Formal Aspects of VLSI Research Group
University of Utah, Department of Computer Science

HOP: A Formal Model for Synchronous Circuits using Communicating Fundamental Mode Symbolic Automata*

GANESH GOPALAKRISHNAN (*ganesh@cs.utah.edu*)
University of Utah
Dept. of Computer Science
Salt Lake City, Utah 84112

Keywords: Formal Hardware Specification, Verification, Synchronous VLSI Design, Symbolic Automaton Models

Abstract. We study synchronous digital circuits in an abstract setting. A circuit is viewed as a collection of *modules* connected through their boundary *ports*, where each port assumes a fixed direction (input or output) over one *cycle* of operation, and can change directions across cycles. No distinction is made between clock inputs and non-clock inputs. A cycle of operation consists of the application of a set of inputs followed by the stabilization of the module state before the next inputs are applied (*i.e.* fundamental mode operation is assumed). The states and inputs of a module are modeled *symbolically*, in a functional notation. This enables us to study not only finite-state controllers, but also large data paths, possibly with unbounded amounts of state. We present the abstract syntax for modules, well-formedness checks on the syntax, the formal semantics in terms of the denotation of a module, and the rule for composing two modules interconnected and operating in *parallel*, embodied in the operator *par*. It is shown that *par* preserves well-formedness, and denotes conjunction. These results are applicable to virtually every kind of synchronous circuit (e.g. VLSI circuits that employ single or multiphase clocks, circuits that employ switch or gate logic structures, circuits that employ uni- or bi-directional ports, etc.), thanks to the small number of assumptions upon which the HOP model is set up.

1 Introduction

Synchronous digital circuits are, perhaps, one of the most widely studied forms of hardware. The most common definition of a synchronous digital system is that it is globally clocked. This definition is often not *directly* applicable to all real-world synchronous hardware systems because some real-world synchronous systems employ polyphase clocks, and their submodules use different subsets of the *set of global clock phases*. An obvious generalization of the above definition of synchronous systems is: "a system with a logical clock that acts as the global time reference, and to which all the actual clock phases are aligned". Though mathematically accurate, this view of synchronous systems violates the principle

*Supported in part by NSF Awards MIP 8710874 and 8902558 – printed 3/13/92 of modularity: a hardware module cannot be understood purely in terms of its interface signals; one is forced to refer to a (fictitious) global signal called the (logical) global clock, and relate all the actual clock signals to it. Another problem with this approach is that often the distinction between data and clocks blurrs – especially in switch-logic structures that employ *level activated* latches controlled by pass-transistors, and dynamic storage. It is then not clear how to accurately talk about global clocking. In addition, in polyphase clocked systems, there often exist embedded submodules that do not have an explicit clock signal, but actually respond to rhythmically arriving data. In this paper, we discuss how an adequate definition that promotes a modular view of synchronous systems can be set up.

A synchronous digital system is often viewed as a pair $(next\_state\_function,\ output\_function)$. This definition does not take into account the boundary *ports* of the module through which inputs are received and outputs are generated. If the set of input and output ports are fixed over time, there is no problem with this view; however, in many real-world synchronous systems, the set of input and output ports are not fixed. The directionality of ports is clearly a matter of concern to a digital designer who wishes to interconnect modules and wants to avoid errors due to two incompatible values "clashing" at a node where two ports that assume the output direction meet. In this paper, we discuss how to take into account varying port directions in a synchronous system.

One often hears "do" and "don't" rules about synchronous hardware. Some familiar rules are: "do not create combinational loops"; do not drive the same node with two sources. In other contexts, for example *two-phase clocked designs*, the following rule is often mentioned: "ensure that every cyclic path is clocked by both phases in an alternating fashion." Yet, it is often not clear whether such usage rules are sufficient to guarantee something desirable, and what some of the desirable properties are. In this paper, we discuss the topic of *adequacy of well-formedness checks.*

In a Nutshell...

In a nutshell, to accommodate the above concerns in a formal setting, we take a fresh look at synchronous systems. We answer the following questions about synchronous hardware, in the framework of our simple, extended automaton based modeling formalism. Can we distill out the essential properties of synchronous hardware and capture it in a simple notation? Can this formalism serve as the basis of a simple HDL? Can well-formedness rules for synchronous hardware be formulated with respect to the syntax of a simple HDL? Are the well-formedness rules adequate, and if so in what sense? Would such an HDL be general enough to ignore differences between multiple clocking schemes, uni- *vs.* bi-directional ports, *etc.*? How is *time* defined in this HDL? Is the HDL intuitive? Are descriptions able to deal with a potentially unbounded amount of state; is this HDL suitable for describing both controllers and data path modules? What is the syntax of such an HDL, well-formedness restrictions on the syntax, and syntax directed semantics? Is there a (binary) parallel composition operator (say *par*) that specifies the behavior of two modules that are interconnected? Does *par* preserve well-formedness? What is the denotation of *par*? Can *par* be implemented as an efficient algorithm to deduce a behavioral description from a structural description? We have not seen anyone else address all these questions in the context of a simple HDL.

In this paper, we answer the above questions in the affirmative. Synchronous behavior is defined using one simple and well-known idea: that of the *fundamental mode* behavior.

