

Spiders for FPGA Applications

DR. GERARD ALLWEIN

CHRISTOPHER BELMONTE

*Center for High Assurance Computer Systems Branch
Information Technology Division*

June 7, 2022

REPORT DOCUMENTATION PAGE				Form Approved OMB No. 0704-0188	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing this collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.					
1. REPORT DATE (DD-MM-YYYY) 07-06-2022		2. REPORT TYPE NRL Memorandum Report		3. DATES COVERED (From - To) 1 Oct 2021 – 30 Sept 2022	
4. TITLE AND SUBTITLE Spiders for FPGA Applications				5a. CONTRACT NUMBER	
				5b. GRANT NUMBER	
				5c. PROGRAM ELEMENT NUMBER 062235N	
6. AUTHOR(S) Dr. Gerard Allwein and Christopher Belmonte				5d. PROJECT NUMBER	
				5e. TASK NUMBER	
				5f. WORK UNIT NUMBER 6C59	
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/5540/MR--2022/4	
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Office of Naval Research 875 N. Randolph Street Arlington VA 22217-1995				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 a report on the abstract structure of spiders for applications. A spider is a realtime monitor of Field Programmable Gate Arrays (FPGAs) and Complex Programmable Logic Devices (CPLDs). The FPGAs and CPLDs are assumed to contain applications coded in VHDL. Each spider is associated with specific logic statements expressing conditions in the application code being monitored. Each spider also contains mechanisms for mitigating the effects of an exploit, either malicious, due to an error in design, or due to a hardware fault. Spiders can be written in any language provided there is a translator into a language that vendor supplied tools support. Eventually, spiders will be automatically compiled from logic statements and mitigation code to produce either VHDL code or ReWire code. ReWire has its own compiler that produces either VHDL or Verilog code, which would then subsequently be included in the application.					
15. SUBJECT TERMS					
16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT U	18. NUMBER OF PAGES 55	19a. NAME OF RESPONSIBLE PERSON Dr. Gerard Allwein
a. REPORT U	b. ABSTRACT U	c. THIS PAGE U			19b. TELEPHONE NUMBER (include area code) (202) 404-3748

This page intentionally left blank.

CONTENTS

EXECUTIVE SUMMARY	E-1
1. INTRODUCTION	1
2. LOGIC PRELUDE	3
2.1 The Logic.....	5
2.2 Interpreting Logic	5
2.3 Interpreting Distributed Logic.....	13
2.4 Distribution	16
3. STATE MACHINES	17
3.1 States vs. Transitions	17
3.2 Multiplying the States	19
3.3 Multiplying the Next-State Relation to Many Relations	19
3.4 Implications.....	20
4. LOGICAL SPECIFICATIONS FOR CHECKING STATES.....	20
4.1 Using Distributed Logic in Specifications.....	21
4.2 Specifying States in the Logic.....	23
4.3 Strong Conditions for Checking States	25
5. WEAKENING THE CONDITIONS FOR CHECKING STATES	28
5.1 Simulations and Bisimulations	28
5.2 Decoupling Spider's States from the Plant's States	33
5.3 Checking the Plant's Outputs	43
6. INVARIANTS	44
6.1 Cheaper Spiders using Invariants	44
6.2 Assigning a Syntactic Map to Local Operators	45
7. MONITORING TIME LIMIT CONDITIONS	46
8. CONCLUSIONS	46
9. APPENDIX	47
9.1 Distributed Logic: Axioms and Rules.....	47
9.2 Base System	48
9.3 Options.....	49
9.4 Definitions and Rules.....	50
REFERENCES	50

This page intentionally left blank

EXECUTIVE SUMMARY

This is a report on the abstract structure of spiders for applications. An application is known as a *plant*. For us, we will be concentrating on spiders for Field Programmable Gate Arrays (FPGAs) and Complex Programmable Logic Devices (CPLDs). Our references to FPGAs in the report is intended to cover both kinds of devices. A spider is a realtime monitor of application code written in any language appropriate for FPGAs. This report will use VHDL to illustrate concepts. Each spider is associated with specific logic statements expressing the conditions in the code being monitored. Each spider also contains mechanisms for mitigating the effects of an exploit, either malicious, due to an error in design, or due to a hardware fault. We will not cover mitigation in this report. Eventually, spiders will be automatically compiled to produce either VHDL code or ReWire code. ReWire has its own compiler that produces either VHDL or Verilog code, which would then subsequently be included with the application and all the code handed to a vendor supplied tool for producing the bitstream downloaded into the FPGA. The NRL Memo Report “Spider for a Traffic Light” is a sequel to this current report and covers a specific example of a spider+plant.

This page intentionally left blank

Spiders for FPGA Applications

1. INTRODUCTION

This is the first report for the NRL Base Program 6.2 (Work Unit 55 T012) Realtime Monitors (Spiders) for FPGA Applications. We term these monitors *spiders* with the idea being that a spider sits above an application with its legs embedded in the application. The legs are signal lines and the body is the center for recognition of error conditions and mitigation of those conditions. The FPGA application is called the *plant*, a term lifted from realtime control of industrial machines. The intuitive picture is

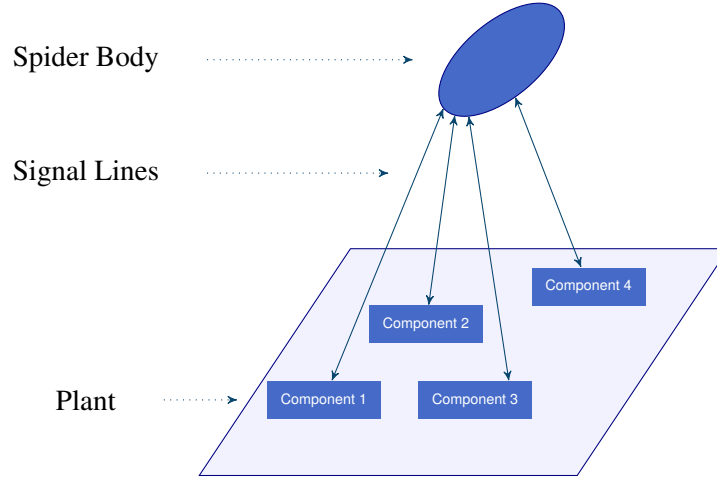


Figure 1.1: Intuitive Diagram of Spider and its Plant

The structure of the system, as shown in Figure 1.2 and as seen from the VHDL code, is usually different. The signal lines between the spider and the plant are not drawn. The hierarchical structure is showing a possible code architecture, not the actual connections (as in the previous diagram).

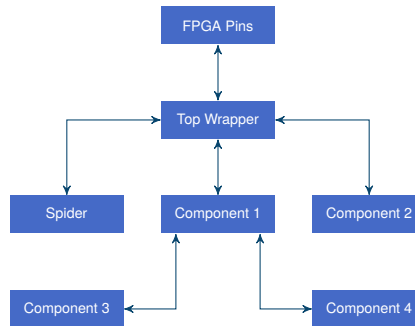


Figure 1.2: Code Structure of Spider and its Plant

This structure makes it relatively easy to place the spider and the plant into a larger system. Of course this structure is only paradigmatic; any actual spider + plant is likely to be much more complicated. There may also be several spiders for any one plant, and spiders can spider other spiders. It is also possible to have a spider observing two different plants if, say, it is required to watch over interaction between the two plants.

We see spiders as being applicable to many types of FPGA applications including controllers for industrial machines. We also use the term *spider* as a verb, i.e., to *spider an application* to have a realtime monitor for that application.

A spider is intended to be compiled along with the plant using vendor supplied tool chains. Recreating those tool chains is infeasible due to the amount of work the vendors have put into those tool chains. Also, the output of those toolchains is specific to each vendor. All the vendors support VHDL and Verilog, however. That said, these languages have their own drawbacks. The applications written in those languages do not lend themselves easily to mechanized verification of the application properties. There is another language, called ReWire [1, 2], which does lend itself to such verification. This is our ultimate target. However, for this report, we will stick with VHDL to make the issues of how to structure a spider + plant combination clear.

It seems obvious that one would want to spider in realtime the actions of an FPGA application. What is not so obvious are how these spider + plant combinations are to be structured. This report covers some salient features of the basic structure of a spider + plant combination.

A spider contains recognition code for detecting the operation of the plant and mitigation code for handling the fallout of a detected fault. One might think that every action of the plant should be checked in the spider. However, a moment's thought should convince one that to do this is essentially to recreate the plant in the spider and add extra circuitry for the checking. While this is feasible for small applications, it is very infeasible for larger applications. As applications get larger, we expect that spiders will target only specific parts of the plant necessary for specific high assurance properties.

The reasons for expecting a spider to be smaller is that it implements what is known in computer science as an *abstract interpretation* [3]. The stock simple example of an abstract interpretation is the rules for a *multiply* operation where two negative numbers yield a positive number, a negative and a positive yields a negative, and a positive and a positive yield a positive. Thus, the interpretation abstracts over the actual values and yields just characteristics. The characteristics are reified into actual abstract values¹

A spider (forgetting about mitigation) seems to act as an abstract interpretation checker. We do not want to check the entirety of the plant's signals but rather a subset of the plant's signals and possibly even abstractions of those.

One can think of a spider much like the output of YACC (Yet Another Compiler Compiler). In YACC, context free expressions describing the language are written as "production" rules of the form

$$\langle A \rangle := \text{if } \langle B \rangle \text{ then } \langle C \rangle; \{ \text{some C code for building a bit of the parse tree} \}$$

¹The wikipedia at https://en.wikipedia.org/wiki/Abstract_interpretation contains a nice synopsis of abstract interpretation.

The non-terminal symbol $\langle A \rangle$ is expanded to the right hand side. Hence the parser is looking for “if”, then some Boolean condition, then the “then”, then some statement $\langle C \rangle$, and finally, the semicolon. The $\langle B \rangle$ and $\langle C \rangle$ are other non-terminal symbols and hence must be expanded further in the description. The “if”, “then”, and “;” are terminal symbols and require no further expansion. The code in the brackets { } is C code that will get inserted into the generated parser. It is this code that connects merely parsing the input and the internal parse tree representation of the parsed input by building that parse tree.

YACC processes a file containing statements like the above and produces a push-down automata, which is C code. The C code can then be run and expects to process your source code file and produce a parse tree. The parse tree is then handed to code generators that produce the object code².

Our intention is that a language for spiders is a combination of a logical language and a mitigation language. The logical language acts much like the production rules (BNF grammar). The mitigation language functions as the active code that builds parse trees in parsers produced by YACC. We would like to build a compiler that reads logical statements + mitigation code and produces a spider. YACC thus is a proof of concept that this sort of architecture is well-tried albeit here it is for an entirely different purpose. At a high-level though, we can think of spiders and YACC produced parsers as being much more similar. Consider the state machine of a plant as accepting *regular expressions* of event types. The actions are quite like the active code telling the plant to do things. A spider is similar; it recognizes regular expressions as event types of the plant plus, maybe, some of its own. The spider’s mitigation code is much like the plant’s active code, except that here the spider’s active code is mitigating the effects of a detected fault.

2. LOGIC PRELUDE

The reason for the use of logic is that conditions that a spider checks must be specified in some formal language if they are to eventually be mechanized to produce the actual spider code. Since they are conditions, they are best specified using a logical language that can combine conditions using notions like implication, conjunction, disjunction, and negation. There is an added complication in that the conditions are about discrete components in a plant. So a language for spiders should have a facility for expressing the component structure built into the language rather than encoded on top of a language. The goal is to keep the layers of abstraction (and there are many here) to a minimum.

The basic logic is detailed in [4] and in the Appendix of this report. We use the term *Distributed Logic* (DL) in a general sense such as *Modal Logic* (ML). A *mode* is a particular aspect of presentation. An example is that a person might be a spouse in one mode, a friend in another mode, and a supervisor in yet another. Traditional Modal Logic allows one to specify *necessity* and *possibility*. Necessity expresses the notion that a proposition “necessary *P*” is true *here* if it is true in every world accessible from *here*. In terms of FPGAs, *worlds* are states, *here* is a state, and an *accessible world* from *here* is a state *there* such that *here* transits to *there* under some event such as the rising edge of a signal. Possibility expresses the notion that there is at least one world accessible from here, i.e., there is at least one state *there* that the current state *here* transits to under some event.

DL also includes ML as simple case. A particular distributed logic is actually a collection of *local* modal logics that are connected in a formal way via distributed connectives. Each local modal logic has a classical

²Modern compilation is more complicated

base that admits the usual necessity and possibility connectives that abstract over next-state relations (these local connectives can be augmented with other modal connectives [5]).

We need a language, preferably a logical language, in order to talk about the meaning or semantics of DL. This language is known as a *meta-language*. A semantics is a map from the object language, in our case DL, to a mathematical model consisting of sets of states, functions, and relations. The semantics is expressed in the meta-language. The semantics is generally given in a *first-order language*. This is a language that allows us the usual logical notions (not, conjunction, etc.) but also allows us to quantify over individuals of some domain, say a domain of states or a domain of signals. The following first 4 symbols are used in both DL and the meta-language. The two quantifier symbols, $\forall$ and $\exists$ are used in the meta-language.

The typical logic symbols are

next-state Relation $\mathcal{H}$		
Symbol	Logical Meaning	Colloquial Example
$\neg$	Negation	Not
$\wedge$	Conjunction	And
$\vee$	Disjunction	Or
$\supset$	Implication	If-then
$\forall$	Universal Quantifier	For Every (Each)
$\exists$	Existential Quantifier	There Exists at Least One
$\in$	“In” Relation	Element of a set, e.g., $x \in P$ “the element x is in the set P ”
$\subseteq$	“Subset Relation”	Subset and Possibly Equal

Table 2.1: The Usual Symbols of Logic and Set Theory and Their Meanings

Set theory is itself written in a first-order language. Set theory has a set of axioms and a set of rules telling us how to get new statements in set theory from old statements.

We frequently use the locution P iff Q for

$$(P \text{ implies } Q) \text{ and } (Q \text{ implies } P).$$

This is sometimes used in the sense of a definition. Let r and s be sets but we wish to treat them as elements of a graph, then

$$r \leftrightarrow s \text{ iff } s \in r.$$

means we are defining the symbol $\leftrightarrow$ over sets.

Two objects x and y are presented as related under a binary relation $\mathcal{F}$ with $\mathcal{F}xy$. Sometimes, when $\mathcal{F}$ is actually a function (as well as a binary relation), then we will use $\mathcal{F}(x)$ and $\mathcal{F}(y)$.

So state a function g as $g : U \rightarrow V$ is to say that g takes a value in U and returns a value in V . To say that for objects u and v that $u \mapsto v$ is to say that u is related to v under some unnamed (usually given by context) relation.

2.1 The Logic

In this report, we do not extensively use DL as a canon of reasoning so much as we need it for its language. The logic will come in handy later in the project when we wish to derive properties of spider + plant combinations. The logic at this point is merely providing a language in which to express properties. In evaluating logical statements for any system, the *propositions* are statements we make about some universe of discourse, i.e., the spider or the plant. We can also, in some instances, treat a proposition under semantic interpretation as a collection of states. We frequently slip back and forth between the two notions of proposition since they are so intimately connected when talking about states.

DL lends itself well to FPGA applications. Each component is seen as having its own *local logic*. The components are connected via their behavior; that behavior is expressed using distributed logical connectives. The distributed connectives abstract over those relations. The abstraction takes the form of the evaluation conditions on the operator as shown in the sequel. The use of the term *distributed relation* reflects that a relation is relating two distinct collections of states in different components. This is in contrast to the local *endo-relations* (such as next-state relations); these latter are limited to a single locality

We use the term *locality* to denote a local logic and its underlying component structure as expressed in its states and relations. Thus, the distributed relations connect localities. A common distributed relation is a *parallelism* relation, called *concurrency*, that represents when states in two different localities can simultaneously occur. Another distributed relation is one that relates all components to each other that share the same clock domain.

2.2 Interpreting Logic

A logic, any logic, is expressed in a language that we call the *object language*. The interpretation of the logic is a map from formulas of the logic to (usually) mathematical terms. This parallels non-formal languages such as English. Think of the syntax of English as given by those diagrams you may have learned in grade school for diagramming sentences. The semantics, or meaning, is usually given in English itself but that is via necessity. If we wanted, we could interpret the language in another language, say, German. The two languages are not considered on par with each other when the second is used in semantics. The language we use to express the meaning of formulas in the object language is called the *metalanguage*. We generally use the language of mathematics for interpreting the logical language. In our case, we use points (representing states), sets, and relations. Functions may be used but these are just binary relations (two-place) relations with special properties, i.e., they are defined over their entire domain, and that no single element can be mapped to two distinct elements in the codomain.

We will interpret a logic in terms of *states*, which for us will typically arise with respect to components in an FPGA application. The term *state* can mean different things to different communities. In a generic sense, a state represents a cross-section through a system where the system is thought of as having a set of

possible trajectories though a state space. The allowable trajectories give a notion of what states can follow other states, the entire space is a sea of states.

