_i \\ \text{if } b_i = \mathrm{ff} & \text{then } P \mathbin{\nabla\!\!\!\!\!+} E_i \end{array} \right\}$.
  $P \mathbin{\nabla\!\!\!\!\!+} \mathsf{copy}(E_1 \beta E_2)$ *is similar, except that we look for* $\beta(b_1, b_2) = \mathrm{ff}$.

Notice that it is quite possible for $P \mathbin{\Delta\!\!\!\!\!+} E$ and $P \mathbin{\nabla\!\!\!\!\!+} E$ both to hold; this simply says that one nondeterministic computation of P succeeds on the experiment E and some other one does not. This is fundamentally different from the notion $P \models \varphi$, where we cannot have $P \models \varphi$ and $P \not\models \varphi$ simultaneously. However, an induction on experiments shows that we always have at least one of $P \mathbin{\Delta\!\!\!\!\!+} E$ and $P \mathbin{\nabla\!\!\!\!\!+} E$ holding for every P and E.

The connection between duplicator experiments and ready simulation is quite close.

Definition 5.2.3 $P \leq_{\mathsf{Dup}} Q$ *if whenever* $P \mathbin{\Delta\!\!\!\!\!+} E$ *then* $Q \mathbin{\Delta\!\!\!\!\!+} E$.

Notice that $P \mathbin{\Delta\!\!\!\!\!+} E$ iff $P \mathbin{\nabla\!\!\!\!\!+} \mathsf{not}\ E$; in particular, $P \leq_{\mathsf{Dup}} Q$ iff whenever $P \mathbin{\nabla\!\!\!\!\!+} E$ then $Q \mathbin{\nabla\!\!\!\!\!+} E$. Thus, we lose no generality by not considering failure.

Theorem 5.2.4 *For finitely branching processes* P *and* Q, $P \leq_{\mathsf{Dup}} Q$ *iff* $P \sqsubseteq_{\rightarrow} Q$

Proof: ($\Rightarrow$) We show that $\leq_{\mathsf{Dup}}$ is a ready simulation relation. Suppose that $P \leq_{\mathsf{Dup}} Q$. We must show that if $P \xrightarrow{a} P'$ then there is some Q' such that $Q \xrightarrow{a} Q'$ and $P' \leq_{\mathsf{Dup}} Q'$, and that if $P \xrightarrow{a}$ then $Q \xrightarrow{a}$. For the former, suppose that $P \xrightarrow{a} P'$, but that for no Q' does $P' \leq_{\mathsf{Dup}} Q'$. Let $Q_1, \ldots, Q_n$ enumerate the a-children of Q. For each i, there is some experiment E_i such that $P' \mathbin{\Delta\!\!\!\!\!+} E_i$ but not $Q_i \mathbin{\Delta\!\!\!\!\!+} E_i$. Let $E' = E_1\ \mathsf{and}\ E_2\ \mathsf{and} \cdots \mathsf{and}\ E_n$. It is easily verified that $P' \mathbin{\Delta\!\!\!\!\!+} E'$ but for no Q_i is it true that $Q_i \mathbin{\Delta\!\!\!\!\!+} E'$. Let $E'' = \mathsf{push}\ a\ \mathsf{then}\ E'$. Then $P \mathbin{\Delta\!\!\!\!\!+} E''$, as witnessed by $P \xrightarrow{a} P'$; however, it is not the case that $Q \mathbin{\Delta\!\!\!\!\!+} E''$. Therefore, $P \not\leq_{\mathsf{Dup}} Q$, which is absurd.

Similarly, if $P \xrightarrow{a}$, then let $E = \mathsf{not}\ (\mathsf{push}\ a\ \mathsf{then}\ \mathsf{succeed})$; we have $P \mathbin{\Delta\!\!\!\!\!+} E$ and therefore $Q \mathbin{\Delta\!\!\!\!\!+} E$, whence $Q \xrightarrow{a}$.

($\Leftarrow$) Easy induction on the structure of experiments. □

5.3 Wild Duplicator Experiments

We have seen that simply copying processes and exploring a fixed number of descendants does not suffice to explain bisimulation. It is necessary to explore all of the alternatives, the entire set of a-children of a process. We might hope that a more powerful duplicator will capture bisimulation; *e.g.*, the *wild duplicator* which countably nondeterministically chooses an integer n and makes n copies of the input process P. In some trials, the wild duplicator will choose n larger than the number of children of P. The experimenter, pressing the a-button on each of these, has the possibility of seeing all of the a-behavior of P in a single experiment. However, this does not add any distinguishing power, and so does not achieve bisimulation:

Theorem 5.3.1 *Wild-duplicator approximation equivalence coincides with ready simulation approximation and equivalence, and hence on finitely-branching processes with ordinary duplicator equivalence.*

A wild duplicator is, intuitively, a sort of duplicator which produces $n \geq 0$ copies of its input process. Wild duplicator experiments must be allowed to branch on n, and in general we may as well allow them to behave differently for each value of n. We will be excessively generous in our definitions of experiments, allowing infinities and divergence to appear in several ways; every informal wild duplicator experiment can be understood as a formal one, but the converse does not hold. Nonetheless, wild duplicator equivalence is precisely ready simulation.

We will allow infinite numbers of tests, and arbitrary Boolean combinations of the results. We do not restrict infinities to be countable. The infinities allow the results of this section apply to arbitrarily branching processes, and *a fortiori* to finitely branching processes.

Definition 5.3.2 *A* wild duplicator experiment *is an ordered tree, possibly countably deep and arbitrarily wide, with node labels and branching as follows. Each node ν is labeled either* choose, *$a \in \mathsf{Act}$, or $B : \mathbf{B}^\kappa \to \mathbf{B}$, where κ is a cardinal ≥ 0, such that*

1. *If ν is labeled a, then ν has exactly two children ν_+ and ν_-.*
2. *If ν is labeled $B : \mathbf{B}^\kappa \to \mathbf{B}$, then ν has κ children.*
3. *If ν is labeled* choose *then ν may have any number of children.*

The intent is that nodes ν labeled with actions a involve pushing the a-button on the process. If the button can be pushed, then the experimenter proceeds with ν_+ on the resultant process; otherwise, the experimenter performs ν_- on the unchanged original process. Nodes labeled with κ-ary Boolean functions instruct the experimenter to make κ copies of the

process, perform the appropriate experiment on each copy, and combine the results with B. Nodes labeled choose allow the experimenter to choose one of the children and perform that experiment.

In particular, an informally-defined wild duplicator experiment may be formalized as follows. Suppose that the wild duplicator experiment instructs us to wild duplicate the process, and if there are n copies perform wild-duplicator experiments e_{n1} through e_{nn}, and use $B_n : \mathbf{B}^n \to \mathbf{B}$ to determine the final outcome. The formal version of this consists of a node ν labeled choose with ω children. The n^{th} child is a node labeled by B_n, with its i^{th} child the translation of e_{ni}. That is, the number of copies that the wild duplicator makes is chosen; that many copies are made; the appropriate experiments are performed on the copies; and the results are combined. We write root(E) for the root node of the tree E.

Definition 5.3.3 *We define $P \mathbin{\Delta\!\!\!\!\!\!+} E$ (resp. $P \mathbin{\nabla\!\!\!\!\!\!+} E$), iff there is some partial function ζ from the nodes of E to pairs of truth values and processes, such that $\zeta(\text{root}(E)) = \langle \mathrm{tt}, P\rangle$, (resp. $\langle \mathrm{ff}, P\rangle$) and whenever $\zeta(\nu) = \langle b, R\rangle$ we have:*

- *If ν is labeled a, then either*
  - *There is some R' such that $R \xrightarrow{a} R'$ and $\zeta(\nu_+) = \langle b, R'\rangle$; or*
  - *$R \overset{a}{\not\to}$, and $\zeta(\nu_-) = \langle b, R\rangle$.*
- *ν is labeled by $B : \mathbf{B}^\kappa \to \mathbf{B}$, and there is some κ-length vector $\vec{b}$ of Booleans such that $B(\vec{b}) = b$ and the α^{th} child ν_α of ν has $\zeta(\nu_\alpha) = \langle b_\alpha, R\rangle$.*
- *ν is labeled* choose*, has κ children ν_α, and there is some $\beta < \kappa$ such that $\zeta(\nu_\beta) = \langle b, P\rangle$.*

We say that ζ demonstrates that $P \mathbin{\Delta\!\!\!\!\!\!+} E$ (resp. $P \mathbin{\nabla\!\!\!\!\!\!+} E$).

Intuitively, ζ assigns to each node a process and the success or failure of the experiment given by that node on that process; the consistency conditions chart the progress of the experiment. As before, it is possible that both $P \mathbin{\Delta\!\!\!\!\!\!+} E$ and $P \mathbin{\nabla\!\!\!\!\!\!+} E$; consider the wild duplicator experiment which succeeds precisely when the duplicator produces a prime number of copies. It is now possible to define experiments which can neither succeed nor fail on some processes; for example, the experiment consisting of repeatedly making two copies of P forever, without actually doing anything with any of the copies.

As usual, we define $P \leq_{\mathsf{Wild}} Q$ as for all experiments E, whenever $P \mathbin{\Delta\!\!\!\!\!\!+} E$ then $Q \mathbin{\Delta\!\!\!\!\!\!+} E$. Predictably, for each experiment E there is an experiment $\neg E$ such that $P \mathbin{\Delta\!\!\!\!\!\!+} \neg E$ iff $P \mathbin{\nabla\!\!\!\!\!\!+} E$ and vice versa; take $\neg E$ to be the experiment E with an extra root node labeled by the negation function. As with ordinary duplicator experiments, we lose no generality by considering only successes.

We will need to construct demonstrations ζ; the following lemma makes the construction easier.

Definition 5.3.4 *The function f is a* consistent choice function *for $P \sqsubseteq_{\rightarrow} P'$ if f is a function from the descendants of P to those of P', such that $f(P) = P'$ and for all $Q \xrightarrow{a} R$ descendants of P, the following holds:*

$$\begin{array}{ccc} Q & \sqsubseteq_{\rightarrow} & f(Q) \\ \downarrow a & & \downarrow a \\ R & \sqsubseteq_{\rightarrow} & f(R) \end{array}$$

Lemma 5.3.5 *Suppose that P and P' are arbitrarily-branching trees such that $P \sqsubseteq_{\rightarrow} P'$. Then there is a consistent choice function for $P \sqsubseteq_{\rightarrow} P'$.*

Proof:

We may non-constructively build such a function f by the Axiom of Choice.[2] Let $\leq$ be a well-ordering of the descendants of P'. Define a sequence of partial functions f_i, taking the i^{th} level of P (counting the root as level 1) to that of P', as follows. Let $f_1(P) = P'$. Suppose that R is on the n^{th} level of P, and $R \xrightarrow{a} S$. Let $f_{n+1}(S)$ be S', the $\leq$-first descendant of $f_n(R)$ such that $S \sqsubseteq_{\rightarrow} S'$; there is at least one such S' by the fact that $R \sqsubseteq_{\rightarrow} f(R)$. Let $f(T) = f_n(T)$ where T is on the n^{th} level of P. It is easy to see that f is a consistent choice function for $P \sqsubseteq_{\rightarrow} P'$. $\Box$

Theorem 5.3.6 *For all (arbitrarily branching) processes P and Q, $P \leq_{\mathsf{Wild}} Q$ iff $P \sqsubseteq_{\rightarrow} Q$*

Proof:

The proof that $P \leq_{\mathsf{Wild}} Q$ implies $P \sqsubseteq_{\rightarrow} Q$ is similar to that in Theorem 5.2.4. If $P \leq_{\mathsf{Dup}} Q$ and $P \xrightarrow{a} P'$, and for none of the Q'_α such that $Q \xrightarrow{a} Q'_\alpha$ does $P' \leq_{\mathsf{Wild}} Q'_\alpha$, then we construct an experiment E' (the conjunction of E_α witnessing $P' \not\leq_{\mathsf{Wild}} Q'_\alpha$) such that $P' \mathbin{\mathbb{A}} E'$ but not $Q'_\alpha \mathbin{\mathbb{A}} E'$ for any α. Let E be the experiment "push a, then do E'"; then $P \mathbin{\mathbb{A}} E$ but not $Q \mathbin{\mathbb{A}} E$.

For the converse, suppose that $P \sqsubseteq_{\rightarrow} Q$ and $P \mathbin{\mathbb{A}} E$. Let ζ be any function demonstrating that $P \mathbin{\mathbb{A}} E$. We will construct a function ζ' demonstrating that $Q \mathbin{\mathbb{A}} E$. Let f be a consistent choice function for $P \sqsubseteq_{\rightarrow} Q$. Define

$$\zeta'(\nu) = \begin{cases} \langle b, f(R)\rangle & \zeta(\nu) = \langle b, R\rangle \text{ and } R \in \text{descendants}(P) \\ \langle b, R\rangle & \zeta(\nu) = \langle b, R\rangle \text{ and } R \notin \text{descendants}(P) \\ \text{undefined} & \text{otherwise} \end{cases}$$

It is straightforward to check that ζ' demonstrates that $Q \mathbin{\mathbb{A}} E$. $\Box$

[2] Note that if P and P' are finitely branching trees, this may be done without the Axiom of Choice.

5.4 Global-Testing Experiments, Tree Isomorphism, and Bisimulation

To explain bisimulation by experiments, we must do more than merely *see* all the alternatives; we must *know that we have seen them all.* Milner, of course, is aware of this. In [Mil81], he postulates a mechanism for exploring all the alternatives available from a given state, by means of the experimenter varying some "ambient ('weather') conditions" which determine which nondeterministic choice the machine will take. Two different weather conditions might cause the machine to behave in the same way, and the experimenter cannot observe this.

For the sake of metaphor — most programmers cannot control the weather — we consider a formulation in terms of a *global-testing duplicator*. The global-testing duplicator is a device with a chamber and a chute. The experimenter places the machine in the chamber. Out of the chute drops one copy of each child P' of the process P, neatly tagged with the action a for which $P \xrightarrow{a} P'$.

It is straightforward to distinguish non-bisimilar processes using a global-testing duplicator as well as an ordinary one. If P and Q are not bisimilar, there is some Hennessy-Milner formula φ which distinguishes them, say $P \models \varphi$ and $Q \not\models \varphi$. We can construct an experiment e_φ from φ on which P can succeed, but Q never will. For instance, the experiment of $\langle a \rangle \varphi$ starts with pressing the a-button, and then performing e_φ on the resulting machine (or failing if the a-button cannot be pressed). The experiment for $[a]\varphi$ involves global-testing duplicating the machine under the a-button, and checking to see that every descendant passes e_φ.

One way to describe global-testing tests is by a program in a small nondeterministic programming language. Aside from ordinary programming constructs, we will have the following:

Data Types: The usual types, and processes and actions as first-class objects. In particular, we allow arrays and lists of processes, and functions to return processes.

`succeed` and `fail`**:** Primitives causing the whole experiment to succeed and fail respectively.

`CanPush`(P, a)**:** A function which returns true iff the a-button on the process P can be pushed.

`DoPush`(P, a)**:** A nondeterministic function — in the programming-language sense rather than the mathematical one — which returns some a-child of P, or causes the whole program to `fail` if there is no such child.

for each b**-child** R **of** P **do :** A looping construct which iterates its body once for each b-child of P, with R bound to that child, chosen in an

```
if CanPush(P, a) then
    Q := DoPush(P, a);
    x := tt
    for each b-child R of Q do
        x := x∧CanPush(R, c);
    if x then succeed else fail;
else
    fail;
```

Figure 5.1: Global-testing experiment distinguishing $P_\star$ and $Q_\star$

arbitrary order. In fact, both $\texttt{DoPush}(P, a)$ and $\texttt{CanPush}(P, a)$ can be implemented in terms of this construct.