Related Work

Finite automaton models were perhaps the first of the formal models to be used for studying synchronous hardware [1]. Though the finite automaton models such as the Mealy machine model are elegant, intuitive, and practical to be used for studying small circuits such as finite state controllers, they prove to be inadequate for modeling large data path modules such as translation lookaside buffers (TLBs), RAM memories, or even registers of large widths, because these data path units have large amounts of internal state that can seldom be explicitly modeled. In response to the need to model larger systems, various hardware description languages (HDLs) were invented; some recent examples are HardwareC [2], ISPS [3], VHDL [4], and Verilog [5].

However, most of the above HDLs are simulation oriented, and do not have a published formal semantic definition, nor the simplicity and the intuitiveness of an automaton model. Therefore the formal design community has searched for something that is expressive as well as semantically well specified. Many such HDLs have been proposed in the last decade. These formally based HDLs are deliberately kept domain specific because otherwise their semantics would become too involved. For example, many of these HDLs adopt a purely functional approach [6, 7, 8, 9, 10], while others adopt a formal process model [11, 12, 13]; languages of the former category are better suited for modeling computation intensive systems while HDLs of the latter category are better suited for modeling control intensive systems. Most of these languages are designed to cater to specific needs such as writing high level specifications, conducting formal verification, or performing high level synthesis. They are often not well suited for capturing the behavior of VLSI modules at a detailed level; details such as single or multiphase clocks, switch *vs.* gate style circuits, the use of uni- *vs.* bi-directional ports, *etc.* cannot be uniformly treated in these approaches. They tend to make assumptions such as "single-phase clocking", "uni-directional ports", *etc.* at the very outset. This is also a weakness of the classical automaton model, as well as many of the "classical HDLs" such as

ISPS, VHDL, and HardwareC.

Formalisms such as the Higher Order Logic [14] or the Boyer-Moore logic [15, 16] have been used for writing both high level and low level hardware descriptions. In principle, any hardware behavior of interest can be specified in these logics and this is one of the claimed advantages of these general purpose logics. For example, in [17], the use of HOL to model switch-logic structures is discussed. So, in principle, approaches that use the language of a formal logic as the HDL are very general and expressive. Despite these advantages, the approach of directly using a formal logic for specifying synchronous hardware systems is not satisfactory because well-formed formulae in the logic of HOL that are used to describe actual hardware need not imply well-formedness (in the hardware sense) of the circuit described. **In other words, what we are aiming for is a notation where the notions of well-formedness of the HDL description and well-formedness of hardware coincide.** The approach advocated by us is to perform the well-formedness checks and *then*, if necessary, translate HDL descriptions into logic for the purposes of verification. However, this translation can be avoided if proof rules are directly formulated in terms of the syntax of the HDL itself [18, 19].

The work reported here is a direct outgrowth of work reported by us over the past several years based on an HDL called "HOP". Two key tools we have developed centered around are PARCOMP [20, 21], and PCA [22, 23]. PARCOMP is used to infer the behavior of a structural description. PCA is used to infer the behavior of regular array structures. These algorithms, especially PARCOMP, has been applied to large systems also. HOP has been used to describe many real-world systems [24, 25, 26], to study composition and abstraction [27, 20], for high-level simulation and test generation [28, 29, 22, 30], and formal verification [31]. All this work has convinced us of the utility of *an extended automaton model to describe hardware*. HOP is an extended automaton model in the sense that its state variables are tuples of individual variables and its transitions are guarded by first order formulae. The questions we have posed in this paper try to identify those aspects that are fundamental to synchronous hardware regardless of the notation in which they are described nor variations in their clocking scheme or circuit implementation style.

Similar automaton models, as well as simple HDLs for modeling hardware have been proposed: Behavioral FSMs [32], Transfer Formulae [33], Lustre [34], and LCF-LSM [35], to name a few. These automaton models do not provide well-formedness conditions and composition rules for synchronous systems. *However*, the emergence of such models is ample evidence to the following fact: designers are leaning towards extended automaton models which are, on the one hand, more powerful than the classical automaton models, but, on the other hand, are much simpler than most HDLs available today, and hence are much more amenable to formal analysis, verification, and even design synthesis.

It is also interesting to compare our work with that reported in [36, 37]. In their approach, they have defined a simple form of *Propositional Temporal Logic*. Formulae in this logic are used to make assertions about circuits that are to be designed and simulated in the COSMOS [38] system. Simulation can be carried out in the scalar mode, using ground values, or in the symbolic mode, using Boolean values. The logic reflects some of the features of the COSMOS model, including the treatment of *inputs and outputs*, and the way `X` values are handled. The HOP model can be viewed as a higher level model that can be built on top of their model – in fact, we have successfully verified many synchronous systems using the COSMOS model for low-level verification and the HOP model for high level verification [30].

Although many compositional models for transistors are under active research [39, 40, 41], they are too complex to be applied directly to synchronous VLSI modeling. Our work can be viewed as a continuation of these works in a sense; for example, in [39], the behavior of a sequential system over a *phase* of activity has been very precisely captured using a non-deterministic iterative process that has a deterministic outcome of stable node values. The net effect achieved by a computation over several sequential phases has not been studied; nor have phase level interactions between a collection of modules, and their collective behavior over several sequential phases studied.