In computer science, a state is thought of as the entire contents of memory, inputs and outputs of a computer system at a particular time. This notion typically does not get one very far in proving properties or describing properties about a system. Usually, as in physics, a notion of coarse-graining is used, i.e., grouping together states into aggregates. The aggregates are frequently termed *macro-states* and the states they arise from are called *micro-states*. Some people in computer science call the macro-states merely states and the micro-states part of a *store*. We will not use this nomenclature in this document. The coarse grained aggregates are those you see called out in case statements of VHDL.

Engineers working on FPGAs typically dissect an application into macro-states and assign a state variable to record which macro-state is in force at any one time. The micro-states are left unmentioned. It can happen that one might construct two FPGA applications for a common problem with each application having different collections of macro-states. This is usually rare because the problem being addressed lends itself to a natural collection, where *natural* means that most engineers would devise the same collection. However, this issue does come up in spidering an application where tracking state changes in a plant need not entail mirroring those same states in the spider. The spider can abstract away states that are not important from a high-assurance perspective.

A proposition in a logical language is some linguistic formula that we wish to reason with, say $x = 2$ where 2 is considered fixed and x is considered to be variable. As a linguistic entity, it is just that, i.e., a string of symbols with some internal structure but that structure is not identifying any specific value for x . The proposition $x = 2$ is uninterpreted at this point. We can consider the collection of states that make this proposition true. Which states are those? All the states in which the variable (memory location) x actually has the same value of the interpretation of the constant 2. We expect any interpretation is sensible in that it evaluates the constant 2 as the number 2. The constant 2 is a linguistic entity and nothing more. The number 2 is an abstract mathematical object. You cannot hold a mathematic object in your hand like you can cut out the 2 on a piece of paper and hold it.

We frequently collect together all the states that turn a proposition true and use that collection and the proposition interchangeably with context disambiguating whether we mean the linguistic entity or the set of states making that proposition true.

A collection of states, either micro or macro, is considered a proposition in a logic of sets. One way to present this is in terms of a satisfaction relation $\models$. The locution $x \models P$ is read as “the object (say a state) x satisfies or makes true the proposition P ”. In symbols, the semantics $\llbracket P \rrbracket$ of a proposition P is specified as

$$x \in \llbracket P \rrbracket \text{ iff } x \models P.$$

Typically, we elide the brackets and simply use $x \in P$ move freely back and forth between thinking of P as a set and P as a linguistic proposition. The reason we are allowed to do this is that we can define

$$P \equiv Q \stackrel{\text{def}}{=} (P \supset Q) \wedge (Q \supset P),$$

then $\equiv$ defines an equivalence relation (reflexive, symmetric, and transitive) given the axioms and rules of classical logic. As such, it divides the collection of logical formulas into equivalence classes which we denote $|P|$ for a proposition P . Hence

$$|P| = \{Q \mid P \equiv Q\},$$

so that

$$|P| = |Q| \text{ iff } P \equiv Q.$$

Algebraically, we are dividing the set of formulas out by bi-implication ($\equiv$). Given the semantic rules, we have

$$|P| = |Q| \text{ iff } \forall x(x \models P \text{ iff } x \models Q).$$

In terms of sets with $\in$ for $\models$, we have

$$|P| = |Q| \text{ iff } \forall x(x \in P \text{ iff } x \in Q).$$

A proposition P is then considered a statement about states. A typical proposition is

$$P(x) = \text{The traffic light is } x,$$

where x is intended to range over red, yellow, or green. Notice that x represents a “hole” in the statement. This could equally have been stated as

$$P(-) = \text{The traffic light is } -.$$

As there is only a single hole, this is adequate. To represent a more complicated proposition, say, $Q(-, -)$, we would have to name the holes; this is precisely what variables do. Also, the variable “ x ” in $P(x)$ is determined, i.e., given a value, in the surrounding mathematical environment, much like for a function f , say,

$$f(x) = x^2$$

is a formula where the value of x is set by the surrounding mathematical environment. We are used to treating x as some undefined quantity ranging over some set, in this case, the set of real numbers. The notation $P(x)$ is entirely analogous. At the level of language, f and x are linguistic entities. We frequently think of them as already interpreted as a function and a point. Similarly, we can treat P and x either as linguistic entities or as a set, the latter is P ’s *interpretation*. In this sense, P is a propositional function, i.e., a function from the set of states to the set of values $\{\top, \perp\}$ where $\top$ represents *true* and $\perp$ represents *false*, and which we shorten to merely “proposition”. In engineering, true and false can be represented as 1 and 0 respectively. However, this can sometimes conflate a semantic (mathematical) value with a voltage on a wire. In mixed hardware logic, this may not hold as part of a circuit may represent true as 0 and false as 1. Using mixed logic allows for a certain simplification since always converting a 0 to a 1 requires an inverter.

Now we will connect the logical language with the model theoretic setting where the locution “model theoretic” means a collection of situations where formulas are either true or false.

Using a concrete situation such as a traffic light, to say that x is either red, yellow, or green, we frequently rely upon set theoretic notation, i.e.,

$$x \in \{\text{red, yellow, green}\}.$$

Also, since there is no order in sets and repetitions do not matter,

$$\{\text{red, yellow, green}\} = \{\text{yellow, red, red, green}\}.$$

where the $=$ sign is standing for equality in the meta-language.

Propositions are either *true* or *false* in a particular setting. The setting for our concrete situation is a traffic light and we must specify a particular state to get a completely defined setting. So we are left with specifying the truth or falsity of a proposition at each state.

We can treat P as a linguistic entity in a logical language or we can treat it as set of those points at which the proposition is true. The locution “points” is more common in logic and we freely use points and states interchangeably.

To say that $P(x)$ is true only for $x = \text{red}$ can be represented as

$$P = \{\text{red}\}.$$

Hence

$$P(x) \text{ is true iff } x \in \{\text{red}\}.$$

The short hand “iff” means this statement stands for

$$(P(x) \text{ is true}) \text{ implies } (x \in \{\text{red}\}) \quad \text{and} \quad (x \in \{\text{red}\}) \text{ implies } (P(x) \text{ is true}).$$

In effect, the iff locution allows us to use $P(x)$ interchangeably with the set $\{\text{red}\}$. We often elide the parentheses since “iff” and “implies” only make sense between complete statements. x by itself is not a complete statement whereas $x \in P$ is a complete statement.

It is helpful to switch between P as a linguistic entity in a language and under an interpretation as a set. The latter is to treat P as a “UCLA proposition”, i.e., a set of points. To evaluate logic formulas, we use the symbol $\models$. The term *models* ($\models$ is its symbol) is a relation between points and propositions and hence a point-proposition pair is either in the relation it is not. In other words, the point-proposition is considered true under the relation or it is not. Similarly, $\in$ when used in $a \in P$ is a relation between points and P as an UCLA proposition.

Definition 2.2.1 A Boolean set lattice $\mathbb{H} = \langle H, \cap, \cup, \neg \rangle$ is a collection of points or states H , set intersection $\cap$, set union $\cup$, and set complement $\neg$. The top of the lattice, $\top^{\mathbb{H}}$ is H , i.e., all the points, and the bottom of the lattice, $\perp^{\mathbb{H}}$ is the empty set $\emptyset$, i.e., none of the points.

Definition 2.2.2 A *local classical frame* or *classical locality* is a structure $H = \langle H, \mathbb{H} \rangle$ such that H is a collection of points called, generically, a *domain*. We use the same symbol for the frame and its collection of states, and let use disambiguate meaning. $\mathbb{H}$ can be alternately thought of as (1) a classical propositional logic, or (2) as a collection of *neighborhoods* or sets of states that are subsets of H and the entire collection is closed under the Boolean operations. Hence $\mathbb{H}$, in this latter case, is a Boolean set algebra. The top proposition is $\top^{\mathbb{H}}$ thought of as always *true* and the bottom proposition $\perp^{\mathbb{H}}$ is thought of as always false. As sets in the set algebra, the set $\top^{\mathbb{H}}$ is the top of the Boolean lattice of sets and is the collection of all states, i.e., H . The set $\perp^{\mathbb{H}}$ is the bottom and is the empty set $\emptyset$.

A diagram of a locality that includes a local logic and its interpreting states is

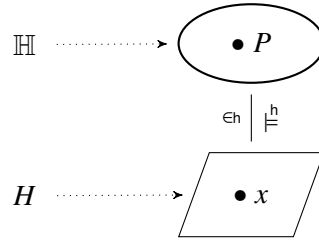


Figure 2.1: Intuitive Diagram of a Locality

where $\models^{\mathbb{H}}$ is used if $\mathbb{H}$ is a collection of propositions and $\in_{\mathbb{H}}$ is used if $\mathbb{H}$ is a collection of sets. The $\mathbb{H}$ in $\in_{\mathbb{H}}$ indicates the frame $\mathbb{H}$ and is the same as the unadorned $\in$ of bog standard set theory.

Since the set $\mathbb{H}$ is either a set of formulas or a collection of sets of states, we will use the locution $P \in \mathbb{H}$ to mean either with use disambiguating meaning. Given the fluidity of how to consider a proposition P , it usually makes no difference which you choose.

The symbols

$$a \models^{\mathbb{H}} P \quad a \not\models^{\mathbb{H}} P$$

mean (respectively) the proposition P is true in state a at locality $\mathbb{H}$ and the proposition P is false in state a at locality $\mathbb{H}$. The former is an assertion that P is true of a at locality $\mathbb{H}$ and the latter is an assertion that P is false of a at locality $\mathbb{H}$. This notation frees us to place extra restrictions on a and on $\models$. In fact, you already have seen one where $\models^{\mathbb{H}}$ is restricting the $\models$ relation to the locality $\mathbb{H}$.

This notation allows us to treat the *models* symbol, $\models^{\mathbb{H}}$, as the set theoretic membership $\in$ at locality $\mathbb{H}$. The latter can also be displayed as $\in_{\mathbb{H}}$. An *atomic* proposition P from a logic has no internal logical structure, i.e., no embedded $\neg$, $\wedge$, etc.. An atomic UCLA proposition is one where there are no other propositions Q such that $\emptyset \subseteq Q \subseteq P$ (where the empty set $\emptyset$ stands for falsity $\perp$).

Let Aprops be the set of *atomic propositions*. A valuation $\llbracket - \rrbracket : \text{Aprops} \rightarrow \{\top, \perp\}$ yields true or false for every atomic proposition. The $\models^{\mathbb{H}}$ relation is extended inductively for $\llbracket - \rrbracket$ a valuation on the atomic propositions:

- $a \models^h P$ iff $a \in \llbracket P \rrbracket$ for $P \in \text{Aprops}$,
- $a \models^h \neg P$ iff $x \not\models^h P$,
- $a \models^h P \wedge Q$ iff $a \models^h P$ and $a \models^h Q$,
- $a \models^h P \vee Q$ iff $a \models^h P$ or $a \models^h Q$,
- $a \models^h P \supset Q$ iff $a \not\models^h P$ or $a \models^h Q$,

where a state a is not merely a list of values for all the propositions but must also record values for individual variables in the logic. That is, $a[v/x]$ means that when is evaluating $P(x)$, it looks up the value of x in its list of variable-value pairs and uses the value stored there, call it v .

Incidentally, the clause for $P \supset Q$ can be written

$$a \models^h P \supset Q \text{ iff } a \models^h P \text{ implies } a \models^h Q.$$

In general, these are the clauses for evaluating classical logic. The logic of the meta-language is also classical and hence when reasoning in it, we use similar valuations.

Notice that we can also code these in terms of set theory. Let H be the collection of all states and here, $\sigma : \text{Aprops} \rightarrow \mathbb{H}$ is a valuation on the atomic propositions that returns a collection of points where for any $P \in \text{Aprops}$, $\sigma P \in \mathbb{H}$:

- $a \models^h P$ iff $x \in \sigma P$ for P an atomic proposition,
- $a \models^h \neg P$ iff $x \in H - P$ iff $x \notin P$ set difference,
- $a \models^h P \wedge Q$ iff $a \in P \cap Q$ set intersection,
- $a \models^h P \vee Q$ iff $a \in P \cup Q$ set union,
- $a \models^h P \supset Q$ iff $a \in \neg P \cup Q$ set inclusion, i.e., if for all a , $a \models^h P \supset Q$, then $P \subseteq Q$.

This means that we can treat the collection of all propositions as a Boolean set lattice.

Next, we extend the *models* symbol, $\models^h$, as a relation, to do more work by leaving the left hand side unspecified:

- $\models^h P$ iff for all $a \in H$, it is the case that $a \models^h P$. Equivalently, $\models^h P$ if and only if for all $a \in \top^{\mathbb{H}}$ (where $\top^{\mathbb{H}}$ is the top of the Boolean set algebra $\mathbb{H}$, i.e. H here), it is the case that $a \models^h P$.
- $\not\models^h P$ iff there is some $a \in H$ and $a \not\models^h P$.

- A local logic at h is consistent just when for all $a \in {}^h H$, it is the case that for all propositions P , not both $a \models^h P$ and $a \not\models^h P$.

This semantics is part of the meta-language, which is the language we use to talk about the language of the logic in which we are interested. Typically, we are not so pedantic and simply use variables $x, y, \dots$ in both the meta-language and the logic letting use disambiguate what is meant. We frequently must quantify over the meta-language to get the correct level of generality. In that case, it is possible to formalize the meta-language a bit with, in addition to the clauses above:

- $a \models^h \forall x(P(x))$ iff for all $v, a[v/x] \models^h P(x)$,
- $a \models^h \exists x(P(x))$ iff there is some $v, a[v/x] \models^h P(x)$.

where the variable v is a quantified variable in the meta-language, a is an arbitrary state in the meta-language, and x is a variable in the object or logical language in which we are interested.

A *theorem* of a logic is one that is provable from the logic. That is, a theorem is either an instance of an axiom of the logic or derivable from the axioms by the use of a finite number of rules of deduction from the axioms. An example of an axiom of classical propositional logic is

$$A \supset (B \supset A).$$

An example of a rule is *modus ponens*

$$\frac{\vdash A \quad \vdash A \supset B}{\vdash B}$$

where $\vdash$ means “provable in the logic”. We typically elide the $\vdash$ in displaying the use of any rule.

A logic is *sound* with respect to a semantics if every theorem is evaluated to true in the semantics. A logic is *complete* with respect to a semantics if every true formula of the logic is a theorem of the logic. We typically present a logic with a finite set of axioms and a finite collection of rules. For example, the formula in classical logic

$$(A \supset B) \supset ((C \supset B) \supset (C \supset A))$$

is true in the truth functional semantics were we assign $\perp$ and $\top$ (standing for *false* and *true*) to the propositions A, B , and C in every combination. Since truth tables are typically use with 0 and 1 rather than $\perp$ and $\top$, we will use 0 and 1 for truth tables. The truth table for $A \supset B$ is

f	P	Q	$P \supset Q$
f_0	0	0	1
f_1	0	1	1
f_2	1	0	0
f_3	1	1	1

Table 2.2: Truth Table: for $\supset$

The f_i are truth functions, i.e., $f_2(P, Q) = 0$.

Now we can compute the following truth table:

f_i	A	B	C	$A \supset B$	$B \supset C$	$A \supset C$	$(B \supset C) \supset (A \supset C)$	$(A \supset B) \supset ((B \supset C) \supset (A \supset C))$
f_0	0	0	0	1	1	1	1	1
f_1	0	0	1	1	1	1	1	1
f_2	0	1	0	1	0	1	1	1
f_3	0	1	1	1	1	1	1	1
f_4	1	0	0	0	1	0	0	1
f_5	1	0	1	0	1	1	1	1
f_6	1	1	0	1	0	1	1	1
f_7	1	1	1	1	1	1	1	1

Table 2.3: All the Truth Functions in Three Variables

The f_i are interpretations, i.e., semantic functions, and the formula $(A \supset B) \supset ((B \supset C) \supset (A \supset C))$ is true under all these interpretations.