The execution of such code with a duplicator and a global-testing duplicator is fairly straightforward. For example, to evaluate a $\texttt{CanPush}(P, a)$, make a copy of P using the ordinary duplicator, press the a-button on the copy, and then discard it, returning true or false as the button could or could not be pushed.

The typical experiment will look something like Figure 5.1, which is an experiment to distinguish between $P_\star$ and $Q_\star$. It corresponds to the modal formula $\langle a\rangle[b]\langle c\rangle\mathrm{tt}$; *viz.*, it checks to see if there is some a-child Q such that every b-child R of Q has a c-child. In fact, it is straightforward to build global-testing experiments which correspond precisely to Hennessy-Milner formulas.

Definition 5.4.1 *If P is a synchronization tree, then $P \restriction n$ is P truncated to depth n. Operationally, we define $P \restriction 0$ is stopped, and $P \restriction (n+1) \xrightarrow{a} P' \restriction n$ whenever $P \xrightarrow{a} P'$.*

For example, $aP \restriction 1 \equiv a$, and $P \restriction 0 \equiv \mathbf{0}$.

Theorem 5.4.2 *(Hennessy and Milner)*

- *$P \models \varphi$ iff P can succeed on e_φ.*
- *Therefore, global-testing equivalence implies bisimulation.*

However, we have gone too far; global-testing equivalence is strictly finer than bisimulation. The experimenter can simply count the black boxes that come out of the chute, or, in Milner's metaphor, count the number of varieties of weather available. In fact,

Theorem 5.4.3

- *P and Q are global-testing duplicator equivalent iff they are isomorphic as unordered trees.*
- *a and $a + a$ are bisimilar, but not global-testing duplicator equivalent.*

Proof:

Clearly, if $P \equiv Q$, then P and Q agree on all global-testing experiments. Conversely, suppose that P and Q are different trees. Then (because P and Q are finitely branching) for some integer n, P and Q truncated to depth n are different trees.

We construct a global-testing program $M(P, Q, n)$ which can return tt precisely if P and Q agree up to depth n (Figure 5.2).

procedure $M(P, Q :$ **process**, $n :$ **integer**$)$:**Boolean**
If $n = 0$ then return tt.
Otherwise,
  For each action a
    Let lp and lq be the lists of a-children of P and Q.
    If lp and lq are different lengths then return ff.
    Otherwise lp and lq are the same length;
      If $M(lp_i, lq_i, n-1)$ returns false for any $i \in [1 \ldots |lp|]$
      then return ff
  P and Q have passed the test for each a, so return tt.

Figure 5.2: Global-testing experiment $M(P, Q, n)$

Clearly, all trees agree to depth 0, and if P and Q have different numbers of children, they cannot agree to any depth $n > 0$. Suppose that P and Q agree to depth n; then there is some ordering of their children $P_1, \ldots, P_k$ and $Q_1, \ldots, Q_k$ such that P_i and Q_i agree up to depth $n-1$. By induction from these, if P and Q agree to depth n, then $M(P, Q, n)$ can return tt.

Conversely, suppose that P and Q do not agree to depth n. Then either they have different numbers of a-children for some a (in which case the algorithm returns ff), or for some a, every arrangement $P_1, \ldots, P_k$ and $Q_1, \ldots, Q_k$ of their a-children has some $P_i \not\equiv Q_i$. By induction, we conclude that $M(P, Q, n)$ must return ff.

So, if $P \not\equiv Q$, let $T = P \upharpoonright n$ (where P and Q disagree at depth n). Then $M(P, T, n)$ can return tt, and $M(Q, T, n)$ cannot; thus P and Q can be distinguished by global-testing experiments. □

We have used a good deal of programming power in our language: integers, recursion, and lists of processes. It is an open question how much of this is necessary for experimental equivalence to be tree isomorphism; *e.g.*,

suitable modifications of the program (as in Lemma 5.4.7) allow us to capture tree isomorphism without the use of lists. However, we can distinguish between a and $a + a$ with a single Boolean variable by counting the number of descendants mod 2.

5.4.1 Wild Global-Testing Duplicators

The reader will have realized that our global-testing duplicator does not precisely match Milner's weathers metaphor: two different forms of weather may cause a process to behave the same way. Milner's description does match the *wild global-testing duplicator*, which may produce *one or more* copies of each child of its input. We might hope that this blurs the distinctions between processes, preventing us from distinguishing between processes that differ only in the numbers of their children; but it does not.

Indeed, it is harder to tell processes apart in at least one sense: it may be that P can pass every test that some different Q can. For example, if $a + a$ can pass the wild global-testing experiment E, then so can a; each time E calls for a process to be wild global-testing duplicated and $a + a$ produces n children (all of which are stopped processes), then $n \geq 2$; it is possible for a to produce n children as well.

However, some more subtle counting experiments suffice to observe tree isomorphism.

Theorem 5.4.4

- *P and Q are wild global-testing duplicator equivalent iff they are isomorphic as unordered trees.*
- *a and $a + a$ are bisimilar, but not wild global-testing duplicator equivalent.*

Proof: The second point follows from the first; it may easily be seen independently. Consider the test E_1 which accepts a process P, uses the wild global duplicator on it, and succeeds precisely if the duplicator produces one a-child. Then a can pass E_1, but $a + a$ cannot.

To show the first point, we will first define a partial order $P \preceq Q$ on finite trees. Unlike most comparisons between processes we have considered, $\preceq$ is antisymmetric; $P \preceq Q$ and $Q \preceq P$ will imply $P \equiv Q$. We will construct experiments $E_{Q,n}$ such that P can pass $E_{Q,n}$ iff $P \upharpoonright n \preceq Q \upharpoonright n$. If P and Q are distinct synchronization trees, then for some n we have $P \upharpoonright n \not\equiv Q \upharpoonright n$, and so either $P \upharpoonright n \not\preceq Q \upharpoonright n$ or $Q \upharpoonright n \not\preceq P \upharpoonright n$. So, $E_{Q,n}$ or $E_{P,n}$ will distinguish P and Q, and the theorem will follow. The necessary mathematics will take up the rest of this section. $\Box$

First, our partial order on finite trees. The condition $P \preceq Q$ holds if, informally, Q can be obtained by repeated duplication of subtrees of P. Formally, this is defined by induction on the depth of finite trees:

Definition 5.4.5 $\mathbf{0} \preceq \mathbf{0}$, *and whenever*

$$P = \sum_{a \in \mathsf{Act}} \sum_{i=1}^{p_a} aP_{ai} \quad , \qquad Q = \sum_{a \in \mathsf{Act}} \sum_{i=1}^{q_a} aQ_{aj} \tag{5.1}$$

we have

1. *For each* Q_{aj} *there is some* P_{ai} *such that* $P_{ai} \preceq Q_{aj}$.
2. *There is a 1-1 function* $f : [1 \ldots p_a] \to [1 \ldots q_a]$ *such that* $P_{aj} \preceq Q_{a,f(j)}$.

That is, each child of Q has a "cousin" which is a child of P, and distinct children of P have distinct cousins in Q. It is easy to show that $\preceq$ is a preorder.

Lemma 5.4.6 $\preceq$ *is a partial order, and a congruence with respect to* $a(\cdot)$, $+$, *and* $\restriction n$.

Proof: Reflexitivity, transitivity, and congruence are straightforward. By a predictable induction on n, we show that it is antisymmetric; that is, if $P \preceq Q \preceq P$ then P and Q are isomorphic synchronization trees. This is trivial if $P = Q = \mathbf{0}$. Let P and Q be expressed as in (5.1), fix an action a and let $f : [1 \ldots p_a] \to [1 \ldots q_a]$ and $g : [1 \ldots q_a] \to [1 \ldots p_a]$ be the 1-1 functions showing that $P \preceq Q$ and $Q \preceq P$ respectively. The existence of 1-1 functions shows that $p_a \leq q_a \leq p_a$ and hence $p_a = q_a$. Therefore f and g are also onto, and so fg and gf are permutations. Recall that if h is a permutation of a finite set and i is an element of that set, then the set $\left\{i, h(i), h^{(2)}(i), \ldots\right\}$ is a finite set, called the *orbit* of i under h; as h is 1-1, we must have $h^{(k)}(i) = i$ for some k, called the *period* of i.

Fix i. We have

$$P_{ai} \preceq Q_{a,f(i)} \preceq P_{a,gf(i)} \preceq \cdots \preceq P_{a,(gf)^{(k)}(i)} = P_{a,i}$$

where k is the period of i. By transitivity, we have $Q_{a,f(i)} \preceq P_{a,i}$, and so by induction $P_{a,i} \equiv Q_{a,f(i)}$. With a little bit of work, this establishes a bijection between the children of P and those of Q as desired. $\Box$

We now define the experiments $E_{Q,n}$ such that P can pass $E_{Q,n}$ iff $P \restriction n \preceq Q \restriction n$. The experiment $E_{Q,0}$ always succeeds. To see if P can pass $E_{Q,n+1}$, wild-duplicate P under each action a, giving $P'_{a1}, \ldots, P'_{a,p'_a}$. If $p'_a \neq q_a$ then the experiment fails. If $p'_a = q_a$ then for each i, see if P'_{ai} passes $E_{Q_{ai},n}$. $E_{Q,n+1}$ succeeds if each $E_{Q_{ai},n}$ succeeds.[3]

[3]The main technical difference between $E_{Q,n}$ and $M(P,Q,n)$ is that Q is hard-wired into $E_{Q,n}$ and a formal argument of the procedure M. It is impossible to tell precisely how many a-children Q has using a wild global-testing duplicator, so we must put this information into the experiment $E_{Q,n}$.

Lemma 5.4.7 *P can pass $E_{Q,n}$ iff $P \restriction n \preceq Q \restriction n$.*

Proof: This clearly is true for $n = 0$. For greater n, suppose that P can pass $E_{Q,n}$. Let P and Q be given as in (5.1). Suppose that the wild global-testing duplicator produced $\left\langle P'_{aj} : a \in \mathsf{Act}, j \in [1 \ldots q_a] \right\rangle$, where each child of P appears in this listing at least once. We have $P \preceq \sum_{a,j} aP'_{aj}$. As each P'_{aj} passes $E_{Q_{aj},n-1}$ we have $P'_{aj} \restriction (n-1) \preceq Q_{aj} \restriction (n-1)$ by induction, and hence we have

$$P \restriction n \preceq \sum_{a,j} aP'_{aj} \restriction (n-1) \preceq \sum_{a,j} aQ_{aj} \restriction (n-1) = Q \restriction n$$

as desired. The other direction is similar. □

Lemma 5.4.8 *P and Q agree on all wild global-testing duplicator experiments iff $P \equiv Q$.*

Proof: Clearly the result of performing an experiment on a process depends only on its synchronization tree. Conversely, suppose that $P \not\equiv Q$. Then there is some n such that $P \restriction n \not\equiv Q \restriction n$. By antisymmetry, we have $P \restriction n \not\preceq Q \restriction n$ or $Q \restriction n \not\preceq P \restriction n$; suppose the former. Then by Lemma 5.4.7, we know that P cannot pass $E_{Q,n}$ but Q can. Hence P and Q are distinguishable with a wild global-testing experiment. □

5.5 Conclusions

In fact, the global-testing duplicator experiments are as bad as possible. To avoid distinguishing bisimilar processes, it is necessary to restrict the kinds of experiments: we cannot pay attention to the number of duplicates produced. The only allowable way to use the global-testing duplicator is to perform the same experiment on all each of its a-children, and take the conjunction or disjunction of the answers. We may call these *modal* global-testing experiments.

Theorem 5.5.1 *(Hennessy and Milner) Two processes are bisimilar iff they agree on all modal global-testing duplicator experiments.*

This modal restriction strikes us as contrived. In previous kinds of machine, we did not have to restrict the use of experimental outcomes. The global-testing duplicator reveals too much, and we must agree — for the purpose of observing bisimulation — to ignore something that we could easily pay attention to. An experimenter with a global-testing duplicator and no preconceptions would probably not be lead to invent the notion of bisimulation; instead, he would invent a notion which is equivalent to tree isomorphism.

Part III

Metatheory

Chapter 6

Finite and Regular Processes

6.1 Overview

Any worthwhile equivalence relation on programs will be undecidable, and ready simulation and bisimulation are no exceptions. However, there are some special cases which do admit decision procedures. In this chapter, we examine their existence and feasibility. The main results are a complete inequational axiom system for ready simulation on finite synchronization trees, and a polynomial time decision procedure for ready simulation on finite automata (*viz.*, regular trees).

These questions are of possible practical as well as theoretical interest. The decision procedure for bisimulation of automata has been implemented on the Concurrency Workbench, and used to analyze restricted cases of nontrivial algorithms [Wal88, Par87c]. The algorithms for bisimulation and ready simulation of finite automata are polynomial in the size of the automaton, which may be exponential in the size of the program; this limits the applicability of these algorithms to rather small problems. There are no reports of attempts to apply theorem provers to the complete axiom systems for bisimulation or ready simulation.

6.2 Complete Axiomatization for Finite Processes

Recall that the operations $\mathbf{0}$, $a(\cdot)$, and $+$ suffice to define all finite synchronization trees. For the remainder of this section, process terms involve only these operations. In this section, we present a complete axiom system for ready simulation on finite processes. We present it as a set of inequational axioms. Using the fact that $P \sqsubseteq_{\rightarrow} Q$ iff $aP + aQ \rightleftharpoons aQ$, it is possible to phrase the axioms in equational terms; however, they become conditional

equations and the conditions are unappealing. For example, the quite natural inequational axiom

$$\frac{P \leq Q}{bP \leq bQ}$$

becomes the rather odd equational axiom

$$\frac{aP + aQ = aQ}{baP + baQ = baQ.}$$

Definition 6.2.1 *We define the axiom system* RS *as follows:*

$$\begin{aligned} P + \mathbf{0} &= P && (6.1)\\ P + Q &= Q + P && (6.2)\\ P + (Q + R) &= (P + Q) + R && (6.3)\\ P + P &= P && (6.4)\\ aP &\leq aP + aQ && (6.5) \end{aligned}$$

We also have the usual axioms for inequational axioms systems, stating that $\leq$ is reflexive, antisymmetric, and transitive, and that all operations are monotone. We write $\mathtt{RS} \vdash P \leq Q$ when $P \leq Q$ is provable from RS, and use similar notation for equations. Notice that (6.1)-(6.4) are precisely the complete axiomatization of bisimulation on finite processes.

Theorem 6.2.2 (Soundness of RS) RS *is sound;* viz., *if* $\mathtt{RS} \vdash P \leq Q$ *then* $P \sqsubseteq_{\rightarrow} Q$.

Proof: Each of the rules of RS is sound. For example, to show that $\sqsubseteq_{\rightarrow}$ is transitive, it is routine to show that the transitive closure $\sqsubseteq_{\rightarrow}^*$ of $\sqsubseteq_{\rightarrow}$ is also a ready simulation relation. So, if $P \sqsubseteq_{\rightarrow} Q \sqsubseteq_{\rightarrow} R$, then $P \sqsubseteq_{\rightarrow}^* R$ and hence $P \sqsubseteq_{\rightarrow} R$. The fact that all operations are monotone can be shown directly; it also follows from the fact that prefixing and summation are GSOS operations. □

By commutativity and associativity we may ignore the order and grouping of summations. We say that P is in *tree form* if it is $\mathbf{0}$, or of the form $\sum a_i P_i$ where each P_i is in tree form.

Theorem 6.2.3 *For any term* P*, there is a term* Q *in tree form such that* $\mathtt{RS} \vdash P = Q$ *and* $P \equiv Q$.

Proof: We induct on the structure of P. In this section, $=$ denotes syntactic equality modulo commutativity and associativity of $+$. If $P = \mathbf{0}$ then P is in tree form. If $P = aP'$, let Q' be the term given inductively for P'; then $\mathtt{RS} \vdash P' = Q'$ and so $\mathtt{RS} \vdash aP' = aQ'$, *viz.*, $\mathtt{RS} \vdash P = Q$. If P is a sum $P_1 + P_2$, then let Q_1 and Q_2 be the terms given inductively for P_1 and P_2. We have four cases, depending on which of Q_1 and Q_2 are $\mathbf{0}$.