What *we* focus on is precisely these aspects of sequential system behavior. We abstract away from the activities within a phase, and capture only the deterministic outcome of computations within a phase that result in the next state of the system. This reduces the complexity of the specification. We also propose a parallel composition algorithm PARCOMP that deduces the collective behavior of systems across multiple phases. In the past, we have shown that PARCOMP has numerous applications, including error checking across symbolic execution paths, behavioral inference prior to formal verification, and in test generation. The provision of an efficient parallel composition algorithm, the definition of structural well-formedness rules, and the ability to handle a wide variety of examples and problem sizes are some of the key features of our work.

Why another automaton model?

One of the shortcomings of the HOP model we have used so far in our work, as well the extended automaton models used by others (cited above) is that they make several assumptions about synchronous hardware which make the model restrictive. Some of the restrictions are the following:

1. System clocks are either not explicitly modeled, or are assumed to be of some standard variety (such as a single phase clock). It is awkward to study interactions among

systems clocked with different (synchronized) clocks.

2. Ports are assumed to be either always inputs or always outputs.

3. Well-formedness conditions are not syntactically characterized. Therefore, many common varieties of errors have to be detected in the semantic domain (e.g. in the process of a laborious designer guided proof of correctness in a system such as the Boyer-Moore [16] system or HOL [18]), rather than much more cheaply detected by means of well-formedness checks.

4. Parallel composition is either not defined, or inadequately characterized (this remark does not apply to LCF-LSM or HOP).

5. Control/data separation is not adequately supported, or this separation is too rigidly enforced. What we have found desirable is a system where control/data separation is supported, but not enforced. By doing so, various "modes of behavior" of a module—the various threads of control flow pertaining to the execution of "operations" as well as sub-cases that arise within operations—can be studied effectively. Often, during PARCOMP, several modes of behavior of a module can be pruned away when it can be predicted that the environment will not invoke these capabilities.

Features of the new model, and Contributions

In the work presented here, we study synchronous digital circuits in an abstract setting. A circuit is viewed as a collection of *modules* connected through their boundary *ports*, where each port assumes a fixed direction (input or output) over one *cycle* of operation, and can change directions across cycles. No distinction is made between clock inputs and non-clock inputs. A cycle of operation consists of the application of a set of inputs followed by the stabilization of the module state before the next inputs are applied (*i.e.* fundamental mode operation is the only assumption explicitly made). The states and inputs of a module are modeled *symbolically*, in a functional notation. This enables us to study not only finite-state controllers, but also large data paths, possibly with unbounded amounts of state.

We base virtually our entire work on *one assumption*: that fundamental mode behavior has to be guaranteed. This assumption, in a sense, captures the *essence* of synchronous system operation. Virtually all the well-formedness checks relate to this single assumption, and, to the best of our knowledge, are being pointed out for the first time in a cohesive manner.

Approach to Formal Semantics

HOP specifications can be written for a black-box, or a network of black-boxes connected through their interface ports. The former are called *behavioral descriptions*, and the latter are called *structural descriptions*. Our approach to the presentation of HOP in this paper consists of first specifying the formal *syntax* of HOP, then the *well-formedness conditions* on the syntax, and finally, *syntax-directed specification* of the formal semantics. This is similar to the approach taken in presenting formal logics or the semantics of programming languages, and in writing *System Semantics* [8]. Our approach differs from the approach taken in "classical automata theory" [42] in which an automaton is directly specified in terms of semantic objects—the domain of states, and the next-state function on states, for instance. We do not follow the latter approach because of many reasons. First of all, "HOP automata" are more general machines; they are not *finite-state*. Secondly, much of our work involves the syntactic manipulation of "automaton specifications", studying their interaction through connected ports, performing parallel composition, and verifying the correctness of specifications *given at a syntactic level*. For doing all this satisfactorily, in a manner that will help guide a software implementation of HOP, a formal syntactic specification is felt necessary. Therefore, we shall deviate from the classical automata theory notation.

Organization

Section 2 presents our notations. Section 3 presents the syntax and informal semantics of behavioral descriptions. Section 4 presents well-formedness checks defined on behavioral descriptions. Section 5 presents the denotational semantics of behavioral descriptions. Section 6 discusses structural descriptions, and parallel composition, PARCOMP. Section 7 proves that PARCOMP preserves well-formedness. Section 8 provides the denotation of *par*. Section 9 presents several examples illustrating HOP, and relating our view of synchronous hardware with existing views. Implementation notes and conclusions follow in 10.

2 Notations

We use $=$ to denote semantic equality (based on familiar rules of the underlying functional calculus). In the following presentation, we adopt the idea that numbers can be viewed as sets. 0 represents the empty set, and N represents the set containing $0 \ldots N-1$. $\{x_i \mid i \in N\}$ is the set $\{x_0, \ldots\}$; for example, $N = 3$ gives $\{1,2,3\}$. $\#\{x_0, \ldots\}$ obtains the cardinality of a set; for example, $\#\{1,2,3\} = 3$. $\{x_i \mid i \in N\} \cup \{x_j \mid j \in M\}$ obtains the union of the sets; for example, $\{1,2,3\} \cup \{4,6,5\}$ gives $\{1,2,3,4,5,6\}$.

Here are the operations on tuples. $(x_i \mid i \in N)$ gives an N-tuple $(x_0, \ldots)$; for example,

$N = 3$ gives $(1, 2, 3)$. $(x_i \mid i \in 1)$ gives a one-tuple (x_0), where $(x_0) = x_0$. If t is a k-tuple, t_n, $n \in k$ is the n-th entry of t. $\#(x_0, \ldots)$ returns the arity of the tuple; example: $\#(1, 2, 3) = 3$. $(x_i \mid i \in N) \cup (x_j \mid j \in M)$ denotes tuple concatenation; example: $(1, 2, 3) \cup (4, 5, 6)$ gives $(1, 2, 3, 4, 5, 6)$.