Completeness is usually argued contrapositively. Let

$$\models P$$

hold when P is true under all interpretations. Let

$$\vdash P$$

hold when P is provable in the logic. Completeness says that

$$\models P \text{ implies } \vdash P.$$

The contrapositive is

$$\not\vdash P \text{ implies } \not\models P.$$

Unwinding this a bit, this says that if P is not provable, then there is some model in which P comes out false. As an example, the formula $(A \supset B) \supset (\neg A)$ is not a theorem. If it is not a theorem, then we must find a truth function which turns it false. Given the truth table for $\supset$ we must find a function that turns $A \supset B$ true and $\neg A$ false. An easy way to get on is the following *pebbling* of its formation (parse) tree:

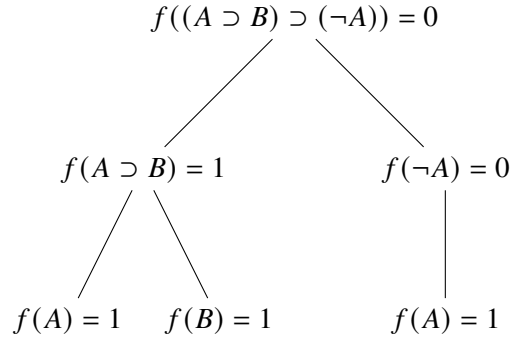


Figure 2.2: Intuitive Diagram of a Locality

where $f((A \supset B) \supset (\neg A)) = 0$ requires that $f(A \supset B) = 1$ and $f(\neg A) = 0$. From there, $f(A) = 1$, and if $f(A) = 1$, then $f(B) = 1$. It is a theorem of classical logic that

$$\vdash \top \supset P \text{ implies } \vdash P$$

where $\top$ is always true under any evaluation. Taking the contrapositive

$$\not\vdash P \text{ implies } \not\vdash \top \supset P.$$

Hence if we assume $\not\vdash ((A \supset B) \supset (\neg A))$, then we can conclude $\not\vdash \top \supset ((A \supset B) \supset (\neg A))$. In order for this latter to be turned false under some interpretation, then that interpretation must turn $\top$ true and $((A \supset B) \supset (\neg A))$ false. The above pebbling turns $((A \supset B) \supset (\neg A))$ false and represents an interpretation. Any interpretation must turn $\top$ true. If any interpretation turns $((A \supset B) \supset (\neg A))$ false, then $\not\models ((A \supset B) \supset (\neg A))$. This argument says that

$$\not\vdash ((A \supset B) \supset (\neg A)) \text{ implies } \not\models ((A \supset B) \supset (\neg A))$$

and taking the contraposition, we have shown

$$\models ((A \supset B) \supset (\neg A)) \text{ implies } \vdash ((A \supset B) \supset (\neg A)).$$

To put a finer point on it, we have shown that $\models ((A \supset B) \supset (\neg A))$ is never true because then the formula must be true under every interpretation. However, we have shown one interpretation where it is not true, hence we cannot conclude the consequent, i.e., that $((A \supset B) \supset (\neg A))$ is provable.

2.3 Interpreting Distributed Logic

Our language is not quite expressive enough, we need to be able to talk about *next states* because our intended interpretation is over state machines. We assume there are at least two localities, which we will denote h and k . These correspond to processes in VHDL and each is structured using a state machine.

Distributed Logic does not care how one associates VHDL constructs to localities. For typical VHDL code, associating processes with localities is sufficient. The fact that each process is a state machine means the logic must have a notion of talking about the underlying state machine at a locality. For now, we will treat the state machine using a single next-state relation. In practice, a state machine changes state because of many different kinds of events and each kind of event will have an associated next state function. We will collect events, which are entities at runtime (e.g., rising edges of signals), together into different event types (their kinds) that we can use in the mathematics. A state transiting to another state upon an event will do that regardless of the first, second, etc. event of the same kind at that state. The notion of *event type* abstracts over these individual events since all these specific share a common set of characteristics.

The next-state relation could represent a deterministic state machine, in which case the next-state relation is a function. Or it could represent a non-deterministic state machine, in which case the next-state relation is not a function but a more general relation. We will also have need of representing relationships between localities, relations will work here for us as well.

The typical logic way of talking about propositions that can talk about situations that may change is via Modal Logic. Distributed Logic is generalized form of Modal Logic where the relations can be next-state relations at a locality or represent relationships between states at different localities such as a *concurrency* relation. A concurrency relation expresses when a state at one locality can co-occur or *concur* with a state at another locality. The local next-state relations are typically termed *Kripke* relations (after Saul Kripke who was instrumental in applying them to Modal Logic). These are *binary* relations and we generally depict two entities x and y being in a relation $\mathcal{H}$ with each other using $\mathcal{H}xy$. Other presentations sometimes used (but not here) are $\mathcal{H}(x, y)$ and $x \mathcal{H} y$.

We think of concurrency relations as *distributed Kripke relations*. The distributed concurrency relations have quite a different character than local next-state relations. DL really does not care what kind of relations you use, any binary relations for the form of DL we use here will work.

Set theoretically, a relation $\mathcal{F}$ is a collection of pairs of points. If the point sets are H and K , then $\mathcal{F} \subseteq H \times K$ where

$$H \times K = \{\langle x, y \rangle \mid x \in H \text{ and } y \in K\}.$$

Technically, $H \times K$ is known as a *cross-product*. We denote $\langle x, y \rangle$ being in the relation $\mathcal{F}$ as either $\langle x, y \rangle \in \mathcal{F}$ or, more usually, $\mathcal{F}xy$. To indicate that $\mathcal{F}$ is a relation between H and K , we use the locution $\mathcal{F} : h \rightarrow k$, where by fiat we say the relation $\mathcal{F}$ is from h to k . This implies that the localities h and k have the sets H and K as their point sets. Technically, relations do not have a direction, only sides. However a direction comes in handy when defining distributed connectives. Note that $H \times K \neq K \times H$, the set of *ordered* pairs $H \times K$ is different than the set of ordered pairs $K \times H$.

The relation $\mathcal{F}$ is model theoretic in that it only exists between point sets used to interpret logical formulas. We will need some way in the logic to refer to such a relation in the logic and for that we use the lower case f . We also use the locution $f : h \rightarrow k$ when concentrating on the logic and not the underlying setting. We use the convention that a $f : h \rightarrow k$ in the logic will be interpreted by a relation $\mathcal{F} : h \rightarrow k$ in a model.

Modal formulas $[f^\circ]$ and $[f^\circ]$ stand for the notions of necessity and possibility. These have specific meanings in Modal Logic and consequently in Distributed Logic interpreted using the underlying relation $\mathcal{F}$. The underlying relation can be a next-state (next-point) relation or some other relation.

The formulas $[f^\circ]\rangle$ and $[f^\circ]\rangle$ are evaluated in the usual way except we must respect the distributed nature now of these modalities:

$$x \models^h [f^\circ]\rangle Q \text{ iff for all } y, \mathcal{F}xy \text{ implies } y \models^k Q \quad x \models^h [f^\circ]\rangle Q \text{ iff there exists } y, \mathcal{F}xy \text{ and } y \models^k Q.$$

The way to read the the first evaluation is that if you are standing at point x and wish to evaluate the proposition $[f^\circ]\rangle Q$, then you look out along the relation $\mathcal{F}$. If every place y you can visit under $\mathcal{F}$, i.e., $\mathcal{F}xy$, implies that Q is true there, then evaluate $[f^\circ]\rangle Q$ as true at x . If there is some place y you can visit under $\mathcal{F}$ that makes Q false, then evaluate $[f^\circ]\rangle Q$ as false at x .

The way to read the the second evaluation is that if you are standing at point x and wish to evaluate the proposition $[f^\circ]\rangle Q$, then you look out along the relation $\mathcal{F}$. If at least one place y you can visit under $\mathcal{F}$, i.e., $\mathcal{F}xy$, makes Q is true there, then evaluate $[f^\circ]\rangle Q$ as true at x . If for all places y you can visit under $\mathcal{F}$ make Q false, then evaluate $[f^\circ]\rangle Q$ as false at x .

Other versions $[f^\circ]\rangle$ and $[f^\circ]\rangle$ run in the other, converse or backwards, direction:

$$y \models^k [f^\circ]\rangle P \text{ iff for all } x, \mathcal{F}xy \text{ implies } x \models^h P \quad y \models^k [f^\circ]\rangle Q \text{ iff there exists } x, \mathcal{F}xy \text{ and } x \models^h Q.$$

Note these are the same as

$$y \models^k [f^\circ]\rangle P \text{ iff for all } x, \check{\mathcal{F}}yx \text{ implies } x \models^h P \quad y \models^k [f^\circ]\rangle Q \text{ iff there exists } x, \check{\mathcal{F}}yx \text{ and } x \models^h Q.$$

where the converse $\check{\mathcal{F}}$ of the relation $\mathcal{F}$ is used. That is $\check{\mathcal{F}}yx$ iff $\mathcal{F}xy$.

Setting $h = k$ yields the evaluation conditions for the corresponding local modality. Hence the distributed modalities are evaluated similar to the local modalities except notice the changes between $\models^h$ and $\models^k$ on the two sides of the evaluations.

The local logics are evaluated according to the previous rules for classical logic extended with these modal rules for formulas that stay within the logic logic. You will notice that every formula has a local logic to which it belongs even if evaluating the distributed modal connective require that the evaluation hops around the graph of localities as needed.

We assume a graph of localities usually denoted with sans serif, i.e., nodes are denoted h , k , etc. At each node is a *classification* consisting of a local logic or Boolean algebra with local modal operators, i.e., a modal algebra, a collection of points, satisfaction relations $\models^h$, $\models^k$, and at least one local relation. The local relation can be the identity relation (for all x , $\mathcal{I}_hxx$) if no feature of a component needs to be represented using a local relation. For the example of spidering a plant, there will be two local next-state relations, $\mathcal{K}$ and $\mathcal{H}$. Incidentally, a logic, being syntactic, has *connectives*. An algebra, being a model theoretic notion, has *operators*. We tend to slip back and forth between logics and their algebraic models. So we sometimes use the term *operators* to refer to both operators and connectives when we ambiguously using either an algebra or a logic.

In the definition of a frame (see below), we use the algebraic structure of a lattice. Here, we use modal set algebras, i.e., Boolean lattices with modal operators on them. In the case of a local logic at a locality, the propositions of the logic are interpreted by sets of the modal set algebra. In the case of a modal propositional logic of sets, the sets are sent into copies of themselves in the Boolean lattice of the frame under an interpretation, or if you like, the propositions constitute the modal set algebra.

Definition 2.3.1 A *local frame* is a structure $\mathfrak{h} = (H, \mathcal{H}, \mathbb{H})$ such that H is a collection of points called, generically, a *domain*. $\mathcal{H} : H \rightarrow H$ (although we sometimes use $\mathcal{H} : \mathfrak{h} \rightarrow \mathfrak{h}$) is a local relation connecting some points of H . $\mathbb{H}$ is a collection of *neighborhoods* or sets of points that are subsets of H and the entire collection is closed under the Boolean operations and under the generic modal operations $[h], [h^\circ] : \mathbb{H} \rightarrow \mathbb{H}$. Hence $\mathbb{H}$ is a modal set algebra. These operators are used to define special collections of points. They have valuation conditions in the sequel given by distributed operators, just set the distribution to a single component. The $\in$ relation between states and elements of the Boolean algebra is left implicit. The set $\top^{\mathbb{H}}$ is the top of the Boolean lattice of sets and is the set H . The set $\perp^{\mathbb{H}}$ is the bottom and is the set $\emptyset$.

We collect the information of two processes into the frames $\langle H, \mathcal{H}, \mathbb{H} \rangle$ and $\langle K, \mathcal{K}, \mathbb{K} \rangle$, where H and K are sets of points, $\mathcal{H}$ and $\mathcal{K}$ are next-state relations, and $\mathbb{H}$ and $\mathbb{K}$ are lattices of propositions. A proposition can be thought of either as some linguistic entity or as a collection of points. The two tuples are called *local frames*. They are local in the sense they do not include any information from each other.

A good mental picture to remember the definition of a frame is the following diagram where P is either proposition of a language or a UCLA proposition. This latter is merely a set of points in a lattice of sets. In the former case, the $\in_{\mathfrak{h}}$ relation becomes $\models^{\mathfrak{h}}$ and in fact, in interpretations, $\in_{\mathfrak{h}}$ is the meaning of $\models^{\mathfrak{h}}$.

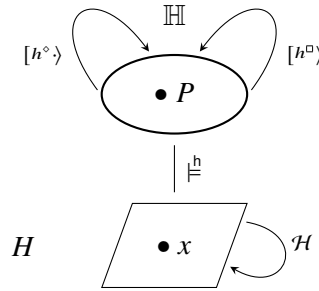


Figure 2.3: Intuitive Diagram of a Locality with a Local Modal Relation

Here x is some point in the domain H and P is either a proposition or some UCLA proposition in $\mathbb{H}$, but it is not necessarily the case that $x \models^h P$. Two example modal operators $[h^\diamond]$ and $[h^\circ]$ are shown. They are the usual modal necessity and possibility operators $\Box$ and $\Diamond$ from modal logic but put into our notation. The relation $\mathcal{H} : \mathfrak{h} \rightarrow \mathfrak{h}$ is some set $\mathcal{H} \subseteq H \times H$. Note that $\mathfrak{h}$ refers to the node at which the frame indicated by the diagram lives. The h in $[h^\diamond]$ and $[h^\circ]$ refers to the relation $\mathcal{H}$ in any interpretation.

2.4 Distribution

The distributed operators take propositions in one locality (component) and translate them to propositions in another.

Definition 2.4.1 A *distributed frame* consists of a graph of nodes where each node is a local frame, distributed Kripke relations link the domains, and distributed modal operators link the logics or the set algebras.

Propositions in the logic are modeled by elements of the set algebras. A distributed frame for modeling two components has two localities. The next diagram depicts this situation (without showing the local modalities or local relations) and two (generic) distributed modalities:

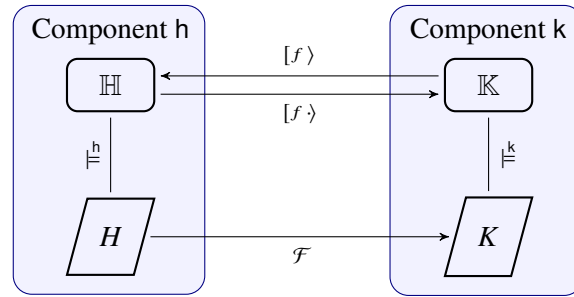


Diagram 2.1: Generic Distribution with Two Components as Localities

where the arrows $[f >$, $[f <$ can be any of the modal forward and backward distributed modalities interpreted by the distributed Kripke relation $\mathcal{F}$, i.e., the lower case f in the modalities is linked with the script $\mathcal{F}$ relation. Example pairs of modalities are $[f^\circ >$ and $[f^\circ <$ and are necessity operators, and $[f^\diamond >$ and $[f^\diamond <$ are possibility operators.

The diagram is only showing the simplest of distributed frames. In any application, even one with only two localities, there can be any number of distributed relations and any number of distributed modalities.

The distributed relation $\mathcal{F} : h \rightarrow k$ used in the evaluation of the operators above uses two localities, h and k . Here, h and k are variables in that any actual FPGA application will fill in those localities by two components; the entire distributed frame will use as many localities as there are components and possibly more if individual processes within a component must be singled out for analysis. Also, any one locality may have many local relations, and any two localities may be connected by many distributed relations. The modalities involved are restricted by the number of relations we have at our disposal to interpret the modalities. Technically, $\mathcal{F} \subseteq H \times K$, we leave this implicit in the notation $\mathcal{F} : h \rightarrow k$.

The distributed relation $\mathcal{F}$ (see diagram) is denoted with an arrow but this is mere convention; $\mathcal{F}$ is a two-place relation that, by fiat, is viewed as a morphism from elements of its first position to elements of its second position. The distributed modalities, on the other hand, really are functions although they have special properties required for us to treat them as modalities.

We will often use the location “ P in h ” to mean P is a proposition at locality h and hence $P \in \mathbb{H}$. A similar statement holds for “ Q in k ”.

3. STATE MACHINES

3.1 States vs. Transitions

As noted above, states of machines are points in a frame. We move back and forth between the two locations. The state machines we consider are *Mealy* machines as opposed to *Moore* machines. We will

be concerned with finite state machines (FSMs) and these can be drawn with graphs. The nodes of a graph represent states and the arcs represent event types. The arcs are event types, not events. An event is a one-time occurrence. An event type is a kind of event. Both types of machine may have actions associated with them such as setting outputs. FSMs do not have local variables such as used in VHDL, but we will simply assume these under the name of outputs. We will not have much to say about actions in this report. In Mealy machines, the actions are associated with arcs and in Moore machines they are associated with states.

We represent FSMs using local frames. There is a complication in doing this; namely there can be many different event types in an FSM. We will associate a different next-state relation to each event type. Consider the following VHDL code fragment:

```

1 case state is
2   when s0 =>
3     if (t0 = '1') then
4       state <= s1;
5     else
6       state <= s0;
7   when s1 =>
8     if (t3 = '1') then
9       state <= s0;
10    else
11      state <= s1;

```

The code corresponds to the following two-state machine:

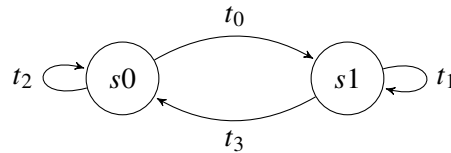


Figure 3.1: Simple State Machine

There are three event types, t_0, t_1, t_2, t_3 . In FSMs, there is a next-state function or relation δ . If the FSM is deterministic, then δ is a function, and if non-deterministic, then δ is a relation. Here, δ is a function and most next-state relations encountered in FPGAs are functions³. Next-state relations come about when we abstract away enough information that specifying a function is not possible, and hence usually occur in analysis of FPGAs, not in the actual FPGA itself, at least not intentionally. Our next-state function has the following definition:

next-state Function δ	
$\delta(s_0, t_0) = s_1$	$\delta(s_1, t_1) = s_1$
$\delta(s_1, t_3) = s_0$	$\delta(s_0, t_2) = s_0$

There are at least two ways to fit this into modal logic.