- If $Q_1 = Q_2 = \mathbf{0}$, then let $Q = \mathbf{0}$. Then

$$\mathtt{RS} \vdash P = P_1 + P_2 = Q_1 + Q_2 = \mathbf{0} + \mathbf{0} = \mathbf{0} = Q.$$

- If $Q_1 = \mathbf{0}$ and Q_2 is a non-$\mathbf{0}$ tree form, then let $Q = Q_2$. By (6.1),

$$\mathtt{RS} \vdash P = P_1 + P_2 = \mathbf{0} + Q_2 = \mathbf{0} = Q.$$

- The reverse case is similar.

- If neither Q_1 or Q_2 is $\mathbf{0}$, then let $Q = Q_1 + Q_2$. Q is in tree form, and clearly $\mathtt{RS} \vdash P = Q$ as desired.

It is straightforward to show that $P \equiv Q$ in each case. □

Theorem 6.2.4 *If $P \equiv Q$, then* $\mathtt{RS} \vdash P = Q$.

Proof: Induction on the depth of trees. Let P' and Q' be terms in tree form provably equal to P and Q. In the base case, $P' = Q' = \mathbf{0}$. Otherwise, there is a 1-1 correspondence between the children of P and those of Q. So, by permuting sums, we have $\mathtt{RS} \vdash P' = \sum_{i=1}^{n} a_i P_i'$ and similarly for Q', where $P_i' \equiv Q_i'$. By induction, $\mathtt{RS} \vdash P_i = Q_i$. Combining these with the tree-form equations for P and Q, we obtain $\mathtt{RS} \vdash P = P' = Q' = Q$. □

Theorem 6.2.5 (Completeness of RS) RS *is complete for finite trees;* viz., *if $P \sqsubseteq_{\rightarrow} Q$ then* $\mathtt{RS} \vdash P \leq Q$.

Proof: We induct on the depth of trees. The basis, depth 0, is trivial. Suppose that the theorem holds for all trees of depth n. Let $P \sqsubseteq_{\rightarrow} Q$ be two terms denoting trees of depth at most $n+1$; without loss of generality, suppose P and Q are in tree form. That is, $P = \sum_{i=1}^{m} a_i P_i$ and $Q = \sum_{j=1}^{n} b_j Q_j$. The children of P are precisely the P_i; so, for each $i \in [1 \ldots m]$, there is a j such that $Q \xrightarrow{b_j} Q_j$, $P_i \sqsubseteq_{\rightarrow} Q_j$, and $a_i = b_j$. Let J be the set of all such j's. By induction, $\mathtt{RS} \vdash P_i \leq Q_j$. By monotonicity, $\mathtt{RS} \vdash a_i P_i \leq b_j Q_j$, and hence

$$\mathtt{RS} \vdash P = \sum_{i=1}^{m} a_i P_i \leq \sum_{j \in J} b_j Q_j$$

Not all summands of Q need appear in the sum on the left; we must add them in.

We will show

$$\sum_{j \in J} b_j Q_j \leq \sum_{j=1}^{n} b_j Q_j \tag{6.6}$$

which suffices to prove the theorem. We will use $\mathtt{RS} \vdash b_j P_j \leq b_j P_j + b_{j'} P_{j'}$ when $b_j = b_{j'}$, an instance of an (6.5) We must make sure that for each j', we can find a j such that $b_j = b_{j'}$. The contrapositive of the second clause of the definition of ready simulation states that if $Q \stackrel{b_{j'}}{\rightarrow}$, then $P \stackrel{b_{j'}}{\rightarrow}$. That is, there is some $a_i = b_{j'}$, and hence some $j \in J$ such that $b_j = b_{j'}$. So, we have $\mathtt{RS} \vdash b_j Q_j \leq b_j Q_j + b_{j'} Q_{j'}$ by (6.5). Summing this inequation for all j', and using (6.4) to eliminate duplicates, we may prove (6.6).

We have shown

$$\mathtt{RS} \vdash P = \sum_{i=1}^{m} P_i \;\leq\; \sum_{j \in J} b_j Q_j \;\leq\; \sum_{j=1}^{n} b_j Q_j = Q$$

as desired. □

6.3 Regular Processes

Another sort of finitely-describable process is the *regular process*; *viz.*, one which may be described as (approximately) a finite automaton. The main result of this section is that it is possible to decide if two regular processes are ready similar in polynomial time. It is known [PT87] that deciding if regular processes are bisimilar can be done in $O(n \lg n)$ time; Milner [Mil89] has given a finite complete axiomatization of this problem. By contrast, many variants and finite approximations of bisimulation are NP-hard or PSPACE-hard [KS90], so a polynomial-time algorithm for ready simulation is hardly a given. The algorithm we present works by brute force and massive ignorance; its n^6 running time (where n is the number of points in the labeled transition system) can probably be improved.[1]

Definition 6.3.1 *A* regular process *is the synchronization tree of a finite labeled transition system.*

Any finite process is regular, and some infinite processes (such as a^ω) are as well. There is a good syntactic characterization of the regular processes as those which can be defined with the operations $\mathbf{0}$, $+$, $a(\cdot)$, and fixed points.

The following algorithm takes as input a labeled transition system $\langle \mathcal{P}, \mathsf{Act}, \dot{\rightarrow} \rangle$, and returns the set of ready simulation approximations in $\mathcal{P}$; that is, the set of pairs $\langle P, Q \rangle \in \mathcal{P}^2$ such that $P \sqsubseteq_{\rightarrow} Q$. This can be used to decide if two regular processes P_0 and Q_0 are ready similar; generate the labeled transition system of the descendants of P_0 and Q_0, run the algorithm, and see if $\langle P_0, Q_0 \rangle$ and $\langle Q_0, P_0 \rangle$ are both in the output.

[1] If the labeled transition system is presented by an adjacency matrix, the input to the algorithm is size n^2; so the algorithm is cubic in the size of the input. It seems likely that most uses of the algorithm would be on sparse processes; the matrix representation is the wrong one to use, and the running time of the algorithm is n^6.

The input representation is as follows. The processes are irrelevant; we will call them $P_1, \ldots, P_n$. Similarly, the identities of the actions will be called $a_1, \ldots, a_m$. We will represent relations between processes as $n \times n$ Boolean matrices, both in input and internally. The transition relation is presented as an $n \times m \times n$ array.

The algorithm maintains a superset of the desired relation in the $n \times n$ array A. A is initialized to contain all pairs of processes with the same ready sets; as $P \sqsubseteq_{\rightarrow} Q$ implies readies(P) = readies(Q), we know that $P \sqsubseteq_{\rightarrow} Q$ implies $A[P, Q] = \mathtt{tt}$ initially. We repeatedly find a place which violates the requirement of ready simulation and remove it, until there is nothing left to remove. The code is given in Figure 6.1.

Showing that `ready-sim` terminates is easy. There is only one unbounded loop (and no recursion). A_k differs from A_{k-1} by having at least one element changed from tt to ff on all but the last pass through the loop. The loop stops when no elements have changed. No element can change more than once, so the loop must terminate.

To show correctness, we first show that the algorithm always returns some ready simulation relation, and then that if $P_i \sqsubseteq_{\rightarrow} P_j$, then $A[P_i, P_j]$ is never set to ff. First, we show that `ready-sim-fails` behaves right. It is convenient to refer to the A_k's generally as A, and say that (*e.g.*) $A[P_i, P_j]$ changes from tt to ff rather than $A_{k-1}[P_i, P_j] = \mathtt{tt}$ and $A_k[P_i, P_j] = \mathtt{ff}$.

Lemma 6.3.2 *For each A_k in a call to* `ready-sim`$(n, m, \dot{\rightarrow})$, *the following are equivalent:*

- `ready-sim-fails`(P_i, P_j, A_k) *returns* tt*;*
- *Either* readies$(P_i) \neq$ readies(P_j) *or for some P_i' such that $P_i \stackrel{a}{\rightarrow} P_i'$ for some a, there is no P_j' such that $P_j \stackrel{a}{\rightarrow} P_j'$ and $A[P_i', P_j'] = \mathtt{tt}$.*

Proof: $A_0[P_i, P_j]$ is initialized to tt just if readies(P_i) = readies(P_j); elements of A are switched from tt to ff but never from ff to tt. The function `ready-sim-fails` searches exhaustively for an instance of the second condition, returning tt iff it finds one. □

Lemma 6.3.3 *The procedure* `ready-sim` *always returns a ready simulation relation.*

Proof: Let A be the matrix returned by `ready-sim`. Suppose that $A[P_i, P_j] = \mathtt{tt}$. There were no changes to the A in the last pass through the **do-until** loop, and so we know that the call `ready-sim-fails`(P_i, P_j, A) returns false for each P_i and P_j. In particular, whenever $A[P_i, P_j] = \mathtt{tt}$, then

- readies(P_i) = readies(P_j) and

```
function ready-sim(n, m, →̇) : array[1...n,1...n] of Boolean
    % The processes are P_1, ..., P_n;
    % The actions are a_1, ..., a_m;;
    % →̇ [i, r, j] is tt iff P_i -a_r-> P_j.
    k := 0
    for each P_i, P_j do
        A_0[P_i, P_j] := if readies(P_i) = readies(P_j) then tt else ff
    do
        for each P_i, P_j do
            k := k + 1
            if ready-sim-fails(P_i, P_j, A_{k-1}) then
                A_k[P_i, P_j] := ff
            else A_k[P_i, P_j] := A_{k-1}[P_i, P_j]
            end if
    until A_{k-1} = A_k
    return A_k



    function ready-sim-fails(P_i, P_j, A) : Boolean
        if A[P_i, P_j] = ff then return ff
        for each a_r do
            for each P_i' such that P_i -a_r-> P_i' do
                found-approx := ff
                for each P_j' such that P_j -a-> P_j' do
                    if A[P_i', P_j'] then found-approx := tt
                if not found-approx then return  tt
        end for each a
        return ff
```

We omit the (easy) code for readies(P).

Figure 6.1: Code for ready simulation algorithm

- Whenever $P_i \stackrel{a}{\rightarrow} P_i'$ there is some P_j' such that $P_j \stackrel{a}{\rightarrow} P_j'$ and $A[P_i', P_j'] =$ tt.

This is precisely the condition that A is (the matrix representation of) a ready simulation relation. □

Lemma 6.3.4 *Let A be the matrix returned by* `ready-sim`$(n, m, \stackrel{a}{\rightarrow})$. *If $P_i \sqsubseteq_{\rightarrow} P_j$ then $A[P_i, P_j] =$* tt.

Proof: Suppose that there was some pair of processes $P_i \sqsubseteq_{\rightarrow} P_j$ and pass k such that $A_k[P_i, P_j] =$ ff. Let P_i and P_j be the first such pair to be eliminated (*viz.*, have $A_k[P_i, P_j]$ set to false). This happened on stage k; in particular,

$$\text{If } P_r \sqsubseteq_{\rightarrow} P_s, \text{ then } A_{k-1}[P_r, P_s] = \text{tt}. \tag{6.7}$$

The only way that $A_k[P_i, P_j]$ will be set to false is if `ready-sim-fails`(P_i, P_j, A_{k-1}) returns tt. That is, there must be some action a and process P_i' such that $P_i \stackrel{a}{\rightarrow} P_i'$, and for no P_j' such that $P_j \stackrel{a}{\rightarrow} P_j'$ does $A_{k-1}[P_i', P_j'] =$ tt. However, we know that there is a P_j' such that $P_i' \sqsubseteq_{\rightarrow} P_j'$. By (6.7), we know that $A_{k-1}[P_i', P_j'] =$ tt. The call `ready-sim-fails`(P_i, P_j, A_k) must have returned ff, a contradiction proving the claim. □

We may now conclude:

Theorem 6.3.5 *If* `ready-sim`$(n, m, \dot{\rightarrow})$ *returns A, then*

$$A[P_i, P_j] = \begin{cases} \text{tt} & P_i \sqsubseteq_{\rightarrow} P_j \\ \text{ff} & \textit{otherwise} \end{cases}$$

Lemma 6.3.6 *The running time of* `ready-sim`$(n, m, \dot{\rightarrow})$ *is $O(mn^6)$; viz., cubic in the size of the input.*

Proof: Each call to `ready-sim-fails` may loop over all m actions, and all n^2 pairs of processes, taking $O(mn^2)$ time. Each pass through the **do-until** loop of `ready-sim` calls `ready-sim-fails` once for each of the n^2 pairs of processes, for $O(mn^4)$ time. At least one pair $\langle P_i, P_j \rangle$ is eliminated in each but the last pass through the loop; so there are at most n^2 passes, for a time of $\mathrm{O}(mn^6)$ as claimed. The representation of $\dot{\rightarrow}$ is an $n \times m \times n$ matrix, making the input size $O(mn^2)$ and giving the algorithm cubic running time. □

The claim of cubic running time is, of course, cheating. It seems likely that processes appearing in practice have quite sparse graphs, with most nodes having one or two children. The natural representation for such processes is an adjacency-list representation, of side $O(n)$ for an n-state process rather than the $O(n^2)$ one presented here. In this representation, this algorithm is $O(n^6)$, which is undesirably large. Reducing this running time remains an open problem.

Chapter 7

Metatheory of GSOS Languages

7.1 Overview

As we have seen, the mere fact that a language is defined by GSOS rules is enough to imply that it satisfies certain essential mathematical properties. In this chapter, we explore the further properties which the GSOS format implies. Perhaps more importantly, they give some hints about the theoretical and analytic power available from well-structured rules. The ability to prove theorems of this sort is another argument that the class of GSOS languages is a useful class of languages.

In the first section, we give an upper bound on the branching of a pure GSOS term (*viz.*, one without synchronization tree constants), and as a corollary exhibit synchronization trees which cannot be achieved by any GSOS terms. In the second section, we show how long it takes for GSOS contexts to distinguish between non-ready-similar synchronization trees. In the third section, we show that negative rules do add to the expressive power of GSOS languages, although they need not add to the discriminatory power.

7.2 Size of Branching

In most of this work, we have ignored the issues of expressive power: which processes, and which operations on processes, are definable in a GSOS language without tree constants. Clearly, we can only hope to produce computably finitely branching trees; can we produce all such trees? In this section, we show that we cannot. If $P \stackrel{a}{\rightarrow} Q$, we bound the size of the term Q in terms of the size of P. Using this, we construct a tree which cannot be generated by any term P.

To make matters more exciting, we will allow guarded recursion in this section. We eliminated it because it did not add to the discriminatory

power of the language; in this section, we are studying the expressive power, which recursion certainly increases. However, we do not consider unguarded recursion; for one thing, the results in this section are false for unguarded recursion, as processes may branch infinitely.

Definition 7.2.1 *The variable X is guarded in the term P if each free occurrence of X in P is inside a term of the form $a(P')$. A term is* well-guarded *or simply* guarded *if, in each subterm of the form* fix $\left[X_i; \vec{X} \Leftarrow \vec{P}\right]$, *each X_i is guarded in each P_j.*

It is a standard result that all guarded terms are finitely branching.

Definition 7.2.2 *Let $|P|$ be the size of P, defined as:*

$$|X| = 1 \tag{7.1}$$

$$|\mathrm{op}(P_1, \ldots, P_n)| = 1 + \sum_{i=1}^{n} |P_i| \tag{7.2}$$

$$|\mathrm{fix}\,[X \Leftarrow P]| = 1 + |P| \tag{7.3}$$

Let

$$r = 1 + \max_{\rho \in \mathcal{G}} \left|C_\rho\right| \tag{7.4}$$

where $\mathrm{op}_\rho(\vec{X}) \stackrel{a_\rho}{\to} C_\rho$ is the consequent of the rule ρ.