We write

$$x \; as \; (e_i \mid i \in N)$$

for some natural number N to denote an N-tuple x. Further, x is "pattern matched" with the tuple pattern $(e_0, \ldots, e_{N-1})$. e_k denotes the k-th field of x, for $k \in N$.

Here are the operations on lists. $[x_i \mid i \in N]$ denotes an N-list $[x_0, \ldots]$; example: $N = 3$ gives $[1, 2, 3]$. If l is a k-list, l_n, $n \in k$ is the n-th entry of l. $\#[x_0, \ldots]$ returns the length of a list; example: $\#[1, 2, 3] = 3$. $[x_i \mid i \in N] \cup [x_j \mid j \in M]$ concatenates the lists; example: $[1, 2, 3] \cup [4, 5, 6]$ gives $[1, 2, 3, 4, 5, 6]$.

map is a higher order function that applies the function given as its first argument to a list/set/tuple given as its second argument, and returns a list/bag/tuple, respectively. Notice that *bag*s are returned as a result of applying *map* to a set. This permits smoother definitions. Example:

$$map(successor, [1, 2]) = [2, 3].$$

Another example:

$$map(square, \{3, 2\}) = \{9, 4\}.$$

A third example:

$$map(odd, \{3, 2, 1\}) = \{True, True, False\}.$$

We will often omit explicit conversions between lists and sets. Where necessary, however, the conversion functions $list2set$ and $tuple2set$ will be used.

The notation $X[\![\, E/D \,]\!]$, where $E = (e_i \mid i \in k)$ is a k-tuple of expressions and $D = (d_i \mid i \in k)$ is a k-tuple of variables denotes the simultaneous replacement of d_i occurring in X by e_i.

We write $(e_1 \; e_2)$ to denote the application of the function denoted by e_1 on the argument denoted by e_2.

Types are regarded as sets. $v \in \tau$ denotes a value of a type τ; example: $5 \in Nat$; $true \in Bool$; $\mathtt{X} \in Bit$.

Let function $ports(X)$ returns the ports used in X, where X is an expression or a formula. Let $vars(X)$ denote the set of free variables in expression/formula X.

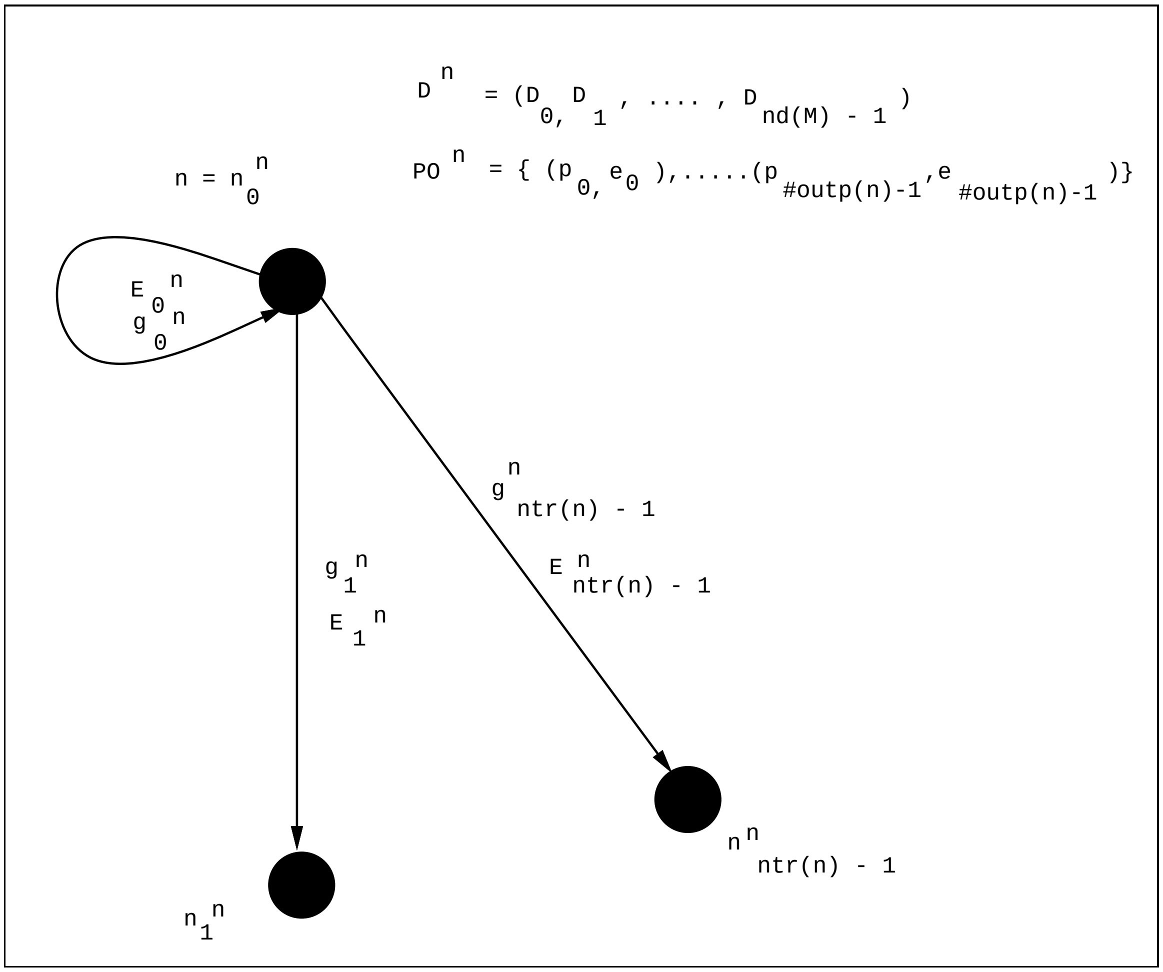
$D^n = (D_{0,} D_1, \ldots, D_{nd(M)-1})$
$PO^n = \{ (p_{0,} e_0), \ldots (p_{\#outp(n)-1}, e_{\#outp(n)-1})\}$
$n = n^n_0$
E^n_0
g^n_0
$g^n_{ntr(n)-1}$
$E^n_{ntr(n)-1}$
g^n_1
E^n_1
$n^n_{ntr(n)-1}$
n^n_1