³Distributed relation are almost never functions

3.2 Multiplying the States

The if-then structure is effectively multiplying the number of states to yield

State	next-state Relation	State	next-state Relation
$\langle t_0, s_0 \rangle$	$\{ \langle \langle t_0, s_0 \rangle, \langle t_1, s_1 \rangle \rangle, \langle \langle t_0, s_0 \rangle, \langle t_3, s_1 \rangle \rangle \}$	$\langle t_2, s_0 \rangle$	$\{ \langle \langle t_2, s_0 \rangle, \langle t_0, s_0 \rangle \rangle, \langle \langle t_2, s_0 \rangle, \langle t_2, s_0 \rangle \rangle \}$
$\langle t_3, s_1 \rangle$	$\{ \langle \langle t_3, s_1 \rangle, \langle t_2, s_0 \rangle \rangle, \langle \langle t_3, s_1 \rangle, \langle t_0, s_0 \rangle \rangle \}$	$\langle t_1, s_1 \rangle$	$\{ \langle \langle t_1, s_1 \rangle, \langle t_1, s_1 \rangle \rangle, \langle \langle t_1, s_1 \rangle, \langle t_3, s_1 \rangle \rangle \}$

Table 3.1: State Explosion

As can readily be seen, there is quite a bit of state and next-state explosion. The reason is that every state must get multiplied by the number of **if** clauses in the original code, or the outgoing arcs if you prefer the state-event diagram. Let E_s be the event types labeling arcs leaving state s , then then for each state s , the multiplication of states satisfies

$$s \mapsto \{s\} \times E_s.$$

The next-state relation explodes because each state cannot predict what the next input will be since the next input is now encoded in the state and the next-state relation has no way of distinguishing among those inputs. Most of the explosion in the next-state relation is due to this. In addition, each state in $\{s_0, s_1\}$ must transit to every state in $\{s_0, s_1\}$ because the graph is strongly connected, i.e., every state in the graph is connected to every other state via a transition.

3.3 Multiplying the Next-State Relation to Many Relations

Typically in modal logic systems, there is a single Kripke relation. In the example Figure 3.1 there are four, one for each kind of event to represent the inputs $\{t_0, t_1, t_2, t_3\}$:

State-Event Relations	
Relation	Tuples
$\mathcal{T}_0$	$\{\langle s_0, s_1 \rangle\}$
$\mathcal{T}_1$	$\{\langle s_1, s_1 \rangle\}$
$\mathcal{T}_2$	$\{\langle s_0, s_0 \rangle\}$
$\mathcal{T}_3$	$\{\langle s_1, s_0 \rangle\}$

These relations are subject to the condition that the overall Kripke relation $\mathcal{K}$ is such that

$$\mathcal{T}_0 \cap \mathcal{T}_1 \cap \mathcal{T}_2 \cap \mathcal{T}_3 = \emptyset \quad \mathcal{K} = \mathcal{T}_0 \cup \mathcal{T}_1 \cup \mathcal{T}_2 \cup \mathcal{T}_3.$$

Were we to change the state machine to

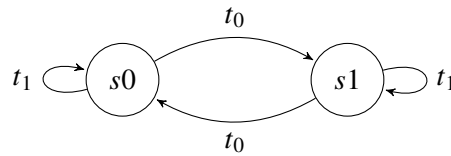


Figure 3.2: Simpler State Machine

we would have the table

State-Event Relations	
Relation	Tuples
$\mathcal{T}_0$	$\{\langle s0, s1 \rangle, \langle s1, s0 \rangle\}$
$\mathcal{T}_1$	$\{\langle s0, s0 \rangle, \langle s1, s1 \rangle\}$

3.4 Implications

This has implications for a spider checking state transitions in a plant. If the goal is to check each and every state and transition of the plant, the most efficient way is for the spider to duplicate the state structure of the plant; that is, for the spider to also be a state machine. The next-state transitions can be checked by including all the inputs in the spider that the plant uses for next-state determination. Also, all the internal conditions that the plant uses for next-state transitions must be exported to the spider as well as the actual state variable. The spider, on the clock strike, checks to see that its state variable is consistent with the exported plant's state variable. There is no way to prevent the plant from making a "bad" transition because at least one clock tick must occur for the spider to realize to what state the plant transisted.

Checking the inputs used by the plant to transist requires the plant record the inputs and export those recorded inputs as signals to the spider. The spider must also record the signals. On the next state, the two records of the signals must be checked. The reason for the recording and waiting until the next state is that signals can only be recorded on a clock strike. This might not be too expensive since the spider is tracking what states are involved and hence need only record the signals relevant to the states; a similar observation holds for the plant.

4. LOGICAL SPECIFICATIONS FOR CHECKING STATES

If the goal is to replicate the state structure of the plant in the spider, and keeping the ultimate goal of mechanizing as much of the work as possible, we can specify a common state structure via some Distributed Logic statements. The ultimate goal is to see how far we can get specifying the entire recognition structure of a spider using DL statements. Success would mean a spider's recognition portion is generated from logical statements. The mitigation of a spider cannot be done in a logical language because it is not specifying conditions but specifying actions. Also, the mitigation must occur as the recognition of conditions occurs. Therefore the mitigation language must be integrated with the recognition language. This report will not go into mitigation due to length considerations and the fact that we are still working out how it can best be accomplished.

4.1 Using Distributed Logic in Specifications

Let $k = \langle K, \mathcal{K}_1, \dots, \mathcal{K}_n, \mathbb{K} \rangle$ represent the locality of the spider and $h = \langle H, \mathcal{H}_1, \dots, \mathcal{H}_n, \mathbb{H} \rangle$ the locality of the plant. As one can tell from the common index n in both localities, the number of event types will be the same for both the spider and the plant. This need not be the case but we do so here because we are interested in duplicating the plant's state machine in the spider in way that the spider can properly check states and transitions.

We will relate states the spider and states in the plant by the relation $\mathcal{F} : k \rightarrow h$. We will view this relation as a concurrence relation which determines which states in the spider can occur with which states in the plant. The relation $\check{\mathcal{F}}$ is the converse of the relation $\mathcal{F}$:

$$\check{\mathcal{F}}_{yx} \text{ iff } \mathcal{F}_{xy},$$

i.e., $\check{\mathcal{F}}$ is the reverse of the relation $\mathcal{F}$.

We wish $\mathcal{F}$ to be at least a function since every state in the spider must be paired with a unique state in the plant. For the relation $\mathcal{F}$ to be a function, it must be constrained to actually be a function. We will assume the spider and plant agree on the event types of the respective state machines. The specification, for all propositions P in h ,

$$[f^\square]P \supset^k [f^\circ]P \quad [f^\circ]P \supset^k [f^\square]P$$

forces the distributed Kripke relation $\mathcal{F}$ to be *entire on its domain* and *functional* respectively. Think of $[f^\square]$ and $[f^\circ]$ as operators that pull propositions P in h from h to k , i.e., against the relation $\mathcal{F}$ used in their interpretations. In a similar fashion, $[f^\square]$ and $[f^\circ]$ pull propositions Q in k from k to h , i.e., against the relation $\check{\mathcal{F}}$, or what is the same, pushes propositions Q in k forward along $\mathcal{F}$ from k to h .

For $\mathcal{F}$ to be entire in its domain means that $\mathcal{F}$ is defined everywhere on its domain, i.e., all of K . For $\mathcal{F}$ to be functional means that it satisfies the characteristic property of a function in that it is single valued, i.e., for every x there is one and only one y such that $\mathcal{F}xy$; typically in mathematics we say that for every x there is a unique y such that $\mathcal{F}xy$. The way to see that this specification forces the properties of $\mathcal{F}$ we need is via the following proofs, using the natural deduction system of [6]. The proofs are somewhat pedantic and use some shorthand so that the object language (Distributed Logic) does not get confused with the model theoretic meta-language, e.g., $\supset$ -Intro stands for implies-Intro

1	$a \models^k [f^\square] P$	assume
2	$\forall y(\mathcal{F}ay \text{ implies } y \models^h P)$	modeling condition for $[f^\square]$, line 1
3	$\exists z(\mathcal{F}az)$	$\mathcal{F}$ is entire over its domain
4	$\boxed{c} \mathcal{F}ac$	assume
5	$\mathcal{F}ac \text{ implies } c \models^h P$	$\forall$ -Elim, line 2
6	$c \models^h P$	$\supset$ -Elim, lines 4, 5
7	$\mathcal{F}ac \text{ and } c \models^h P$	$\wedge$ -Intro, lines 4, 6
8	$\exists z(\mathcal{F}az \text{ and } z \models^h P)$	$\exists$ -Intro, line 7
9	$a \models^k [f^\circ] P$	modeling condition for $[f^\circ]$, line 8
10	$a \models^k [f^\circ] P$	$\exists$ -Elim, lines 3, 4
11	$a \models^k [f^\square] P \text{ implies } a \models^k [f^\circ] P$	$\supset$ -Intro, line 1
12	$[f^\square] P \supset^k [f^\circ] P$	modeling condition for $\supset^k$, line 11

and

1	$a \models^k [f^\circ] P$	assume
2	$\exists y(\mathcal{F}ay \text{ and } y \models^h P)$	modeling condition for $[f^\circ]$, line 1
3	$\boxed{b} \mathcal{F}ab \text{ and } b \models^h P$	assume
4	$\boxed{c} \mathcal{F}ac$	assume
5	$\mathcal{F}ab$	$\wedge$ -Elim, line 3
6	$\mathcal{F}ab \text{ and } \mathcal{F}ac$	$\wedge$ -Intro, lines 4, 5
7	$b = c$	$\mathcal{F}$ is functional, line 6
8	$b \models^h P$	$\wedge$ -Elim, line 3
9	$c \models^h P$	$=$ -Elim, lines 7, 8
10	$\forall y(\mathcal{F}ay \text{ implies } y \models^h P)$	$\forall$ -Intro, line 4
11	$a \models^k [f^\square] P$	modeling condition for $[f^\square]$, line 10
12	$a \models^k [f^\square] P$	$\exists$ -Elim, lines 2, 3
13	$a \models^k [f^\circ] P \text{ implies } a \models^k [f^\square] P$	$\supset$ -Intro, line 1
14	$[f^\circ] P \supset^k [f^\square] P$	modeling condition for $\supset^k$, line 13

The specification, for all propositions Q in k ,

$$[f^\square]Q \supset^h [f^\circ]Q \quad [f^\circ]Q \supset^h [f^\square]Q$$

forces the distributed Kripke relation converse $\check{\mathcal{F}}$ to be *entire* on its domain and *functional* respectively. The proofs are similar now using $\check{\mathcal{F}}$, the converse relation of $\mathcal{F}$. When the properties for $\mathcal{F}$ and $\check{\mathcal{F}}$ hold, $\mathcal{F}$ is called a *bijection*. This means that every state in the spider can be paired with a unique state the plant and visa versa. Put another way, the two state sets have the same cardinality and the relation $\mathcal{F}$ shows how to pair up the states.

The sense in which the formulas “force” the relation to have the properties we want is that the only way the proofs can succeed is with relation $\mathcal{F}$ having the correct properties. If the formula holds as valid, then the indicated relation must have the correct properties.

This feature of the formulas, now as additional axioms, to constrain the properties of the relation is going to be problematic. The reason is that there is nothing in the form of the DL statements that can be mechanically lifted to form the properties of the relation being entire on its domain and functional. The only way these properties were seen to be necessary is to look at the proofs and ascertain that were the properties to obtain, then the proofs could go through. This is a soundness argument. The completeness argument relies on *canonical* frames, which are those frames that can be formed from merely the logic itself. This last statement is a bit mysterious but we must leave it at that because to explain it properly would require another chapter and without being a logician, fairly unintelligible. Suffice it to say it is not obvious and it also would not be enlightening since the canonical frame too contains not enough information to extract these properties.

4.2 Specifying States in the Logic

Since our machines are finite, there are only a finite number of states we need be concerned with. Hence we have singleton sets available to us. This implies that the set $\{i\}$ for i a coarse-grained state in a machine (i.e., we see it as part of a case statement in the VHDL) can be used as a proposition and can denote it P_i .

Semantically, we have

$$Q_i = \{i\} = \{z \mid z = i \text{ and } z \text{ is a variable over macro-states} \}$$

using the interpretation of those propositions in terms of states. In the logic when not under interpretation, we can simply use Q_i and rely upon any interpretation as mapping Q_i onto $\{i\}$ via defining

$$Q_i(x) \text{ is } x = i.$$

This says that $Q_i(x)$ is literally the string $x = i$. Note that here the $=$ is in the object language not the metalanguage. This shows that a propositional object language can have more internal structure than just the Boolean connectives. However, that structure is ignored in any of the logical rules but not so in the semantics since x and i must be interpreted to yield the same object, in our case, a state. Also, since there is usually only one variable in any of our propositions of the object language, we can elide that variable and use

$$Q_i \text{ is } x = i.$$

where i is assumed to be a constant that will evaluate to the i the state in a case statement. The i in the case statement is part of the syntax of VHDL, and hence is still considered part of the syntax of the object

language. Given the = sign and the fact that the = as a sign is always interpreted by the equality relation in the semantics, this says that whatever x evaluates to must be the same as whatever i evaluates to.

Now we come to the rather delicate definition of how to match Q_i in the spider with some $P_{i'}$ in the plant. The reason that $i \neq i'$ is that the i and i' are in two different localities and all we have is a bijection between the locality state sets. The way to see this is to consider the following two bijections between two sets of states:



Figure 4.1: Two Bijections

The issue is there is no way to solve the identification problem just by specifying bijections. A solution can be had by specifying a common representation for states in the spider and states in the plant. We will deal with this later. For now, we simply assume a bijection.

Let the proposition Q_i be in the local logic at k (the local logic of the spider) and $P_{i'}$ be in the local logic at h (the local logic of the plant), then we require that

$$y \in_k Q_i \text{ and } \mathcal{F}yx \text{ implies } x \in_h P_{i'}. \quad x \in_h P_{i'} \text{ and } \check{\mathcal{F}}xy \text{ implies } y \in_k Q_i.$$

These validate

$$Q_i \overset{k}{\supset} [f^\square] P_{i'} \quad P_{i'} \overset{h}{\supset} [f^\square] Q_i.$$

The formula $[f^\square] Q_i$ will identify a proposition, call it $P_{i'}$, containing a single state. Similarly, $[f^\square] P_{i'}$ will identify a single proposition, call it Q_i , containing a single state. We then wish the following formulas to hold:

$$Q_i \overset{k}{\supset} [f^\square] [f^\square] Q_i \quad P_{i'} \overset{h}{\supset} [f^\square] [f^\square] P_{i'}.$$

Using the rules D3 and D4 from the appendix, these are equivalent to

$$[f^\square] Q_i \overset{k}{\supset} [f^\square] Q_i \quad [f^\square] P_{i'} \overset{h}{\supset} [f^\square] P_{i'}.$$

We have seen these before, they state (respectively) that $\check{\mathcal{F}}$ and $\mathcal{F}$ are functional. Since we are assuming that $\mathcal{F}$ is a bijection, both $\mathcal{F}$ and $\check{\mathcal{F}}$ are indeed functional.

We need event types to refer to a conjunction of actual inputs. An example is

```

1 case state is
2   when s0 =>
3     if ((t0 = '1') and (t1 = '1')) then
4       state <= s1;
5     elsif (t2 = '1') then
6       state <= s0;

```

The event types e_0 and e_1 are defined by

$$e_0 \stackrel{def}{=} (t_0 = 1) \wedge (t_1 = 1) \qquad e_1 \stackrel{def}{=} (t_2 = 1) \wedge \neg e_0$$

Note that the extra conjunction $e_0 = \neg((t_0 = 1) \wedge (t_1 = 1))$ is needed in defining e_1 since VHDL executes the **elsif** only when the **if** fails.

We explain the notation a bit further and how it relates to semantic interpretations. The meaning of $t_0 = '1'$ is $t_0 = 1$. This implies we are using the variable t_0 in the semantic metalanguage to interpret the symbol t_0 in VHDL and the symbol $'1'$ is mapped to the mathematical value 1. Also, the $=$ symbol on the left is interpreted as the equality relation $=$ on the right.

Technically, we would need to set up a separate interpretation for VHDL which can be a bit of a problem since there is no agreed upon semantics for VHDL. If we were to recode the VHDL into ReWire, then we could interpret the logical object language in ReWire code, and ReWire does have a mathematical semantics. By virtue of the mappings, we get an interpretation for the logical object language. Here, we will make do with a bit of *semantic archeology* and assume the mapping we have chosen makes sense and is sound.

