Naval Research Laboratory

Washington, DC 20375-5320

NRL/MR/5543--20-10,140

Process Algebras for Distributed Logic and ReWire

Gerard Allwein

Center for High Assurance Computer Systems Branch
Information Technology Division

September 2, 2020

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	**2. REPORT TYPE**	**3. DATES COVERED** *(From - To)*
02-09-2020	NRL Memorandum Report	1 Oct 2020 – 30 Sept 2020

4. TITLE AND SUBTITLE

Process Algebras for Distributed Logic and ReWire

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER
062235N

6. AUTHOR(S)

Gerard Allwein

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER
6B23

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Naval Research Laboratory
4555 Overlook Avenue, SW
Washington, DC 20375-5320

8. PERFORMING ORGANIZATION REPORT NUMBER

NRL/MR/5543--20-10,140

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Naval Research Laboratory
4555 Overlook Avenue, SW
Washington, DC 20375-5320

10. SPONSOR / MONITOR'S ACRONYM(S)

ONR

11. SPONSOR / MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

DISTRIBUTION STATEMENT A: Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This is an interim report on connecting Distributed Logic and the ReWire language. The connection is via process algebras. Process algebras are used to generate Kripke frames for evaluating Distributed Logic. The generated Kripke frames can also be used to evaluate terms in the ReWire language. The assumptions to read this tech report are heavy. One needs to know Distributed Logic, process algebras, the lambda calculus, and ReWire. Needless to say, the background on those topics cannot be covered here. Process algebras and the lambda calculus are basic computer science theory. The circuit used in section on ReWire does not occur in any device. It is merely a generic circuit devised for this report to illustrate how to connect an FPGA application with a process algebra. The circuit is in no way complete in that one could not actually build it without supplying many additional assumptions and constructions.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
					Gerard Allwein
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**	Unclassified Unlimited	34	**19b. TELEPHONE NUMBER** *(include area code)*
Unclassified Unlimited	Unclassified Unlimited	Unclassified Unlimited			(202) 404-3748

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39.18

This page intentionally left blank.

CONTENTS

This page
intentionally
left blank

EXECUTIVE SUMMARY

This is an interim report on connecting Distributed Logic and the ReWire language. The connection is via process algebras. Process algebras are used as mathematics for evaluating Distributed Logic (DL) and describing FPGA applications written in the language ReWire. High assurance statements in Distributed Logic are thus evaluated against applications in ReWire.

Process algebras in the literature are inappropriate for describing both Distributed Logic and ReWire together. Changes were required to capture the proper connection between these two rather disparate formal systems. DL is a logic whose statements should made true or false of an application written in ReWire. However, ReWire cannot directly interpret DL statements. Only a mathematical semantics of ReWire can interpret DL statements.

The existing mathematical semantics for ReWire is too low level. The DL statements we wish to interpret express high assurance properties of FPGA applications. The proper level at which to evaluate DL statements and serve as a mathematical semantics for ReWire is that of a process algebra. However, existing process algebras are too tightly constrained for systems that cannot express the complexities of ReWire. Hence, in this report, we investigate a process algebra that can function both as a mathematical model of DL statements and interpret applications written in ReWire.

This page
intentionally
left blank

PROCESS ALGEBRAS FOR DISTRIBUTED LOGIC AND REWIRE

1. INTRODUCTION

Distributed Logic and ReWire were developed in a complementary fashion although each can stand very well on its own. Distributed Logic (DL) is a logic for distributed systems reminiscent of channel theory [1] in that the distribution structure makes it stand out against more traditional logics. DL is most frequently used with intensional connectives such as modalities. The distribution structure is captured in a graph for a particular use of the logic. The graph is considered syntactic structure. Typically, a particular situation, such as an FPGA application, supplies the graph. Various axioms may be added much like modal logic. Each node of the graph is a locality and may contain its own modal logic. The distributed structure supports modalities between localities where a modal connective takes propositions in one locality to propositions in another. It is also possible in interpretations to have three-place relations in place of the usual two-place relations. The resultant intensional connectives are then two-place connectives like those found in relevance logic [2, 3]. The relevant publications for DL are [4–7].

ReWire (`http://mu-chaco.github.io/ReWire/`) is a functional language for FPGA applications. The main features of these applications are components running in parallel exchanging signals with each other. The components provide the localities for DL. The signals and parallel execution provide the relations that underly the graph of localities. ReWire has a compiler which produces VHDL. We are currently working on another possible code generator for FIRRTL (`https://www.chisel-lang.org/firrtl/`). The relevant publications for ReWire are [8–14].

The game plan is to have DL be the programming logic for ReWire. We are interested in showing high assurance properties of FPGA applications in ReWire. In the service of doing this, we must evaluate both Distributed Logic and ReWire in a common mathematical semantic framework. The framework of this report is a form of process algebra.

The evaluations are of two very different kinds. Statements in Distributed Logic can be evaluated over a process algebra term. The term represents a Kripke frame in *intension* ("intension" is a technical term used in model theory of logics and not a misspelling of "intention"). That means that as the process algebraic term is unwound using the rules, new terms are generated. The new terms correspond to states in an FPGA application or worlds in the Kripke model. Relationships between the process algebra terms correspond to the relations of a Kripke frame. A model in extension (for contrast) would start with state sets and several relations as already given.

A ReWire FPGA application, on the other hand, can be "compiled" into or interpreted as a process algebra term. This term is the one that is used to evaluate Distributed Logic statements about the ReWire application. ReWire is providing logical models for statements in Distributed Logic via the process algebra. Thus, the FPGA application and high assurance statements about the application become linked.

Manuscript approved August 27, 2020.

2. LAMBDA CALCULUS AND PROCESS ALGEBRA

The use of *localities* or *lanes* (we use them interchangeably) are examples of *contexts*. A component or device in ReWire is a context in which certain regularities hold. See [1] for use of the term "regularities" with respect to *channel theory* or [15] for some pre-channel theory.

2.1 General Observations

Most process algebras appear to have been modeled on the Lambda Calculus (LC). The LC is a very simple process algebra, there is only allowed to be one process. It might be possible to consider a term in the LC to be threaded if there are internal reductions that can be performed before outer reductions. Essentially though, a term is a single process. The reduction scheme

$$(\lambda x.t(x))u \rightsquigarrow t(x)[u/x]$$

where we substitute u for every free x in t represents a single step. Further internal reductions might proceed independently of one another, and these would contribute to independent threads of reductions. The Church-Rosser property assures that they are indeed independent.

The juxtaposition of $\lambda x.t(x)$ with u is actually an operation. More aptly, this should be written

$$(\lambda x.t(x)) \cdot u$$

With juxtaposition, there is a single operation determined by the juxtaposition. With $\cdot$, one can imagine more than a single operation, say, $\cdot_0$, $\cdot_1$, ... and so on.

Juxtaposition also has another property, namely that the terms necessarily must be juxtaposed together linearly on a line. A step away from this is to consider a multiset of terms, any two of which can be composed if of the correct form.

One further step is to divide the multiset into regions where with a single region, the lambda calculus reduction can take place but between regions, a signal reduction can take place where the signal reduction is the usual particle reduction of most process algebras.

In this document, the regions are localities or lanes, we use the two terms interchangeably.

2.2 Algebras of Regular Events

Expressions are algebraic terms in regular languages and can represent intensional multimodal Kripke frames. Each expression is a state or, in modal logic terms, a world; the subexpressions yield the states of a particular Kripke frame, including the expression we start with. The environment presents each expression with all of the *anti-events* used in the expression. Events and anti-events combine to form particle reduction. These reduction can be symbolized by a and $\overline{a}$ combining to form an inner particle τ which is often elided.

Since the environment provides all the anti-events to the regular events in the algebraic term, the environment drives the execution of the regular expression as a term in a process algebra. By ordering in time the anti-events presented to the expression, the expression represents a recognizer for the language being presented by the environment. The reduction relation then shows how to pull the expression apart yielding the next state relation.

2.3 Church-Rosser

From [16], the theorem is

Theorem 2.3.1 *If $t \rightsquigarrow u$ and $t \rightsquigarrow v$, one can find w such that $u \rightsquigarrow w$ and $v \rightsquigarrow w$.*

In diagram form, this is

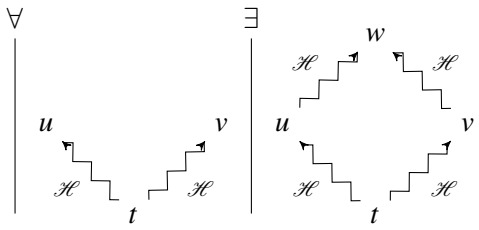


Figure 2.1: Church-Rosser

where there is a single relation $\mathscr{H}$ modeling the reduction relation. This condition will validate the usual forward simulation conditional

$$\langle h \rangle [h] P \supset [h] \langle h \rangle P.$$

So there appears to be modal logic of the typed and untyped λ-calculus. In the untyped case, it appears we must have coequations describing formulas since normalization (viewing lambda terms as coding proofs) does not hold.

Looked at in this manner, the typed and untyped λ-calculus is really a process algebra for the modal logic over its terms where its terms are thought of as states.

2.4 Logical Relations

Logical relations in this report are from [17], who attribute it to [18] but actually goes back to Plotkin [19], the latter is a difficult to read paper. A concise definition of a logical relation is one that satisfies

$$\mathscr{R} xy \text{ implies } \mathscr{R}(fx)(fy)$$

for all f in some collection of functions thought of as operations. This is a disguised form of a simulation relation owing to the fact that functions are everywhere defined and single valued. We will use h in place of their f and $\mathscr{F}$ in place of their R so that it does not clash with our default conventions:

$$\mathscr{F} xy \text{ implies } \mathscr{F}(hx)(hy)$$

We generally use the sans serif h,k to refer to domains, and we type the relation as $\mathscr{F} : \mathsf{h} \rightharpoonup \mathsf{k}$. Hence the preceding statement can be generalized a bit by considering two functions h and k in two different domains. Domain are collections of states or worlds:

$$\mathscr{F}xy \text{ implies } \mathscr{F}(hx)(ky)$$

This says that the two functions, h and k are somewhat similar in their actions on their respective domains. One more generalization is

$$\mathscr{F}xy \text{ implies } \mathscr{G}(hx)(ky)$$

where the original can be recovered by letting $\mathscr{G} = \mathscr{F}$.

Lemma 2.4.1 $\mathscr{F}xy$ implies $\mathscr{G}(hx)(ky)$ *is a bisimulation.*

Proof: Assume this last generalized logical relation definition. We will treat h,k as relations and denote them $\mathscr{H},\mathscr{K}$ so that $\mathscr{H}xu$ iff $hx = u$ and $\mathscr{K}yv$ iff $ky = v$. Let $\mathscr{F}xy$ and $\mathscr{H}xu$. Hence $\mathscr{G}(hx)(ky)$. So there is some v, namely $v = ky$, such that $\mathscr{K}yv$ and $\mathscr{G}uv$. Similarly, let $\mathscr{F}xy$ and $\mathscr{K}yv$. Hence $\mathscr{G}(hx)(ky)$. So there is some u, namely $u = hx$, such that $\mathscr{H}xu$ and $\mathscr{G}uv$.

We have just shown that

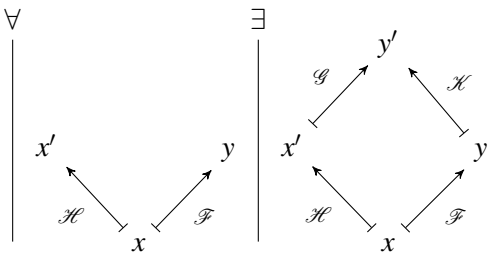
$$\mathscr{H}xu \text{ and } \mathscr{F}xy \text{ implies } \exists v(\mathscr{G}uv \text{ and } \mathscr{K}yv) \qquad \mathscr{K}yv \text{ and } \mathscr{G}xy \text{ implies } \exists u(\mathscr{F}uv \text{ and } \mathscr{H}xu)$$

These are (respectively) forward and converse simulations. Together, they form a bisimulation. ∎

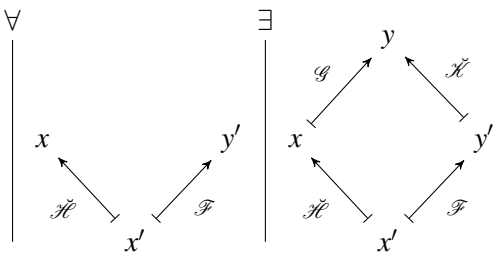
Hence logical relations are simplified bisimulations. More generally,

Name	Frame Condition	Axiom
Forward Simulation	$\mathscr{F}xy$ and $\mathscr{H}xx'$ implies $\exists y'(\mathscr{K}yy'$ and $\mathscr{G}x'y')$	$\langle f\rangle[k]Q \supset [h]\langle g\rangle Q$
Proverse Simulation	$\mathscr{F}x'y'$ and $\mathscr{H}xx'$ implies $\exists y(\mathscr{K}yy'$ and $\mathscr{G}xy)$	$\langle f\rangle[\cdot k\cdot]Q \supset [\cdot h\cdot]\langle g\rangle Q$
Converse Simulation	$\check{\mathscr{F}}yx$ and $\mathscr{K}yy'$ implies $\exists x'(\mathscr{H}xx'$ and $\check{\mathscr{G}}y'x')$	$\langle f\cdot\rangle[h]P \supset [k]\langle g\cdot\rangle P$
Backward Simulation	$\mathscr{F}x'y'$ and $\mathscr{K}yy'$ implies $\exists x(\mathscr{H}xx'$ and $\mathscr{G}xy)$	$\langle f\cdot\rangle[\cdot h\cdot]P \supset [\cdot k\cdot]\langle g\cdot\rangle P$

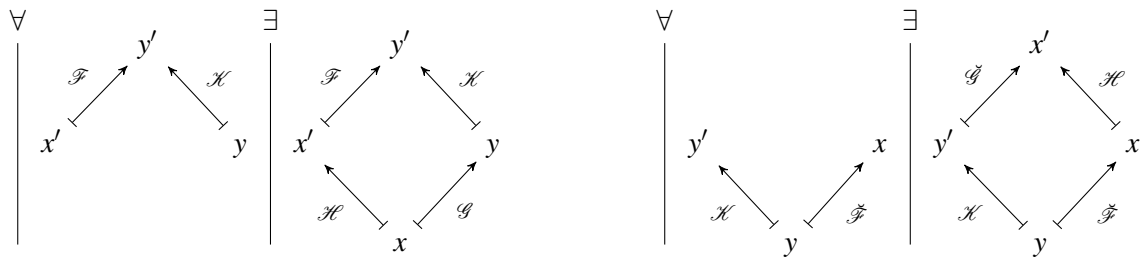
Diagrammatically:



Forward



Proverse



Backward Converse

From [20], pp. 19,

Definition 2.4.2 Let $\langle A, \mathscr{R} \rangle$ and $\langle B, \mathscr{S} \rangle$ be two relational structures. An *isomorphism* between $\langle A, \mathscr{R} \rangle$ and $\langle B, \mathscr{S} \rangle$ is a bijection $f : A \twoheadrightarrow B$ with the property that $\mathscr{R}aa'$ iff $\mathscr{S}(fa)(fb)$ for all $a, a' \in A$.