Figure 1: A Generic State Diagram

3 Behavioral Descriptions

Module: $M = (P, N)$

Figure 1 shows the general nature of the state transition graph of a HOP module. Much of the terminology to be introduced is illustrated in this picture. Though this picture brings out some relationships with classical automata, HOP is an extended symbolic automaton model tailored for the description of synchronous systems viewed as fundamental mode machines. In the rest of this paper, we avoid drawing such pictures because they become unwieldy for large examples.

A hardware module M is as a pair (P, N). P is a possibly empty set of boundary ports of M. A port p in P can serve as an input port during certain cycles, and as an output port during other cycles. The *type* of values carried by a port remains the same throughout. (We assume that the types of objects that ports can carry are bit-vectors of length ≥ 1.)

N is a non-empty finite set of *nodes*[1]of M. By convention, module M starts its execution at node n_0. The nodes in N are connected by transitions (as explained below) and thus forms an "automaton diagram". We introduce selectors P and N to select the components P and N of M. Also, $\mathsf{n}_0(M)$ selects the starting node of M.

Each $n \in N$ is, informally, a tree of height 1 whose root is a symbolic state, and whose leaves are the next-states that are reached through transitions that form the tree edges. Transitions are guarded by first order formulae. Formally, a node $n \in N$ is a lambda expression

$$\lambda D^n.(PO^n,\ \{(g_i^n, n_i^n, E_i^n) \mid i \in ntr(n)\})$$

that takes a tuple of values for the data states D^n and returns a pair $(port\_outputs, set\_of\_moves)$, where the port outputs made at node n are PO^n and the ith move made out of node n is described by (g_i^n, n_i^n, E_i^n) – to be explained below. The lambda expression is *closed* in the sense that all occurrences of variables in its body are bound by D^n and port identifiers that occur in its body belong to P.

The starting (symbolic) state for node n is D^n. D^n is a tuple of variables that denotes the *internal data* state:

$$D^n = (v_i \mid i \in nd(M)),$$

where $nd(M)$ is the number of data path state elements of M—an a priori fixed quantity (*i.e.* the module doesn't, for example, "dynamically allocate memory"). v_i belong to Var, the universe of variables. (Note: We will omit the superscript n, writing D^n, n_i^n, *etc.*, as D, n_i, *etc.*, when the node in question, n, is clear from the context.)

The first action performed by the module while in state D^n is to generate port outputs PO^n. PO^n is a set of *port outputs* (port-expression pairs):

$$PO^n = \{(p_i, e_i) \mid p_i \in P,\ e_i \in EXP^P\}$$

such that if $(p_i, e_i) \in PO$, there exists no *other* $(p_i, e_j) \in PO$. Selectors p and e are used to select the components of a port-expression pair. Define

$$iports(PO^n) = \cup_{i \in \#(PO^n)}\ ports(e_i)\ s.t.\ (p_i, e_i) \in PO^n$$

to be the input ports, the values coming through which are used to generate the port outputs on ports p_i.

EXP^P are expressions defined over ports P:

$$EXP^P ::= Var \mid P \mid f^k(EXP_i^P \mid i \in k)$$

[1]In the sense of a graph node.

where Var denotes the universe of variables, and (according to the notation introduced so far) $f^k(EXP_i^P \mid i \in k)$ represents a k-ary function applied to $k \geq 0$ arguments. An example is $plus(1, x, p)$. This expression evaluates to the sum of 1, the value denoted by variable x, and the value incoming into the module through port p.

There are $ntr(n)$ transitions outgoing from D^n, each of them guarded by guard g_i. Guards g_i belong to the language $Guard^P$ and have the syntax

$$Guards^P ::= BVar \mid BP \mid p^k(EXP_i^P \mid i \in k) \mid b^k(Guards_i^P \mid i \in k)$$

where $BVar$ are Boolean variables ($BVar \subseteq Var$), BP are Boolean ports ($BP \subseteq P$), p^k are first-order predicate constants of various arities $k \geq 0$, and b^k are boolean connectives of various arities $k \geq 0$.