4.3 Strong Conditions for Checking States

We wish to assert that the spider is correctly tracking the plant's states by using its own states. So far we have that the relation $\mathcal{F}$ is a bijection. Assume the next-state relations $\mathcal{K}_j$ at k and $\mathcal{H}_j$ at h connected with the event type e_j . So e_j refers to some collection of inputs that the spider and plant use to change state. One method of assuring this is to use graph isomorphisms. One can view $\mathcal{K}_j$ and $\mathcal{H}_j$ as directed graphs and hence we want $\mathcal{F}$ to be a graph isomorphism. A graph isomorphism $\mathcal{F}$ between $\mathcal{K}_j$ and $\mathcal{H}_j$, under the assumption of $\mathcal{F}$ is a bijection, is to assume either of

$$\mathcal{K}_j x x' \text{ iff } \mathcal{H}_j \mathcal{F}(x) \mathcal{F}(x') \qquad \mathcal{H}_j y y' \text{ iff } \mathcal{K}_j \check{\mathcal{F}}(y) \check{\mathcal{F}}(y'),$$

where we are using $\mathcal{F}$ now as a function and not a two-place relation. These statements say that $\mathcal{F}$ is an isomorphism between the relation $\mathcal{K}_j$ and $\mathcal{H}_j$ for all j .

Each of those conditions is actually two conditions since they are stated using iff. Since either will do, we will work with the first formula. Assuming $\mathcal{F}$ is a bijection, the first formula validates

$$[k_j^\circ] Q_i \stackrel{k}{\supset} [f^\circ] [h_j^\circ] [f^\circ] Q \qquad [f^\circ] [h_j^\circ] [f^\circ] Q \stackrel{k}{\supset} [k_j^\circ] Q_i.$$

The two proofs are

1	$a \models^k [k_j^\circ] Q_i$	assume
2	$\exists y(\mathcal{K}_j a y \text{ and } y \models^k Q_i)$	modeling condition for $[-^\circ]$, line 1
3	$\boxed{b} \quad \mathcal{K}_j a b \text{ and } b \models^k Q_i$	assume
4	$\mathcal{H}_j \mathcal{F}(a) \mathcal{F}(b)$	first condition left to right, line 3
5	$\exists x(\mathcal{F}(a) = x)$	$\mathcal{F}$ is a function, line 4
6	$\exists x'(\mathcal{F}(b) = x')$	$\mathcal{F}$ is a function, line 4
7	$\boxed{c} \quad \mathcal{F}(a) = c$	assume
8	$\boxed{d} \quad \mathcal{F}(b) = d$	assume
9	$\mathcal{F} a c \wedge \mathcal{F} b d$	$\mathcal{F}$ is also a relation, lines 7, 8
10	$\mathcal{F} b d \wedge b \models^k Q_i$	$\wedge$ -Elim, $\wedge$ -Intro, lines 5, 9
11	$d \models^h [f^\circ] Q_i$	modeling condition for $[-^\circ]$, line 10
12	$\mathcal{H}_j c d$	=-Elim, lines 4, 7, 8
13	$c \models^h [h_j^\circ] [f^\circ] Q_i$	modeling condition for $[-^\circ]$, lines 11, 12
14	$a \models^k [f^\circ] [h_j^\circ] [f^\circ] Q_i$	modeling condition for $[-^\circ]$, lines 9, 13
15	$a \models^k [f^\circ] [h_j^\circ] [f^\circ] Q_i$	$\exists$ -Elim, lines 6, 8
16	$a \models^k [f^\circ] [h_j^\circ] [f^\circ] Q_i$	$\exists$ -Elim, lines 5, 7
17	$a \models^k [f^\circ] [h_j^\circ] [f^\circ] Q_i$	$\exists$ -Elim, lines 2, 3
18	$[k_j^\circ] Q_i \supset^k [f^\circ] [h_j^\circ] [f^\circ] Q_i$	modeling condition for $\supset^k$, line 1

and

1	$a \models^k [f^\circ] [h_j^\circ] [f^\circ] Q_i$	assume
2	$\exists x (\mathcal{F}ax \text{ and } x \models^h [h_j^\circ] [f^\circ] Q_i)$	modeling condition for $[-^\circ]$, line 1
3	$\boxed{b} \quad \mathcal{F}ab \text{ and } b \models^h [h_j^\circ] [f^\circ] Q_i$	assume
4	$\exists x' (\mathcal{H}_jbx' \text{ and } x' \models^h [f^\circ] Q_i)$	modeling condition for $[-^\circ]$, line 3
5	$\boxed{c} \quad \mathcal{H}_jbc \text{ and } c \models^h [f^\circ] Q_i$	assume
6	$\exists y' (\mathcal{F}y'c \text{ and } y' \models^k Q_i)$	modeling condition for $[-^\circ]$, line 5
7	$\boxed{d} \quad \mathcal{F}dc \text{ and } d \models^k Q_i$	assume
8	$\mathcal{F}ab \wedge \mathcal{H}_jbc \wedge \mathcal{F}dc$	$\wedge$ -Elim, $\wedge$ -Intro, lines 3, 5, 7
9	$\mathcal{F}(a) = b \wedge \mathcal{F}(d) = c$	$\mathcal{F}$ is also a function, line 8
10	$\mathcal{H}_j\mathcal{F}(a)\mathcal{F}(d)$	$=$ -Elim, lines 8, 9
11	$\mathcal{K}_jad$	first condition right to left, line 10
12	$a \models^k [k_j^\circ] Q_i$	modeling condition for $[-^\circ]$, lines 7, 11
13	$a \models^k [k_j^\circ] Q_i$	$\exists$ -Elim, lines 6, 7
14	$a \models^k [k_j^\circ] Q_i$	$\exists$ -Elim, lines 4, 5
15	$a \models^k [k_j^\circ] Q_i$	$\exists$ -Elim, lines 2, 3
16	$[f^\circ] [h_j^\circ] [f^\circ] Q_i \supset^k [k_j^\circ] Q_i$	modeling condition for $\supset^k$, line 1

We are not quite finished. While we now know that the graphs connecting $\mathcal{K}_j$ and $\mathcal{H}_j$ are isomorphic, we do not yet know which isomorphism we have forced via the DL formulas. Consider the following two diagrams:

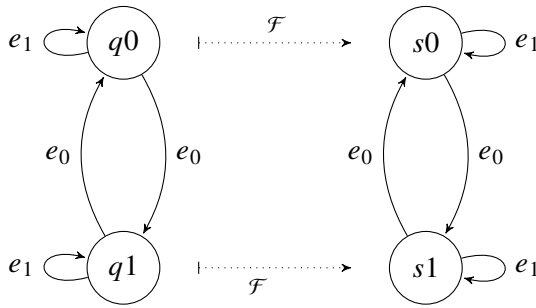


Figure 4.2: First Isomorphism

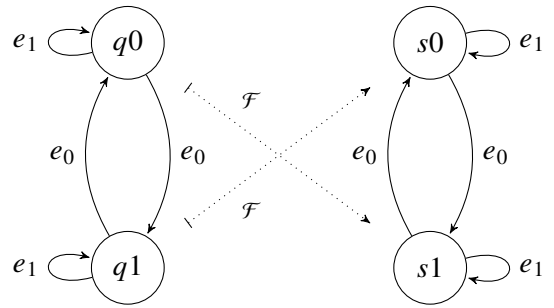


Figure 4.3: Second Isomorphism

If we were only concerned with inputs and state changes, then it would not matter. However, those state machines might also set outputs. We have not taken those into account. First note that $\mathcal{K}_j$ and $\mathcal{H}_j$ are defined directly from the VHDL code while up until now the relation $\mathcal{F}$ has been rather free without any direct connection to the machines other than the cardinality of machines' state sets and the interaction of $\mathcal{F}$ with $\mathcal{K}_j$ and $\mathcal{H}_j$. If we define the state sets in the VHDL to use the same enumeration, i.e.,

```
1  type state_type is (q0, q1, q2, q3, q4);
```

and

```
1  type state_type is (s0, s1, s2, s3, s4);
```

then q_i maps to the ordinal i and s_i to the ordinal i , this will set up the bijection and subsequently isomorphism we want.

A safer way to accomplish this is to not assume the VHDL compiler maps the `state_type` values to the same ordinals with the two uses. Rather, we can use a common file that uses a single set of symbols in the enumerated type and use those same symbols in both the spider and the plant. The mathematics can continue to use q_i and s_i and we can be assured that they both map to the ordinal i in the implementation and hence can count on $q_i = s_i$ under any interpretation.

5. WEAKENING THE CONDITIONS FOR CHECKING STATES

A graph isomorphism is a fairly strong and rigid framework for checking states. We can loosen the requirements a bit. To do this, we use the notion of *simulation* from modal logic and also systems theory. Actually, the notion of simulation in various guises comes up repeatedly throughout logic and computer science. The notion of simulation gives us more flexibility in writing spiders.

Simulations allow us to describe machines with respect to their behavior, not how they are constructed. The latter is an *algebraic* notion, The former is termed *coalgebraic*. Algebraic systems have been known throughout mathematics for several hundred years starting with Descartes. Coalgebraic systems have only recently been recognized as central to much of mathematics and examples abound of them appearing before recently though without the modern view of them as coalgebraic.

The next-state function of finite state machines is an example of a coalgebraic sort of description. So are the Kripke relations from modal and distributed logics. A spider with no notions of the internal state structure of a plant can still constrain the plant's behavior through just observation and mitigation. The observation part is a coalgebraic notion. In broad terms, algebras show how a system is constructed. Coalgebras show how a system is destructed.

5.1 Simulations and Bisimulations

We will assume that the event types determine local Kripke relations, and that the spider and the plant agree on event types. Let $E_{spider} = E_{plant}$ where E stands for event types. Note that this is establishing a set theory identification as well as bijection between event types. So for now we can simply use the unadorned E to indicate the set of event types of cardinality $|E| = n$. Each $e_j \in E$ gives rise to a next-state relation specific to the event type e_j although it must be parameterized with a state set from a machine or Kripke frame. We generally let $\mathcal{K}_j : k \rightarrow k$ and $\mathcal{H}_j : h \rightarrow h$ be next-state relations associated with event type e_j .

Definition 5.1.1 Let $\mathcal{K}_j : k \rightarrow k$ and $\mathcal{H}_j : h \rightarrow h$ be next-state relations in a spider and plant respectively. A *bisimulation* $\mathcal{F} : k \rightarrow h$ between a spider and plant must satisfy, for all $e_j \in E$,

- M1. $\mathcal{K}_j yy'$ and $\mathcal{F}_{yx}$ implies there is some x' such that $\mathcal{F}_{y'x'}$ and $\mathcal{H}_j xx'$,
- M2. $\mathcal{H}_j xx'$ and $\check{\mathcal{F}}_{xy}$ implies there is some y' such that $\check{\mathcal{F}}_{x'y'}$ and $\mathcal{K}_j yy'$.

Each of the conditions on its own defines a *simulation*.

The Definition can be put into graphical form with a variation on the diagrammatic language of [7]

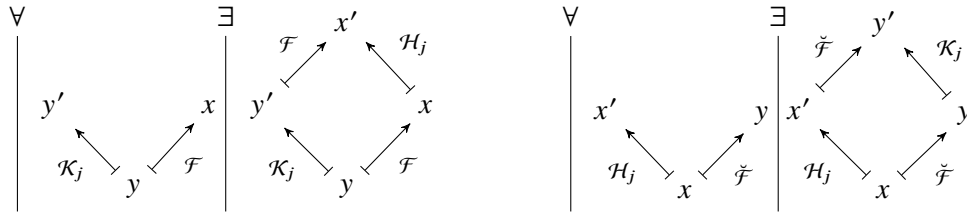


Figure 5.1: Conditions for Bisimulation

Each clause in the definition of bisimulation is referred to as a *simulation*; hence each of the diagrams is a simulation condition. The first simulation condition (first diagram) starting at k says that if y is related via $\mathcal{F}$ to x and y is also locally related to y' via $\mathcal{K}_j$, then there is some x' such that y' is related via $\mathcal{F}$ to x' and x is related via $\mathcal{H}_j$ to x' . Put in colloquial terms, if the spider can make a transition, then the plant can match that transition. A similar statements holds from the plant's perspective using the second diagram, which expresses that if the plant can make a transition then the spider can match that transition.

We will use general Q and P rather than Q_i and P_i since the conditions are not peculiar to the latter specialized propositions. These simulation conditions will validate (respectively) for all P in h and Q in k .

$$\text{L1. } [f^\circ] [h_j^\circ] P \stackrel{k}{\supset} [k_j^\circ] [f^\circ] P,$$

$$\text{L2. } [f^\circ] [k_j^\circ] Q \stackrel{h}{\supset} [h_j^\circ] [f^\circ] Q.$$

In effect, these are prescriptions for what we want to hold if the plant is being spidered and no errors occur. If either of these conditions fail, then an error has occurred.

We show the validation of the first formula:

1	$a \models^k [f^\circ] [h_j^\circ] P$	assume
2	$\exists x(\mathcal{F}ax \text{ and } x \models^h [h_j^\circ] P)$	modeling condition for $[-^\circ]$, line 1
3	$\boxed{c} \quad \mathcal{F}ac \text{ and } c \models^h [h_j^\circ] P$	assume
4	$\boxed{b} \quad \mathcal{K}_{jab}$	assume
5	$\mathcal{F}ac \text{ and } \mathcal{K}_{jab}$	$\wedge$ -Elim, $\wedge$ -Intro, lines 3, 4
6	$\exists x'(\mathcal{H}_{jcx'} \text{ and } \mathcal{F}bx')$	Simulation Condition, line 5
7	$\boxed{d} \quad \mathcal{H}_{jcd} \text{ and } \mathcal{F}bd$	assume
8	$\forall x'(\mathcal{H}_{jcx'} \text{ implies } x' \models^h P)$	modeling condition for $[-^\circ]$, line 3
9	$\mathcal{H}_{jcd} \text{ implies } d \models^h P$	$\forall$ -Elim, line 8
10	$d \models^h P$	$\supset$ -Elim, lines 7, 9
11	$\mathcal{F}bd \text{ and } d \models^h P$	$\wedge$ -Elim, $\wedge$ -Intro, lines 7, 10
12	$\exists u(\mathcal{F}bu \text{ and } u \models^h P)$	$\exists$ -Intro, line 11
13	$b \models^k [f^\circ] P$	modeling condition for $[-^\circ]$, line 12
14	$b \models^k [f^\circ] P$	$\exists$ -Elim, lines 6, 7
15	$\forall v(\mathcal{K}_{jav} \text{ implies } v \models^k [f^\circ] P)$	$\forall$ -Intro, line 4
16	$a \models^k [k_j^\circ] [f^\circ] P$	modeling condition for $[-^\circ]$, line 15
17	$a \models^k [k_j^\circ] [f^\circ] P$	$\exists$ -Elim, lines 2, 3
18	$a \models^k [f^\circ] [h_j^\circ] P \text{ implies } a \models^k [k_j^\circ] [f^\circ] P$	$\supset$ -Intro, line 1
19	$[f^\circ] [h_j^\circ] P \supset [k_j^\circ] [f^\circ] P$	modeling condition for $\supset$, line 18

Notice what must be the case if the spider is checking these conditions. The plant must export its state variable to the spider and the spider and plant must share the same inputs.

The proof of the second condition is

1	$a \models^h [f^\circ] [k_j^a] Q$	assume
2	$\exists y (\check{\mathcal{F}} ay \text{ and } y \models^k [k_j^a] Q)$	modeling condition for $[-^\circ]$, line 1
3	$\boxed{c} \quad \check{\mathcal{F}} ac \text{ and } c \models^k [k_j^a] Q$	assume
4	$\boxed{b} \quad \mathcal{H} ab$	assume
5	$\check{\mathcal{F}} ac \text{ and } \mathcal{H} ab$	$\wedge$ -Elim, $\wedge$ -Intro, lines 3, 4
6	$\exists y' (\mathcal{K}_j cy' \text{ and } \check{\mathcal{F}} by')$	Simulation Condition, line 5
7	$\boxed{d} \quad \mathcal{K}_j cd \text{ and } \check{\mathcal{F}} bd$	assume
8	$\forall u (\mathcal{K}_j cu \text{ implies } u \models^k Q)$	modeling condition for $[-^\circ]$, line 3
9	$\mathcal{K}_j cd \text{ implies } d \models^k Q$	$\forall$ -Elim, line 8
10	$d \models^h Q$	$\supset$ -Elim, lines 7, 9
11	$\check{\mathcal{F}} bd \text{ and } d \models^k Q$	$\wedge$ -Elim, $\wedge$ -Intro, lines 7, 10
12	$\exists z (\check{\mathcal{F}} bz \text{ and } z \models^k Q)$	$\exists$ -Intro, line 11
13	$b \models^h [f^\circ] Q$	modeling condition for $[-^\circ]$, line 11
14	$b \models^h [f^\circ] Q$	$\exists$ -Elim, lines 6, 7
15	$\forall v (\mathcal{H}_j av \text{ implies } v \models^h [f^\circ] Q)$	$\forall$ -Intro, line 4
16	$a \models^h [h_j^a] [f^\circ] Q$	modeling condition for $[-^\circ]$, line 15
17	$a \models^h [h_j^a] [f^\circ] Q$	$\exists$ -Elim, lines 2, 3
18	$a \models^k [f^\circ] [k_j^a] Q \text{ implies } a \models^k [h_j^a] [-^\circ] Q$	$\supset$ -Intro, line 1
19	$[f^\circ] [k_j^a] Q \subseteq_k [h_j^a] [f^\circ] Q$	modeling condition for $\supset$, line 18

The simulation conditions are weaker than strict equality. We can recast them in prose as

- P1. $\mathcal{F} : k \rightarrow h$ being a simulation means that if the spider can make a move in its state machine, then the plant must make a move in its state machine.
- P2. $\check{\mathcal{F}} : h \rightarrow k$ being a simulation means that if the plant can make a move in its state machine, then the spider must make a move in its state machine.