This appears to be something like a combination of simulations of various flavors. Again we change the notation a bit:

Definition 2.4.3 Let $\langle H, \mathscr{H} \rangle$ and $\langle K, \mathscr{K} \rangle$ be two relational structures, with H and K collections of states and $\mathscr{H}$ and $\mathscr{K}$ relations (respectively) on those state sets. An *isomorphism* between $\langle H, \mathscr{H} \rangle$ and $\langle K, \mathscr{K} \rangle$ is a bijection $f : H \twoheadrightarrow K$ with the property that

$$\mathscr{H}xu \text{ iff } \mathscr{K}(fx)(fu) \text{ for all } x, u \in H.$$

Notice that this is different that the logical relation definition. The two conditions for logical relations and isomorphism (the latter where f is a bijection) are

$$\mathscr{F}xy \text{ implies } \mathscr{G}(hx)(ky) \qquad\qquad \mathscr{H}xx' \text{ iff } \mathscr{K}(fa)(fb).$$

Assume the isomorphism definition, we will convert it to simulations.

Lemma 2.4.4 *The previous definition constitutes a simulation in the left to right direction and a backwards simulation in the right to left direction.*

Proof: We will treat f as a relation and denote it $\mathscr{F}$ so that $\mathscr{F}xy$ iff $fx = y$. We treat the forward direction first. Assume $\mathscr{H}xu$ implies $\mathscr{K}(fx)(fu)$ for all $x, u \in H$. To show this is a simulation, assume $\mathscr{H}xu$ and $\mathscr{F}xy$. So $\mathscr{K}(fx)(fu)$. So there is some v such that $\mathscr{K}yv$ and $\mathscr{F}uv$, namely $fx = y$ and $fu = v$. We have just shown that

$$\mathscr{H}xu \text{ and } \mathscr{F}xy \text{ implies } \exists v(\mathscr{F}uv \text{ and } \mathscr{K}yv).$$

Now for the reverse direction. Assume $\mathscr{K}(fx)(fu)$ implies $\mathscr{H}xu$ for all $x,u \in H$. To show this is a backwards simulation, assume $\mathscr{K}yv$ and $\mathscr{F}uv$. Hence $fu = v$. Since f is a bijection, there is some x such that $\mathscr{F}xy$, so $fx = y$. Therefore $\mathscr{H}xu$. We have just shown that

$$\mathscr{K}yv \text{ and } \mathscr{F}uv \text{ implies } \exists x(\mathscr{F}xy \text{ and } \mathscr{H}xu).$$

■

2.5 Applicative Similarity

This is a notion from Abramsky who attributes it to Robin Miler [21].

Milner

From [21], there are several domains: input, computations, and output, and a total function $\mathscr{F} : D \to D$ ($D = (D_{\text{in}} \cup D_{\text{comp}} \cup D_{\text{comp}})$) with

$$\mathscr{F}(D_{\text{in}} \cup D_{\text{comp}}) \subseteq D_{\text{comp}} \cup D_{\text{out}}.$$

Think of $D_{\text{comp}} = N \times E$, where E is the set of possible state-vector values, and for non-recursive programs N is the finite set of nodes of the flowchart while for recursive programs N is the infinite set of possible states of a pushdown store.

He asserts the following definition:

Definition 2.5.1 A program $\mathscr{A}$ determines its associated partial function

$$\hat{\mathscr{A}} : D_{\text{in}} \to D_{\text{out}}$$

in an obvious way.

In [21] pp. 7:

> Now assume two programs, $\mathscr{A} = \langle D_{\text{in}}, D_{\text{comp}}, D_{\text{out}}, \mathscr{F} \rangle$ and $\mathscr{A}' = \langle D'_{\text{in}}, D'_{\text{comp}}, D'_{\text{out}}, \mathscr{F} \rangle$.
>
> **Definition 2.5.2** Let $\mathscr{R} \subseteq D \times D'$. Then $\mathscr{R}$ is a weak simulation of $\mathscr{A}$ by $\mathscr{A}'$ if
>
> (i) $\mathscr{R} \subseteq D_{\text{in}} \times D'_{\text{in}} \cup D_{\text{comp}} \times D'_{\text{comp}} \cup D_{\text{out}} \times D'_{\text{out}}$
> (ii) $\mathscr{R}\mathscr{F}' \subseteq \mathscr{F}\mathscr{R}$.
>
> Condition (ii) simply states that $\mathscr{R}$ is a weak homomorphism between the algebraic structures $\langle D, \mathscr{F} \rangle, \langle D', \mathscr{F}' \rangle$. This concept is used in automata theory to define the notion of covering – see for example Ginzburg [22][p. 981].
>
> Now denote $\mathscr{R} \cap (D_{\text{in}} \times D_{\text{out}})$ by $\mathscr{R}_{\text{in}}$ and $\mathscr{R}_{\text{out}}$, $\mathscr{R}_{\text{out}}$ similarly so that $\mathscr{R} = \mathscr{R}_{\text{in}} \cup \mathscr{R}_{\text{comp}} \cup \mathscr{R}_{\text{out}}$ and these parts are disjoint.

Theorem 2.5.3 *If $\mathscr{R}$ is a weak simulation of $\mathscr{A}$ by $\mathscr{A}'$ then*

(i) $\mathscr{R}_{\text{in}}\hat{\mathscr{A}'} \subseteq \hat{\mathscr{A}}\mathscr{R}_{\text{out}}$

(ii) $\mathscr{R}^{-1}$ *is a weak simulation of* $\mathscr{A}'$ *by* $\mathscr{A}$

(iii) $\mathscr{R}_{\text{in}}^{-1}\hat{A} \subseteq \hat{\mathscr{A}'}\mathscr{R}_{\text{out}}^{-1}$

The main condition is

$$\mathscr{R} \cdot \mathscr{F}' \subseteq \mathscr{F} \cdot \mathscr{R}$$

where the relational composition operation $\cdot$ is used rather than rely upon juxtaposition. Hence this becomes, in set theoretic relations,

$$x(\mathscr{R} \cdot \mathscr{F}')y \text{ implies } x(\mathscr{F} \cdot \mathscr{R})y.$$

Expanding the relational composition $\cdot$, we get

$$\exists y(\mathscr{R}xy \text{ and } \mathscr{F}'yy') \text{ implies } \exists x'(\mathscr{F}xx' \text{ and } \mathscr{R}xy').$$

Pulling the first nested existential to the outside yields

$$\forall y\big((\mathscr{R}xy \text{ and } \mathscr{F}'yy') \text{ implies } \exists x'(\mathscr{F}xx' \text{ and } \mathscr{R}xy')\big).$$

Since the x and y' are free variables, they are universally quantified so we get

$$\forall y,x,y'\big((\mathscr{R}xy \text{ and } \mathscr{F}'yy') \text{ implies } \exists x'(\mathscr{F}xx' \text{ and } \mathscr{R}xy')\big).$$

In diagrammatic form, this is

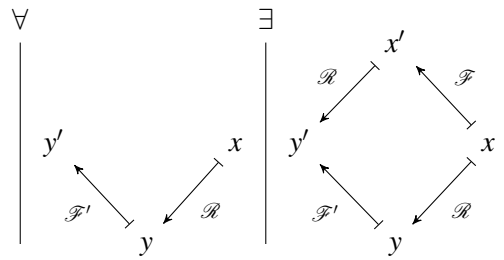


Figure 2.2: Millner Weak Simulation of $\mathscr{A}$ by $\mathscr{A}'$

Note that Milner's locution of weak simulation runs essentially opposite of the way we describe simulation.

Abramsky

The paper [23] is an exposition of the lazy lambda calculus. This calculus underlies Haskell upon which ReWire is built. The difference between the lazy lambda calculus and the call-by-value lambda calculus is the former will allow arguments (to functions) that do not necessarily terminate into a definite value. It may be that the function never needs to evaluate such an argument in which case not attempting to expand it will avoid the entire function to become non-terminating.

The following is from [23] pp. 177, where $\rightharpoonup$ is a partial function (partial map), $\mathrm{dom}(\mathsf{eval})$ is the domain eval, and A^A is the collection of partial maps from A to A:

Definition 2.5.4

(i) A *quasi-applicative transition system* (q-ats) is a structure $\langle A, \mathsf{eval}\rangle$ such that $\mathsf{eval} : A \rightharpoonup A^A$ and $\mathrm{dom}(\mathsf{eval}) \neq A$.

Notation

$$a \Downarrow f \stackrel{def}{=} a \in \mathrm{dom}(\mathsf{eval}) \text{ and } \mathsf{eval}(a) = f,$$
$$a \Downarrow \stackrel{def}{=} a \in \mathrm{dom}(\mathsf{eval}),$$
$$a \Uparrow \stackrel{def}{=} \neg(a \Downarrow).$$

A q-ats is *pointed* if $\exists \bot \in A.\, \mathrm{dom}(\mathsf{eval}) = A \setminus \{\bot\}$.

(ii) Let $\langle A, \mathsf{eval}\rangle$ be a q-ats and $\mathsf{Rel}(A) \stackrel{def}{=} \mathscr{P}(A \times A)$. Define $F : \mathsf{Rel}(A) \to \mathsf{Rel}(A)$ by

$$F(R) \stackrel{def}{=} \{(a,\ b) \mid a \Downarrow f \Rightarrow [b \Downarrow g \text{ and } \forall c \in A.\ f(c)\ R\ g(c)]\}.$$

Then $R \in \mathsf{Rel}(A)$ is called an *applicative bisimulation* if $R \subseteq F(R)$.

There is more from that definition but this in what we are interested now. The following is from [23] pp. 243,

Definition 2.5.5 A relation $\mathscr{R} \subseteq \prod_{i \in \mathscr{I}} K_i$, where each $\mathscr{K}_i \stackrel{def}{=} \langle K_i, \Downarrow_i\rangle$ is an lts (lambda transition system), it is a *logical relation* if

$$\vec{d} \in R \text{ implies } \forall \vec{e} \in R(\vec{d} \cdot \vec{e} \in \mathscr{R}),$$

where $\vec{d} \cdot \vec{e} \stackrel{def}{=} \langle d_i e_i : i \in \mathscr{I}\rangle$.

where $d_i e_i$ is a lambda application of d_i to e_i.

The following is from [24]:

Definition 2.5.6 (Applicative Similarity) An *applicative simulation* is a binary relation $\mathscr{S}$ between closed λ-terms satisfying if $P \mathscr{S} P'$ and $P \Downarrow \lambda x.M$, then $P' \Downarrow \lambda x.M'$ for some M' such that for all closed Q, $M[Q/x] \mathscr{S} M'[Q/x]$.

What is interesting about this definition is that Abramsky is reaching inside a closed term and pulling it apart using a universal quantifier. Diagrammatically, the form is

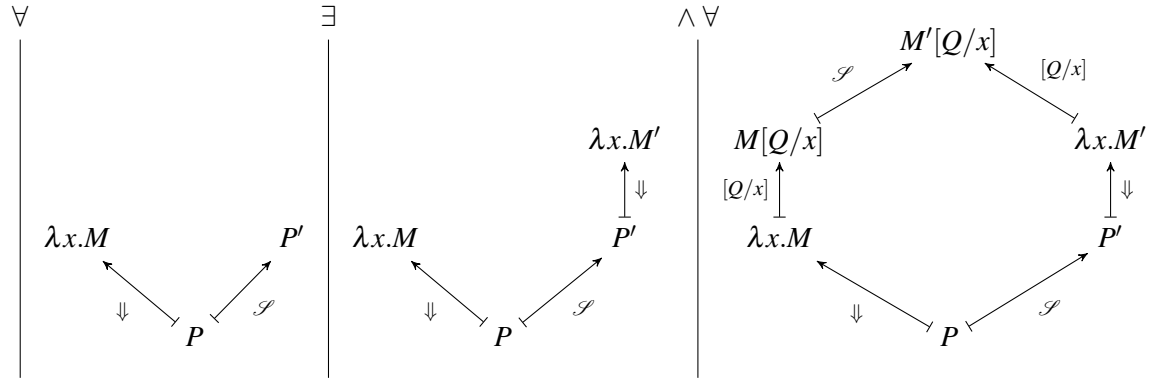


Figure 2.3: Applicative Simulation

The tricky bit is that the last universal quantifier is quantifying over all closed terms Q. The $[Q/x]$ is a function in the form

$$M \mapsto M[Q/x].$$

Hence the last universal quantifier is quantifying over an entire collection of functions. This is a second-order concept that is outside of the purview of modal logic. The best that can be done is that the quantifier is left to the meta-logic where the semantics lives.