First, we consider terms without recursion.

Theorem 7.2.3 *Let P be a fix-free closed term of $\mathcal{G}$, such that $P \stackrel{a}{\to} Q$. Then $|Q| < r^{|P|}$.*

Proof: Induction on $|P|$. The base case is for P a nullary operator. If $P \stackrel{a}{\to} Q$, then this must be an axiom of $\mathcal{G}$; whence Q is one of the C_ρ. Thus $|Q| < r = r^{|P|}$.

Now suppose that the lemma holds for all terms of size less than n, and $P = \alpha(P_1, \ldots, P_k) \stackrel{a}{\to} C[\vec{P}, \vec{Q}]$ where $|P| = n$, and $P_i \stackrel{a_{ij}}{\to} Q_{ij}$ for each i, j.[1] We have $|P_i| \leq |P| - 1$ for each P_i, and inductively $\left|Q_{ij}\right| < r^{|P_i|} \leq r^{|P|-1}$. Let $k = \left|C[\vec{X}, \vec{Y}]\right|$; we know $k < r$ by the definition of r. There are at most k symbols in $C[\vec{X}, \vec{Y}]$, and hence at most k occurrences of P_i's and Q_{ij}'s in $C[\vec{P}, \vec{Q}]$. Let l be the length of the longest of these. Either for some i, $l = |P_i| < |P| \leq r^{|P|-1}$ or for some i, j, $l = \left|Q_{ij}\right| < r^{|P|-1}$. Then $|Q| = \left|C[\vec{P}, \vec{Q}]\right| < k \cdot l \leq k \cdot r^{|P|-1} \leq r \cdot r^{|P|-1} = r^{|P|}$ as desired. □

In fact, there is an exponential lower bound. Define the operation $\cdot^2$ by the rule:

$$\frac{X \stackrel{a}{\to} Y}{X^2 \stackrel{a}{\to} Y|Y}$$

[1] The negative rules are irrelevant for this theorem.

Let $P_0 = a$, and $P_{n+1} = P_n^2$. Each P_n is monogenic; let Q_n be its unique child. We calculate $Q_0 = \mathbf{0}$, and $Q_{n+1} = Q_n|Q_n$, and discover that indeed $|Q_n| = 2^{n+1} - 1$.

The situation with fixed points is more exciting; we do not have a uniform upper bound. If R is a subterm of P, we say that R is *loose* in P if R does not occur inside a subterm of the form $a(C[R])$; otherwise it is *penned*. For example, in $P = \mathrm{fix}\,[X \Leftarrow a(R_1) + R_2]$, the subterms R_2 and P itself are loose, while R_1 is penned. We define $/P/$ to be the number of *loose fixes* in P; that is, the number of loose subterms of P of the form $\mathrm{fix}\,[X \Leftarrow R]$. Formally,

$$\begin{array}{rcl} /X/ & = & 0 \\ /aS/ & = & 0 \\ /\mathrm{op}(\vec{S})/ & = & \sum /S_i/ \\ /\mathrm{fix}\,[X \Leftarrow S]/ & = & 1 + /S/ \end{array} \tag{7.5}$$

The following lemma follows by trivial induction on the structure of S, and says that when X is guarded in S and we substitute something for X, we neither add nor delete loose fixes.

Lemma 7.2.4 *If X is guarded in S and T is any term, then $/S[X := T]/ = /S/$.*

We are now ready to derive a bound on the size of children of terms involving recursion.

Theorem 7.2.5 *Define $f(p, l, r) = r^{p^{2^l}}$. Let P be a closed guarded term, and $p = |P|$ and $l = /P/$. Suppose $P \xrightarrow{a} Q$. Then $|Q| < f(p, l, r)$. In particular, $|Q| < f(p, p, r)$.*

Proof: The following facts are routine calculations with exponents; f was chosen to make them true:

$$\begin{array}{rcl} p & \leq & f(p, l, r) \\ r \cdot f(p-1, l, r) & \leq & f(p, l, r) \\ f(p^2, l-1, r) & \leq & f(p, l, r) \end{array}$$

The proof is by induction on l and p in that order. Suppose that the theorem holds for all terms with either smaller l than P, or the same l as P but smaller size. There are three cases:

1. $P = aP'$. In this case, $Q = P'$ and $|Q| = |P| - 1 < p \leq f(p, l, r)$.
2. $P = \mathrm{op}(\vec{P})$. The reasoning in this case is similar to that of Theorem 7.2.3. $Q = C[\vec{P}, \vec{Q}]$; there are fewer than r free variables in the context $C[\vec{X}, \vec{Y}]$, and each P_i and Q_{ij} is shorter than $f(p-1, l, r)$. We thus have $|Q| < r \cdot f(p-1, l, r) \leq f(p, l, r)$.

3. $P = \mathrm{fix}\,[X \Leftarrow P']$. Then $P \xrightarrow{a} Q$ iff $P'[X := P] \xrightarrow{a} Q$. By Lemma 7.2.4, we have $/P'[X := P]/ = l - 1 < l$. There are fewer than $\left|P'\right| < p$ free occurrences of the variable X in P', and each of them is replaced by a term of size $|P| = p$, so $\left|P'[X := P]\right| < p^2$. By induction, we have $|Q| < f(p^2, l-1, r) \leq f(p, l, r)$.

□

We conjecture that this bound is not tight.

Now, an easy diagonalization will show that there are computably branching trees which cannot be realized in any pure GSOS language (that is, one without tree constants), even with recursion. We define $T_/$ as follows. Say that the root of a tree is at level 1, and a child of a level-n node is at level $n+1$.

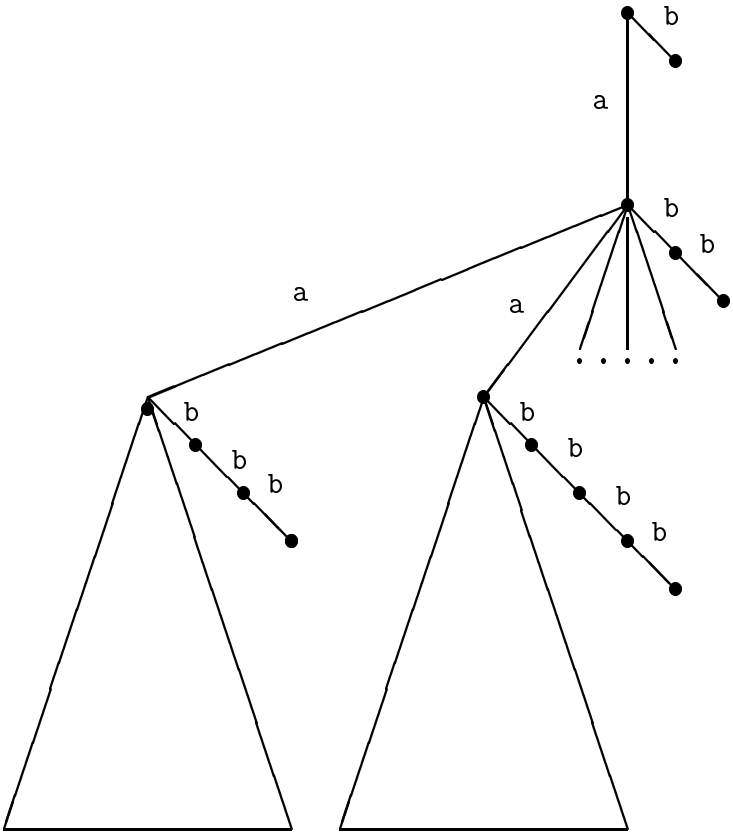


Figure 7.1: The top of $T_/$

Theorem 7.2.6 *There is a tree $T_/$ which is not bisimilar to any term P over any pure GSOS language $\mathcal{G}$ with guarded recursion. In particular, there is no such P such that $P \equiv T_/$.*

Proof:

We will bound the number of non-bisimilar descendants of a term P in terms of r, $|P|$, and the number of operations in $\mathcal{G}$. We will construct $T_/$ by diagonalizing over this bound, so that for any fixed $\mathcal{G}$ and P, $T_/$ will have too many non-bisimilar descendants at some level n.

Suppose

$$P_0 \stackrel{a}{\to} P_1 \stackrel{a}{\to} \cdots \stackrel{a}{\to} P_m$$

We have, for each P_i, $|P_{i+1}| < f(|P_i|, |P_i|, r)$. Let $f^{(1)}(p,r) = f(p,p,r)$ and $f^{(n+1)}(p,r) = f(f^n(p,r), f^n(p,r), r)$. We therefore have $|P_m| < f^{(m)}(|P_0|, r)$.

We now bound the number of $\mathcal{G}$-terms of size $< k$ for an arbitrary integer k. Suppose that there are g operations in $\mathcal{G}$. We have represented $\mathcal{G}$-terms as trees; however, by the fact that each operation has a fixed arity, we may map terms 1-1 into strings of operations in Polish prefix form. So, there are fewer than g^{k+1} $\mathcal{G}$-terms with size at most k.

If $P \stackrel{a^m}{\to} Q$ and $p = |P|$, then $|Q| < f^{(m)}(p,r)$; hence there are fewer than $g^{f^{(m)}(p,r)}$ distinct a^m-descendants of P. Now define the diagonal function $f_/(n) = 1 + n^{1+f^{(n)}(n,n)}$. Clearly, if $n = \max(m,g,p,r)$, there are fewer than $f_/(n)$ distinct a^m-descendants of P.

Let T_* be the tree in which each node at level n has $f_/(n)$ a-children. Let $v_1, v_2, \ldots$ be some recursive enumeration of the nodes of T_* level by level. We may now define $T_/$ to be T_* with the node v_n given a subtree of the form b^n. The top portion of this tree is sketched in Figure 7.1. Note that all the nodes of $T_/$ which are a^*-descendants of the root are tagged with different numbers of b's, and hence are distinct up to bisimulation.

Suppose that P were a term in a pure GSOS language $\mathcal{G}$ with g operations, such that $P \leftrightarrows T_/$. Let r be defined by (7.4), and $n = \max(r, g, |P|)+1$. Choose a term P' such that $P \stackrel{a^n}{\to} P'$, and let T' be any node on the n^{th} level of $T_/$ such that $T' \leftrightarrows P'$. Then T' has $f_/(n)$ a-children distinct up to bisimulation, and hence P' has at least that many. However, by Theorem 7.2.5, we know that P' has fewer than $g^{f^{(n)}(|P|,r)} < f_/(n)$ classes of distinct, and hence non-bisimilar, a-children. This is impossible; hence there is no such P.

□

7.3 Single-Bit Observations

We have taken traces of arbitrary length as our primitive observations. Experiments on processes have a moderately elaborate structure: they consist of running a process in a context and watching to see if a particular trace is produced. It would be slightly cleaner to have the result of an experiment be a single bit: *e.g.*, an experiment could consist simply of a context; P

succeeds on the experiment $C[X]$ iff $C[P] \stackrel{t}{\rightarrow}$, where we are using t as a distinguished success action.

In many settings, it suffices to observe bits. For example, consider observing Turing machines computing partial recursive functions. The basic experiment is a Turing-machine context $C[X]$, and a possible output n; the observation is whether or not $C[M]$ halts with n on the output tape. In fact, given such a $C[X]$ and n, it is routine to build a context $C'_n[X]$ such that $C'_n[M]$ halts with the first bit of its tape set to 1 iff $C[M]$ halts with n. That is, the complex observation of the output n can be reduced to observing the first bit of the tape. This is due, of course, to the programming power available in Turing-machine contexts.

In any GSOS language — CCS and CSP in particular — this sort of experiment does not suffice to observe ready simulation, or any of the other important equivalence relations. Intuitively, only one step of lookahead is possible in the rules. In particular, there is no context $C[X]$ such that $C[aa]$ and $C[ab]$ are distinct after one step. This is necessary to avoid countable branching, as shown by our counterexamples. It may be considered a weakness in the defining power of GSOS languages, but it is forced by our other desired characteristics.

Theorem 7.3.1 *For any process P and context $C[X]$ over a GSOS language, $C[P] \upharpoonright n \equiv C[P \upharpoonright n] \upharpoonright n$.*

Proof: The induction hypothesis (for of course this proof is by induction on n) is this. Let $\vec{P}$ be a vector of synchronization trees, and $\vec{m}$ a vector of integers with $n \leq m_i$ for each i; let $P'_i = P_i \upharpoonright m_i$. Then $C[\vec{P}] \upharpoonright n \equiv C[\vec{P'}] \upharpoonright n$. Clearly this holds when $n = 0$; suppose, then, that $n > 0$ and the hypothesis holds for all numbers below n.

Consider any transition $C[\vec{P}] \stackrel{a}{\rightarrow} R$. By Theorem 4.2.6, there is some ruloid ρ

$$\frac{X_i \stackrel{a_{ij}}{\rightarrow} Y_{ij}, \quad X_i \stackrel{b_{ik}}{\not\rightarrow}}{C[\vec{X}] \stackrel{a}{\rightarrow} D[\vec{X}, \vec{Y}]}$$

and vector of processes $\vec{Q}$ such that $\vec{P}$ and $\vec{Q}$ satisfies the antecedents of ρ and $R = D[\vec{P}, \vec{Q}]$. Let $Q'_{ij} = Q_{ij} \upharpoonright (m_i - 1)$. As $m_i \geq n > 0$, we have $P'_i \stackrel{a_{ij}}{\rightarrow} Q'_{ij}$, and $P'_i \stackrel{b_{ik}}{\not\rightarrow}$. Hence ρ applies to $C[\vec{P'}]$ as well, and we have $C[\vec{P'}] \stackrel{a}{\rightarrow} D[\vec{P'}, \vec{Q'}]$. As $C[\vec{P}] \upharpoonright n \stackrel{a}{\rightarrow} D[\vec{P}, \vec{Q}] \upharpoonright (n-1)$, and by induction

$$D[\vec{P}, \vec{Q}] \upharpoonright (n-1) \equiv D[\vec{P'}, \vec{Q'}] \upharpoonright (n-1)$$

we have shown that each child of $C[\vec{P}]$ corresponds to some child of $C[\vec{P'}]$. Similar methods show the reverse. □

For example, we have $aa \upharpoonright 1 = a = ab \upharpoonright 1$, and hence $C[aa] \upharpoonright 1 = C[ab] \upharpoonright 1$ for all contexts $C[X]$. Thus these quite distinct processes cannot be distinguished by single-action observations.

7.4 Expressive Power of Negative Rules

Negative rules are a peculiar feature of GSOS languages. In many formal settings, such as proof theory, they are dangerous; they may lead to inconsistencies or non-monotone properties. GSOS languages keep tight rein on negative rules, by making sure that the behavior of a term depends only on the behavior of smaller terms. However, it is hard to see how negative rules should be interpreted in the case of silent actions, and not necessarily easy to see how to implement them in any useful sense.

It is therefore of some interest to see how powerful a language can be crafted without negative rules. As we have seen, the full *discriminatory* power of GSOS languages can be achieved without negation. Glancing at the proofs in Section 7.2, we see that negative rules played no particular part in the branching bound on process expressions; in particular, it is still possible to generate exponentially bushy trees with positive rules. It is an open problem (of greater difficulty than importance) whether there are trees which require the full power of GSOS rules to define.

There is one aspect of expressive power for which negative rules are certainly necessary, *viz.* the ability to define operations on trees. Some operations, such as true sequencing, require negative rules.