Spiders can be constructed to only satisfy one of these properties. The first property holding precludes the spider from making state changes for purely internal housekeeping such as responding to query attempts from the outside. The outside could be another spider or someone sitting at a terminal. The outside could also include conditions for turning the spider on and off. The list could go on. The second condition holding forces the spider to check state changes in the plant.

That said, there is another way to weaken the spider. That is, make either $\mathcal{F}$ or $\check{\mathcal{F}}$ not be entire over their respective domains. If $\mathcal{F}$ is not entire, then the spider can make state changes that are not answered

by the plant. If $\check{\mathcal{F}}$ is not entire, the plant can make state changes that the spider need not care about. Care must be taken under both of these scenarios to make sure the spider's states correctly take into account these weakenings.

Let $\mathcal{F} : k \rightarrow h$ be a bisimulation. That is, both $\mathcal{F} : k \rightarrow h$ and $\check{\mathcal{F}} : h \rightarrow k$ are simulations. It is helpful to understand what happens if the spider fails:

- If $\mathcal{F} : k \rightarrow h$ is a simulation and fails to see the plant make an incorrect move, then the spider generates a *false positive*.
- If $\check{\mathcal{F}} : h \rightarrow k$ is a simulation and the plant makes a correct move which the spider detects as an incorrect move, then the spider is generates a *false negative*.

These observations suggest some further language concerning simulations.

Definition 5.1.2 Let $\mathcal{F} : k \rightarrow h$ be a distributed relation connecting the spider at k with the plant at h . Call a spider *correct* with respect to $\mathcal{F}$ if for all $P \in h$ and all j iff

$$[f^\circ] [h_j^\circ] P \subseteq_h [k_j^\circ] [f^\circ] P.$$

Call a spider *closed* with respect to all $Q \in k$ and all j iff

$$[f^\circ] [k_j^\circ] Q \subseteq_k [h_j^\circ] [f^\circ] Q.$$

Correctness says that the spider can check all state sets (propositions) in the plant. Closure says that any conditions that the spider does check must hold of the plant.

If these conditions always hold, then $\mathcal{F}$ is a bisimulation. A weaker notion then occurs when we only demand that they hold for certain P and Q . In that case, the bisimulation only holds for some pairs of states y and x such that $\mathcal{F}xy$.

It sometimes makes sense to recast the logical specifications of L1 and L2. They are equivalent to

$$\text{L1}'. [f^\circ] [k_j^\circ] Q \overset{h}{\supseteq} [h_j^\circ] [f^\circ] Q,$$

$$\text{L2}'. [f^\circ] [h_j^\circ] P \overset{k}{\supseteq} [k_j^\circ] [f^\circ] P.$$

Recall that $[f^\circ]$ pulls propositions back along the relation $\mathcal{F}$ from h to k , and that $[h_j^\circ]$ is an operator on the logic at h and hence $[h_j^\circ] P$ is a proposition in h . Similarly, $[f^\circ]$ pulls propositions back along the relation $\check{\mathcal{F}}$ from k to h and that $[k_j^\circ] Q$ is a proposition in k .

Condition L2 can seem a bit odd from the spider's perspective in that this condition is pulling some proposition Q of the spider back along $\check{\mathcal{F}}$. Hence condition L2' can be used in its place to remove the awkwardness.

If we identify P_i and Q_i with states in the plant and spider respectively, the simulation formulas then become

- $[f^\circ] [h_j^\circ] P_{i'} \supset [k_j^\circ] [f^\circ] P_{i'}$,
- $[f^\circ] [h_j^\circ] Q_i \supset [k_j^\circ] [f^\circ] Q_i$.

These conditions on their own do not enforce a bijection between the plant's state sets and the spider. Notice that the index is i' on the plant and i on the spider. What this is saying is that the specific propositions $P_{i'}$ and Q_i are identified with states in the plant and spider respectively. We do not want to restrict the propositions in this way, but merely allow the $P_{i'}$ and Q_i to be used along with other P s and Q s.

The simulation conditions are what we can observe from the outside about our state machines. This is as opposed to the actual construction of our state machines. Bisimulations identify machines that are indiscernible from observing their behavior. The machines of Figures 4.3 are indiscernible from each other by observation and we say they are *bisimilar*. The following machines are also bisimilar:

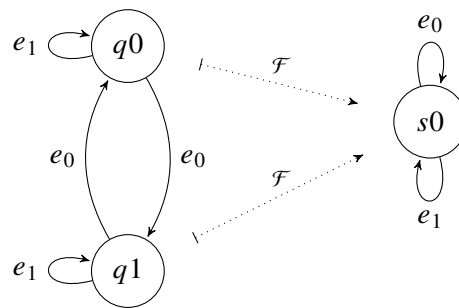


Figure 5.2: Simple Bisimulation

One consequence is that a bisimulation between two possible spiders can only tell when the spiders are identical as far as their observational behavior. Further, while the spider and plant may have different state machines internally, their external behavior can indicate they essentially have the same abstract structure. Another consequence is that a possible trojan may be able to detect enough behavior of a plant or spider to exfiltrate information we would rather keep secret. Think of the trojan as constructing a bisimilar image of the plant's or spider's behavior. The trojan would not have to identify all behavior, just enough to compromise a system. In the machines of Figure 5.2, the machines could have a lot of internal behavior not discernible from the outside.

5.2 Decoupling Spider's States from the Plant's States

Typically, a state is a list of values of variables and outputs. This presents a problem for spiders in general because it may not always be possible to use the same variables and outputs in the spider as are used in the plant. In that case, we wish to discover conditions under which states and their transitions in the plant can be properly tracked.

We require a representation of a state machine in mathematics that we can also use in setting up machinery for comparing state machines. The representation will use a variant of the representation used in Section 3.2 but requires a small bit of set theory. Set theory itself would become very noisy unless we drop standard set theory and adopt non-wellfounded set theory.

Remark 5.2.1 *One word about set theory and abstraction. The presentation*

$$\{x\}$$

is not actually a set. A set is an abstract thing in the same sense the number one is an abstract entity that we present using the numeral 1 that can be put on a page. Likewise a function we depict as $f = x^2$ on the page is not the actual function. A function is an abstract object. We can also think of a function extensionally as a set of pairs of elements, i.e., a relation. Or we can think of a function as depicted in a rule like we depict with $f = x^2$. The actual function is an abstract object. Similar statements hold for relations and all mathematical objects. The language of mathematics allows us to talk about these objects and depict the rules that govern our use of them. Logic is similar, it requires a language to talk about propositions and propositional functions, e.g., $\wedge$. We use presentations of abstract things to refer to them, but they are not in themselves the actual abstract thing. This remark must be kept in mind to prevent confusion about the mathematical objects we require.

In pure standard set theory, everything is a set. One might ask, how do we get elements? The only set guaranteed to exist is the empty set, $\emptyset$. From this, quite a lot can be built up. Consider the von Neumann ordinals, i.e., representations of the natural numbers, i.e., 0, 1, 2, ... The representation is

$$\begin{aligned} 0 &\stackrel{\text{def}}{=} \emptyset \\ 1 &\stackrel{\text{def}}{=} \{\emptyset\} \\ 2 &\stackrel{\text{def}}{=} \{\emptyset, \{\emptyset\}\} \\ 3 &\stackrel{\text{def}}{=} \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}\} \\ &\stackrel{\text{def}}{=} \cdot \end{aligned}$$

This allows for operations such as $+$ to be defined on the sets so that $+$ truly represents $+$ in the natural numbers. That is,

$$n + 1 = n \cup \{n\}$$

Hence

$$n + m = \begin{cases} n & \text{if } m = 0 \\ (n + (m - 1)) \cup \{n\} & \text{otherwise} \end{cases}$$

That said, encoding everything in sets gets a bit onerous. An addition to set theory is to consider a collection of things called *urelements*. Urelements are things that are not sets. Hence we can include things like the natural numbers as urelements and avoid the encoding. Of course this involves defining the algebra of natural numbers with its 0 element and $+$ operation. Regardless, we will assume we have a ready stock of urelements and use them to represent events and actions in states that set outputs.

Standard set theory has the axiom of foundation which says that for any set A and chain of sets such that

$$\dots a_{n-1} \in a_{n-2} \in \dots \in a_1 \in a_0 \in A,$$

is finite, i.e., there is an n such that the a_{n-1} is end of the chain. Non-wellfounded set theory relaxes this consideration and put quickly, allows us to model “things that go ’round” such as operating systems, bicycle tires, or finite state machines with loops through a state set.

A stock example of a set that the axiom of foundation rules out is

$$\Omega \stackrel{def}{=} \{\Omega\}.$$

The set seems to chase its tail. A picture of this set is

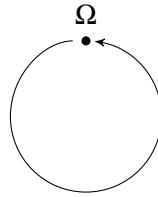


Figure 5.3: Picture of the Set Ω

where the arrow is a graphical representation of the membership $\in$ relation. This says that Ω is a member of itself. An attempt to display Ω in a non-referential equation by expanding its definition leads inevitably to ellipses:

$$\Omega = \{\Omega\} = \{\{\Omega\}\} = \{\{\{\Omega\}\}\} \dots$$

Were we to represent Ω as a data structure in a language with pointers, then we could represent the first equality as a data structure that had a pointer pointing to itself. Were we to represent this in a functional language that has no pointers, then the language has to use lazy evaluation. That is, we only evaluate as far as is needed for any computation without generating an infinite list of objects. The remaining list is represented as a computation that when evaluated gives us the next object in the sequence along with a further unexpanded computation.

State machines with cycles are like that. If we intend to model state machines with cycles, then we need representation that can handle cycles. We also want a representation that is mathematically based which we can mechanize. To put the problem in better relief, consider the graph

An example from [8] modified to represent not just a pure set but a finite state machine is

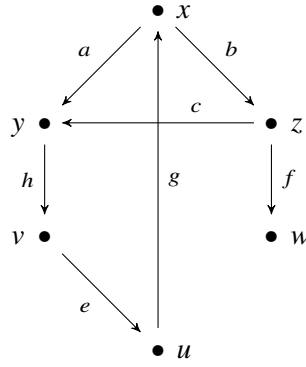


Figure 5.4: Example Graph

Forgetting about the event types on the arcs, this graph can represent a pure set using a function d , called a *decoration of the graph*, from nodes to sets in V_{afa} (the collection of non-wellfounded sets) using the following set equations:

$$\begin{aligned}
 d(x) &= \{d(y), d(z)\} & d(v) &= \{d(u)\} \\
 d(y) &= \{d(v)\} & d(u) &= \{d(x)\} \\
 d(z) &= \{d(y), d(w)\} & d(w) &= \emptyset
 \end{aligned}$$

An example of a tuple in the membership relation is $d(y) \in d(x)$. Notice the cycle $d(x), d(y), d(v), d(u), d(x)$. Let s and r refer to arbitrary nodes, then the sets are built up using

$$d(s) \in d(r) \text{ iff } r \bullet \rightarrow s.$$

We can incorporate the event types by the use of *atoms*. Atoms are objects that may have internal structure, possibly even be sets but which do not contribute to the set theoretical structure but rather used as ancillary information objects in a set. There is also a class⁴ of what are known as *urelements*, which we depict with $\mathcal{U}$. Those are “atoms” that are objects not in V_{afa} . They can well be sets but we do not use any of their internal structure if they possess that. They merely represent whatever we wish them to represent from some domain of discourse. For now, we can leave the atoms and urelements as devoid of any internal structure. We may also use as atoms elements from the class of urelements.

In this case, d is a function from graph points to $V_{afa}[A]$, the collection of non-wellfounded sets which use atoms from A . Our equations become

$$\begin{aligned}
 d(x) &= \{\langle a, d(y) \rangle, \langle b, d(z) \rangle\} & d(v) &= \{\langle e, d(u) \rangle\} \\
 d(y) &= \{\langle h, d(v) \rangle\} & d(u) &= \{\langle g, d(x) \rangle\} \\
 d(z) &= \{\langle c, d(y) \rangle, \langle f, d(w) \rangle\} & d(w) &= \emptyset
 \end{aligned}$$

⁴A class is like a set but can be larger. We will not go into a discourse on set theory to cover them. There are many good books on the subject.

where pairs $\langle a, b \rangle$ are also encoded in set theory as $\{a, \{b\}\}$. Hence the entire encoding uses sets with possibly embedded sets. The way to remember this is to consider the graph of Figure 5.4 to be a state machine. Choose a state in the machine and all the event types leading out of it where the event types label the arrows. While not the case in this graph, if you had to states that looked the same with respect to just the event type arrows, then you would consider those two states to be essentially the same state. To represent that state as a set means to represent it as shown above. Now what happens to the two states that are essentially the same? They are represented by the same set. So what is happening is that the set theoretic observation

$$\{x, x\} = \{x\}$$

from the fact that repetitions do not exist in any actual set means that any two states identical from the local view of their event types leading out of the state are collapsed in set theory to be the same set. The equation above is merely showing that the two *presentations* of the set $\{x\}$ are to be considered as representing the same set, namely a singleton set with one element x . So the set theoretic representation of state machines done in this manner automatically collapses states that we cannot tell apart by mere observation.

Let $V[A]$ be well-founded sets using urelements from A . Let $\mathcal{P}$ be the powerset operator on sets; that is, if give $\mathcal{P}$ a set, then it gives you back the set of all subsets of that set. So it causes a jump in set theoretic typing from sets to sets of sets.

Definition 5.2.2 ([8]) A *(labeled) graph* $G = (G, \bullet \rightarrow, l)$ is a triple such that $(G, \bullet \rightarrow)$ is a graph and $l : G \rightarrow \mathcal{P}(A)$. A *decoration* of a graph G over A is a function $d : G \rightarrow V_{afa}[A]$ such that for all $g \in G$

$$d(g) = \{d(h) \mid g \bullet \rightarrow h\} \cup l(g).$$

The function d for the diagram of Figure 5.4 is such decoration.

Another way to describe the state machines is as *labeled transition systems* (LTSs). LTSs are not limited to having a finite collection of states or event types. We do not need such generality, however much of the theory for finite state machines is stated in terms of LTSs and so we adopt their terminology. You will not go wrong in just thinking of them as state machines. The main difference is in the representation of the next-state function, δ .

Definition 5.2.3 A *labeled transition system* over Act is a set pair $\mathcal{T} = \langle S, \delta \rangle$, where S is a set, and $\delta : S \rightarrow \mathcal{P}(Act \times S)$ is a *(non-deterministic) transition function*.

The set S can be thought of as a collection of states, but in the literature, it is typically just a set and then it is applied to model to some situation, say state machines. The term Act is shorthand for *Actions*. If the transition function is such that

$$\langle a, q \rangle, \langle a, q' \rangle \in \delta s,$$

then the LTS is non-deterministic because s transits to both q and q' under the action a . Regardless, δ is still a function because its codomain is $\mathcal{P}(Act \times S)$ so δs is a set.

In LTSs and deterministic FSMs, the next-state function is truly a function. Where they differ is in their domains and codomains. They have the types (respectively)

$$\delta : \text{states} \times \text{event types} \rightarrow \text{states} \quad \delta : S \rightarrow \mathcal{P}(\text{Act} \rightarrow S).$$

In FSMs, if you give δ state and an event type, then δ gives you back a state. This is an algebraic notion in that we think of the pair $\langle s, e \rangle$ being treated as giving all the information necessary to identify the next state. In LTSs, if you give δ a member of S , then it gives you back a relation which is a subset of $\text{Act} \times S$. This is a coalgebraic notion. Giving the function δ a member of S gives you back the local structure of S , namely a relation telling you how to get to other members of S . A relation $\mathcal{R} \subseteq \text{Act} \times S$ is a member of $\mathcal{P}(\text{Act} \times S)$, i.e., a collection of pairs whose first member is in Act and whose second member is in S ; thus $p \in \mathcal{R}$ has the form $\langle a, s \rangle$.