This above form is a bit hard to handle modally. Just a brief aside, Abramsky is not even requiring that $(\lambda x.M)\ \mathscr{S}\ (\lambda x.M')$ obtain. Regardless, the prescription can be altered to

Definition 2.5.7 (Modified Applicative Similarity) An *applicative simulation* is a binary relation $\mathscr{S}$ between closed λ-terms satisfying if $P \mathscr{S} P'$ and $P \Downarrow \lambda x.M$, then $P' \Downarrow \lambda x.\ M'$ for some M' such that for all closed Q, $M\ [Q/x]\ N$ implies there is some N' such that $M'\ [Q/x]\ N'$ and $N \mathscr{S} N'$.

Stripping the lambda calculus from the above and converting it to the Kripke frames from Distributed Logic where the $\mathscr{F}$ is a co-occurrence relation modeling $\mathscr{S}$, $\mathscr{H}$ is a next state relation modeling $P \Downarrow \lambda x.\ M'$, $\mathscr{H}'$ and $\mathscr{K}'$ model $[Q/x]$:

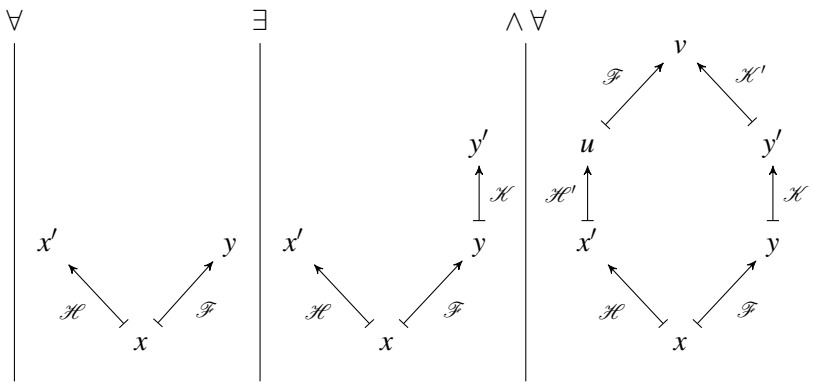


Figure 2.4: Applicative Simulation

The diagram should come to

$$\mathscr{G}xx' \text{ and } \mathscr{F}xy \text{ implies } \exists y'\big(\mathscr{K}yy' \text{ and } \forall u,v(\mathscr{G}'x'u \text{ and } \mathscr{K}'y'v \text{ implies } \mathscr{F}uv)\big)$$

The modal formula should be

$$\langle f\rangle[k]\langle k'\rangle Q \overset{\mathsf{h}}{\supset} [h]\langle h'\rangle\langle f\rangle Q.$$

The soundness proof is

1	$x \in_{\mathsf{h}} \langle f\rangle[k]\langle k'\rangle Q$	assume
2	$\mathscr{F}xy$ and $y \in_{\mathsf{k}} [k]\langle k'\rangle Q$	def. $\langle f\rangle$ for some y, line 1
3	$\quad \mathscr{G}xx'$	assume
4	$\mathscr{G}xx'$ and $\mathscr{F}xy$	$\wedge$-Elim, $\wedge$-Intro, lines 2, 3
5	$\mathscr{K}yy'$ and $\forall u,v(\mathscr{G}'x'u$ and $\mathscr{K}'y'v$ implies $\mathscr{F}uv)$	Applicative Simulation for some y', line 4
6	$\mathscr{K}yy'$ and $y \in_{\mathsf{k}} [k]\langle k'\rangle Q$	$\wedge$-Elim, $\wedge$-Intro, lines 2, 5
7	$y' \in_{\mathsf{k}} \langle k'\rangle Q$	def. $[k]$, line 6
8	$\mathscr{K}'y'v$ and $v \in_{\mathsf{k}} Q$	def. $\langle k'\rangle$ for some v, line 7
9	$\quad \mathscr{G}'x'u$	assume
10	$\mathscr{G}'x'u$ and $\mathscr{K}'y'v$	$\wedge$-Elim, $\wedge$-Intro, lines 8, 9
11	$\mathscr{G}'x'u$ and $\mathscr{K}'y'v$ implies $\mathscr{F}uv$	$\forall$-Elim, $\wedge$-Elim, lines 5, 10
12	$\mathscr{F}uv$	$\supset$-Elim, lines 10, 11
13	$\mathscr{F}uv$ and $v \in_{\mathsf{k}} Q$	$\wedge$-Elim, $\wedge$-Intro, lines 8, 12
14	$u \in_{\mathsf{h}} \langle f\rangle Q$	def. $\langle f\rangle$, line 13
15	$x' \in_{\mathsf{h}} [h']\langle f\rangle Q$	def. $[h']$, line 9
16	$x \in_{\mathsf{h}} [h]\langle h'\rangle\langle f\rangle Q$	def. $[h]$, line 3

The point is that Applicative Simulation is a more rigorous notion than simulation *simpliciter*. Essentially, Applicative Simulation forces the simulation to look farther into the next state relations, $\mathscr{H}$ and $\mathscr{H}'$ for locality h and $\mathscr{K}$ and $\mathscr{K}'$ for locality k, before declaring two states similar. This will be of use, although we do not cover it in this report, for process algebras where we will need the extra conditions to skip past the functional execution code of a device after it receives an input. It will contribute to understanding when one device (component) can track what another device (component) is doing in an FPGA application. This has implications for high assurance where that sort of tracking may be unwanted behavior.

2.6 Evaluation Strategies

Different evaluation strategies yield different reduction relations. Hence it is plausible that a modal algebra could pick apart, say, call-by-value and call-by-name. Given that applicative simulation is very tied to call-by-name from Abramsky's work, and we now have a modal formula expressing it, there's a good chance that call-by-value fails to satisfy this modal formula. Call-by-value might underly some other modal formula that fails for call-by-need.

Call-by-need is likely underlying most process algebras and allows arguments to functions (lambda abstractions) to not necessarily terminate. Call-by-value requires that arguments to functions terminate. From [25], a term of the form $(\lambda x.\ M)$ is an *abstraction*; of the form (MN) is a combination. A term is a *value* iff it not a combination.

Call-by-value is likely to satisfy Applicative Simulation because the universal quantifier at the end will skip over an x' where there is no u such that $\mathscr{H}'x'u$. In effect, it will ignore pairs of substitutions where the h computation fails to converge while the k computation does converge. The use of next-state relations rather than next-state functions is part of the problem. They are too loose because there is no way to say that there is no follow on state. Another issue is the $\mathscr{H}'$ and $\mathscr{K}'$ stood for all possible $[Q/-]$ substitutions. To bring back individual substitutions where one might cause a lambda term to diverge would mean bringing back second-order quantification. This might be glossed over by relegating that quantification to the meta-logic in which we express Applicative Simulation. The lambda calculus theorists appear to have simulations for call-by-value, but to delve into that literature here would take us too far afield for this report.

3. APPLICATIONS

Systems-on-a-chip, which includes FPGA applications, may contain several computing cores, some memory, buses, I/O ports, etc. They are notoriously difficult to handle in terms of security. The overriding feature of course is the distribution of the components around the chip. One can only go so far using equational logic and typing systems for the semantics of the designs for system-on-a-chip. Eventually, the typing schemes become so complex that it is better to use a more general logic. Our work in system-on-a-chip involves a language for hardware, ReWire, with a formal semantics. For our purposes, that semantics is a not a good fit; rather a process algebra semantics is a better alternative. The process algebra evaluation of distributed logic statements is a work in progress but we feel there is much potential here.

There is a notion of labeled process algebras with static or dynamic labels. The labels correspond to components in an application. The static approach is in [26]. There are a number of publications in the area so this may not be the seminal paper. It concentrates on static labels which remain fixed throughout the evaluation of algebraic terms. We presume the dynamic approach allows one to pass labels around, something like we would need for passing device addresses around. Our initial applications are not complicated

enough for the dynamic approach. There does seem to be a rough equivalence between the two approaches according to the literature.

We use the term *event annihilation* to refer to two process algebra terms using synchronized communication so that each may advance. In symbols

$$a.E \mid \overline{a}.E = \tau.(E \mid E).$$

The τ is essentially a no-operation (as a mere algebraic artifact) could be elided but it seems preferable to leave it in the expressions to make clear which events were executed. The $\mid$ is a parallel operator that separates two computations running in parallel. The equals sign is not to be thought of as a mathematical equality. There is a direction from left to right through the equals sign.

Components are modeled as processes. To allow for future expansion, i.e., a hardware process spanning components in the sense that the components conspire to advance a calculation that spans the components, we will keep the terms *component* and *processes* separate. A component in an algebra looks like an algebraic term as does a process. At some point we may wish to use different syntax. For now, we will not use hardware processes but only hardware components and refer to them as such.

There are some assumptions that are made to simplify the exposition in this section. The assumptions may be relaxed with a rise in algebraic complexity. The assumptions are:

A1. Two components on different clock cycles will require some algebraic apparatus since each component may be connected to other components on the same clock cycle. In effect, there would be two collections of components, each collection has its own clock cycle.

A2. If events on the same clock cycle can annihilate each other, then they must annihilate each other. At each strike of the clock, all possible progress must occur.

A3. Removing τ events takes no underlying clock cycle, they are merely algebraic artefacts.

3.1 A Simple Application

More specifically, in [27], Robin Milner uses the following diagram

to illustrate bisimulation. The initial coequations he uses are

$$\mathscr{H} \stackrel{def}{=} a.\mathscr{H}' \qquad \mathscr{K} \stackrel{def}{=} r.\mathscr{K}'$$
$$\mathscr{H}' \stackrel{def}{=} \overline{r}.\mathscr{H} \qquad \mathscr{K}' \stackrel{def}{=} \overline{b}.\mathscr{K}$$

In English, $\mathscr{H}$ receives a and then communicates with $\mathscr{K}$ via r and $\overline{r}$, and then $\mathscr{K}$ emits $\overline{b}$. These are coequations because if you look closely, they seem to be chasing their tails with what is known as *tail recursion*. They are still equations in the normal sense. However, for solutions, we must fix on a set theory. Well-founded (wf) set theory would say that the solutions to these equations are all empty. Non-well-founded (nwf) set theory does permit solutions. The difference is in fixpoints to the the sets modeling the solutions. In well-founded set theory, there is no difference between least and greatest fixpoints. In nfs, there is a difference and we take the greatest fixpoints of the resulting set coequations. We will not go into the intricacies of set theories here other than to note that in nwf solutions are guaranteed to exist for the kinds of coequations we use.

One has to be a bit careful of the process algebra equations, they can destroy the distribution and in effect, cause one to take the cross product of states in the two processes. Avoiding that promises to make a model checking algorithm for distributed logic statements cheaper than a traditional model checker.

Assume there is an environment supplying $\overline{a}$ and b via two coequations:

$$\mathscr{A} = \overline{a} \,.\, \mathscr{A}$$
$$\mathscr{B} = b \,.\, \mathscr{B}$$

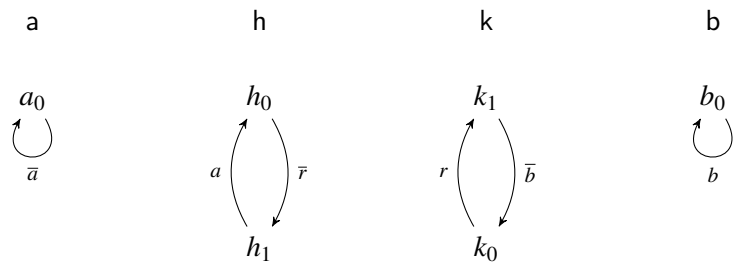
We execute assuming $[r]$ represents co-occurrence relation $\mathscr{R} : \mathsf{h} \rightharpoonup \mathsf{k}$ where h, k are the localities (or components, if you wish) holding the state sets H and K respectively for the relations $\mathscr{H}, \mathscr{K}$ also respectively. The terms trace out a trajectory in the distributed state space $H \times K$ (the cross-product of the state space H and K). P is some proposition of k, $[r]P$ is that proposition's image under $[r]$ in h:

$$\begin{aligned}
(\mathscr{A} \mid \mathscr{H} \mid \mathscr{K} \mid \mathscr{B}) \models_h [r]P \text{ iff } & (\overline{a} \,.\, \mathscr{A} \mid a \,.\, \mathscr{H}' \mid \overline{b} \,.\, \mathscr{K}' \mid b \,.\, \mathscr{B}) \models_h [r]P \\
\text{iff } & (\tau \,.\, (\mathscr{A} \mid \mathscr{H}') \mid \tau \,.\, (\mathscr{K}' \mid \mathscr{B}) \models_h [r]P) \\
\text{iff } & (\mathscr{A} \mid \mathscr{H}' \mid \mathscr{K}' \mid \mathscr{B}) \models_h [r]P \\
\text{iff } & (\mathscr{A} \mid r \,.\, \mathscr{H} \mid \overline{r} \,.\, \mathscr{K} \mid \mathscr{B}) \models_h [r]P \\
\text{iff } & (\mathscr{A} \mid \tau \,.\, (\mathscr{H} \mid \mathscr{K}) \mid \mathscr{B}) \models_k P \\
\text{iff } & (\mathscr{A} \mid \mathscr{H} \mid \mathscr{K} \mid \mathscr{B}) \models_k P
\end{aligned}$$

There is an underlying assumption, that if events can annihilate each other, then they must annihilate each other. This models that all the components in the above term are on the same clock signal. At each strike of the clock, all possible progress must occur. Two components on different clock cycles will require more algebraic apparatus since each component may be connected to other components on the same clock cycle. In effect, there would be two collections of components, each collection has its own clock cycle. We will not treat multiple clock scenarios in this report.