Definition 7.4.1 *If P and Q are synchronization trees, the* true sequencing *$P\,;Q$ of P and Q is the synchronization tree given by P with a copy of Q at each leaf node.*

Few if any concurrent languages defined by operational semantics use true sequencing. CCS does not have a sequencing operation per se; it makes do with the restricted case aQ of running an action and then a process. CSP has a form of sequencing in which processes announce that they are finished by performing a distinguished action $\surd \notin \mathsf{Act}$, and an operation $P;Q$ which runs P until it performs a $\surd$ and then runs Q. CSP sequencing is definable by positive rules. (That is, the standard "stopped process" can perform a single $\surd$.) It is possible in CSP for a process to be unable to perform any action, and also unable to signal that it is finished; such processes are *deadlocked*. If P is deadlocked, then $P;Q$ is also deadlocked. This sequencing operation is not true sequencing, as it removes some arcs labeled $\surd$ from P's tree. It is debatable whether true sequencing is a better operation than either CCS or CSP sequencing; it is probably more powerful (allowing some kinds of deadlock detection) but harder to implement (requiring some kind of deadlock detection).

We show that true sequencing is not definable up to ready simulation, and *a fortiori* up to tree isomorphism or bisimulation. In fact, this theorem holds for any reasonable notion of equivalence; *viz.*, any notion in which the process ab (which cannot perform a b action initially) is not equivalent to any process which can perform a b action initially.

Theorem 7.4.2 *True sequencing is not definable (up to ready simulation) by positive GSOS rules.*

Proof: Let $\mathcal{G}$ be a positive GSOS language. By Proposition 4.2.10, the ruloid set of $\mathcal{G}$ contains only positive ruloids. Suppose that $C[X, X']$ defined sequencing; that is, the tree of $C[P, P']$ is the true sequencing of P and P'. Consider $C[\mathbf{0}, b]$. As $\mathbf{0}; b \rightleftharpoons b$, we must have some transition $C[\mathbf{0}, b] \stackrel{b}{\rightarrow} Q$, where Q is stopped; and so we have some positive ruloid ρ

$$\frac{X \stackrel{a_j}{\rightarrow} Y_j, \quad X' \stackrel{a'_{j'}}{\rightarrow} Y'_{j'}}{C[X, X'] \stackrel{b}{\rightarrow} D[X, X', \vec{Y}, \vec{Y'}]}$$

enabling this transition. As $\mathbf{0}$ is stopped, no positive transition formula $X \stackrel{a_j}{\rightarrow} Y_j$ holds with $X = \mathbf{0}$; so, there can be no transition formulas concerning X in ρ, and there are no variables Y_j. Furthermore, each $a'_{j'}$ must be b, as $b \stackrel{b}{\rightarrow} \mathbf{0}$ is the only transition possible for b; also each $Y'_{j'}$ is instantiated by $\mathbf{0}$ for the same reason. So, $Q = D[\mathbf{0}, b, \vec{\mathbf{0}}]$.

Now consider $C[a, b]$. Clearly b satisfies the same transition formulas as itself, so $C[a, b]$ has behavior under the ruloid ρ:

$$C[a, b] \stackrel{b}{\rightarrow} D[a, b, \vec{\mathbf{0}}].$$

However, we have assumed that $C[X, X']$ defines sequencing, and in particular that $C[a, b] \rightleftharpoons ab$. However, we have discovered that $C[a, b]$ has an initial b-transition and ab does not; therefore the two processes are not ready similar, nor yet equivalent under any reasonable notion of process equivalence.

□

Part IV

Approaches to Bisimulation

Chapter 8

Global Testing

8.1 Global Testing and Bisimulation

In this chapter, we present a language defined by a sort of structured operational rules in which bisimulation is trace congruence. The rules of this language necessarily depart from the GSOS format.

This language is a variant of a language proposed by Samson Abramsky [Abr87].[1] Roughly, we code Hennessy-Milner formulas φ as processes F_φ, and construct a context $\mathrm{Sat}(F_\varphi, P)$ which computes whether or not $P \models \varphi$. The $[a]\varphi$ modality is achieved by *global testing*; that is, we have rules of the general form

$$\frac{X \stackrel{a}{\Rightarrow} \langle Y_1, \ldots, Y_n \rangle}{\alpha(X) \stackrel{a}{\rightarrow} \beta(X_1) \diamond \beta(X_2) \diamond \cdots \diamond \beta(X_n)}$$

In this rule, α is testing the formula $[a]\varphi$ by listing in order all of the a-descendants $Y_1, \ldots, Y_n$ of X, and applying the operator β testing for satisfaction of φ to each of them, and using the operation $\diamond$ to combine the results.

It is reasonably straightforward to design a language of this form with an infinite set of operators, op_φ, one for each Hennessy-Milner formula φ, in which bisimulation coincides with trace congruence. This, however, violates our requirement that the set of operations be finite. It is similarly straightforward to build a two-sorted language, of *processes* and *formulas*, with the same property[Abr87]; of course, this violates our requirement that the language be one-sorted. Jan Friso Groote [Gro89] has an alternate approach for building a language in which bisimulation is trace congruence, involving much simpler-looking rules without global testing; however, the transition relation for his language is not r.e., and in some variants exceeds the arithmetic hierarchy.

[1] In Abramsky's language, there were two sorts of processes, and bisimulation was not a congruence on one of the sorts. Our language, unlike Abramsky's, fits the basic requirements of Section 3.5.

There is, however, a single-sorted, finite global-testing language in which bisimulation coincides with trace congruence, and has the other basic properties of Section 3.5. We achieve this goal at some cost to the elegance of the language; the operation we add is harder to describe in familiar terms than the splitter and synchronizer of CCSSS. Indeed, the same proof shows how to add three operations and possibly a few actions to any GSOS language to give a global-testing language in which bisimulation coincides with trace congruence.

We assume that the alphabet of actions contains at least the following symbols: t, f, $\diamond$, $\Box$, c_1, c_2, d_1, d_2, o. The satisfaction-tester will produce o's while it is running, and a t or f when it has made a decision true or false. The remaining actions are used to code modal formulas.

First, we define the operations of global-testing conjunction and disjunction, $P \triangle Q$ and $P \nabla Q$. Computing these requires an inordinate amount of caution; we cannot allow the computations any asynchrony or branching, or bisimulation will cease to be a congruence. We use the notation $\vec{P} = \langle P_i : i = 1, \ldots, n \rangle$ for vectors of process terms. We define $\boxed{\triangle}_{i=1}^{0} P_i = t$, and

$$\boxed{\triangle}_{i=1}^{n+1} P_i = \left(\boxed{\triangle}_{i=1}^{n} P_i\right) \triangle P_{n+1} \tag{8.1}$$

Here, and for the rest of this section, conjunctions $\triangle$ and disjunctions ∇ are symmetric; *e.g.*, $\boxed{\nabla}_{i=1}^{n} P_i$ is defined in the obvious fashion. We also use the notations $\boxed{\triangle}\vec{P}$ and $\boxed{\nabla}\vec{P}$ to operate on the entire vector.

$$\frac{P \overset{o}{\Rightarrow} \langle P_1, \ldots, P_m \rangle,\ Q \overset{o}{\Rightarrow} \langle Q_1, \ldots, Q_n \rangle,\ (m, n > 0)}{\begin{array}{c} P \triangle Q \overset{o}{\rightarrow} \left(\boxed{\triangle}_{i=1}^{m} P_i\right) \triangle \left(\boxed{\triangle}_{i=1}^{n} Q_i\right) \\ P \nabla Q \overset{o}{\rightarrow} \left(\boxed{\nabla}_{i=1}^{m} P_i\right) \nabla \left(\boxed{\nabla}_{i=1}^{n} Q_i\right) \end{array}} \qquad \frac{P \overset{a}{\rightarrow} P', Q \overset{b}{\rightarrow} Q',\ (a, b \in \{t, f\})}{\begin{array}{c} P \triangle Q \overset{c}{\rightarrow} \mathbf{0}, \quad (c = a \wedge b) \\ P \nabla Q \overset{d}{\rightarrow} \mathbf{0}, \quad (d = a \vee b) \end{array}} \tag{8.2}$$

$$\frac{P \overset{o}{\Rightarrow} \langle P_1, \ldots, P_m \rangle,\ Q \overset{o}{\not\rightarrow},\ (m > 0)}{\begin{array}{c} P \triangle Q \overset{o}{\rightarrow} \left(\boxed{\triangle}_{i=1}^{m} P_i\right) \triangle Q \\ P \nabla Q \overset{o}{\rightarrow} \left(\boxed{\nabla}_{i=1}^{m} P_i\right) \nabla Q \end{array}} \qquad \frac{P \overset{o}{\not\rightarrow},\ Q \overset{o}{\Rightarrow} \langle Q_1, \ldots, Q_n \rangle,\ (n > 0)}{\begin{array}{c} P \triangle Q \overset{o}{\rightarrow} P \triangle \left(\boxed{\triangle}_{i=1}^{n} Q_i\right) \\ P \nabla Q \overset{o}{\rightarrow} P \nabla \left(\boxed{\nabla}_{i=1}^{n} Q_i\right) \end{array}} \tag{8.3}$$

It is convenient to package the three global-testing rules for $\triangle$ into a single rule in the following lemma, using the function $P^{\triangle}$. P^{∇} is defined dually.

$$P^{\triangle} = \begin{cases} \boxed{\triangle}_{i=1}^{n} P_i & P \overset{o}{\Rightarrow} \langle P_1, \ldots, P_n \rangle, n > 0 \\ P & P \overset{o}{\not\rightarrow} \end{cases} \tag{8.4}$$

The behavior of a conjunction is a simple combination of the behavior of its conjuncts. It is worth noting that $P \triangle Q$ is *o-monogenic*; it has at most one o-descendant. We omit the statement of the dual of this lemma.

Lemma 8.1.1 $\left(\boxed{\triangle}_{i=1}^{n} P_i\right) \xrightarrow{o} S$ *iff* $S = \boxed{\triangle}_{i=1}^{n} P_i^{\triangle}$ *and at least one* $P_i \xrightarrow{o}$. $\left(\boxed{\triangle}_{i=1}^{n} P_i\right) \xrightarrow{b} S$ *iff* $S = \mathbf{0}$, $b \in \{t, f\}$, *and each* $P_i \xrightarrow{b_i}$ *for some* b_i*'s with* $\bigwedge b_i = b$.

The real workhorse of Modal-CCS is the Modal$(C \ggg P, Q)$ operator. It is hard to disguise this as a synchronizer. C determines the operator structure, and P the actions in the modalities. We will define the rules and codings of formulas simultaneously.

$$\frac{C \xrightarrow{t} C'}{\text{Modal}(C \ggg P, Q) \xrightarrow{t} \mathbf{0}} \qquad \begin{aligned} C_{tt} &= t \\ P_{tt} &= \mathbf{0} \end{aligned}$$

and similarly for falsehood.

$$\frac{C \xrightarrow{c_1} C_1,\ C \xrightarrow{c_2} C_2,\ P \xrightarrow{c_1} P_1,\ P \xrightarrow{c_2} P_2}{\text{Modal}(C \ggg P, Q) \xrightarrow{o} \text{Modal}(C_1 \ggg P_1, Q) \triangle \text{Modal}(C_2 \ggg P_2, Q)}$$

$$\begin{aligned} C_{\varphi \wedge \psi} &= c_1 C_\varphi + c_2 C_\psi \\ P_{\varphi \wedge \psi} &= c_1 P_\varphi + c_2 P_\psi \end{aligned}$$

Disjunctions are treated similarly, using d_i in place of c_i and ∇ in place of $\triangle$.

$$\frac{C \xrightarrow{\Box} C_1, P \xrightarrow{a} P_1, Q \overset{a}{\Rrightarrow} \langle Q_1, \ldots, Q_n \rangle, n \geq 0}{\text{Modal}(C \ggg P, Q) \xrightarrow{o} \boxed{\triangle}_{i=1}^{n} \text{Modal}(C_1 \ggg P_1, Q_i)} \tag{8.5}$$

$$\begin{aligned} C_{\langle a \rangle \varphi} &= \Box C_\varphi \\ P_{\langle a \rangle \varphi} &= a P_\varphi \end{aligned}$$

Possibility modalities are defined by similar rules using $\boxed{\nabla}_{i=1}^{n}$.

We define $Q \overset{a}{\Rrightarrow} \langle Q_1, \ldots, Q_n \rangle$ so that it can be satisfied in precisely one way. We fix an order on all proofs of $P \xrightarrow{a} P'$; $Q \overset{a}{\Rrightarrow} \langle Q_1, \ldots, Q_n \rangle$ is satisfied if and only if there are exactly n proofs that $Q \xrightarrow{a} Q'$ for any Q', and $Q_1, \ldots, Q_n$ are the Q''s in the fixed order. For example, $a + a \overset{a}{\Rrightarrow} \langle \mathbf{0}, \mathbf{0} \rangle$.[2]

[2]The technical peculiarities of this definition are aesthetic, not essential. The ordering of processes allows us to avoid cluttering the proofs of this section with innumerable unedifying "up to permutation of the Q_i's."

Global-testing rules are schema, with one instance of the rule for each n, in much the same way that the β-rule of the lambda calculus is a rule scheme; the notation $\boxed{\bigtriangledown}$, like the λ-calculus $M[x := N]$, is a metanotation rather than an operation in the language. It is not clear just what a global-testing rule is in general; for example, we could imagine a system in which the consequent was not a recursive function of n. Whatever the general definition, however, Modal-CCS is definitely the kind of thing we want to allow as a global-testing system.

We will now show that bisimulation is precisely trace congruence with respect to Modal-CCS. The proof is in two parts, Theorem 8.1.2 showing that bisimulation is a congruence, and Theorem 8.1.7 showing that the coding of Hennessy-Milner logic in Modal-CCS is correct and hence that Modal-CCS can distinguish between non-bisimilar processes.

Theorem 8.1.2 *Modal-CCS respects bisimulation; that is, if* $\vec{P} \underline{\leftrightarrow} \vec{P'}$, *then* $C[\vec{P}] \underline{\leftrightarrow} C[\vec{P'}]$.

Proof:

The proof uses the standard bisimulation methodology. We define a bisimulation relation $\underline{\Leftrightarrow}$ which is essentially the congruence extension of bisimulation. We will have, if $\vec{P} \underline{\leftrightarrow} \vec{P'}$, then $\vec{P} \underline{\Leftrightarrow} \vec{P'}$. The definition of $\underline{\Leftrightarrow}$ will imply that $\mathrm{op}(\vec{P}) \underline{\Leftrightarrow} \mathrm{op}(\vec{P'})$ and thus $\mathrm{op}(\vec{P}) \underline{\leftrightarrow} \mathrm{op}(\vec{P'})$.

We define $\underline{\Leftrightarrow}$ between processes, and the auxiliary relations $\leq_\circ$ and $=_\circ$ between sequences of processes, by mutual recursion.

Definition 8.1.3

- $\vec{P} \leq_\circ \vec{P'}$ *iff for each* P_i *there is a* $P'_{i'}$ *such that* $P_i \underline{\Leftrightarrow} P'_{i'}$.
- $\vec{P}$ *and* $\vec{P'}$ *are* $\underline{\Leftrightarrow}$-tangled, $\vec{P} =_\circ \vec{P'}$, *iff* $\vec{P} \leq_\circ \vec{P'}$ *and* $\vec{P'} \leq_\circ \vec{P}$.
- $\underline{\Leftrightarrow}$ *is defined by:*
  1. *Whenever* $P \underline{\leftrightarrow} P'$ *then* $P \underline{\Leftrightarrow} P'$.
  2. *Whenever* $P_i \underline{\Leftrightarrow} P'_i$ *for* $i = 1, \ldots, \mathrm{arity}(\mathrm{op})$, *then* $\mathrm{op}(\vec{P}) \underline{\Leftrightarrow} \mathrm{op}(\vec{P'})$.
  3. *Whenever* $\vec{P} =_\circ \vec{P'}$, *then* $\boxed{\bigtriangledown}\vec{P} \underline{\Leftrightarrow} \boxed{\bigtriangledown}\vec{P'}$ *and* $\boxed{\bigtriangleup}\vec{P} \underline{\Leftrightarrow} \boxed{\bigtriangleup}\vec{P'}$.