The ith next-state is specified by the pair (n_i^n, E_i^n) thus: n_i^n is the node reached via the ith transition out of n, and E_i^n are the "actual parameter expressions" corresponding to the lambda-abstracted "formal parameters" of node n_i^n. Suppose the actual parameter expressions E_i^n when evaluated yield values V_i^n. Then, after taking the ith transition, the module resumes execution as per $(n_i^n\ V_i^n)$ (the lambda expression n_i^n applied to the value tuple V_i^n). We call a form $(n\ V)$ a *node instance*.

The arities of n_i^n (the number of formal parameters) and E_i^n (the number of actual parameters) are required to be the same. We require that each module have at least one transition going from n back to n; by convention, this is the *zero*th transition. In other words, $ntr(n) > 0$ for any node n, and n_0^n is the same as node n.

We define

$$inp(n) = (\cup_{i \in ntr(n)}\ ports(g_i^n) \cup ports(E_i^n)) \cup iports(PO^n)$$

to be the set of ports that are configured in the input direction while at node n. Also define

$$outp(n) = map\ \mathsf{p}\ PO^n$$

to be the set of ports configured as outputs while at node n.

It is clear that $inp(n) \cap outp(n)$ must be equal to the empty set, for every n. However this is not enough: those ports that are, at the present time (while at node n) used as inputs must *not* be configured to be outputs during the next time (while at any successive node). This restriction arises because of the fundamental mode behavior: external inputs are held stable until the system stabilizes in its next state; the external world, if it chooses to do so, should be allowed to continue to drive the external inputs even after the system attains the next state. In other words, after reaching the next state, the input ports must not suddenly "turn around" and drive the external world which has not yet switched off its drive. So, we

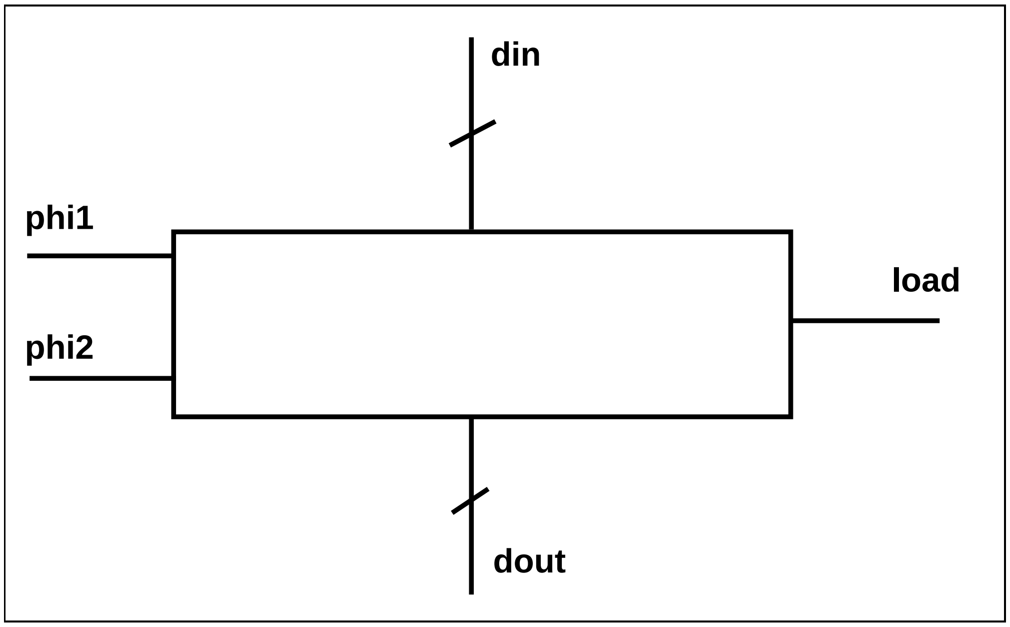

Figure 2: A Modulo Counter 3

modify the above requirement to the following two requirements:

$$inp(n) \cap (outp(n) \cup outp(n_i^n)) = \emptyset \; for\ every\ i \in ntr(n) \tag{1}$$

$$outp(n) \cap (\cup_{i \in ntr(n)} inp(n_i^n)) = \emptyset \tag{2}$$

Condition 1 says that those ports that are configured as inputs in the present state must not be configured as outputs either in the present state or in any of the next states. Condition 2 says that those ports that are configured as outputs in the present state must not be configured as inputs in any of the next states. Notice that we do not require $inp(n) \cup outp(n) = P$ because there may be ports that are neither used as inputs nor as outputs. Some of these ports may be in the process of changing their previously assigned directions; in fact, this is exactly how ports change their directions – they go through a "neutral state" before changing their directions.