3.2 An Expanded Simple Application

Using the example of the previous section, it can be expanded to reveal some features of an evaluation system. First, we will alter the previous diagram to be be a collection of state diagrams:



where the environment localities a and b have been added. The above diagrams are subject to the equations

$$\mathscr{A} = \overline{a}\,.\,\mathscr{A} \qquad \mathscr{H} = a\,.\,\mathscr{H}' \qquad \mathscr{K} = r\,.\,\mathscr{K}' \qquad \mathscr{B} = \overline{b}\,.\,\mathscr{B}$$
$$\mathscr{H}' = \overline{r}\,.\,\mathscr{H} \qquad \mathscr{K}' = \overline{b}\,.\,\mathscr{K}$$

We will use the coformulas

$$A = [\overline{a}]A$$
$$P = [a][f][r][\cdot f \cdot]P$$
$$Q = [\cdot f \cdot][\overline{r}][f][b]Q$$
$$B = [\overline{b}]B$$

The process algebra formulas on the left side of the $\models$ satisfaction relation below each correspond to states in their respective localities. The left side uses $\mid$ to separate localities while the right side uses ; in a similar fashion. Use of $\mid$ on the right to indicate parallel evaluation of formulas was too disconcerting. Each lane (locality) on the right contains a sequence of formulas separated by commas, to be considered conjoined. They get to move between lanes by losing their distributed modalities. The idea is that if $x \models [f]S$ and $\mathscr{F}xy$, then $y \models S$. The x and y are in different lanes and we would have $x \mid y$ on the left side of the $\models$ relation. There is no use of the conjunctive sequences in the example below.

$\mathscr{F} : \mathsf{h} \rightharpoonup \mathsf{k}$ is a co-occurrence relation. The modalities corresponding to $\mathscr{F}$ are $[f]$ and $[\cdot f \cdot]$. There are actually co-occurrence relations possible between any pair of localities, but we will only use $\mathscr{F}$.

The next-state Kripke relations adhere to their lanes. Hence $\mathscr{A}$ is the local relation for a and similarly for the rest. However, the h lane contains subrelations for those involving the a and r particles. Particles can never leave their lanes. We could annotate the $\mathscr{H}$ relation with $\mathscr{H}_a$ and $\mathscr{H}_r$ satisfying the inclusions $\mathscr{A}, \mathscr{R} \subseteq \mathscr{H}$, and to be complete, $\mathscr{H} = \mathscr{H}_a \cup \mathscr{H}_r$. The use of $\mathscr{H}$ washes or abstracts out the difference between the two relations. We could use partially-ordered modalities if it were necessary to represent these inclusions modally. There are similar statements that can be made for the localities k and b.

We will use $[a]$ to indicate all transitions with respect to a in the $\mathscr{H}$ relation induced by the particle a. The left side lane containing an algebraic expression for the h lane effectively indexes the $\mathscr{H}$ relation at the state represented by that algebraic expression. The implication is that the particle a can never be associated with another lane. If necessary, we could use a_{h} to indicate the a particle in the h lane.

Now we compute:

$$
\begin{array}{rll}
(\overline{\mathscr{A}} \mid \mathscr{H} \mid \mathscr{K} \mid \mathscr{B}) \models_h A\,;P\,;Q\,;B & & \text{initial state} \\
(\overline{a}\,.\,\mathscr{A} \mid a\,.\,\overline{r}\,.\,\mathscr{H} \mid r\,.\,b\,.\,\mathscr{K} \mid \overline{b}\,.\,\mathscr{B}) \models_h [\overline{a}]A\,;[a][f][r][\cdot f\cdot]P\,;[\cdot f\cdot][\overline{r}][f][b]Q\,;[\overline{b}]B & & \text{unwind coequations} \\
(\overline{\mathscr{A}} \mid \overline{r}\,.\,\mathscr{H} \mid r\,.\,b\,.\,\mathscr{K} \mid \overline{b}\,.\,\mathscr{B}) \models_h A\,;[f][r][\cdot f\cdot]P\,;[\cdot f\cdot][\overline{r}][f][b]Q\,;[\overline{b}]B & & \text{satisfaction } [\overline{a}] \text{ and } [a] \\
(\mathscr{A} \mid \overline{r}\,.\,\mathscr{H} \mid r\,.\,b\,.\,\mathscr{K} \mid \overline{b}\,.\,\mathscr{B}) \models_h A\,;[\overline{r}][f][b]Q\,;[r][\cdot f\cdot]P\,;[\overline{b}]B & & \text{satisfaction } [\cdot f\cdot] \text{ and } [f] \\
(\mathscr{A} \mid \mathscr{H} \mid b\,.\,\mathscr{K} \mid \overline{b}\,.\,\mathscr{B}) \models_h A\,;[f][b]Q\,;[\cdot f\cdot]P\,;[\overline{b}]B & & \text{satisfaction } [\overline{r}] \text{ and } [r] \\
(\mathscr{A} \mid \mathscr{H} \mid b\,.\,\mathscr{K} \mid \overline{b}\,.\,\mathscr{B}) \models_h A\,;P\,;[b]Q\,;[\overline{b}]B & & \text{satisfaction } [f] \text{ and } [\cdot f\cdot] \\
(\mathscr{A} \mid \mathscr{H} \mid \mathscr{K} \mid \mathscr{B}) \models_h A\,;P\,;Q\,;B & & \text{satisfaction } [b] \text{ and } [\overline{b}]
\end{array}
$$

Note that the only way to advance is to go through particle annihilation or a co-occurrence relation. We must still accommodate each lane able to proceed internally without any communication to the outside. However, any internal computation is merely function application and that will be handled further on. A process term could signal but only if it used a parallel operator. In that case, the two terms communication would exist in the same lane, but it would be implicit that they are on different clock ticks. Actually, sending a signal and receipt of that signal occur on different clock ticks. More on this will be supplied later in this report.

The alternative it to treat the process algebra as abstracting away local events (computation). That is likely to interfere with ferreting out high assurance arguments, say, where we are satisfying a simulation formula. Let us analyze the following simulation formula:

$$\langle f\rangle[k]Q \overset{\mathsf{h}}{\supset} [h]\langle\cdot f\cdot\rangle Q.$$

Buried in the analysis above was that we needed to look at all execution paths for necessity. However, the formulas only depict a single possible execution path for each lane so there is no real difference between necessity and possibility.

There is another issue. Just considering the relationship between a and h, the relations $\overline{\mathscr{A}}$ and $\mathscr{A}$ are linked in that a and h cannot advance past the $\overline{a},a$ annihilation unless the both do simultaneously. This feature is not captured in the logic, or *specified* would be a better word. The relationship should be, for $\mathscr{C} : \mathsf{a} \rightharpoonup \mathsf{h}$ the co-occurrence relation between a and h (we need better ways naming relations),

$$\overline{\mathscr{A}}\breve{}\,uu \text{ and } \mathscr{C}\,ux \text{ and } \mathscr{A}\,xv \text{ implies } \mathscr{C}\,uv.$$

Writ large, any annihilation must be constrained by a co-occurrence pair of states before the annihilation resulting in the co-occurrence relation holding afterward. We know how to do this, the following formula prescribes it:

$$[c]S \overset{\mathsf{a}}{\supset} [\cdot\overline{a}\cdot][c][a]S$$

in that the previous relational condition underwrites the soundness of this formula as an axiom of this producer-consumer example.

This is ungainly in that S is not in the collection of formulas we are using. It will not do to add it and use a sequence in the a lane since the A formula must be constrained by it. One might be tempted to use A

and reverse the direction of $\mathscr{C}$, and what amounts to the same thing, to use the backward modality $[\cdot c\cdot]$. This gives us

$$[\cdot c\cdot]A \overset{\mathsf{h}}{\supset} [\cdot a\cdot][\cdot c\cdot][\bar{a}]A$$

However, the P formula must be constrained by the interaction with a as well. The only recourse seems to be to use

$$[c]S \overset{\mathsf{a}}{\supset} [\cdot\bar{a}\cdot][c][a]S$$

Now our naming conventions are becoming intolerable as seen by the double accents over the $\mathscr{A}$. Also, $\overline{\mathscr{A}}$ looks like the Boolean complement of $\mathscr{A}$.

Stepping back a bit. Essentially we are using the satisfaction relation within a classical metalogic. It would then seem that we should just jump in with both feet and use the natural deduction system of Barwise and Etchemendy [28]. Each step contains a satisfaction formula. We allow intermediate steps to be underwritten by process algebra reductions. Universal quantifiers, seeing as we are operating in a finite world, come down to being large case statements for each nerve (thread of states, necessarily in a binary relation) in a relation. Possibility quantifiers only require a single nerve.

The soundness proof of the above formula in the presence of an $\mathscr{F}$ being a simulation relation is:

1	$x \in_{\mathsf{h}} \langle f\rangle[k]Q$	assume
2	$\mathscr{F}xy$ and $y \in_{\mathsf{k}} [k]Q$	def. $\langle f\rangle$ for some y, line 1
3	$\mathscr{H}xx'$	assume
4	$\mathscr{F}xy$ and $\mathscr{H}xx'$	$\wedge$-Elim, $\wedge$-Intro, lines 2, 3
5	$\mathscr{K}yy'$ and $\mathscr{F}x'y'$	Simulation Condition for some y', line 4
6	$y' \in_{\mathsf{k}} Q$	def. $[k]$, lines 2, 5
7	$x' \in_{\mathsf{h}} \langle f\rangle Q$	def. $\langle f\rangle$, lines 5, 6
8	$x \in_{\mathsf{h}} [h]\langle f\rangle Q$	def. $[h]$, line 3

We attempt to combine this with the intensional Kripke structure described by the algebraic co-equations. From the process algebra computation above, we can read off which terms are in the relations:

$$\begin{aligned}\mathscr{F} = \{&\langle a\,.\,\bar{r}\,.\,\mathscr{H}, r\,.\,b\,.\,\mathscr{K}\rangle,\\ &\langle \bar{r}\,.\,\mathscr{H}, r\,.\,b\,.\,\mathscr{K}\rangle,\\ &\langle \mathscr{H}, b\,.\,\mathscr{K}\rangle\}\end{aligned}$$

where

$$\langle a\,.\,\bar{r}\,.\,\mathscr{H}, r\,.\,b\,.\,\mathscr{K}\rangle = \langle \mathscr{H}, \mathscr{K}\rangle.$$

The relations $\mathscr{H}$ and $\mathscr{K}$ are similar but now we are looking from one line to the next in the algebraic proof. These relations must be divided up into the relations associated with a and $\overline{r}$, and with b and r respectively:

$$\mathscr{H}_a = \{\langle a\,.\,\overline{r}\,.\,\mathscr{H}, \overline{r}\,.\,\mathscr{H}\rangle\} \qquad \mathscr{H}_{\overline{r}} = \{\langle \overline{r}\,.\,\mathscr{H}, a\,.\,\overline{r}\,.\,\mathscr{H}\rangle\},$$

and

$$\mathscr{K}_b = \{\langle b\,.\,\mathscr{K}, r.b\,.\,\mathscr{K}\rangle\} \qquad \mathscr{K}_r = \{\langle r.b\,.\,\mathscr{K}, b\,.\,\mathscr{K}\rangle\}.$$

To be explicit, the simulation condition is

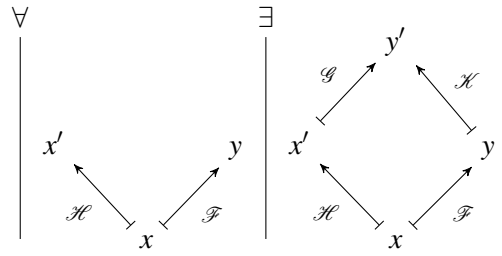


Figure 3.1: Forward Simulation Condition

where $\mathscr{G}$ is $\mathscr{F}$. The $\mathscr{G}$ has been left in in the hopes it will be of use later. There is at least one use case. The simulation condition does not care what relations are used. It is possible that for spider (a form of runtime monitor) purposes to divide the states at h into two groups and the states at k into two groups. Let $\mathscr{H}$ be from states in the first group to states in the second group and similarly for k. Now let $\mathscr{F}$ be a co-occurrence relation between states in the first group of h and the first group of k. Similarly, let $\mathscr{G}$ be the co-occurrence relation between the second two groups. This might be a bit contrived but maybe it has a use.

We note that $\mathscr{H}$ and $\mathscr{F}$ must agree on the first element. This is somewhat ignored in the form of the proof where the simulation antecedent would simply fail to be met and hence the proof could not succeed. The first-order logic does not know anything about the particular condition we are working upon. As a result, the first element of the h lane must be $\overline{r}\,.\,\mathscr{H}$.