The difference is a bit easier to see if we assume the LTS is deterministic. Let δ be the next-state function of an FSM. We can define the $\vec{\delta}$ function over states and strings of event types inductively where ε is the empty string of event types:

$$\begin{aligned} \vec{\delta}(s, \varepsilon) &= s \\ \vec{\delta}(s, a_{n+1} \cdots a_1) &= \vec{\delta}(\delta(s, a_{n+1}), a_n \cdots a_1) \end{aligned}$$

This defines $\vec{\delta}$ inductively via a recursion scheme. Next, assume δ is the next-next state function of a deterministic LTS, δ has type

$$\delta : S \rightarrow [\text{Act} \rightarrow S].$$

where $[\text{Act} \rightarrow S]$ is the set of all functions from Act to S . This says if you give δ a member of S , it will give you back a function from Act to S . The fact that it gives you back a function means that the LTS is deterministic.

We then have the definition of $\vec{\delta}$

$$\vec{\delta}s = \lambda a . \vec{\delta}(\delta s a)$$

where the λ is from logic (and computer science). The λ says “I am a function, give me an a and I will tell you what to do next”, which is $\vec{\delta}(\delta s a)$ in our case. The δs yields a function, call it, g , such that $g a$, which yields another state, say, s' . The prescription $\vec{\delta}s'$ then uses the co-recursive definition which expands out to

$$\vec{\delta}s' = \lambda a . \vec{\delta} s' a.$$

The two instances of the variable a in $\vec{\delta}s$ and $\vec{\delta}s'$ are not related, the λ acts like a quantifier in this respect, i.e.,

$$\forall x Q(x) \text{ and } \forall y Q(y)$$

are equivalent. The variables x and y just indicate “holes” in the formula to be filled in. Similarly the λa just indicates a hole to be filled in the subsequent expression.

Notice that the entire line seems to define $\vec{\delta}$ with an expression chasing its tail; that is, it recurses but there is no way to stop the recursion (in this case). This form is called *corecursion* and is defining $\vec{\delta}$ by *coinduction*. This feature makes coalgebras and coinduction an easy way to define things that go 'round, like state machines that loop.

Definition 5.2.4 A labeled transition system $\mathcal{T}$ is *canonical* if for each state $s \in S$,

$$s = \{\langle a, s' \rangle \in \text{Act} \times S \mid s \xrightarrow{a} s'\}.$$

In other words, $\mathcal{T}$ is canonical if its transition function δ is the identity function on the state set S . If S is a canonical labeled transition system which is bisimilar to $\mathcal{T}$, then we say that S is a *canonical form* of $\mathcal{T}$.

Notice that in a canonical labeled transition system, the states are literally sets that tell you how to progress to the next states. A state in this sense is not some abstract atom or urelement, it is an actual and possibly non-wellfounded set.

As an example, let us rewrite the graph of Figure 5.4 (the graph on the left) using the representation we used in Section 3.1 (the graph on the right):

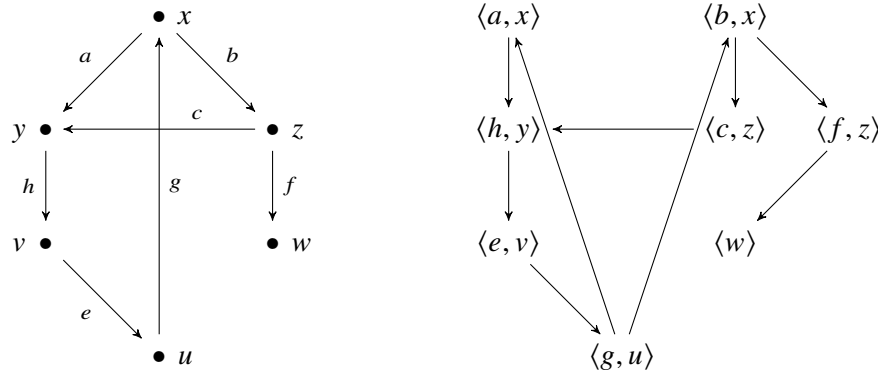


Figure 5.5: LTS Representation

Suppose that we add an arc e from w to u , then we have

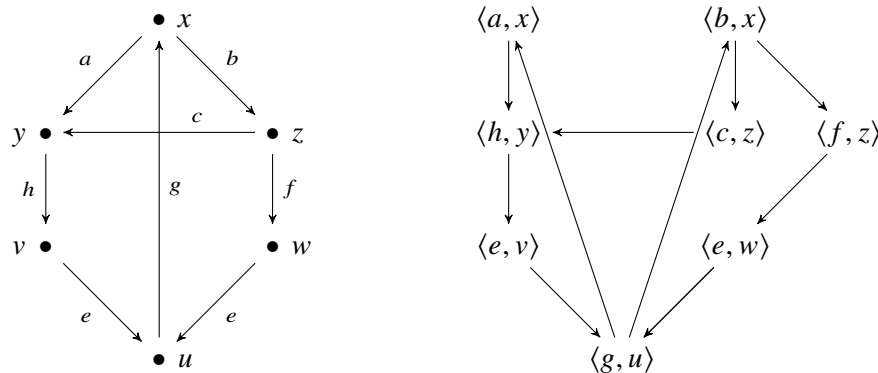


Figure 5.6: Extended LTS Representation

The nodes v and w now look the same locally. In the set theory representation below, the sets representing these two nodes will be the same set and hence the two nodes become one in the set theory representation. The graph is not the set theory representation. It is rather a presentation of the underlying set. This is an important fact about the set theory representation. It is brought out by noticing in the set bisimulation below, the bisimulation actually says these two sets are the same.

It can be somewhat tricky to present graphs in a document describing their representation in set theory. In the graph for Figure 5.5, the $x, y, \dots$ are not officially *labels* in terms of our set theory representation. They are rather artifacts of presenting the graph on the page and in essence those designations qua designations do not exist in the graph. The nodes are there but the presentation with $x, y, \dots$ is merely for our convenience in talking about the graph.

Labeled graphs will provide the capability of added extra kinds of information at a node. We define a label function l for the graph of Figure 5.6 with

$$\begin{aligned} l\{\langle a, x \rangle\} &= \{a\} \\ l\{\langle e, v \rangle\} &= \{e\} \end{aligned}$$

and so on. So $l : \text{Nodes} \rightarrow \mathcal{P}(\mathcal{U})$, where the members of $\mathcal{P}(\mathcal{U})$ are sets of urelements. Our extended LTS representation can now code $\langle p, s \rangle$ as $\{p, d(s)\}$.

The non-wellfounded set decoration of the LTS of 5.6 is

$$\begin{aligned} d\langle a, x \rangle &= \{\langle a, d\langle h, y \rangle\}\} & d\langle f, z \rangle &= \{\langle f, d\langle e, w \rangle\}\} \\ d\langle b, x \rangle &= \{\langle b, d\langle c, z \rangle, d\langle f, z \rangle\}\} & d\langle e, v \rangle &= \{\langle e, d\langle g, u \rangle\}\} \\ d\langle h, y \rangle &= \{\langle h, d\langle e, v \rangle\}\} & d\langle g, u \rangle &= \{\langle g, d\langle a, x \rangle, d\langle b, x \rangle\}\} \\ d\langle c, z \rangle &= \{\langle c, d\langle h, y \rangle\}\} & d\langle e, w \rangle &= \{\langle e, d\langle g, u \rangle\}\} \end{aligned}$$

As you can see

$$d\langle e, v \rangle = d\langle e, w \rangle$$

and hence $d\langle e, v \rangle$ and $d\langle e, w \rangle$ refer to the same set. So the non-wellfounded set theoretic representation collapses nodes that have the same local structure. Just for the record, this is related to *Mostowski's collapse lemma*, which is fundamental to well-founded set theory. Here, the Lemma can be promoted to a formulation of the anti-foundation axiom (AFA) [9] for non-wellfounded sets.

(AFA) Every graph has a unique decoration.

Now we can state bisimulation for decorated labeled graphs::

Definition 5.2.5 ([8]) A *bisimulation relation on sets* is a binary relation R on set which satisfy the following condition: if $\mathcal{R}\varphi\psi$ then

- (1) for every set $\sigma \in \varphi$ there exists a set $\tau \in \psi$ such that $\mathcal{R}\sigma\tau$.
- (2) for every set $\tau \in \psi$ there exists a set $\sigma \in \varphi$ such that $\mathcal{R}\sigma\tau$.
- (3) $\varphi \cap \mathcal{U} = \psi \cap \mathcal{U}$.

This works because no urelement is a set in $V_{afa}[\mathcal{U}]$ even though every urelement is in $\mathcal{U}$. Put another way, no urelement a is of the form $\{b\}$ regardless of whether $b \in V_{afa}$ or $b \in V_{afa}[\mathcal{U}]$.

Proposition 5.2.6 ([8]) For every binary relation $\langle A, \mathcal{R} \rangle$ there is a set B such that $\langle A, \mathcal{R} \rangle$ is isomorphic to $\langle B, \ni_B \rangle$ where $\ni_B$ is the converse of the membership relation on B .

Lemma 5.2.7 ([8]) If G is any labeled graph and d is its decoration, then $d[G]$ is a set of sets that is transitive on sets; transitive on sets means if $a \in b \in d[G]$ and a is a set (not an urelement) then $a \in d[G]$.

The identity relation $\mathcal{I}$ on a set C is that relation such that $\mathcal{I}ab$ iff $a = b$ for all a, b in C .

Theorem 5.2.8 (Strong Extensionality, [8]) Let $\mathcal{I}$ be the identity relation on sets. Then $\mathcal{I}$ is the largest identity bisimulation relation on sets. That is,

1. $\mathcal{I}$ is a bisimulation relation on sets.
2. If $\mathcal{F}$ is a bisimulation relation on sets, then $\mathcal{F}$ is a subrelation of the identity relation. That is, if $\mathcal{F}ab$ then $a = b$.

The reason Theorem 5.2.8 is useful is that exporting the states from the plant to the spider represents somewhat of a security issue. If the spider rather uses its own states but there is still a need to check states, then it must construct states that match with at least some of the plant's states. The representation of states as sets reduces the state machine to its minimal configuration which matches the Myhill-Nerode theorem for machine minimization. Hence if both the spider and plant are in their minimal configuration, i.e., their set representation matches precisely their actual structure, then a bisimulation will be some subset of a bijection.

Dovetailing with the above, there is a proposition that should be of use in determining whether the spider's state machine and the plant's have the same structure. It relies upon labeled graphs and the use of urelements. It requires a bit of set up. The overview is that we will collapse a state machine into a labeled graph of (possibly) non-wellfounded sets. This will wipe out the use of separate relations for the event types but that information has been encoded in the states. Using separate relations is useful for other purposes.

Definition 5.2.9 ([8]) Let G and G' be any labeled graphs over $\mathcal{U}$. A *bisimulation relation* between G and G' is a relation $\mathcal{R} \subseteq G \times G'$ satisfying the following conditions:

- (1) If $g\mathcal{R}g'$ and $g \twoheadrightarrow_G h$ then there is h' with $h\mathcal{R}h'$ and $g' \twoheadrightarrow_{G'} h'$.
- (2) If $g\mathcal{R}g'$ and $g' \twoheadrightarrow_{G'} h'$ then there is h with $h\mathcal{R}h'$ and $g \twoheadrightarrow_G h$.
- (3) if $g\mathcal{R}g'$ then $l(g) = l(g')$.

Proposition 5.2.10 *Let g and g' be nodes in labeled graphs G and G' over A . There is a bisimulation $\mathcal{R}$ between G and G' such that $g\mathcal{R}g'$ iff $d(g) = d'(g')$, where d, d' are the decorations of G and G' respectively. Hence, G and G' are bisimilar iff $d[G] = d'[G']$.*

Now all that is left is to code the state machines.

Every state transition in a state machine is accompanied by an event type, we used a above. However, there are also operations associated with an event of that type occurring. For $s \xrightarrow{a} s'$, let $\text{ops}(s, a)$ refer to the operations that are executed when the machine is in state s and the event a occurs. We can then recode the next-state function LTS's, now representing state machine transitions as

$$s \xrightarrow{\text{ops}(s, a)} s'.$$

We recode the canonical state machines as

$$s = \{ \langle a, \text{ops}(s, a), s' \rangle \in \text{Act} \times \mathcal{P}(\text{ops}) \times S \mid s \xrightarrow{\text{ops}(s, a)} s' \},$$

where the tuple $\langle a, \text{ops}(s, a), s' \rangle$ is $\langle \langle a, \text{ops}(s, a) \rangle, s' \rangle$

Now we can realize the conditions under which the spider has the same state structure as the plant without the plant exporting its state symbols. We recode both state machines as LTSs in non-wellfounded set theory and use Strong Extensionality for the criteria needed for identity of structure.

Consider the state machine of Figure 3.1. We can write this as a set of what are known as *coequations*. These are equations that seem to chase their tail and are readily represented in a lazy functional language such as Haskell or ReWire. In fact, a state machine in ReWire is represented a collection of coequations. The state machine of Figure 3.1 in coequational form is

$$\begin{aligned} s_0 &= t_0 s_1 + t_2 s_0 \\ s_1 &= t_1 s_1 + t_3 s_0 \end{aligned}$$

where the $+$ has the force of disjunction or choice. This rather transparently encodes the state machine. Operations can also be represented by extending the syntax a bit:

$$\begin{aligned} s_0 &= t_0 \text{ ops}(s_0, t_0) s_1 + t_2 \text{ ops}(s_0, t_2) s_0 \\ s_1 &= t_1 \text{ ops}(s_1, t_1) s_1 + t_3 \text{ ops}(s_1, t_3) s_0 \end{aligned}$$

A solution to this set of equations in V_{afa} is a non-wellfounded set. These equations are *flat* in the sense that there are no left hand sides embedded in sets on the right hand sides. Put another way, there is no depth in the equations. This feature makes them quite like algebraic equations and in fact, because they seem to chase their tails, are called *coalgebraic equations*.

5.3 Checking the Plant's Outputs

Checking a plant's output can get a bit tricky in accounting for the fact that state changes occur on the clock ticks. Consider again the following spider+plant system

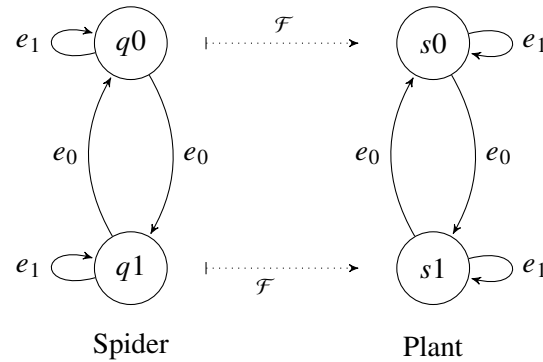


Figure 5.7: Simple Spider+Plant

The code in the spider must read

```

1 case qstate is
2   when q0 =>
3     if (pstate = s0) then
4       if (e0) then
5         qstate <= q1;
6       elsif (e1 = '1') then
7         qstate <= q0;
8       else
9         error;
10    else
11      error;
12  when q1 =>
13    if (pstate = s1) then
14      if (e0) then
15        qstate <= q_0;
16      elsif (e1 = '1') then
17        qstate = q1;
18      else
19        error;
20    else
21      error;

```

This seems simple enough. However, if the plant is also setting outputs when it makes a state change, then it can be reading values in the current clock tick but the output only gets set in the next clock tick just like the state variable.

However, now suppose that s0 on e0 sets an output

$$o \leq p \text{ and } r$$

for some signals p and r . These are being read in the current clock tick. The spider cannot check the value of o in the current clock tick because that value will only get updated in the next clock tick. Hence the code must read

```

1  case qstate is
2    when q0 =>
3      if (pstate = s0) then
4        if (e0) then
5          qstate <= q1;
6          saved_p <= p;
7          saved_r <= r;
8        elsif (e1 = '1') then
9          qstate <= q0;
10       else
11         error;
12     else
13       error;
14   when q1 =>
15     if (pstate = s1) then
16       if (o = saved_p and saved_r) then
17         if (e0) then
18           qstate <= q_0;
19         elsif (e1 = '1') then
20           qstate = q1;
21         else
22           error;
23       else
24         error;
25     else
26       error;

```

6. INVARIANTS

6.1 Cheaper Spiders using Invariants

An invariant is a statement or proposition that holds regardless of the state in which the plant or spider finds itself. The terminology comes from Floyd-Hoare logic [10, 11]. Floyd-Hoare logic treats the necessity modal operator as the weakest precondition operator. That is, given a triple

$$\{P\} C \{Q\}$$

where P and Q are logic statements and C is a piece of computer code (the $\{ \}$ are mere syntax), the weakest precondition for Q results in a logic statement Q' such that $P \supset Q'$ and if we were to execute C while $P \supset Q'$ is true, then Q would become true. We essentially adapted this for finite state machines where the weakest precondition Q' with respect to C becomes $[k_j^o]Q$ for statements in the spider associated with event type e_j . The statement P is the condition expected to hold of a state just before C is executed and Q is expected to hold in any state immediately following C .