We must now show that $\underline{\Leftrightarrow}$ is a bisimulation relation. The proof is by induction on the definition of $\underline{\Leftrightarrow}$, showing that whenever $R \underline{\Leftrightarrow} R'$ and $R \stackrel{a}{\rightarrow} S$, then there is a term S' such that $R' \stackrel{a}{\rightarrow} S'$ and $S \underline{\Leftrightarrow} S'$, and similarly whenever $R' \stackrel{a}{\rightarrow} S'$.

The following useful fact appears several times in the proof. Suppose that $P \underline{\Leftrightarrow} P'$ and the induction hypothesis holds for P and P'. Let $P \stackrel{a}{\Rightarrow} \langle P_1, \ldots, P_n \rangle$ and $P' \stackrel{a}{\Rightarrow} \langle P'_1, \ldots, P'_{n'} \rangle$. Then either $n = n' = 0$ (*viz.*, $P \stackrel{a}{\not\rightarrow}$ and $P' \stackrel{a}{\not\rightarrow}$), or $n, n' > 0$ and $\vec{P} =_\circ \vec{P'}$; this follows immediately from the

induction hypothesis. By 3, under these conditions we have $P^{\triangle} \underline{\Leftrightarrow} P'^{\triangle}$ and $P^{\triangledown} \underline{\Leftrightarrow} P'^{\triangledown}$.

The induction proceeds by cases according to why $P \underline{\Leftrightarrow} P'$.

1. If $R \underline{\leftrightarrow} R'$, then the lemma is obvious.

2. If $R = \mathrm{op}(\vec{P}) \underline{\Leftrightarrow} \mathrm{op}(\vec{P'}) = R'$ where $\vec{P} \underline{\Leftrightarrow} \vec{P'}$ and the induction hypothesis holds for each P, and $R \stackrel{a}{\rightarrow} S$, then we must show that $R' \stackrel{a}{\rightarrow} S'$ for some $S' \underline{\Leftrightarrow} S$. When the transition $R \stackrel{a}{\rightarrow} S$ is given by a GSOS rule, then arguments like those in Section 4.2.2 apply. We must check the GSOS operations individually. We present one sample global testing rule; the rest are similar.

   Suppose that $R = P \triangle Q \stackrel{o}{\rightarrow} S$, $R' = P' \triangle Q'$, and $P \underline{\Leftrightarrow} P'$ and $Q \underline{\Leftrightarrow} Q'$. By Lemma 8.1.1, we know that $S = P^{\triangle} \triangle Q^{\triangle}$, and that at least one of P and Q can take an o-step. By the useful fact, we know that at least one of P' and Q' can take an o-step, and hence that $R' \stackrel{o}{\rightarrow} P'^{\triangle} \triangle Q'^{\triangle} = S'$. We know $P^{\triangle} \underline{\Leftrightarrow} P'^{\triangle}$ and $Q^{\triangle} \underline{\Leftrightarrow} Q'^{\triangle}$ by induction and the useful fact, so we have $S \underline{\Leftrightarrow} S'$ as desired.

3. Finally, suppose that $R = \left(\boxed{\triangle}_{i=1}^{n} P_i\right) \underline{\Leftrightarrow} \left(\boxed{\triangle}_{i'=1}^{n'} P'_{i'}\right) = R'$, where $\vec{P} =_{\circ} \vec{P'}$. Suppose that $R \stackrel{a}{\rightarrow} S$. The case where $a \neq o$ is trivial; suppose $a = o$. Then $S = \boxed{\triangle}_{i=1}^{n} P_i{}^{\triangle}$, and some $P_{i_0} \stackrel{o}{\rightarrow}$. By the induction hypothesis, some $P'_{i'_0} \stackrel{o}{\rightarrow}$.

   We now show that $\vec{P^{\triangle}} = \left\langle P_i{}^{\triangle} : i = 1, \ldots, n \right\rangle =_{\circ} \left\langle P'_{i'}{}^{\triangle} : i' = 1, \ldots, n' \right\rangle = \vec{P'^{\triangle}}$. We know that for each i there is an i' such that $P_i \underline{\Leftrightarrow} P'_{i'}$; by induction and the useful fact, this implies that $P_i{}^{\triangle} \underline{\Leftrightarrow} P'_{i'}{}^{\triangle}$, showing that $\vec{P^{\triangle}} \leq_{\circ} \vec{P'^{\triangle}}$ The other direction is similar.

   So, know that $R' \stackrel{o}{\rightarrow} S' = \boxed{\triangle}_{i=1}^{n'} P'_{i'}{}^{\triangle}$ and $S \underline{\Leftrightarrow} S'$ as desired.

We have shown that $\underline{\Leftrightarrow}$ is a bisimulation relation, and so $R \underline{\Leftrightarrow} R'$ implies $R \underline{\leftrightarrow} R'$; by the first clause in the definition of $\underline{\Leftrightarrow}$, the converse holds as well. Therefore, $\underline{\Leftrightarrow}$ and $\underline{\leftrightarrow}$ are the same relation. Clearly $\underline{\Leftrightarrow}$ is a congruence relation; a simple induction on the structure of contexts shows that if $\vec{R} \underline{\Leftrightarrow} \vec{R'}$ then $C[\vec{R}] \underline{\Leftrightarrow} C[\vec{R'}]$. The theorem follows immediately. □

Corollary 8.1.4 *If P and P' are bisimilar, then they are trace congruent with respect to Modal-CCS.*

Now, we show that we have achieved bisimulation. First, we describe the behavior of $\boxed{\triangle}$ and $\boxed{\triangledown}$ on monogenic processes.

Lemma 8.1.5 *If $P_i \equiv o^{n_i} b_i$ where $b_i \in \{t, f\}$, then*

$$\begin{aligned} \boxed{\triangle}_{i=1}^{m} P_i &\equiv o^n \bigwedge_{i=1}^{m} b_i \\ \boxed{\triangledown}_{i=1}^{m} P_i &\equiv o^n \bigvee_{i=1}^{m} b_i \end{aligned}$$

where $n = \max_{i=1}^{m} n_i$.

Proof: Easy induction on m. □

If φ is a Hennessy-Milner formula, let $C_\varphi[X] = \text{Modal}(C_\varphi \ggg P_\varphi, X)$.

Lemma 8.1.6 *If Q is a process and φ a Hennessy-Milner formula, then for some j $C_\varphi[Q] \equiv o^j t$ if $Q \models \varphi$, and $C_\varphi[Q] \equiv o^j f$ otherwise.*

Proof: Predictably, the proof is by induction on φ. The base cases tt and ff are trivial. If the lemma holds for φ and ψ, consider a trace of $C_{\varphi \wedge \psi}[P]$:

$$C_{\varphi \wedge \psi}[P] \xrightarrow{o} C_\varphi[P] \triangle C_\psi[P]$$

(which is the only transition from $C_{\varphi \wedge \psi}[P]$), and by Lemma 8.1.5 the lemma follows. Similarly, the only transition from $C_{[a]\varphi}[P]$ is

$$C_{[a]\varphi}[P] \xrightarrow{o} \boxed{\triangle}_{i=1}^{n} C_\varphi[P]$$

and again the lemma follows from Lemma 8.1.5. (Note that if $n = 0$, then $\boxed{\triangle}_{i=1}^{n} C_\varphi[P] = t$ as desired.) The remaining logical connectives are similar □

Corollary 8.1.7 *$Q \models \varphi$ iff $\text{tr}(C_\varphi[Q]) = \left\{o^j t\right\}$ for some j, and hence if $Q \not\underline{\leftrightarrow} Q'$ then Q and Q' are not trace congruent.*

Proof: The first statement is immediate from Lemma 8.1.6; the second follows from the first and Theorem 2.4.8. □

Combining Corollary 8.1.4 and Corollary 8.1.7, we have:

Theorem 8.1.8 *Bisimulation and Modal-CCS congruence coincide.*

8.1.1 Discussion of Modal-CCS

We have presented Modal-CCS as an attempt to give a reasonable language in which bisimulation is trace congruence, and indeed Modal-CCS is much less disturbing than the language of Section 3.2. However, it has some peculiar features.

It seems to use nondeterminism in the wrong way. Ordinarily, nondeterminism allows a process or system to *choose* among several alternatives, or to state that all the alternatives are *possible*; *e.g.*, in the process $P|Q$ in which P and Q are running in interleaving parallel, either P or Q may take the first step. Global testing rules treat nondeterminism more as a way of *listing* a collection of processes. Applying a global-testing operation to $P|Q$ will somehow determine all the ways in which the system could proceed, and combine these in some way. It is less than clear how to implement or even understand such an operation.

The formal contrast between CCSSS and Modal-CCS is worth noting. As CCSSS is a GSOS language, we know that ready simulation refines CCSSS-congruence. The corresponding fact for Modal-CCS, Theorem 8.1.2, is a nontrivial theorem. More significantly, this theorem must be reproven whenever the language changes; when changes are made to the global-testing rules, even the induction hypothesis may change. The alterations are generally nontrivial. The proofs that the codings of modal logic work, Lemma **??** and Lemma 8.1.6, are roughly equivalent in difficulty if fully written out.

Global-testing rules appear in surprising places in Modal-CCS. They were introduced in the Modal($C \ggg P, Q$) operation to deal with $\langle a \rangle \varphi$ and $[a]\varphi$. However, we must use them elsewhere, in the $\triangle$ and $\triangledown$ operations, which sound as if they could be defined quite sensibly by GSOS rules such as $P \stackrel{o}{\to} P', Q \stackrel{o}{\to} Q' \Longrightarrow P \,\&\, Q \stackrel{o}{\to} P' \,\&\, Q'$. (We call this *nondeterministic* conjunction.) However, this language does not respect bisimulation.[3]

Part of the reason that Theorem 8.1.2 is so difficult is that it is a delicate theorem; with small changes to the language will make it false. For example, if we had used nondeterministic rather than global-testing conjunction in the definition of Modal($\ggg$,), then bisimulation would cease to be a congruence.

It is straightforward to find a global-testing language in which trace congruence is precisely synchronization tree isomorphism; in fact, it is nontrivial to find a language using global testing in an interesting way in which a and $a + a$ are congruent. There is a general consensus in the CCS/CSP community that a and $a + a$ should be identified, and so that tree isomorphism is too refined an equivalence, it is clear that arbitrary global-testing languages are undesirable.

Moreover, the difference between Modal-CCS and its too-refined variants is small and subtle. There might be a reasonably general theorem saying

[3] It is possible to use nondeterministic rather than global-testing connectives, at the cost of making the Modal($C \ggg P, Q$) rules considerably more complex.

that if $\bigtriangledown$ and $\bigtriangleup$ possess suitable algebraic properties, then Modal($\cdot \ggg \cdot, \cdot$) respects bisimulation.

Although it is dangerous to predict that no good theory of global-testing processes will ever be developed, it would be surprising if it were as generally applicable, appealing, and closely matched to bisimulation (rather than tree isomorphism) as the theory we have sketched for GSOS languages. For example, it is trivial to decide if a GSOS language respects ready simulation (it does); it is probably undecidable whether or not a global-testing language respects bisimulation.

Part V

Concluding Remarks

Chapter 9

Conclusions

9.1 Conclusions

The main purpose of this work was to give an approach to sorting through the variety of semantics for concurrency which have been proposed. We worked in the setting of CCS and related languages, concentrating on notions of program comparison. It is well-known that bisimulation, the notion of process equivalence usually used in CCS, is not fully abstract with respect to observing traces. We started this research looking for an explanation of bisimulation; this lead us to the study of CCS-like languages and the discovery of ready simulation.

We considered the possibility of extending CCS in some way to make bisimulation fully abstract. We did not want the extension to violate the spirit of CCS, and so we developed a class of CCS-like languages: the GSOS languages. Rather than simply picking an arbitrary generalization, we made an attempt to find a maximal reasonable generalization. We isolated some essential properties of the original rules — in our case, finite computable branching and respecting bisimulation and the synchronization tree semantics were the most important — and found a large class of languages which guaranteed these properties, and a constellation of counterexamples, languages just outside our class which violated some of the essential properties. As a good pragmatic check, most of the languages proposed have either been in our class or violated the essential properties. This class, then, is our candidate formalization of a class including all reasonable languages.

We then introduced the notion and theory of ready simulation. Ready simulation is a weakening of bisimulation, which is adequate for all GSOS languages and fully abstract for an extended version of CCS. Bisimulation and ready simulation have fairly similar theories. Each one has two main equivalent definitions, existential and modal. Each has a complete axiomatization on finite processes, a polynomial-time decision procedure on regular processes, and so on. Both have an associated methodology of proving pro-

grams equal, using the existential definition. Mathematically, the two are fairly similar in character.

Ready simulation seems to have a philosophical advantage. Ready simulation semantics are fully abstract, with respect to a wide variety of kinds of observations and languages. Bisimulation is not fully abstract with respect to any known reasonable observations or languages. It seems unlikely that bisimulation will ever be fully abstract with respect to something reasonable; it cannot be fully abstract with respect to, for example, GSOS languages and any of the plausible experiments presented in Chapter 5.

As the two semantics are roughly equivalent in their formal properties, and ready simulation is better justified philosophically and computationally, ready simulation seems to be a better choice for the fundamental strong equivalence for CCS-like languages — in particular, those provided with process-control and copying operations as appearing in CCSSS.

The ideas and methods of this work should apply to other areas in semantics. It is quite common for a semantic model to be adequate but not fully abstract. Full abstraction, though not an absolute requirement, is a pleasing property; full abstraction of the sort achieved here, applying over a large class of languages, suggests that the semantics is a good one.

The methodology of quantifying over languages defined by structured operational semantics has been useful in at least one other context. Typed lambda calculus with arithmetic and recursion (called PCF) is fully abstract with respect to the Scott cpo model with the addition of some suitable parallel combinators[Plo77]. However, it is not fully abstract with respect to the Scott *lattice* model with respect to any set of combinators definable by some powerful rules, of a form which strongly resemble positive GSOS rules[Blo88a].

9.2 Open Problems

1. The notion of ready simulation resembles several notions proposed for meeting a specification [LT88, AL88]. It would be informative to attempt to verify some protocols with respect to ready simulation.

2. We have made a fairly strong case that the GSOS languages include the reasonable ones. We have some theorems showing that some basic essential properties hold of them, and a constellation of counterexamples showing that these theorems fail for many natural larger classes of language. Further, the class of GSOS languages is amenable to further mathematical study. However, we have no such theory of what "reasonable" or "well-structured" observations is. We have a fairly solid metaphor, of a scientist pushing buttons on a black box. Formalizing this metaphor in particular scenarios is easy. Formalizing the notion of a good experimental scenario remains to be done.

3. The theory of ready simulation requires further work. Bisimulation of finite processes has a complete axiom system, and bisimulation of regular processes has a polynomial time algorithm; the same holds of ready simulation. However, the results for bisimulation are stronger. The algorithm for bisimulation is $O(n \lg n)$, while for ready simulation it is $O(n^6)$. Furthermore, there is a complete axiomatization of bisimulation on *regular* processes. It seems likely that similar results should hold for ready simulation; however, they have not yet been achieved.

4. Strong bisimulation, though a useful relation, is not the relation of primary concern in CCS theory. *Weak bisimulation*, a variant of strong bisimulation which ignores silent moves, is the notion commonly used. We expect that the results of this thesis, suitably generalized, will carry over to the case of weak bisimulation. However, this remains to be explored.

   In fact, the problem becomes somewhat murkier. Strong bisimulation is a very nice relation; for example, all the CCS operations respect it. Weak bisimulation is not as nice: summation, one of the most essential combinators, does not respect weak bisimulation. The actual notion used is congruence with respect to $+$, which happens to be equal to the congruence closure of weak bisimulation. We now have something of a puzzle: what are the essential properties of "well-structured operations" to be? Should they respect weak bisimulation, like all of the CCS operations other than $+$? Are they permitted to be as bad as $+$? Can they be worse?