1	$(\overline{a}\,.\,\mathscr{A} \mid a\,.\,\overline{r}\,.\,\mathscr{H} \mid r.b\,.\,\mathscr{K} \mid \overline{b}\,.\,\mathscr{B}) \models^{\mathsf{h}} \langle f\rangle[k]Q$	assume
2	$\mathscr{F}(\overline{r}\,.\,\mathscr{H})(r.b\,.\,\mathscr{K})$ and $r.b..\mathscr{K} \models^{\mathsf{k}} [k]Q$	def. $\langle f\rangle$ for some y, line 1
3	$\mathscr{H}_r(\overline{r}\,.\,\mathscr{H})(\mathscr{H})$	assume
4	$\mathscr{F}(\overline{r}\,.\,\mathscr{H})(r.b\,.\,\mathscr{K})$ and $\mathscr{H}(\overline{r}\,.\,\mathscr{H})(\mathscr{H})$	$\wedge$-Elim, $\wedge$-Intro, lines 2, 3
5	$\mathscr{K}(r.b\,.\,\mathscr{K})(b\,.\,\mathscr{K})$ and $\mathscr{F}(\mathscr{H})(b\,.\,\mathscr{K})$	SC for some y', line 4
6	$(b\,.\,\mathscr{K}) \models^{\mathsf{k}} Q$	def. $[k]$, lines 2, 5
7	$\mathscr{H} \models^{\mathsf{h}} \langle f\rangle Q$	def. $\langle f\rangle$, lines 5, 6
8	$(\overline{r}\,.\,\mathscr{H}) \models^{\mathsf{h}} [h]\langle f\rangle Q$	def. $[h]$, line 3

Note that there is no sense of particle annihilation with respect to r in this proof although it is required to set up the relations. Annihilation at this point is not being expressed in the logic.

3.3 Non-Wellfounded Sets and Coformulas

Consider

$$P = [f]P.$$

Notice that this is not

$$P \equiv [f]P,$$

which comes to

$$(P \supset [f]P) \wedge ([f]P \supset P).$$

The $=$ is syntactic equality, not logical bi-implication. For any classical propositional context, $\Gamma(-)$, the logic cannot tell the difference from swapping in P or $[f]P$ to yield $\Gamma(P)$ and $\Gamma([f]P)$ under either $=$ (certainly) or $\equiv$, the latter because P and $[f]P$ are truth functionally equivalent.

The valuation condition is

$$\begin{aligned} x \models P &\text{ iff } x \in P \\ &\text{iff } x \in [f]P \\ &\text{iff } \forall y(\mathscr{F}xy \text{ implies } y \in P) \end{aligned}$$

where the first line should technically read

$$x \models P \text{ iff } x \in [\![P]\!]$$

but where we introduce the sleight of hand to make the formulas easier to read.

From Abramsky and referencing [29], there are four axioms that define a semilattice for a fragment of CCS:

- $x+0=x$
- $x+y=y+x$
- $x+(y+z)=(x+y)+z$
- $x+x=x$

An algebra satisfying these axioms, say $(S,0,+)$ is a semilattice. He notes there is no prefixing operator and wished to consider the free algebra with no generators, which seems a bit odd since 0 is certainly an element of any algebra satisfying these axioms. He also notes that [30] uses a free Boolean algebra here with one generator.

He then considers the following semantics (where his P and Q have been to p and q)

$$\begin{aligned}[[0]] &= \emptyset \\ [[p+q]] &= [[p]] \cup [[q]] \\ [[e \cdot p]] &= \{[[p]]\}\end{aligned}$$

Given a collection of process formulas, all involving P and Q and the action e, a model with these operations, i.e., 0, $+$, and $x \mapsto \{x\}$ satisfies the semilattice equations. The way he connects the two sets of equations is via (where I have changed his $[P]$ into $[[p]]$ since there is no definition for $[P]$ and I presume he means $[[P]]$)

$$[[p]] \in [[q]] \iff \exists r \,.\, q = e \cdot p + r.$$

Hence $[[q]]$ also contains whatever $[[r]]$ comes to. This is essentially the semantics that [20] use for Kripke models.

We can generalize a bit by using, say, e_1 and e_2 (more than a single e) where upon we must rely upon urelements as in [20], i.e.,

$$[[e \cdot p]] = \{[[p]], e\}$$

where the symbol e is overloaded to be both a particle in the algebra and an urelement in the semantics. This amounts to allowing more than one "next state" relation in the Kripke model version, say $\mathscr{H}_1$ and $\mathscr{H}_2$. This situation might occur if we want to model the effect of two different inputs and the next state is determined by which input occurs or which is recognized.

The interpretation of $[f]P$ is (as usual)

$$\begin{aligned}[[[f]P]] &= \{x \mid \forall y(\mathscr{F}xy \text{ implies } y \in P)\} \\ &= \{x \mid \forall y(y \cup \{f\} \in x \text{ implies } y \in P)\}\end{aligned}$$

where f is a urelement.

The canonical model might appear a bit odd. We start with a join semilattice with a bottom element and use all the ideals. Hence typical ideal satisfies

$$a, b \in x \text{ iff } a + b \in x,$$

where a, b are urelements. So x is rather flat, no internal set structure... yet. Now we must add to x all the y accessible under $\mathscr{F}$:

$$(\!|f|\!)(x) = x \cup \{\{y, f\} \mid \mathscr{F}xy\}$$

where f is a urelement and $(\!|f|\!)$ turns f into an accessibility function. We now have two collections of urelements, those from the lattice and those from encoding the next state relations.

Just for the record, Abramsky [31] contains axioms for when $[-]$ and $\langle - \rangle$ live on a distributive lattice since then they are not Boolean duals of each other. A suitable weakening of those should work for the semilattice case we have here.

3.4 Non-Wellfounded Sets and Domains

In [32], they say that a set algebra has the same operations as the semantics from Abramsky citing Milner [29]:

> Our starting point is the observation that the set HF of well-founded, hereditarily finite sets can be characterized in algebraic terms. Consider HF as an algebra whose operations are the constant 0, the binary operation union, and the unary operation that forms a singleton set by taking x to $\{x\}$. This structure satisfies familiar algebraic laws. In particular, it is a commutative monoid whose binary operation, union, is idempotent. In the appropriate category of algebras, HF is an initial algebra. Our idea is to enlarge HF to a structure satisfying the same laws and in which equations such as $x = \{x\}$ have unique solutions. The study of abstract data types provides a natural choice for such a structure, namely, an initial continuous algebra. In such an algebra, there are unique solutions to systems of equations of the kind that interest us. The initial continuous algebra we construct is our algebra W of partial sets.

The semilattice operations are a commutative monoid. As they go on to mention, there are initial continuous algebras but those are too abstract for their purposes and also involve congruence classes of trees. They rather start with a preordered algebra of *protosets* which they then complete (ideal completion) to yield the initial algebra of partial sets. The partial sets contain the non-wellfounded sets as ideal elements at the top of approximating chains of protosets.

3.5 Signaling using Three-Place Relations

There is an oddity in the process algebras. If all possible annihilations are not done from step to step, then formulas in co-occurrence before any one annihilation need not have any reducts of those formulas in co-occurrence after annihilation. This can occur if one formula gets its annihilation while being in co-occurrence with another yet both must receive their annihilation to be in co-occurrence after. In short, the process algebra is not picking up the nuances of FPGA state machines. Put another way, without the assumption of all possible reductions are done from step to step, reductions are not occurring in lock step in some cases where they must necessarily occur in lockstep.

An added complication is that a signal can be received by more than one component. This seems to imply that a relation modeling the signal should be n-ary. Only one component can drive a signal (unless there is some design error), so it sounds like the modeling relation has a single input. Let us consider the three place relation

$$\mathscr{R}^{\mathsf{hlk}}xyz$$

where $z \in \mathsf{k}$ is considered the input. We have a local algebraic term defined

$$z \in_{\mathsf{k}} A \overset{\mathsf{hlk}}{\circ} B \text{ iff } \exists x, y(x \in_{\mathsf{h}} A \text{ and } y \dashv B \text{ and } \mathscr{R}^{\mathsf{hlk}}xyz).$$

There is nothing specifically about $\mathscr{R}^{\mathsf{hlk}}$ that makes it a signaling relation as opposed to a co-occurrence relation. This issue is dealt with in the next subsection. A distributed logic whose semantics uses three-place relations is in [7].

3.6 Generalized Simulations and Signaling Relations

First we need to define co-occurrence relations. Assume two components as localities h and k. We wish to construct a co-occurrence relation $\mathscr{F} : \mathsf{h} \rightharpoonup \mathsf{k}$. The two components are assumed to be state machines and have two initial collections of states H_0 and K_0 respectively. We use an inductive definition for co-occurrence:

$$\begin{aligned}\mathscr{F}_0 &= \{\langle x,y\rangle \mid x \in_{\mathsf{k}} H_0 \text{ and } y \in_{\mathsf{k}} K_0\},\\ \mathscr{F}_{n+1} &= \{\langle x',y'\rangle \mid \exists x,y(\langle x,y\rangle \in \mathscr{F}_n \text{ and } \mathscr{H}xx' \text{ and } \mathscr{K}yy')\},\\ \mathscr{F} &= \bigcup_n \mathscr{F}_n.\end{aligned}$$

The intuition is that any x,y in the initial state sets H_0, K_0 respectively must form a pair in the co-occurrence relation. This is assuming that each machine can start in any of its start states. One can always limit H_0, K_0 to be singleton sets.

Next, we realize that no state is a dead end, i.e., each component will make some transition from any state (even if only to the same state):

$$\forall x,y,\exists x',y'(\mathscr{H}xx' \text{ and } \mathscr{K}yy').$$

Another way of stating this is as a liveness condition for the entire collections of states:

$$H \times K = \{\langle x,y\rangle \mid \exists x',y'(\mathscr{H}xx' \text{ and } \mathscr{K}yy')\}.$$

The relation $\mathscr{F}$ is a bisimulation. Let $\mathscr{F}xy$, then there is some n such that $\mathscr{F}_n xy$. Assume $\mathscr{H}xx'$, then there is some y' such that $\mathscr{K}yy'$. Hence $\mathscr{F}_{n+1}x'y'$ and so $\mathscr{F}x'y'$. The other direction of the bisimulation argument is similar.

Notice that the notion of co-occurrence relies upon initial states. Co-occurrence does not appear coherent if only the mere state sets H and K are given under the assumption that every state will transist to some state. There is a related notion of starting from mere state sets and declaring *co-tenability* to be the largest bisimulation between the two localities as Kripke structures. This must exist as the union of all bisimulation relations between the two Kripke frames. The different between co-occurrence and co-tenability is that co-tenability does not rely upon any starting states. Put another way, it treats the entire state sets as starting states.

Signaling relations satisfy a generalized simulation condition. Let $\mathscr{S}$ be a signaling relation associated with a signal s sent from h to k. By this is meant that at time t, say, h is in state x. When the clock strikes, x will emit a signal to k and specify a move via $\mathscr{H}$ to state x'. However, h will not actually move to state x' until the next clock strike. Hence the signal is sent from state x, not state x'. The locality k will not receive the signal in the state in which k was when the signal was sent (at time t). Instead, k will transist to a new state at time $t+1$ to receive the signal.

The relation $\mathscr{K}_s \subseteq \mathscr{K}$ refers to that subset of $\mathscr{K}$ in which $\langle y,y'\rangle \in \mathscr{K}_s$ if the transition from y to y' is in response to the signal s. For now, we will ignore any other conditions that must be present in order for

that transition to take place. That is, it is assumed the transition might also be contingent upon some other conditions than the mere receipt of the signal. Since k makes no transitions unless it receives a signal from somewhere, even itself, we can define for some signal set S to k

$$\mathscr{K} = \bigcup_{s \in S} \mathscr{K}_s.$$

Note that S is not $\mathscr{S}$ (note the font difference), the former is a set of signals, the latter is a relation modeling a particular signal s. The following simulation condition will be met:

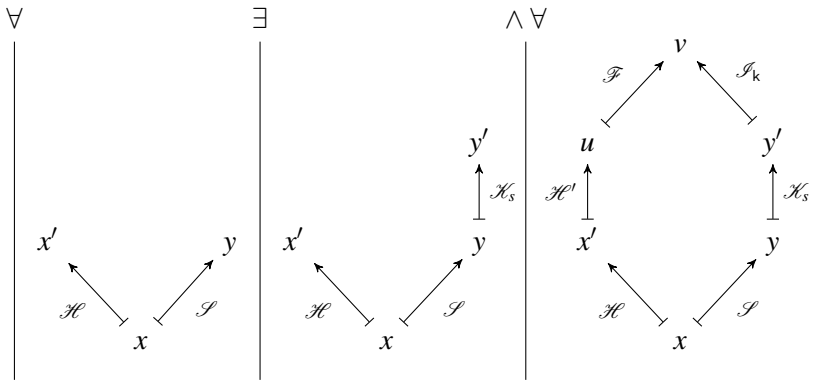


Figure 3.2: Signaling Relationship

where $v = y'$ because $\mathscr{I}_\mathsf{k}$ is the identity relation on K. From the previous section on applicative bisimulation, the diagram should come to

$$\mathscr{H} x x' \text{ and } \mathscr{F} x y \text{ implies } \exists y' \big(\mathscr{K}_s y y' \text{ and } \forall u, v (\mathscr{H}' x' u \text{ and } \mathscr{I}_\mathsf{k} y' v \text{ implies } \mathscr{F} u v) \big)$$

The modal formula should be

$$\langle s \rangle [k_s] \langle i_\mathsf{k} \rangle Q \overset{\mathsf{h}}{\supset} [h] \langle h' \rangle \langle f \rangle Q$$

which reduces to

$$\langle s \rangle [k] Q \overset{\mathsf{h}}{\supset} [h] \langle h' \rangle \langle f \rangle Q.$$

since $\langle i_\mathsf{k} \rangle Q \equiv Q$. The condition

$$[k] Q \subseteq_\mathsf{k} [k_s] Q$$

will force $\mathscr{K}_s \subseteq_\mathsf{k} \mathscr{K}$.