3.1 An Example of HOP Specification

The example in figures 2 and 3 specifies a counter operating under a two-phase clocking discipline. The specification is provided in two syntaxes: one corresponding to the above mathematical definitions, and the latter being a readable version of the former. We shall discuss the latter syntax as it is more readable. This syntax is based on familiar programming language syntax instead of the lambda notation.

```
ctr = (P,N)
P   = {phi1, phi2, load, din, dout}
N   = {n0, n1}

n0  = lambda(D0,DOUT0).
      ( (dout,DOUT0) ,
         { (~phi1,                          n0, (D0 ,  DOUT0) ),
           (phi1 /\ ~phi2 /\ load,           n1, (din,  Xdt)   ),
           (phi1 /\ ~phi2 /\ ~load /\ D0<MAX, n1, (D0+1, D0)    ),
           (phi1 /\ ~phi2 /\ ~load /\ D0=MAX, n1, (0,    D0)    )
         }
      )

n1  = lambda(D1,DOUT1).
      ( (dout,DOUT1) ,
         { (~phi2,                          n1, (D1,   DOUT1)  ),
           (~phi1 /\ phi2,                   n0, (D1,   D1)     )
         }
      )
```

```
MODULE ctr(size)                (* macro parameters - can be of any type
                                 - the macro is expanded & then type-checked *)
TYPE
 dT : vector size of bit

PORT
 INPUT  phi1, phi2, load : bit;
 BIDIR  din : dT;               (* Deliberately specified as BIDIR *)
 OUTPUT dout: dT

BEHAVIOR

ctr0(D0,DOUT0); {dout=DOUT0}
       = {  ~phi1                     -> ctr0(D0,DOUT0)
          |  phi1 /\ ~phi2 /\
                {  load               -> ctr1(din,XdT)
                 | ~load /\ {  D0<MAX   -> ctr1(D0+1,D0)
                            | D0=MAX   -> ctr1(0,D0)
                           }
                }
         }

ctr1(D1,DOUT1); {dout=DOUT1}
       = {  ~phi2                     -> ctr1(D1,DOUT1)
          | ~phi1 /\  phi2            -> ctr0(D1,D1)
         }
END ctr
```

Figure 3: A Modulo Counter in Formal and High-level Syntaxes

The first aspect of the high level syntax is that it allows fixed direction ports to be separately specified in the `INPUT` and `OUTPUT` sections, and ports that can change their directions in the `BIDIR` section. Directions are specified in this manner to permit better static checks. The types of ports can also be defined. From now on, we focus on the `BEHAVIOR` section.

The counter begins its operation in control state `ctr0`, in data state `(D0,DOUT0)`, and with its only output port `dout` holding the value `DOUT0`. Control states ("`cs`") are names given to nodes. They are shown as the names of tail-recursive functions, whose formal arguments are the data states ("`ds`"). Port outputs are written as "`port=expression,...`", denoting a set of `port,expression` pairs. In the textual order, these port outputs are specified after the term `cs(ds)` and a semi-colon.

While in state `ctr0(D0,DOUT0)`, so long as guard `~phi1` is true, the state is maintained at `ctr0(D0,DOUT0)`. Both the clock inputs must not be simultaneously made true in state `ctr0`. If the inputs are changed so as to satisfy `phi1/\~phi2/\load` and held steady at that state, the execution reaches control state `ctr1`, with data state `(D1,DOUT1)` set to `(din,Xdt)` where `Xdt` is the value acquired through the input port `din`, and `XdT` is an *unknown value* of type `dT`. (This behavior is shown purely for illustration, and need not correspond to an actual counter.)

In state `ctr1(D1,DOUT1)`, so long as `~phi2` holds, the system stays in the same state, producing the output `DOUT1` on port `dout`. Thus, notice that the condition that brought the system into state `ctr1`—namely `phi1/\~phi2/\Y` where `Y` is some formula, is stronger than the condition necessary to hold the attained state—i.e. only the condition `~phi2` need exist *once* the state `ctr1(D1,DOUT1)` has been attained. This property will be discussed further in section 4.

In this state, the counter is prepared only to recognize one other condition: `~phi1/\phi2`. For example, if both clocks were to be found high, no "move" is defined, indicating that this combination is not meant to be supplied from outside. When the allowed combination (`phi1` low and `phi2` high) occurs, the state `ctr0` is attained. In state `ctr0` notice how the guards `D0<MAX` and `D0=MAX` are used to guide the execution to proceed to two different states.

Guards capture the dependence of control flow on data. Thus, in our symbolically specified "generic automata", the execution is *classified* into a small number of classes (= paths) by the guards, and within each class, the next-state and output functions are symbolically specified using a functional notation, thus specifying many actual executions very compactly.

The above specification of the counter allows the combination `phi2`=1 and `phi1`=0 to immediately follow the combination `phi2`=0 and `phi1`=1. If the clocks have to be forced to go through the combination `phi2`=0 and `phi1`=0 before assuming the combination `phi2`=1 and `phi1`=0, the specification can be suitably modified.

Before we examine other examples, it is appropriate to know about the exact details of execution of HOP processes, and how it corresponds to the fundamental mode execution model. It is also appropriate to know about well-formedness conditions formulated at the HOP syntax level that, in turn, imply the well-formedness of the hardware being modeled. These are now presented.

4 Well-formedness

In this section, we discuss all the well-formedness conditions on HOP modules. The well-formedness conditions explained in section 3 as part of the definition of modules are first repeated below:

WF1: $ntr(n) > 0$ for any node n.

WF2: $n_0^n = n$ for any node n.

WF3: A node is a closed lambda term.

WF4: The arities of n_i^n and E_i^n are equal.

WF5:

$$inp(n) \cap (outp(n) \cup outp(n_i^n)) = \emptyset \ for\ every\ i \in ntr(n)$$
$$outp(n) \cap (\cup_{i \in ntr(n)} inp(n_i^n)) = \emptyset$$

WF6: If $(p_i, e_i) \in PO$, there exists no $(p_i, e_j) \in PO$ such that $e_j \neq e_i$.

Additional well-formedness conditions are now presented and explained:

WF7: $g_i^n \Rightarrow g_i^n [\![E_i^n / D^n]\!]$

In other words, every guard that is true at node n before a transition must remain true after the transition. Or, the transition must not disable the guard.