   A further issue is the treatment of *divergence*. In the languages we have considered, it is not possible for a process to compute forever without producing any output. In CCS with unguarded recursion, this is no longer true: $\mathrm{fix}\,[X \Leftarrow X]$ is a process which thinks forever about what it is going to do next. We can eliminate such processes from consideration by only working with guarded recursions. In CCS with silent moves, we can write the process $\mathrm{fix}\,[X \Leftarrow \tau X]$, which performs silent moves forever. This is a kind of divergence; the process computes forever, though it does produce silent actions while it is computing. The proper relations between these two sorts of divergence are not clear: are the two kinds of divergence equally bad? Should one be considered catastrophic but the other benign?

5. How essential is the restriction to finite GSOS languages?

6. Can we get Theorem 7.2.6 to work for ready simulation?

9.3 Acknowledgments

I am vastly grateful to my advsior Albert R. Meyer, whose superb mathematical intuition and clear philosophical understanding have shaped this thesis. Sorin Istrail, a co-author of [BIM88], made significant contributions to Chapter 3 and Chapter 4, both in ideas and in boundless energy. Kim Larsen and independently Rob van Glabbeek discovered the state-comparison form of ready simulation, which was a substantial improvement on the denial-logic form which we had been working with. Special thanks are due to Lalita Jategaonkar, Arie Rudich, and especially Jon Riecke, who have suffered through innumerable drafts of this work as it grew from conference paper to job talk to thesis. Voluminous computer-mail discussions with Frits Vaandrager and Jan Friso Groote about GSOS and other kinds of structured operational rules greatly contributed to all our understanding. Discussions with Robin Milner about the "weathers" model inspired a very memorable walk with Albert Meyer in the weather in Edinburgh, in which we sketched the research discussed in Chapter 5. Samson Abramsky provided a variety of helpful and insightful comments, as well as the idea of a global-testing language. Nancy Lynch and John Guttag, the readers of this thesis, provided a wealth of advice and commentary, and brought a variety of somewhat related work to my attention.

I would also like to thank Vicki Borah for a good deal of physical and emotional support during the final throes of this thesis, and a great many other things as well. Ray Hirschfeld, my friend, officemate, and frequent competitor for our one high-quality terminal, deserves a place in this thesis (like so many others) for his energetic and skillful maintenance of the Theory Group's computers, and his gifts of chocolate to people writing acknowledgements. Be Hubbard and Sally Bemus deserve special thanks for assistance in many of the emergencies over the course of my graduate study.

Finally, I would like to thank the NSF and ONR; an NSF graduate fellowship, MIT research and teaching assistantships, and NSF Grant No. 8511190-DCR and ONR grant No. N00014-83-K-0125 supported much the research in this thesis.

Appendix A

Appendix

A.1 Mathematical Notation

We use fairly conventional mathematical notation, but as mathematical notation has several conflicting conventions, we will be explicit.

We write $A \subsetneq B$ and $A \subseteq B$ for proper and improper set containment. $A \cup B$ and $A \cap B$ are set union and intersection. The sets $\mathbf{N}$ and ω are the sets of natural numbers, $\{0, 1, 2, \ldots\}$; ω is more commonly used as the first infinite cardinal. Ranges of integers are also useful; $[m \ldots n]$ is $\{i \in \mathbf{Z} : m \leq i \leq n\}$.

$\mathbf{B}$ is the set $\{\mathrm{tt}, \mathrm{ff}\}$ of *Booleans*; tt is used for truth, ff for falsehood. We write if b then x else y for the mathematical conditional; this expression is x when $b = \mathrm{tt}$ and y when $b = \mathrm{ff}$. It is not used when $b \notin \mathbf{B}$.

The notation for strings is similar to that for vectors. If Σ is a set, Σ^* is the set of all finite strings of elements of Σ, which might as well be given concretely as

$$\{s : s \text{ is a function } [1 \ldots n] \to \Sigma\}$$

The symbol $\langle\rangle$ is the empty string; $|s|$ is the length of s. We write s_i for $s(i)$, the i^{th} character in s; the substring from position i to position j is denoted $s_{i\ldots j}$. Catenation is written by juxtaposition st. We take the usual liberties with the mathematics, *e.g.*, we are quite careless about the distinction between characters and one-element strings.

The notions and notations $f : A \to B$, 1-1 (injection), and onto (surjection) are quite standard. A 1-1 onto function is called a *bijection*; a bijection from A to A is called a *permutation*. The identity function $\mathrm{id}_A : A \to A$ is defined $\mathrm{id}(x) = x$. If $f : A \to B$ and $g : B \to C$, then $gf : A \to C$ is the composition of f and g, given by $(gf)(x) = g(f(x))$. If $f : A \to A$, then $f^{(n)}$ is the n^{th} iterate of f, given by $f^{(0)} = \mathrm{id}$ and $f^{(n+1)} = ff^{(n)}$.

A relation $\preceq$ between elements of A and elements of B is simply a subset of $A \times B$. If $\preceq$ is a relation on elements of A (that is, between elements of A and elements of A), and $f : A \to A$, then $\preceq$ is a *congruence* with respect

to f if, whenever $a \preceq a'$, then $f(a) \preceq f(a')$. Similar terminology applies to n-ary functions from A to A.

A *preorder* is a reflexive, transitive relation; a *partial order* is an anti-symmetric preorder.

A multiset is, intuitively, a set-like thing which may contain multiple copies of objects. Formally, a multiset is a mapping M from some set A to positive numbers. The basic definitions for manipulating multisets are fairly natural, and generally similar to manipulating sets. For example, the union of $M : A \to \mathbf{N}$ and $N : B \to \mathbf{N}$ is a map from $A \cup B$:

$$(M \cup N)c = \begin{cases} M(c) + N(c) & c \in A \cap B \\ M(c) & c \in A - B \\ N(c) & c \in B - A \end{cases}$$

Appendix B

List of Notation

Symbol	Def.	Page	Description
$C[P]$		7	Process P in context $C[X]$.
P **can-yield** o		7	Process P can yield observation o.
$P \sqsubseteq_{\mathcal{O},\mathcal{L}} Q$	1.4.1	8	P is an approximation of Q with respect to the observations in $\mathcal{O}$ and contexts over language $\mathcal{L}$
$P \equiv_{\mathcal{O},\mathcal{L}} Q$	1.4.1	8	P is equivalent to Q with respect to the observations in $\mathcal{O}$ and contexts over language $\mathcal{L}$
$a(P)$		25	Prefixing: do a, then do P
$P \mid Q$		25	P and Q running in CCS-style parallel.
$P \parallel Q$		25	P and Q running in interleaving parallel without communication
$P\backslash L$		25	Run the process P, forbidding the actions in the set L.
$\mathrm{fix}\left[X_i; \vec{X} \Leftarrow \vec{P}\right]$		25	Recursive definition of several processes. Represents the X_i solution of the equations $X_i = P_i[\vec{X}]$
$\mathrm{fix}\,[X \Leftarrow P]$		25	Recursive definition of a single process satisfying the equation $X = PX$. Abbreviation for $\mathrm{fix}\,[X; X \Leftarrow P]$
$P \equiv Q$	2.4.1	29	P and Q are isomorphic synchronization trees.
Synchronization Tree	2.3.1	28	Rooted, unordered, finitely branching tree edge-labeled by actions; the intended meaning of a process in a very fine semantics.
$P \underline{\Leftrightarrow} Q$	2.4.2	29	Notation used for strong bisimulation relations, and in particular ones which are finer than bisimulation.
$P \underline{\leftrightarrow} Q$	2.4.3	30	Bisimulation
tt,ff	2.4.6	31	Boolean constants for truth and falsehood

Symbol	Def.	Page	Description
$\varphi \wedge \psi, \varphi \vee \psi$	2.4.6	31	Logical conjunction and disjunction
$\langle a \rangle \varphi$	2.4.6	31	Possibility modality: possibly after doing an a-action, φ holds
$[a]\varphi$	2.4.6	31	Necessity modality: necessarily after doing an a, φ holds.
$P \models \varphi$	2.4.7	31	The process P satisfies (is a model of) the formula φ. Equivalently, φ holds for P
$\mathbf{0}$		37	The canonical stopped process
aP	Eqn. 3.1	37	The process which performs a, then behaves like P
$P + Q$	Eqn. 3.2	37	The process which nondeterministically chooses to behave like either P or Q.
$P \stackrel{s}{\rightarrow} Q$	3.1.2	39	The process P can perform the actions in the string s and evolve into Q.
$P \sqsubseteq_{tr} Q$	3.1.6	39	Every trace of P is a trace of Q (in isolation)
$P \equiv_{tr} Q$	3.1.6	39	P and Q have the same traces
$P \sqsubseteq_{tr}^{\mathcal{L}} Q$	3.1.6	39	P is a trace approximation of Q with respect to $\mathcal{L}$-contexts.
$P \equiv_{tr}^{\mathcal{L}} Q$	3.1.6	39	P and Q are trace congruent with respect to the language $\mathcal{L}$
GSOS Rule	3.4.2	44	CCS-like rule
$\dot{\rightsquigarrow}, \sigma \models t$	3.4.5	46	The formula t is true of transition relation $\dot{\rightsquigarrow}$ and the substitution σ.
$P; Q$		49	Sequencing: perform P, and when and if it finishes perform Q.
$P \sqsubseteq_{\supseteq} Q$	4.2.1	55	P is a ready simulation approximation of Q
$P \rightleftharpoons Q$	4.2.1	55	P and Q are ready similar; each approximates the other.

Symbol	Def.	Page	Description
readies(P)	Eqn. 4.1	55	The set of actions that P is ready to perform.
Can't(a)	4.2	60	The atomic denial formula, satisfiable if the process is unable to take an a-step
$P \leq_{DL} Q$ $P \equiv_{DL} Q$	4.3.1	60	Approximation and equivalence with respect to denial formulas.
$\langle\!\langle\!\rangle\!\rangle(P)$	Sec. 4.4	62	Copying operation of CCSSS: when P signals that it wants to be copied, $\langle\!\langle\!\rangle\!\rangle(P)$ copies it.
$S \triangleright P$	4.4	62	S running in controlled communication with P. S runs independently until it signals that it wants to communicate with P; then S and P cooperate for one step.
$S \triangleright\!\!\triangleright P$	4.4	62	An auxiliary operation used by $\triangleright$; $S \triangleright\!\!\triangleright P$ is $S \triangleright P$ in the state after S has signalled that it wants to communicate with P.
$P \Uparrow E$, $P \Downarrow E$	5.2.2	72	The process P can pass (resp. fail) the experiment E.
$P \leq_{\mathsf{Dup}} Q$	5.2.3	72	Whenever P can succeed on a duplicator experiment, so can Q.
$P \leq_{\mathsf{Wild}} Q$	Sect. 5.3	74	P approximates Q with respect to wild duplicator experiments.
$P \restriction n$	5.4.1	77	The tree P truncated to depth n
$P \preceq Q$	5.4.5	80	Q may be obtained from P by repeated duplication of subtrees. A partial order on trees used in wild global-testing duplicator equivalence.
RS	6.2.1	84	A sound and complete inequational axiom system for ready simulation of finite processes.
$\|P\|$	Eqn. 7.2.2	91	The size of the term P.
$/P/$	7.5	92	The number of loose fixes in P.
$T_{/}$	Thm. 7.2.6	93	A tree which cannot be defined in any pure GSOS language
$P \triangle Q$,$P \triangledown Q$	Eqn. 8.1	100	Parallel conjunction and disjunction testing.
$\boxed{\triangle}\,\vec{P}$,$\boxed{\triangledown}\,\vec{P}$	Eqn. 8.1	100	Parallel conjunction (disjunction)of a vector of processes
$P^{\triangle}$, $P^{\triangledown}$	Eqn. 8.4	100	
Modal($C \lll P, Q$)	8.1	101	Operator which interprets C and P as a Hennessy-Milner formula and computes whether or not Q satisfies that formula.
$\leq_{\circ}$, $=_{\circ}$, $\underline{\Leftrightarrow}$	8.1.3	102	Technical gadgets used to show that Modal-CCS respects bisimulation.

Bibliography

Bibliography

[AB84] Didier Austry and Gérard Boudol. Algèbre de processus et synchronisation. *Theoretical Computer Sci.*, 30(1):91–131, 1984.

[Abr] Samson Abramsky. Tutorial on concurrency. Slides of an invited lecture at POPL '89.

[Abr87] Samson Abramsky. Observation equivalence as a testing equivalence. *Theoretical Computer Sci.*, 53(2/3):225–241, 1987.

[ACM83] *Conference Record of the Tenth ACM Symposium on Principles of Programming Languages*, 1983.

[ACM85] *Conference Record of the Twelth Annual ACM Symposium on Principles of Programming Languages*, 1985.

[ACM89] *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, 1989.

[AL88] Mart´

n Abadi and Leslie Lamport. The existence of refinement mappings. In *Third Annual Symposium on Logic in Computer Science* [IEE88], pages 165–177.

[Arv89] Arvind. Personal communication from Professor Arvind of the MIT Laboratory for Computer Science, July 1989.

[AS89] Bowen Alpern and Fred Schneider. Verifying temporal properties without temporal logic. *ACM Trans. on Programming Languages and Systems*, 11(1):147–167, 1989.

[AV] Samson Abramsky and Steve Vickers. Observational logic and process semantics. In preparation as of March 1989.

[Bar81] Henk P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic*. North-Holland, 1981. Revised Edition, 1984.

[BB88] J. C. M. Baeten and J. A. Bergstra. Recursive process definitions with the state operator. Technical Report P8802, University of Amsterdam Programming Research Group, The Netherlands, January 1988.

[BC87] Gérard Boudol and Ilaria Castellani. Concurrency and atomicity. Technical Report 748, INRIA Sophia-Antipolis, France, 1987.

[BC89] G. Boudol and I. Castellani. Permutation of transitions: an event structure semantics for CCS and SCCS. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *REX School/Workshop on Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency,* Noordwijkerhout, volume 354 of *Lect. Notes in Computer Sci.*, pages 411–427. Springer-Verlag, 1989.

[BHR84] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *J. ACM*, 31(3):560–599, 1984.

[BIM88] Bard Bloom, Sorin Istrail, and Albert R. Meyer. Bisimulation can't be traced (preliminary report). In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 229–239, 1988. Also appears as MIT Technical Memo MIT/LCS/TM-345.

[BK82] J. A. Bergstra and J. W. Klop. Fixed point semantics in process algebras. Technical Report IW 206/82, Department of Computer Science, Mathematisch Centrum, Amsterdam, The Netherlands, 1982.

[BK84a] J. A. Bergstra and J. W. Klop. Algebra of communicating processes. Technical Report CS-R8421, Centre for Mathematics and Computer Science, Amsterdam, 1984.

[BK84b] J. A. Bergstra and J. W. Klop. Process algebra for synchronous communication. *Information and Computation*, 60(1–3):109–137, 1984.

[BK85] J. A. Bergstra and J. W. Klop. Algebra of communicating processes with abstraction. *Theoretical Computer Sci.*, 37(1):77–121, 1985.

[Blo88a] Bard Bloom. Can LCF be topped? flat lattice models of the simply typed λ-calculus. In *Third Annual Symposium on Logic in Computer Science* [IEE88], pages 282–295.

[Blo88b] Bard Bloom. Constructing two-writer registers. *IEEE Transactions on Computers*, 37(12):1506–1514, December 1988.

[Blo89] Bard Bloom. Partial traces and the semantics and logic of CCS-like languages. Technical Report 89-1066, Cornell, 1989.

[BM89] Bard Bloom and Albert R. Meyer. A remark on bisimulation between probabilistic processes. In A. R. Meyer and M. A. Taitslin, editors, *Logic at Botik '89: Symposium on Logical Foundations of Computer Science*, volume 363 of *Lect. Notes in Computer Sci.*, pages 26–40. Springer-Verlag, 1989.