Next, it is always the case that if h in state x signals a state y in k, and then h proceeds to state x' (which is necessarily must do, any x' will do), then x and y will not necessarily co-occur simply because it took one clock tick for the message leaving x at t and is recognized by y at time $t+1$. Consequently, x' and y will co-occur, with $\mathscr{F}$ the co-occurrence relation. The following condition obtains:

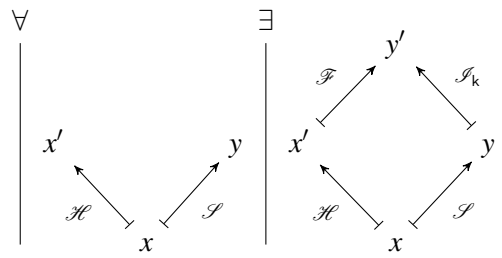


Figure 3.3: Signaling Simulation

where necessarily $y' = y$ because $\mathscr{I}_{\mathsf{k}}$ is the identity relation at k. The following generalized simulation formula will then hold

$$\langle s \rangle [i_{\mathsf{k}}] Q \subseteq_{\mathsf{h}} [h] \langle f \rangle Q,$$

and which reduces to

$$\langle s \rangle Q \subseteq_{\mathsf{h}} [h] \langle f \rangle Q.$$

We could have added the above diagram to the previous diagram in two ways, adding a link between x and y after the existential bar, or adding the entire diagram as a conjunct after the existential bar.

Given the constant availability of next states under next state relations (components cannot just die), simulation relations by themselves do not tell us very much. Connecting two components with a simulation relation sounds like it should tell us when one component can track another. However, the constant availability condition means that something more must be added for the simulation condition to be meaningful. In the signaling exposition above, the next state relation for k had to be constructed from next states as the result of certain signals. Not every state in k has a next state under $\mathscr{K}_s$.

The next state relation $\mathscr{K}_s$ need not depend solely upon the signal s. The signal s just needs to contribute to $\mathscr{K}_s$ along side other conditions or signals. Consequently, it may be that $\mathscr{K}_s yy'$ and $\mathscr{K}_{s'} yy'$ for $s \neq s'$.

4. BUS MASTER

The Bus Master is a simple circuit for displaying the main features of FPGA applications. It does not exist in any current device and it entirely generic. Consider two CPUs that make requests to the Bus Master for data from memory. The Bus Master in turn manipulates the Muxes to access the memory in such a way as to keep their accesses separate so that one CPU does not get the other CPU"s data. The Muxes themselves are not mere muxes but rather more involved components. The circuit diagram of the Bus Master is:

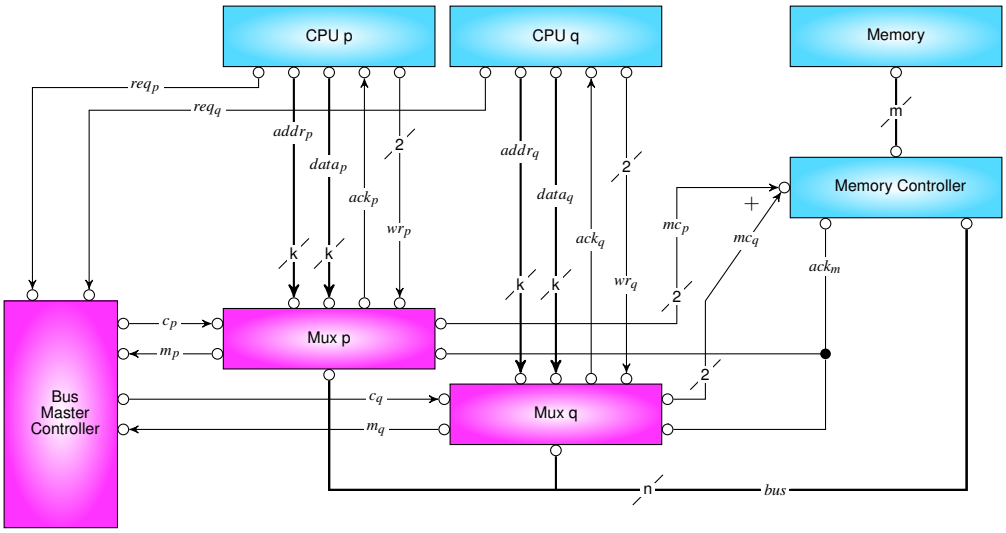


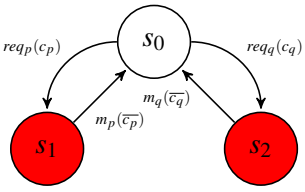
Figure 4.1: Bus Master Component Diagram

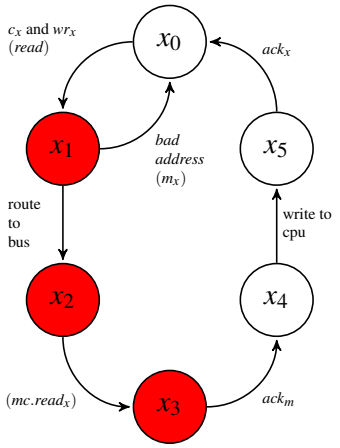
There is a font change between the diagram and the mathematics mainly due to getting Latex to accurately respond to font size commands. In general, the sf font is changed to math it, and the p and q are now subscripts.

For $x \in \{p, q\}$,

- m_x are signals from the Muxes
- c_x are signals to the Muxes
- req_x are signals from the CPUs

Restricting operation to reading from the memory, the state machine for the Bus Master Controller, and the state machines for both Muxes (they have the same state machine description) where $x \in \{p, q\}$, are:





Not all of the arcs are from external signals to the subcomponent (either the Bus Master Controller or one of the Muxes). Some are internally generated conditions labeling the arcs so it is clear why a particular arc is being taken in that state machine. A clock cycle is necessary to recognize any event, either external signals or internal conditions.

Also, the data and address lines and the bus lines are not processed in the state machine. They could have been. We assume upon receipt of c_x and wr_x, the address and data lines are routed to the bus properly and this is initiated via the read output. Typically, there is an entire protocol just for manipulation of the bus, we ignore that here in the interest of simplicity since it would not add much to the exposition. We do however use one state change to represent the work necessary to do this by labeling the transition from x_1 to x_2 with $(read)$ which sets the read bit of the two line signal $((read)/(write))$.

The $((read)/(write))$ lines are tri-state signals to the memory controller. They normally float low, i.e., 0 with a pull down resistor.

The Bus Master Controller's state machine can be rewritten using a linear script in the form of coequations. These are coequations in a process algebra language that we will develop by adding assumptions and indicating the reductions as the exposition of the Bus Controller proceeds. The assumption is positive logic for signals, i.e., 1 means *true* and 0 means *false*. The rising edge means the step from a signal being 0 to the signal being 1.

(S1) $s_0 = req_p\ (c_p)\ s_1 + req_q\ (c_q)\ s_2$

(S2) $s_1 = m_p\ s_0$

(S3) $s_2 = m_q\ s_0$

The first coequation is interpreted as

- s_0: has the definition $req_p\ (c_p)\ s_1 + req_q\ (c_q)\ s_2$ meaning if req_p is asserted, then assert c_p and move to state s_1; if req_q is asserted, then assert c_q and move to state s_2, and if both req_p and ref_q is asserted, non-deterministically chose $req_p\ (c_p)\ s_1$ or $req_q\ (c_q)\ s_1$.
- $req_p\ (c_p)$: on the rising edge of the signal req_p, set the signal c_p to 1.

The Mux state machines can be rewritten

(X1) $x_0 = c_x\ x_1$

(X2) $x_1 = bad\ address\ x_0 + read\ req\ x_2$

(X3) $x_2 = memory\ ready\ (m_x)\ x_3$

(X4) $x_3 = cpu\ ack\ x_4$

(X5) $x_4 = (write\ to\ cpu\ req)\ x_5$

(X6) $x_5 = (ack)\, x_0$

The main security property to be satisfied is

$$bus\ active \supset ([x]bus(x) \supset \neg\langle\bar{x}\rangle bus(\bar{x})).$$

where $bus(x)$ for $x \in \{p,q\}$ indicates that Mux_x owns the bus. In terms of predicates

$$bus\ active = \{s_1, s_2\}, \quad bus(x) = \{x_1, x_2, x_3\}, \quad x \in \{p,q\}.$$

The relations underlying $[x]$ and $\langle x\rangle$ are P and Q. They are co-occurrence relations. That is to say, $P \subseteq B \times M_p$ and $Q \subseteq B \times M_q$ for B the collection of Bus Master Controller states, M_p is the collection of Mux_p states, and M_q the collection of Mux_q states. The relations simply declare which states can occur simultaneously. These can be computed from the state machines above. However, in practice, they will be computed from the process algebra expressions that will be used to encode the state machines. The process algebra expressions *generate* the states or more accurately, generate the Kripke model over which the logical formulas are evaluated.

4.1 Parallel Execution: Multiple Simultaneous Reductions

Here, we are somewhat breaking the rules of algebraic systems in general by performing two reductions at step 2 (the arrow 2). Step 1 is not technically a reduction, it is merely replacing expressions with the right hand sides of their identities. Another addition is the use of $\langle -,-\rangle$ to indicate a pair of signals where receipt of the pair does not imply the pair was sent by a single locality. Rather, the elements of the pair (more generally, a tuple) can originate from different localities (components):

$$\begin{array}{c}
s_0 \mid p_0 \mid q_0 \mid cpu_p \mid cpu_q \\
= \Big\downarrow \text{step 1} \\
req_p.\overline{c_p}.m_p.\overline{c_p}.s_1 \mid \langle c_p, wr_p\rangle.\overline{read}.p_0 \mid \langle c_q, wr_q\rangle.\overline{read}.q_0 \mid \overline{req_p}.\overline{wr.p}.ack_p.cpu_p \mid \overline{req_q}.\overline{wr.q}.ack_q.cpu_q \\
\tau = req_p \cdot \overline{req_p} \Big\downarrow \text{step 2} \\
\overline{c_p}.m_p.\overline{c_p}.s_1 \mid \langle c_p, wr_p\rangle.\overline{read}.p_0 \mid \langle c_q, wr_q\rangle.\overline{read}.q_0 \mid \overline{wr.p}.ack_p.cpu_p \mid \overline{req_q}.\overline{wr.q}.ack_q.cpu_q \\
\tau = c_p \cdot \overline{c_p}, wr_p \cdot \overline{wr_p} \Big\downarrow \text{step 3} \\
m_p.\overline{c_p}..s_1 \mid \overline{read}.p_0 \mid (c_q, wr_q).\overline{read}.q_0 \mid cpu_p \mid \overline{req_q}.\overline{wr.q}.cpu_q
\end{array}$$

Figure 4.2: Computing with the Process Algebra

Notice that each algebraic term has a *lane* that it stays within even as it morphs into further terms. Each lane is a *label*. For our simple applications, there is no need to explicitly use the *label* :: *term* syntax since the lanes do not change.

4.2 Parallel Execution: Parallel Reduction Paths in the Algebra

We wish to avoid parallel threads of execution. They would occur if the machine had some built in non-determinism. We will just record a brief exposition here in case they come up in actual designs. Were they to occur in an actual designs, it might indicate an error in the design. The result would be a machine that had somewhat arbitrary behavior.

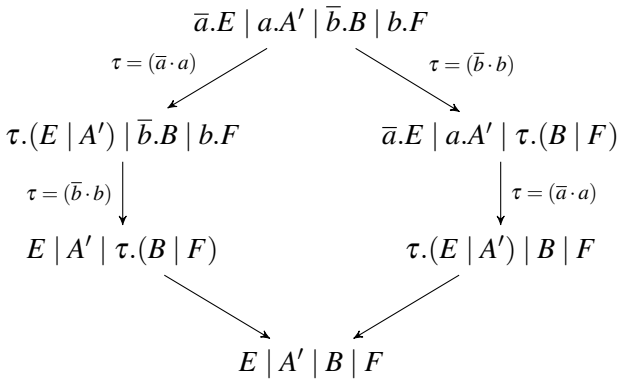


Figure 4.3: Computing with the Process Algebra

Again, each algebraic term has a *lane* that it stays within even as it morphs into further terms. There is no need to explicitly use the *label* :: *term* syntax since the lanes do not change.

4.3 Parallel Formulas

The lanes of the algebra are localities in process algebra and in Distributed Logic. The reductions in the algebra within a lane are pairs in the intensional next state relation in the lane. The reductions involving two lanes indicate pairs in a co-occurrence relation. Just for the record, there is nothing stopping us from have reductions that involve n (for $n > 2$) lanes in which case the indicated relation is n-ary. An example of a logic built on 3 place relations is [7].

In a sense, we already have lanes for logical formulas, they are again the localities. A typical move is a formation rule

$$\frac{P_1 \mid \cdots \mid Q \mid R \mid \cdots \mid P_n}{P_1 \mid \cdots \mid Q, [f]R \mid \top \mid \cdots \mid P_n}$$

It makes more sense to treat the lanes as having sets of formulas. The $\mid$ operators are certainly not $\vdash$ relations. In this case, moving a formula is removing it from one lane's set and added it to another with a necessity prepended. The $[f]$ appellation must be underwritten by a co-occurrence relation $\mathscr{F}$.