An invariant for Algol style computer languages such as C, Pascal, etc. is a statement that holds before a loop and continues to hold throughout every iteration of the loop. For us, a finite state machine looks like a

very complicated looping structure requiring goto statements were we to recode it in a Algol-like language. In a functional language, an invariant holds throughout every (co)recursive call.

For state machines, an invariant is simply a statement. Suppose the statement P is expected to hold throughout all the states of a plant. The spider then looks at $[f^\circ]P$ (where $\mathcal{F}$ is the concurrence relation between the spider and the plant) and evaluates it regardless of what state in which the spider might actually be. Of course there are variations on this. A spider might have more than one state, some in which it checks the invariant and some in which it does not.

6.2 Assigning a Syntactic Map to Local Operators

In Floyd-Hoare, the weakest precondition $[k_j^\circ]Q$ computed using

$$\{P\} C \{Q\}$$

is a formula Q' derived from the bit of program code. That is, C is conceived as a function that given the statement Q , returns a statement Q' that we call $[k_j^\circ]Q$ where $\mathcal{K}_j$ represents in relational form the code C ; that is, $\mathcal{K}_j$ is the collection of state pairs with the first state from before C is executed and the second state from after C is executed. Used this way, C is the *intensional*⁵ form of the relation $\mathcal{K}_j$. That is, $\mathcal{K}_j$ is not represented extensionally as a collection of pairs of states. Rather, $\mathcal{K}_j$ is represented as the piece of syntax C that tells us how to get from states before C is executed to states after C is executed. An example is

$$\{P\} x = y + x \{x = 10\}.$$

The syntactic computation is

$$(x = 10)[y + x \mapsto x] \doteq (y + x = 10)$$

where the $\doteq$ is equality in the semantic metalanguage and $=$ signs are equality in the logic (object) language. Hence in order for $x + 10$ to obtain after the computation, then $P \supset (y + x = 10)$ must obtain before the computation. P is any precondition that makes sense in the larger program. “If” statements are also suitably handled:

```

1      {P}
2      if (c1) then
3          pstate <= p1;
4      {pstate = r + 2}
```

Let the bit of program at lines 2 and 3 be called C , then

$$pstate = r + 2[C] \doteq (P \wedge c_1) \supset (p_1 = r + 2)$$

where we use similar variables in the VHDL and the logic formulas.

⁵The word “intensional” is not a misprint. It comes from philosophical logic and in this case, means some finite description of a relation. One can think of it as the “intent” of the relation as opposed to the “extent” of the relation which means listing the pairs of the relation.

In order for a condition Q to hold for some state s' , the weakest precondition of Q must hold in all states s that can transist to s' . Given the manner in which we represent states and if statements, this amounts to $[k_j^a]Q$ holding at s' for all events types e_j such that $s \xrightarrow{\text{ops}(s,a)} s'$ and $e_j = \langle a, \text{ops}(s, a) \rangle$.

7. MONITORING TIME LIMIT CONDITIONS

Monitoring time limits is a bit difficult at the logical level *simpliciter*. Logic has no way of recording time or anything else. Logic can probe a time variable but it cannot count because it is mostly a (propositional) functional language. The same strategy is nearly applicable to spiders, except where something active like a clock tick count is involved. There are various paths for a solution:

- Rely upon the FPGA application to provide the clock count. This is trusting the FPGA application to do it accurately.
- Include mitigation code, in this case, VHDL that has a clock count variable that can be updated and then further referred to in the logic.
- Produce another *sidecar* spider that only counts clock ticks and keeps a clock count. The main spider then reads this count variable. The term *sidecar* is not a technical term. We use it to refer to code connected to a main process but does some very specific work for the main process.

8. CONCLUSIONS

We have shown there is ample room to apply logic for the construction of spiders. In this report, we only used logic as a guide for how to structure a spider to check the conditions written in the logic. Eventually, we would like to express the spider in terms of logic + mitigation code and have a special compiler process this into an actual spider that is then included with application source code. The result is then handed to vendor supplied tools to produce the bitstreams necessary for vendors' FPGAs. This goal is not pre-ordained to be achievable. Since spiders are ultimately FPGA code and that is usually constructed to implement a state machine, the logic + mitigation must generate a state machine.

We can ignore, for now, mitigation code since that code would merely be included in any generated spider. It is easier to see how to generate a state machine from logic if only invariants are used. There is a method called *filtration* used in modal logic to generate finite models. The models have a finite collection of worlds, i.e., states, and finite relations among them. This seems a likely place to start. If state checking of the plant's FSM by the spider's FSM is required, it is unclear whether the filtration method will produce a machine that is bisimulation equivalent to the plant's FSM. However, now we at least have a set theoretic representation that could be checked for bisimulation.

This report assumed VHDL was used for both the spider and the plant. In the future, we would like to use the language ReWire. ReWire has a compiler that produces either VHDL or Verilog with the change of a switch. The reason to use ReWire is that we have more control over the language. VHDL, and to a lesser extent, Verilog have less formal underpinnings. Both languages were originally developed as simulation languages (using the engineering meaning of the term "simulation"). As such, they contain some inelegant constructs not directly having to do with FPGAs. It is the vendors' tools that produce FPGA bitstreams.

The issue of how intrusive to make a spider is not an easy question to answer. In a very high assurance system, many tight connections might be necessary but the resultant cost must then be factored in. The issue comes down to how expensive can a spider be allowed to be given the application being spidered and available resources to the spider.

There is an important issue with respect to a very intrusive spider. The more intrusive a spider is, the more signals must be exported from the plant to the spider. This opens up those signals to possible snooping by adversaries depending upon how the spider and plant are implemented. If the spider is very close to the plant in terms of chip layout, then the signals need not be exported beyond the confines of the FPGA and may not be such a problem. If the spider and plant are in two separate FPGAs, then the signals are exposed to snooping by an adversary.

The issue of where the spider exists relative to the plant in terms of chips becomes more pressing if the spider is working with a SCADA system. There, it may not be possible to keep the spider and plant on the same chip. A further problem with these sorts of designs is that veracity of the signals may now be suspect, i.e. open to tampering.

There are new issues with *safety* and *liveness* with respect to spiders and plants. From [14],

- a *safety* property is a trace property that proscribes “bad things” and can be proved using an invariance argument, and
- a *liveness* property is a trace property that prescribes “good things” and can be proved using a well-foundedness argument.

From [15],

Distinguishing between safety and liveness properties is useful because proving that a program satisfies a safety property involves an invariance argument while proving that a program satisfies a liveness property involves a well-foundedness argument.

Trace properties look at finite sequences of states. In FPGAs, finite sequences appear to have little value. While we have developed notions of invariants for FSMs, it is not at all clear whether these are susceptible to trace properties. Well-foundedness arguments work with well-founded set models. The most natural models for spiders and plants are non-wellfounded. It is not clear at this point whether these older notions of safety and liveness makes sense for FPGAs. We may have to revisit these notions and recast them for FPGAs.

9. APPENDIX

9.1 Distributed Logic: Axioms and Rules

A local logic is “local” in that it is associated with one node in the graph. In this paper, the accompanying notion of a global logic does not entail formulas “spanning” two local logics in the sense of P in one logic implying Q in another where “implying” is reified as an implication connective (and similarly with other two

place connectives). Each formula is entirely within a single local logic although it may contain subformulas from others.

The distributed logic graphs we use have *endo-diagrams*, each of which is a labeled node and a single endo-arc (self-arc). Each endo-arc will be translated into an endo-morphism. Each node is required to have a special endo-arc that will be translated under interpretation as an identity morphism. This is necessary since the models for the logic will be a category. In category theory, every object (here node) is required to have an identity morphism. The graph specifies which local logics there are to be, which morphisms are to appear in any model, and force identity morphisms to exist. Each local logic may have its own propositional atoms and local modal connectives. In the Base System (see below), the **S** specification and **A** and **B** axioms are not optional.

In axiom A2 below, the local modal logic is a classical modal logic that contains all the operators, for a node at h , contains the operators in $\{[h^\circ], [h^\circ], [h^\circ], [h^\circ]\}$ with interpreting relation $\mathcal{H}$. There can be as many local collections of modal operators and interpreting relations as one like, but here to simplify the presentation, we only insist on one.

The local modal logic can contain any additional modal axioms deemed necessary to adequately describe conditions modeled by that node.

9.2 Base System

Graph Specification S:

For an arc $f : k \rightarrow h$ in a graph, let $\text{dom}(f) = k$ and $\text{cod}(f) = h$.

- | | |
|---|--|
| S1. A graph $\mathfrak{G}$ of nodes and arcs
A set $\mathfrak{D}$ of endo-diagrams | S2. An endo-diagram with an
arc i for each node in $\mathfrak{G}$ |
|---|--|

Axiom Schemes A: For each node in $\mathfrak{G}$,

- | | |
|--|--|
| A1. all (two-valued) truth functional
theorems of a propositional logic | A2. Modal axioms for a logic
at this node |
|--|--|

Each node h must contain an endo-diagram for each class of modal connectives in its local logic.

Axiom Schemes B: These axioms force arcs to be interpreted as morphisms in a category. For arcs $r : h \curvearrowright k$ and $s : k \curvearrowright l$,

- | | |
|---|---|
| B1. $P \equiv [i^\circ] P$
B1'. $P \equiv [i^\circ] P$ | B2. $[r^\circ] [s^\circ] P \equiv [s \circ r^\circ] P$
B2'. $[r^\circ] [s^\circ] P \equiv [s \circ r^\circ] P$ |
|---|---|

where $[i^\circ]$ and $[i^\circ]$ is interpreted by the identity relation for any node h .

Axiom Schemes C: Taken all together these axioms would force the distributed modal connectives to be normal.

$$C1. [r^\square]P \wedge [r^\square]Q \supset [r^\square](P \wedge Q)$$

$$C2. [r^\square](P \wedge Q) \supset [r^\square]P \wedge [r^\square]Q$$

$$C3. \top \supset [r^\square]\top$$

$$C1'. [r^\diamond]P \wedge [r^\diamond]Q \supset [r^\diamond](P \wedge Q)$$

$$C2'. [r^\diamond](P \wedge Q) \supset [r^\diamond]P \wedge [r^\diamond]Q$$

$$C3'. \top \supset [r^\diamond]\top$$

The Axiom Schemes **C** should be present to specify simulation logic [12]; they also allow the specification of backward looking connectives residuated (see [5]) with their forward looking counterparts.

Axiom Schemes D: These schemes assure that the forward and backward connectives are connected by a condition known as *residuation*. For $r : k \multimap h$,

$$D1. [r^\diamond][r^\square]P \supset P$$

$$D2. [r^\diamond][r^\square]Q \supset Q.$$

The schemes **D** are, in the presence of the monotonicity of the operators that are derived from their distribution properties with the Boolean $\wedge$ and $\vee$, equivalent to the following rules:

$$D3. \frac{[r^\diamond]P \supset Q}{P \supset [r^\square]Q}$$

$$D4. \frac{[r^\diamond]Q \supset P}{Q \supset [r^\square]P}$$

9.3 Options

Axiom Schemes E: The **E** axioms are examples of extra properties to be enforced on the interpreting morphisms. Other axioms can be added as well. We use Axioms **E** as paradigm examples, for $r : k \multimap h$:

$$E1. [r^\square]P \supset [r^\diamond]P$$

$$E2. [r^\diamond]P \supset [r^\square]P$$

$$E1'. [r^\square]P \supset [r^\diamond]P$$

$$E2'. [r^\diamond]P \supset [r^\square]P$$

In normal systems, the first specifies the interpreting relation be total on its domain and the second that it act functionally.

Axiom Schemes F: The axiom **B1** is only necessary if you wish the classical proposition logic at $\text{dom}(r)$ to be included in the logic at $\text{cod}(r)$, for $r : k \multimap h$:

For all propositional letters p and q ,

$$F1. p \supset [r^\square]p$$

$$F2. q \supset [r^\diamond]q$$

In normal distributed logics, the axiom G1 forces the arcs in the graph to be interpreted as simulation relations. The axioms B2 and B2' force composition of relations to hold, where a simulation relation is one “half” of a bisimulation [13]. One common use of the simulation relation is when the interpretation of $[r^\circ]$ via a relation $\mathcal{R}$ is a p-morphism. To force this, add the Axiom Schemes **C** and **E** to the simulation axiom.

Axiom Scheme G: Simulation logic [12] requires for an arc $r : h \curvearrowright k$ in $\mathfrak{G}$, and modal connectives $[h^\circ] \in \text{dom}(r)$, $[k^\circ] \in \text{cod}(r)$,

$$\text{G1. } \langle r \rangle [k^\circ] P \supset [h^\circ] [r^\circ] P$$

$$\text{G2. } [r^\circ] [h^\circ] P \supset [k^\circ] [r^\circ] P$$

9.4 Definitions and Rules

Definition of Possibility: For any $m : l \curvearrowright l$ or $m : k \curvearrowright h$,

$$[m^\circ] U \stackrel{\text{def}}{=} \neg [m^\circ] \neg U, \quad [m^\circ] V \stackrel{\text{def}}{=} \neg [m^\circ] \neg V.$$

Rules A: For each local logic at l interpreted by the relation $\mathcal{L}$,

$$\frac{\vdash_l (U_1 \wedge \cdots \wedge U_n) \equiv U}{\vdash_l ([l^\circ] U_1 \wedge \cdots \wedge [l^\circ] U_n) \equiv [l^\circ] U} \quad \frac{\vdash_l (V_1 \wedge \cdots \wedge V_n) \equiv V}{\vdash_l ([l^\circ] V_1 \wedge \cdots \wedge [l^\circ] V_n) \equiv [l^\circ] V}$$

$$\frac{\vdash_l U \quad \vdash_l U \supset V}{\vdash_l V}$$

where the appellation of $\vdash$ indicates the local logic to which the proof sign attaches.

Rules B: For each $f : k \curvearrowright h$ arc in $\mathfrak{G}$,

$$\frac{\vdash_h (P_1 \wedge \cdots \wedge P_n) \equiv P}{\vdash_k ([f^\circ] P_1 \wedge \cdots \wedge [f^\circ] P_n) \equiv [f^\circ] P} \quad \frac{\vdash_k (Q_1 \wedge \cdots \wedge Q_n) \equiv Q}{\vdash_h ([f^\circ] Q_1 \wedge \cdots \wedge [f^\circ] Q_n) \equiv [f^\circ] Q}$$

Rules D: The **Axioms D** can be replaced in the presence of **Axioms C** with the following rules (which are frankly much easier to use). For each $f : k \curvearrowright h$ arc in $\mathfrak{G}$, the following two way rules hold from top to bottom, and bottom to top:

$$\frac{\vdash_h P \supset [f^\circ] Q}{\vdash_k [f^\circ] P \supset Q} \quad \frac{\vdash_k Q \supset [f^\circ] P}{\vdash_h [f^\circ] Q \supset P}$$

REFERENCES

1. 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.

2. 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.
3. P. Cousot and R. Cousot, “Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints,” Proceedings of the Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA (ACM Press), 1977, pp. 238?–252.
4. 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).
5. J.M. Dunn and G. Hardegree, *Algebraic Methods in Philosophical Logic*, Oxford Logic Guides 41 (Oxford University Press, 2001).
6. D. Barker-Plummer, J. Barwise, and J. Etchemendy, *Language, Proof, and Logic*, second ed. (CSLI Publications, 2011).
7. P.J. Freyd and A. Scedrov, *Categories, Allegories* (North–Holland, 1990).
8. J. Barwise and L. Moss, *Vicious Circles* (CSLI Publications, 1996).
9. P. Aczel, *Non-Wellfounded Sets* (CSLI Publications, 1988). Lecture Notes 14.
10. C.A.R. Hoare, “An Axiomatic Basis for Computer Programming,” *Communications of the ACM* **12**, 576–580, 583 (1969).
11. K.R. Apt, “Ten Years of Hoare’s Logic: A Survey–Part 1,” *ACM Transactions on Programming Languages and Systems* **3**, 431–483 (1981).
12. G. Allwein, W.L. Harrison, and D. Andrews, “Simulation logic,” *Logic and Logical Philosophy* **23**(3), 277–299 (2014).
13. D. Sangiorgi, *Introduction to Bisimulation and Coinduction* (Cambridge University Press, 2012).
14. M.R. Clarkson and F.B. Schneider, “Hyperproperties,” Proceedings of the Proceedings of the 2008 21st IEEE Computer Security Foundations Symposium (IEEE Computer Society), 2008, pp. 51–65.
15. B. Alpern and F.B. Schneider, “Recognizing Safety and Liveness,” *Distributed Computing* **2**, 117–126 (1987).