[BM90] Bard Bloom and Albert R. Meyer. Experimenting with process equivalence. In *Proceedings of the International BCS-FACS Workshop on Semantics for Concurrency*. Springer-Verlag, July 1990.

[Bou85] Gérard Boudol. Notes on algebraic calculi of processes. Technical Report 395, INRIA Sophia-Antipolis, France, 1985.

[Bra85] Wilfried Brauer, editor. *Automata, Languages and Programming: 12^{th} Colloquium*, volume 194 of *Lect. Notes in Computer Sci.* Springer-Verlag, July 1985.

[BRdS85] Gérard Boudol, Gérard Roucairol, and Robert de Simone. Petri nets and algebraic calculi of processes. Technical Report 410, INRIA Sophia-Antipolis, France, 1985.

[Bro83a] Stephen Brookes. On the relationship of CCS and CSP. In J. D´

az, editor, *Automata, Languages and Programming: 10^{th} Colloquium*, volume 154 of *Lect. Notes in Computer Sci.*, pages 83–96. Springer-Verlag, July 1983.

[Bro83b] Stephen Brookes. On the relationship of CCS and CSP. Technical Report CMU-CS-83-111, Carnegie-Mellon University, 1983.

[Bro83c] Stephen Brookes. A semantics and proof system for communicating processes. Technical Report CMU-CS-83-134, Carnegie-Mellon University, 1983.

[BT86] J. A. Bergstra and J. V. Tucker. Algebraic specifications of computable and semicomputable datatypes. Technical Report CS-R8619, Stichting Mathematisch Centrum, Amsterdam, May 1986.

[BV89] J. C. M. Baeten and Frits Vaandrager. An algebra for process creation. Technical Report CS-R8907, Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands, 1989.

[BvG87] J. C. M. Baeten and Rob van Glabbeek. Another look at abstraction in process algebra. In Ottmann [Ott87], pages 84–94.

[CES83] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite state concurrent systems using temporal logic specifications: A practical approach. In ACM [ACM83], pages 117–126.

[CH87] Ilaria Castellani and Matthew Hennessy. Distributed bisimulations. Technical Report 5/87, University of Sussex, England, July 1987.

[CH88] Rance Cleaveland and Matthew Hennessy. Priorities in process algebra. In *Third Annual Symposium on Logic in Computer Science* [IEE88], pages 193–202.

[CMP87] Luca Castellano, Giorgio De Michelis, and Lucia Pomello. Concurrency vs interleaving: an instructive example. *Bull. Europ. Ass. Theoretical Computer Sci.*, 31:12–15, February 1987.

[dBBKM82] J. W. de Bakker, J. A. Bergstra, J. W. Klop, and J.-J. Ch. Meyer. Linear time and branching time semantics for recursion with merge. Technical Report IW 211/82, Mathematisch Centrum Amsterdam, 1982.

[dBZ83] J. W. de Bakker and J. I. Zucker. Compactness in semantics for merge and fair merge. Technical Report IW 238/83, Mathematisch Centrum Amsterdam, 1983.

[dMOZ86] J. W. de Bakker, J.-J. Ch. Meyer, E.-R. Olderog, and J. I. Zucker. Transition systems, metric spaces, and ready sets in the semantics of uniform concurrency. Technical Report 86-04, Department of Computer Science, SUNY at Buffalo, February 1986.

[dNH84] R. de Nicola and M. C. B. Hennessy. Testing equivalences for processes. *Theoretical Computer Sci.*, 34(2/3):83–133, 1984.

[dS85] Robert de Simone. Higher-level synchronising devices in MEIJE-SCCS. *Theoretical Computer Sci.*, 37(3):245–267, 1985.

[FHLd79] Nissim Francez, C. A. R. Hoare, Daniel J. Lehman, and Willem P. de Roever. Semantics of nondeterminism, concurrency and communication. *J. Computer and System Sci.*, 19(3):290–308, 1979.

[FLS87] Alan Fekete, Nancy Lynch, and Liuba Shira. A modular proof of correctness for a network synchronizer. Technical Report MIT/LCS/TM-341, Massachusetts Institute of Technology, 1987.

[GG89] R.J. Glabbeek and U. Goltz. Equivalence notions for concurrent systems and refinement of actions. Arbeitspapiere der GMD 366, Gesellschaft für Mathematik und Datenverarbeitung, Sankt Augustin, 1989. An extended abstract appeared in: Proceedings 14^{th} Symposium on Mathematical Foundations of Computer Science, Porąbka-Kozubnik, Poland, August/September 1989 (A. Kreczmar and G. Mirkowska), LNCS 379, Springer-Verlag, pp. 237–248.

[Gis84] Jay Loren Gischer. Partial orders and the axiomatic theory of schuffle. Technical Report STAN-CS-84-1033, Stanford University, 1984.

[Gro89] Jan Friso Groote. Personal communication. Electronic mail communications, March 1989.

[GS85] Susanne Graf and Joseph Sifakis. From synchronisation tree logic to acceptance. In Rohit Parikh, editor, *Logics of Programs*, volume 193 of *Lect. Notes in Computer Sci.*, pages 128–142. Springer-Verlag, 1985.

[GS86] Susanne Graf and Joseph Sifakis. A modal characterization of observational congruence on finite terms of CCS. *Information and Computation*, 68(1–3):125–145, 1986.

[GS87] Susanne Graf and Joseph Sifakis. Readiness semantics for regular processes with silent actions. In Ottmann [Ott87], pages 115–125.

[Gut75] John Guttag. The specification and application to programming of abstract data types. Technical Report CSRG-75, Computer Systems Research Group, University of Toronto, Canada, 1975. (Thesis).

[GV89] Jan Friso Groote and Frits Vaandrager. Structured operational semantics and bisimulation as a congruence (extended abstract). In G. Ausiello, M. Dezani-Ciancaglini, and S. Ronchi Della Rocca, editors, *Automata, Languages and Programming: 16^{th} International Colloquium*, volume 372 of *Lect. Notes in Computer Sci.* Springer-Verlag, 1989.

[Hen83] Matthew Hennessy. Synchronous and asynchronous experiments on processes. *Information and Computation*, 59(1–3):36–83, 1983.

[Hen85] Matthew Hennessy. An algebraic theory of fair asynchronous communicating processes. In Brauer [Bra85], pages 260–269.

[HM80] Matthew Hennessy and Robin Milner. On observing nondeterminism and concurrency. In J. W. de Bakker and J. van Leeuwen, editors, *Automata, Languages and Programming: Seventh Colloquium*, volume 85 of *Lect. Notes in Computer Sci.*, pages 299–309. Springer-Verlag, July 1980.

[HM85] Matthew Hennessy and Robin Milner. Algebraic laws for nondeterminism and concurrency. *J. ACM*, 32(1):137–161, 1985.

[HO85] Brent Hailpern and Susan Owicki. Modular verification of concurrent programs. In ACM [ACM85], pages 322–336.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Comm. ACM*, 12(10):576–580, 1969.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Comm. ACM*, 21(8):666–677, 1978.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Series in Computer Science. Prentice-Hall, 1985.

[IEE88] IEEE. *Third Annual Symposium on Logic in Computer Science*, 1988.

[Kel76] R. Keller. Formal verification of parallel programs. *Comm. ACM*, 19(7):561–572, 1976.

[KH89] Richard Kelsey and Paul Hudak. Realistic compilation by program transformation. In ACM [ACM89], pages 281–292.

[KS90] P. Kanellakis and S. Smolka. CCS expressions, finite state processes and three problems of equivalence. *Information and Computation*, 86(1):43–68, 1990.

[Lam80] Leslie Lamport. The 'Hoare logic' of concurrent programs. *Acta Informatica*, 14(1):21–37, 1980.

[Lam88] Leslie Lamport. Control predicates are better than dummy variables for reasoning about program control. *ACM Trans. on Programming Languages and Systems*, 10(2):267–281, April 1988.

[LG81] Gary Marc Levin and David Gries. A proof technique for communicating sequential processes. *Acta Informatica*, 15(3):281–302, 1981.

[LS84] Leslie Lamport and Fred B. Schneider. The "Hoare logic" of CSP and all that. *ACM Trans. on Programming Languages and Systems*, 6(2):281–296, 1984.

[LS88] Kim Larsen and Arne Skou. Bisimulation through probabilistic testing (preliminary report). Technical Report R 88-16, Institut for Elektroniske Systemer, Aalborg Universitetscenter, Aalborg, Danmark, June 1988.

[LT87] Nancy A. Lynch and Mark R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, pages 137–151, 1987.

[LT88] Nancy Lynch and Mark Tuttle. An introduction to I/O automata. Technical Report MIT/LCF/TM-373, Massachusetts Institute of Technology, 1988. (TM-351 revised).

[Mey88] Albert R. Meyer. Semantical paradigms: Notes for an invited lecture, with two appendices by Stavros Cosmadakis. In *Third Annual Symposium on Logic in Computer Science* [IEE88], pages 236–253.

[Mil77] Robin Milner. Fully abstract models of typed λ-calculus. *Theoretical Computer Sci.*, 4(1):1–22, 1977.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lect. Notes in Computer Sci.* Springer-Verlag, 1980.

[Mil81] Robin Milner. A modal characterisation of observable machine-behaviour. In E. Astesiano and C. Böhm, editors, *CAAP '81: Trees in Algebra and Programming, 6th Colloquium*, volume 112 of *Lect. Notes in Computer Sci.*, pages 25–34. Springer-Verlag, 1981.

[Mil83] Robin Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Sci.*, 25(3):267–310, 1983.

[Mil84] Robin Milner. Lectures on a calculus for communicating systems. In S. D. Brookes, A. W. Roscoe, and G. Winskel, editors, *Seminar on Concurrency: CMU, July 9-11, 1984*, volume 197 of *Lect. Notes in Computer Sci.*, pages 197–220. Springer-Verlag, 1984.

[Mil88] Robin Milner. Operational and algebraic semantics of concurrent processes. Technical Report ECS-LFCS-88-46, Laboratory for Foundations of Computer Science, University of Edinburgh, Scotland, 1988.

[Mil89] Robin Milner. A complete axiomatisation for observational congruence of finite-state behaviours. *Information and Computation*, 81(2):227–247, May 1989.

[MP83] Zohar Manna and Amir Pneuli. How to cook a temporal proof system for your pet language. In ACM [ACM83], pages 141–154.

[MS76] Milne and Strachey. *A Theory of Programming Language Semantics*. Chapman and Hall, 1976.

[Mul85] Ketan Mulmuley. *Fully Abstract Submodels of Typed Lambda Calculi*. PhD thesis, Carnegie-Mellon University, 1985.

[NGO85] Van Nguyen, David Gries, and Susan Owicki. A model and temporal proof system for networks of processes. In ACM [ACM85], pages 121–131.

[OG81] Susan Owicki and David Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6(4):319–340, 1981.

[OH86] E.-R. Olderog and C. A. R. Hoare. Specification-oriented semantics for communicating processes. *Acta Informatica*, 23(1):9–66, 1986.

[OL82] Susan Owicki and Leslie Lamport. Proving liveness properties of concurrent programs. *ACM Trans. on Programming Languages and Systems*, 4(3):455–495, 1982.

[Ott87] Thomas Ottmann, editor. *Automata, Languages and Programming: 14^{th} International Colloqium*, volume 267 of *Lect. Notes in Computer Sci.* Springer-Verlag, July 1987.

[Par81] D. Park. Concurrency and automata on infinite sequences. In P. Deussen, editor, *Theoretical Computer Science*, Lect. Notes in Computer Sci., page 261. Springer-Verlag, 1981.

[Par87a] Joachim Parrow. Submodule construction as equation solving in CCS. Technical Report ECS-LFCS-87-26, Laboratory for Foundations of Computer Science, University of Edinburgh, Scotland, 1987.

[Par87b] Joachim Parrow. Synchronization flow algebra. Technical Report ECS-LFCS-87-35, Laboratory for Foundations of Computer Science, University of Edinburgh, Scotland, 1987.

[Par87c] Joachim Parrow. Verifying a CSMA/CD-protocol with CCS. Technical Report ECS-LFCS-87-18, Laboratory for Foundations of Computer Science, University of Edinburgh, Scotland, January 1987.

[Phi86] Iain Phillips. Refusal testing. In Laurent Kott, editor, *Automata, Languages and Programming: 13^{th} International Colloquium*, volume 226 of *Lect. Notes in Computer Sci.*, pages 304–313. Springer-Verlag, July 1986.

[Plo77] G. D. Plotkin. LCF considered as a programming language. *Theoretical Computer Sci.*, 5(3):223–255, 1977.

[Plo81] Gordon Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, Computer Science Department, Denmark, 1981.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *18^{th} Annual Symposium on Foundations of Computer Science*, pages 46–57. IEEE, 1977.

[Pnu85] Amir Pnueli. Linear and branching structures in the semantics and logics of reactive systems. In Brauer [Bra85], pages 15–32.

[PP88] Gordon Plotkin and Vaughn Pratt. The resolving power of multiple observers (extended abstract). (The final version has not yet appeared, to my knowledge), October 1988.

[PS87] Prakash Panangaden and Vasant Shanbhogue. On the expressive power of indeterminate primitives. Technical Report 87-891, Cornell University, Computer Science Department, November 1987.

[PT87] Robert Paige and Robert Tarjan. Three partition refinement algorithms. *SIAM J. Computing*, 16(6):973–989, 1987.

[Sch88] Russel Schaffer. Constructing multi-writer atomic registers. Technical report, Massachusetts Institute of Technology, 1988. MIT/LCS/TM-364.

[Sco76] D. Scott. Data types as lattices. *SIAM J. Computing*, 5(3):522–587, 1976.

[SPE83] D. E. Shasha, A. Pnueli, and W. Ewald. Temporal verification of carrier-sense local area network protocols. In *Conference Record of the Eleventh ACM Symposium on Principles of Programming Languages*, pages 54–65, 1983.

[ST87] Donald Sanella and Andrzej Tarlecki. On observational equivalence and algebraic specifications. *J. Computer and System Sci.*, 34(2/3):150–178, 1987.

[Sto77] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, 1977.

[Tho89] Bent Thomsen. A calculus of higher order communicating systems. In ACM [ACM89], pages 142–154.

[Vaa89] Frits Vaandrager. A simple definition for parallel composition in prime event structures. Technical Report CS-R8903, Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands, March 1989.

[VG88] Frits Vaandrager and Jan Friso Groote. Structured operational semantics and bisimulation as a congruence. Technical Report CS-R8845, Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands, 1988.

[vGR89] Rob van Glabbeek and Jan Rutten. The processes of de Bakker and Zucker represent bisimulation equivalence classes. Unpublished manuscript, 1989.

[vGV87] Rob van Glabbeek and Frits Vaandrager. Petri net models of algebraic theories of concurrency. In J. W. de Bakker, A. J. Nijman, and P. C. Treleaven, editors, *PARLE: Parallel Architectures and Languages Europe, Volume II*, number 259 in Lect. Notes in Computer Sci., pages 224–242. Springer-Verlag, 1987.

[vGW89] Rob van Glabbeek and W. P. Weijland. Branching time and abstraction in bisimulation semantics. In G. X. Ritter, editor, *Information Processing 89: Proceedings of the IFIP 11th World Computer Congress*, pages 613–618. North-Holland, August 1989.

[Wal87] David Walker. Bisimulation equivalence and divergence in CCS. Technical Report ECS-LFCS-87-29, Laboratory for Foundations of Computer Science, University of Edinburgh, Scotland, 1987.

[Wal88] David Walker. Analysing mutual exclusion algorithms using CCS. Technical Report ECS-LFCS-88-45, Laboratory for Foundations of Computer Science, University of Edinburgh, Scotland, January 1988.

[WGS87] Jennifer Widom, David Gries, and Fred B. Schneider. Completeness and incompleteness of trace-based network proof systems. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages*, pages 27–38, 1987.