Let us try for some better notation before we develop more of the logic. For $\mathscr{F} : \mathsf{h} \rightharpoonup \mathsf{k}$,

$$\frac{\mathsf{p}_1 \mid \cdots \mid \mathsf{h} \mid \mathsf{k}(R) \mid \cdots \mid \mathsf{p}_n}{\mathsf{p}_1 \mid \cdots \mid \mathsf{h}([f]R) \mid \top \mid \cdots \mid \mathsf{p}_n}$$

Let us use the trick from Gentzen systems of using contexts:

$$\frac{\Gamma(\mathsf{h} \mid \mathsf{k}(R))}{\Gamma(\mathsf{h}([f]R) \mid \top)}$$

The $\mid$ operator is too distracting in this setting, let us try a ;:

$$\frac{\Gamma(\mathsf{h}; \mathsf{k}(R))}{\Gamma(\mathsf{h}([f]R); \top)}$$

This is very close to the logical rule of necessitation for Distributed Logic expanded to include the other localities as being parametric. It also accommodates an infinite collection of localities all hiding in Γ. The bit of syntax Γ is a collection of lanes and very much like Relevance Logic's notion of fusion. In RL, the left sides of Gentzen systems are finite multisets with elements separated by ;. Under interpretation, ; is RL's $\circ$.

A global logic then contains (possibly infinite) formulas with each locality containing a set of formulas. All the formulas in a particular set of a locality can be considered conjoined together. Evaluation of a locality's set at a state amounts to checking the satisfiability of each formula at the state.

The tuples in the global logic with a different locality's set at each element of the tuple. Hence a formula has the form

$$P \in \mathrm{Log}(\mathsf{h}_1) \times \cdots \times \mathrm{Log}(\mathsf{h}_i) \times \cdots$$

for $\mathrm{Log}(\mathsf{h}_i)$ the logic at locality h_i.

It would appear that the tuples are elements of something like a tensor product. Take two formulas, P and Q in localities h and k respectively. If the locality designation is a part of the formula, then a simple cross product will suffice to represent the formulas because we can always keep track of the localities. More in keeping distributed logic is to assign the localities to the operators since we already do this for the distributed modal operators. The only operator we have is a semicolon, which is usually a part of sequent. It makes more sense to use a relevance logic fusion

Another idea is to put a consequence logic in each of the lanes with thinning, cut, and axiom rules. The goal is then to define valid rules for moving formulas from either side of the sequents in one lane to sides of the sequents in another lane.

5. A WORD ON INFINITE FORMULAE

In [20], Chapter 11 on Modal Logic, there is a rendition of infinite formulae. In particular, they use the syntax:

$w \models_G p$	iff	$p \in l(w)$ (*for all* $p \in \Lambda$)
$w \models_G \mathsf{T}$		*for all* w
$w \models_G \neg\varphi$	iff	$w \not\models_G \varphi$
$w \models_G \Diamond\varphi$	iff	*there is a* $w' \in G$ *such that* $w \to w'$ and $w' \models_G \varphi$
$w \models_G \varphi \wedge \psi$	iff	$w \models_G \varphi$ and $w \models_G \psi$
$w \models_G \bigwedge \Phi$	iff	$w \models_G \varphi$ (*for all* $\varphi \in \Phi$)

The Kripke model G is in the form of a graph with labels $l(w)$ at the nodes w. The labels are formulae. The infinite formula $\bigwedge \Phi$ is a bit odd. They seem to want to use any set Φ of formulae. However, it is hard to call this syntax if the formulae in Φ are not generated by some sort of rule so that the formulas share a common form or template. Let us assume Φ is so generated.

From Dynamic Logic [33], there are formulae

$$[a^*]\varphi$$

where the infinity of the Kleene * operator is pushed into the model theory. If we instead code this formula as

$$\begin{aligned}[a^*]\varphi &= \varphi \wedge [a]\varphi \wedge [a][a]\varphi \wedge \cdots \\ &= \bigwedge_{0\leq i<\omega} [a]^i\varphi\end{aligned}$$

for ω the first infinite ordinal. We can analyze the modeling condition for the two which some out to the same conditions. That is, $[a^*]$ is to be evaluated using the reflexive-transitive closure of the relation interpreting $[a]$ where

$$\mathscr{A}^* = \bigcup_{0\leq i<\omega} \mathscr{A}^i, \quad \mathscr{A}^0 = \mathscr{I}$$

with $\mathscr{I}$ being the identity relation. Note that $\mathscr{A}^*$ only makes sense within a locality, not between localities because of the properties of reflexivity and transitivity of a relation. Hence

$$\begin{aligned}x \models [a^*]\varphi \text{ iff } &\forall y(\mathscr{A}^* xy \text{ implies } y \models \varphi) \\ &\text{iff } (0 \leq \forall i < \omega)\forall y(\mathscr{A}^i xy \text{ implies } y \models \varphi) \\ &\text{iff } x \models \bigwedge_{0\leq i<\omega} [a]^i\varphi\end{aligned}$$

The consequence is that we can express invariants in the form

$$Q = P \wedge [a]Q$$

as long as we adopt non-wellfounded set theory for the formalization of the logical syntax. If we were to use the well-founded set theory, we would have

$$Q \equiv P \wedge [a]Q$$

which pushes the recursion into the model theory. This does not seem to make sense in our case since a finite state machine does not require an infinite set of pairs in its modeling relation. When the machine cycles back to a previous state, it is back to the previous state *simpliciter*.

If we consider a single state machine as a Kripke relation in intension, then it would seem that we ought to have

$$wkp(\mathscr{M}, Q) = [m]Q$$

where m is the modality interpreted by $\mathscr{M}$ and *wkp* is the *weakest precondition*.

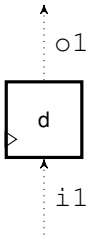
6. REWIRE

This section is the conversion of the ReWire-centric (as opposed to Haskell-centric) language constructions into process algebra equivalents. Some minimal conventions and assumptions are:

- The notation $\overline{i(v)}$ represents an input on the signal named i and v is the value of that input. Separating the signal from its value makes the syntax easier to handle.
- Some equations are coequations with tail recursion. Hence the coequations appear not to have solutions in well-founded set theory. We assume non-wellfounded set theory where every collection of flat coequations (no nested indeterminates) has a solution.
- It is intended that every serial operator, i.e., the dot in $a\,.\,b$, requires one clock tick.
- Regarding FPGA applications, they are composed of islands of finite state machines. Typically these are called components, ReWire terms them *devices*. All the devices are running in parallel and exchange signals among themselves and the outside world. There is a sleight of hand necessary to understand the constructors listed below. All devices are run at a top level that get run on a clock tick. There are no nested devices. The sleight of hand is that devices are constructed as functions and ReWire's surrounding *run* infrastructure schedules them to run when the clock strikes. If a constructor, which is building a device, requires a pre-existing device, the pre-existing device is treated as function that is call from within the newly constructed device.

6.1 Basic Device Constructor

- Built-in Type `Dev i o`
  - Parameterized by input and output types, `i` and `o`
- Construct devices by `Dev i o` hardware type components
- ReWire compiler translates `Dev i o` into synthesizable VHDL
- `Dev i o` is mathematically based
  - Algebraic structure for clocked, synchronous parallelism
  - Useful for specifying secure systems [LCTES15,JCS09]



The process algebra equivalent is

$$d' = \overline{i_1(v_1)}\,.\,d(v_1) >>= \lambda v_2.\ (o_1(v_2) \mid d')$$

Some explanation is in order. We cannot use entirely process algebra notation because ReWire is essentially written in a functional notation. We wish to keep the acceptance of a signal, i.e., the part preceding the first dot, as requiring one clock tick. The functional lambda λ notation however also uses a dot, so the dot

become overloaded. The rule is that a dot following a λ and its bound variable is part of the lambda notation. The $>>= \lambda -$. notation uses Haskell's bind operator, $>>=$, which collects the output from the device (as a monad) on the left of the operator and feeds it to the lambda abstraction on the right of the operator.

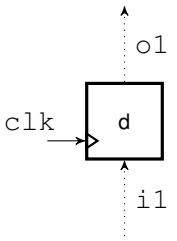
In the expression above, the monad d is applied to an argument v_1 where we break with Haskell and denote this as $d(v_1)$ rather than $d\ v_1$ where the v_1 appears somewhat disembodied. The bind operator takes the result and feeds it to the following lambda function. That function outputs v_2 on o_1 in parallel with another copy of d' to be executed on the next clock cycle.

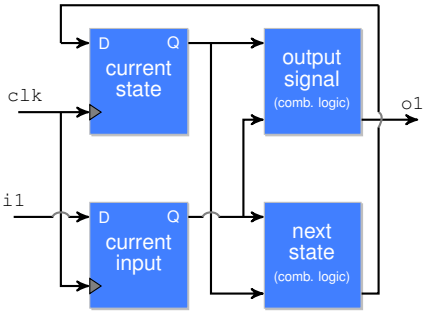
Any term of the form $\overline{i(v)}$ refers to receiving a signal on line i and the value of that signal is v. Any term of the form $o(v)$ refers to sending a signal on line o and the value of that signal is v.

The lambda calculus generally causes formulae to reduce once when a function is applied to an argument. In this sense, a lambda abstraction is an instance of function. Once reduced by application, that instance is no more. Process algebra terms are no different. This poses a problem for representing hardware devices as process algebra terms. We use coequations and a judicious use of the parallel operator to "replenish" a process algebra term after the current clock tick.

To get another copy of the device for running in the next clock cycle, the equation written above is called a coequation and the trailing d' looks like a tail-recursive call. That is one way to think of it. We, however, think of it as something to be replaced by its definition. This is underwritten by non-well-founded set theory and that set theory is consistent. Remember this is an algebraic presentation of what the hardware is actually doing. The hardware is not making a copy of itself, but mathematics cannot work with hardware, it must represent hardware and this is as close as we can get to a device which is always on and waiting to run every clock cycle.

6.2 Representing `Dev i o` as a Circuit





The process algebra equivalent is constructed at a lower level than device as represented in ReWire. Still, we will have a go at just to see how complicated the algebra needs to be. We let *ci* refer to the current input, *cs* the current state, *os* the output signal, and *ns* the next state. As a first cut, we start with the following simplified coequations:

$$\begin{aligned}
\overline{i_1}\,.\,d\,.\,o_1 &= cs \mid ci \mid os \mid ns \\
cs &= \overline{D_{cs}}\,.\,CS\,.\,Q_{cs}\,.\,cs \\
ci &= \overline{D_{ci}}\,.\,CI\,.\,Q_{ci}\,.\,ci \\
os &= \langle \overline{I_{os1}}, \overline{I_{os2}} \rangle\,.\,OS\,.\,O_{os}\,.\,os \\
ns &= \langle \overline{I_{ns1}}, \overline{I_{ns2}} \rangle\,.\,NS\,.\,O_{ns}\,.\,ns
\end{aligned}$$

The problem with these coequations is that clock ticks are not represented by process algebra's dot notion. Process algebra is a bit anemic in that all sequencing is represented with the dot notion. However we have two forms of sequencing, that induced by clock ticks and reading a value on an input line and that of function invocation. This latter requires no clock cycles. The above notation does give the gist of the hardware circuit. A better, albeit more complicated notation, will require Haskell's hind operator. A device (from above) has the description

$$d' = \overline{i_1(v_1)}\,.\,d(v_1) >>= \lambda v_2.\ (o_1(v_2) \mid d')$$

The implementing circuit is as at a lower hardware level and hence this notation is inappropriate. The following notation is closer to what is required:

$$\begin{aligned}
d' &= cs \mid ci \mid os \mid ns \\
cs &= \overline{D_{cs}(v_1)}\,.\,CS(v_1) >>= \lambda v_2.\ (Q_{cs}(v_2) \mid cs) \\
ci &= \overline{D_{ci}(v_3)}\,.\,CI(v_3) >>= \lambda v_4.\ (Q_{ci}(v_4) \mid ci) \\
os &= \langle \overline{I_{os1}(v_5)}, \overline{I_{os2}(v_6)} \rangle\,.\,OS(v_5, v_6) >>= \lambda v_7.\ (O_{os}(v_7) \mid os) \\
ns &= \langle \overline{I_{ns1}(v_8)}, \overline{I_{ns2}(v_9)} \rangle\,.\,NS(v_8, v_9) >>= \lambda v_1 0.\ (O_{ns}(v_{10}) \mid ns)
\end{aligned}$$

The linkages among the inputs and outputs are missing. This requires that we make certain identifications. So to the above coequations we add some equations:

$$\begin{aligned}
Q_{cs} &= I_{os1} \\
Q_{cs} &= I_{ns2} \\
Q_{ci} &= I_{os2} \\
Q_{ci} &= I_{ns1} \\
O_{ns} &= D_{cs}
\end{aligned}$$

Note that the bar notation will cause inputs to link to outputs properly. The coequations can be flattened by adding intermediate indeterminates ([20]) immediately past their first action symbol and then adding coequations for the new indeterminates, recursively. From non-wellfounded set theory, there is a model for these. That model is the set-theoretic model for the device d.

6.3 Iteration Constructor

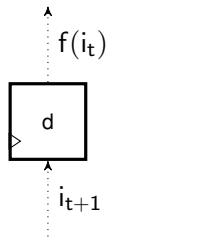
```
iter :: (i -> o) ->
        o        ->
        Dev i o
```

`iter` takes a function of type `(i -> o)` and a value of type `o` and returns an object of type `Dev i o`

`iter` "iterates" `f` over the input for each clock cycle. `f` is "combinational" logic as a ReWire function.

At time $t+1$, f has been applied to the previous input at time t

d = iter f o



Now we must represent a constructor. The diagram has the following process algebra description: The process algebra equivalent is

$$d = \overline{i(v)}\,.\,(o(f(v)) \mid d)$$

The use of the parallel process algebra operator (not ReWire's parallel operator) is because the device reads an input at clock t and outputs at time $t+1$ but is still able to process the next input at time $t+1$. Hence, the device does two things at time $t+1$, reads the input and sends an output where the output relies upon the previous value of the input. Necessarily, the value of v is (possibly) different at times t and $t+1$.

The constructor must produce this coequation, to wit:

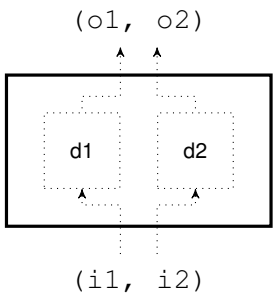
$$iter(f,o) \stackrel{def}{=} d = \overline{i(v)}\,.\,(o(f(v)) \mid d)$$

where $f : i \to o$, i.e., a function from i as an input type to o as an output type. In typical Haskell fashion, the definition of the device uses i and o as variables as well as types.

6.4 Parallelism Constructor

```
<&> :: Dev i1 o1 ->
       Dev i2 o2 ->
       Dev (i1,i2) (o1,o2)
```

d1 and d2 are non-interfering by construction



The possible process algebra equivalent is

$$d = \overline{i\langle v_1, v_2\rangle}\,.\,\big(d_1(v_1) \mid d_2(v_2)\big) >>= \lambda(v_3, v_4).\ (o\langle v_3, v_4\rangle \mid d)$$

where $>>= \lambda(-).$ gathers the output of the previous computation and pipes it to the following lambda term with the understanding that any output of the parallel computations on the left of $>>= \lambda(-).$ is bundled together into a tuple. This has the unfortunate syntactic problem of overloading the dot for both lambda abstraction and process algebra sequencing.

Let

$$d = \overline{i_1(v_1)}\,.\,d_1(v_1) >>= \lambda v_3.\ (o_1(v_3) \mid d) \qquad\qquad d' = \overline{i_2(v_2)}\,.\,d_2(v_2) >>= \lambda v_4.\ (o_2(v_4) \mid d')$$

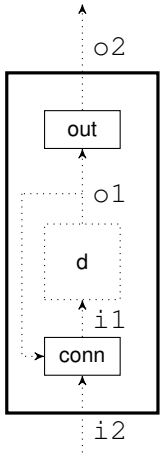
The constructor's definition is

$$<\&>(d,d') \stackrel{def}{=} d'' = \overline{\langle i_1(v_1), i_2(v_2)\rangle}\,.\,\big(d_1(v_1) \mid d_2(v_2)\big) >>= \lambda(v_3, v_4).\ (\langle o_1(v_3), o_2(v_4)\rangle \mid d'')$$

6.5 Feedback Constructor

feedback out conn d

```
feedback ::
    (o1 -> o2) ->
      (o1 -> i2 -> i1) ->
        Dev i1 o1 ->
          Dev i2 o2
```



Notice `feedback` is an actual language element not something implied of a design in VHDL

The process algebra equivalent is

$$d' = \overline{\langle i_2(v_2), o_1(a_1)\rangle}\,.\,d(conn(v_2,a_1)) >>= \lambda v_1.\ (\langle o_1(v_1), o_2(out(v_1))\rangle \mid d')$$

Let

$$g = \lambda v_4.\ o_2(out(v_4)) \qquad f = \lambda v_2.\ \lambda a_1.\ conn(v_1,a_1) \qquad d = \overline{i_1(v_1)}\,.\,d_1(v_1) >>= \lambda v_3.\ (o_1(v_3) \mid d)$$

The process algebra transform is

$$feedback(g,f,d) \stackrel{def}{=} d' = \overline{\langle i_2(v_2), o_1(a_1)\rangle}\,.\,d(conn(v_2,a_1)) >>= \lambda v_1.\ (\langle o_1(v_1), o_2(out(v_1))\rangle \mid d')$$

with the following equation

$$o_1 = i_2.$$

Another new feature is $\overline{\langle i_2(v_2), o_1(a_1)\rangle}$. This is not the same as $\overline{i(v,v')}$. The latter inputs a pair. The former represents forming a pair from two separate inputs and then boxing them together.

7. APPENDIX

7.1 Hennessy-Milner

From [34] pp. 138, we have the following statement;

> In this section we introduce a way of defining equivalence between programs that is based entirely on operational considerations; informally, two programs are equivalent when no observations can distinguish them. Further, two subprograms or program phrases are congruent if the

result of placing each of them in any program context yields two equivalent programs. Then, considering the phrases as modules, one can be exchanged for the other in any program without affecting the observed behavior of the latter.

There is the following characterization of algebraic theories:

Theorem 7.1.1 (Birkhoff) *The following axiom and rules suffice to characterize those algebras that are definable by equations:*

$$x = x$$

$$\frac{x = y \qquad y = z}{x = z}$$

$$\frac{x = y}{y = x}$$

$$\frac{x = y}{\sigma(x_1, \ldots, x, \ldots, x_n) = \sigma(x_1, \ldots, y, \ldots, x_n)}$$

$$\frac{s = s'}{s(t_1/x_1, \ldots, t_n/x_n) = s'(t_1/x_1, \ldots, t_n/x_n)}$$

*(where $s(t_1/x_1, \ldots, t_n/x_n)$ denotes the result of uniform substitution of $t_1, \ldots, t_n$ for the variables $x_1, \ldots, x_n$ in the term s; similarly for s'). We call this equational logic **EL**. This we have*

$$Q \models s = t \text{ iff } Q \vdash_{\mathbf{EL}} s = t.$$

The x, y, etc. variables and s, t, etc. variables are metalinguistic variables. The former are metalinguistic variables for variables and the latter are metalinguistic variables for terms. In the object language, variables are terms.

The point is that Hennessy and Milner's prose definition of *equivalence* seems to correspond to reading the last rule backwards where the s and s' are terms and we twiddle with their inputs to see if there are any discrepancies. Their notion of congruence seems to correspond to reading the third rule (skipping the axiom $x = x$) backwards where the x and y refer to computations and the σ is an algebraic context, i.e., algebraic term with *holes* in it.

From [35], they have some material on image finite relations which stem from [34]. A relation is image finite (thinking of the relation as a graph) if every node can only go to a finite number of other nodes under the relation. They state that a model's modal theory is the collection of all theorems valid in model, i.e., theorems that are valid at every state in the model. If two models are bisimulation equivalent, then their theories are equal. The converse is supplied by image finiteness: if two models are image finite and their theories are equal, then they are bisimulation equivalent.

7.2 Monads and Algebras

From [36], we know that modal algebras are monadic over Boolean algebras and that the Lindenbaum-Tarski algebra is the same whether we start with a Lindenbaum-Tarski Boolean algebra (over a set of generators A) and using generators of the form $a, \Box a$ or merely the same set of generators A and generating the modal algebra from scratch.

REFERENCES

1. J. Barwise and J. Seligman, *Information Flow: The Logic of Distributed Systems* (CUP, 1997). Cambridge Tracts in Theoretical Computer Science 44.

2. A. Anderson and N. Belnap, Jr., *Entailment: The Logic of Relevance and Necessity*, volume I (Princeton University Press, 1975).

3. A. Anderson, N. Belnap, Jr., , and J. Dunn, *Entailment: The Logic of Relevance and Necessity*, volume II (Princeton University Press, 1992).

4. G. Allwein and W. L. Harrison, "Partially Ordered Modalities," Proceedings of the Proceedings of the Advances in Modal Logic Conference, 2010 (Springer-Verlag), 2010, pp. 1–20.

5. G. Allwein, W. L. Harrison, and D. Andrews, "Simulation logic," *Logic and Logical Philosophy* **23**(3), 277–299 (2014).

6. G. Allwein and W. L. Harrison, "Distributed Modal Logic," in K. Bimbó, ed., *J. Michael Dunn on Information Based Logic: Outstanding Contributions to Logic*, pp. 331–362 (Springer-Verlag, 2016).

7. G. Allwein, W. L. Harrison, and T. Reynolds, "Distributed Relation Logic," *Logic and Logical Philosophy* **26**(1), 19–61 (2017).

8. A. Procter, W. L. Harrison, I. Graves, M. Becchi, and G. Allwein, "Semantics-directed machine architecture in ReWire," Proceedings of the Proceedings of the 2013 International Conference on Field-Programmable Technology, 2013, pp. 446–449.

9. A. Procter, W. L. Harrison, I. Graves, M. Becchi, and G. Allwein, "Semantics Driven Hardware Design, Implementation, and Verification with ReWire," Proceedings of the Proceedings of the Conference on Languages, Compilers, Tools and Theory for Embedded Systems, Lecture Notes in Computer Science (ACM Digital Library), 2015, pp. 1–10.

10. A. Procter, W. L. Harrison, I. Graves, M. Becchi, and G. Allwein, "A Principled Approach to Secure Multi-Core Processor Design with ReWire," Proceedings of the Proceedings of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, Tools and Theory for Embedded Systems (ACM Digital Library), 2016.

11. W. L. Harrison, A. Procter, I. Graves, M. Becchi, and G. Allwein, "A Programming Model for Reconfigurable Computing Based in Functional Concurrency," Proceedings of the Proceedings of the 11th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC 2016) (IEEE), 2016, pp. 1–8.

12. T. Reynolds, A. Procter, W. L. Harrison, and G. Allwein, "A Core Calculus for Secure Hardware: Its Formal Semantics and Proof System," Proceedings of the Submitted to Mathematical Foundations of Programming Semantics (MFPS) 2017 (Electronic Notes in Theoretical Computer Science), 2017.

13. W. L. Harrison and G. Allwein, "Semantics-directed Prototyping of Hardware Runtime Monitors," Proceedings of the Proceedings of the 29th International Symposium on Rapid System Prototyping (RSP), 2018, pp. 42–48.

14. W. L. Harrison and G. Allwein, "Language Abstractions for Hardware-based Control-Flow Integrity Monitoring," Proceedings of the Proceedings of the 2018 International Conference on Reconfigurable Computing and FPGAs, 2018, p. (to appear).

15. J. Barwise, "Information Links in Domain Theory," Proceedings of the Mathematical Foundations of Programming Semantics, 7th International Conference (Springer-Verlag), 1991, pp. 168–192.

16. J. Y. Girard, P. Taylor, and Y. LaFont, *Proofs and Types* (Cambridge University Press, 1989).

17. Y. Kinoshita, P. W. O'Hearn, A. J. Power, M. Takeyama, and R. D. Tennent, "An axiomatic approach to binary logical relations with applications to data refinement," Proceedings of the International Symposium on Theoretical Aspects of Computer Software, TACS 1997: Theoretical Aspects of Computer Software, volume Lecture Notes in Computer Science book series (LNCS, volume 1281), 1997, pp. 191–212.

18. Q. Ma and J. C. Reynolds., "Types, abstraction, and parametric polymorphism, part 2," in S. B. et al., ed., *Mathematical Foundations of Programming Semantics, Proceedings of the 7th International Conference,*, volume Volume 598 of Lecture Notes in Computer Science, 1991, pp. 1–40.

19. G. D. Plotkin, "Lambda Definability and Logical Relations," University of Edinburgh, School of Artificial Intelligence, October, 1973.

20. J. Barwise and L. Moss, *Vicious Circles* (CSLI Publications, 1996).

21. R. Milner, "An Algebraic Definition of Simulation Between Programs," Stanford University, 1971.

22. A. Ginzburg, *Algebraic Theory of Automata* (Academic Press, 1968).

23. S. Abramsky and C. H. L. Ong, "Full abstraction in the lazy lambda calculus," *Information and Computation* **105**, 159–267 (1993).

24. A. Pitts, "A note on logical relations between semantics and syntax," *Logic Journal of the IGPL* **5**(4), 589–601 (1997).

25. G. D. Plotkin, "Call-by-Name, Call-by-Value and the λ Calculus," *Theoretical Computer Science* **1**, 125–159 (1975).

26. L. Aceto, "A static view of localities," *Formal Aspects of Computing* **6**, 201–222 (1994).

27. R. Milner, *Concurrency and Communication* (Prentice Hall, 1989). Series in Computer Science.

28. D. Barker-Plummer, J. Barwise, and J. Etchemendy, *Language, Proof, and Logic*, second ed. (CSLI Publications, 2011).

29. R. Milner, *A Calculus of Communicating Systems* (Springer-Verlag, 1980). Lecture Notes in Computer Science 92.

30. J. Bergstra and J. W. Klop, "Process algebra for synchronous communication," *Information and Control* **60**, 109?–137 (1984).

31. S. Abramsky, "A Cook's Tour of the Finitary Non-Well-Founded Sets," https://arxiv.org/abs/1111.7148v2 (2011).

32. M. Mislove, L. Moss, and F. Olesl, "Non-Well-Founded Sets Modeled as Ideal Fixed Points," *Information and Computation* **93**, 16–54 (1991).

33. D. Harel, D. Kozen, and J. Tiuryn, *Dynamic Logic* (MIT Press, 2000).

34. M. Hennessy and R. Milner, "Algebraic Laws for Nondeterminism and Concurrency," *Journal for the Association of Computing Machinery* **32**(1), 137–161 (1985).

35. P. Blackburn, M. de Rijke, and Y. Venema, *Modal Logic* (Cambridge University Press, 2001). Cambridge Tracts in Theoretical Computer Science, No. 53.

36. M. M. Bonsangue and A. Kurz, "Presenting Functors by Operations and Equations," Proceedings of the Foundations of Software Science and Computation Structures, LNCS 3921 (Springer-Verlag), 2006, pp. 172–186.