Consider a guard g that gets disabled by a transition. In terms of circuits, we have a boolean circuit C that evaluates guard g_i^n and produces output on an internal port o. The outputon o is true before the transition and false after the transition. Since the true output on o is used to generate the next state which, when shifted into the "present state" latches causes output o to become false again, certainly an internal combinational loop must have formed. This is being ruled out by insisting that o remain true even after the transition.

It is assumed that if g_i^n is true, then all the transient intermediate states that the data state goes through in the process of changing from D^n to E_i^n also preserve the truth of g_i^n.

Usually the stability of the guards g_i^n is guaranteed by the fact that g_i^n does not use any data state component that changes: *i.e.*, g_i^n does not use D_k^n if $D_k^n \neq (E_i^n)_k$ (*i.e.*, the kth component of the tuple (E_i^n)), for $k \in nd(M)$.

WF8: $g_i^n \Rightarrow g_0^{n_i^n} [\![E_i^n / D^n]\!]$

In other words, if ports in $inp(n)$ are held steadily at values V that make the guard g_i^n true, the system will attain the next-state corresponding to node n_i^n (the ith next-node from node n) and then remain in that state by taking the *zero*th transition from n_i^n back to itself (the "self loop"). Therefore guard $g_0^{n_i^n}$ that guards this self loop will have to be true under the very same inputs, V. If the module cannot remain in state n_i^n under the inputs V, this state proves to be a "transient state" which we do not model as a HOP automaton state.

The implication sign signifies that some of the external inputs may actually turn out to be don't cares *after* the transition, because of the change of state. In short, "the perturbation condition for node n is stronger, or at worst, as strong as the holding condition for the next node attained, n_i^n".

WF9: $\forall i, j.\ i \neq j \Rightarrow \neg(g_i^n \wedge g_j^n)$;

i.e., the guards of transitions going out of a state are mutually exclusive. This means that HOP processes specify deterministic computations.

WF10: $\vee_{i \in ntr(n)}\ g_i^n = true$;

i.e., the guards are exhaustive. Some of the transitions may lead to error states, though. Thus the behavior is totally specified.

The following is a useful (pessimistic) error checking procedure that covers many of the above well-formedness conditions: *every guard must involve at least one input port*; *i.e.*, $ports(g_i^n) \neq \emptyset$. The argument in favor of this check proceeds as follows. Assume g_k^n has no port identifiers occurring in it. There are three cases:

1. $g_k^n = false$: in this case the guard, and the transition it guards can be eliminated.

   The next two cases assume $g_k^n = true$ for some value assignment of D^n.

2. $k = 0$: then, the module, after reaching node n remains stuck in the self-loop going from n back to n. (Recall that if g_0^n is true, no other guard g_i^n can be true.) While a module remains stuck in the self-loop going from n back to n, it can still serve as a combinational unit, because the PO^n component of a node is a set of combinational functions specifying values for output ports $outp(n)$ in terms of input ports $inp(n)$. However, the situation of a module remaining stuck in a state is unusual, and calls for a warning to be issued.

3. $k \neq 0$: the module, after reaching node n can immediately slip away to the next node n_k. However, this would violate WF9 because after reaching node n, by WF8, g_0^n must also be true.

Many of these well-formedness checks are illustrated in section 9.3.2 which may, actually, be read now.

5 Denotational Semantics[2]

Informal Semantics

First we examine the informal semantics of a module. Consider a module $M = (P, N)$ at node instance $(n\ V)$. It provides output values corresponding to PO^n. Suppose the environment applies input values through ports $ports(g_i^n)$ such that g_i^n is made true. Then, the module evaluates E_i^n at the current time, yielding values V_i^n and resumes its behavior at node instance $(n_i^n\ V_i^n)$.

Since, by WF8, the inputs applied are guaranteed to satisfy $g_0^{n_i^n}$, the module will be held in state n_i^n until the next external perturbation comes along.

The "execution" of M thus proceeds in discrete steps. Each such step measures one *tick* of time on an abstract time scale. All the modules in the system use the same time scale because we require a collection of interconnected modules also to respond to one perturbation, attain a steady state collectively, and then only accept the next perturbation.

Time t begins with the tth application of the external perturbation, and lasts through the attainment of steady state by M following the perturbation. A move through g_0^n, for any n, also constitutes one tick.

[2]The term "denotational semantics" is used in a broader sense, to mean "semantics in terms of abstract semantic domains such as sets, relations, etc." - similar to Boute's System Semantics [8].

Formal Semantics

We define the behavior of a module M as its behavior when started at node $\mathsf{n_0}(M)$. We associate with each module M its boundary *waveforms*, W. W is a mapping from a port name p and a time t to the value held by the port at t. $(W\ p)$ will be defined for all those points in time, t, where p is configured as an output port. For the remaining points in time, $(W\ p)$ is to be supplied by the external world, in the form of input values, or is undefined, if the value at this time is of no interest.

We use the valuation relation $\mathcal{M}_m$ for determining the meaning of a module, valuation relation $\mathcal{M}_n$ for the meaning of a node, and valuation function $(\mathcal{M}_0\ W\ t)$ for the basis case of expressions and formulae. The evaluator $\mathcal{M}_0$ takes the waveforms W and time t to become a "specialized evaluator" that can then evaluate expressions E with respect to W and at time t. We indicate this by the application $(\mathcal{M}_0\ W\ t)(E)$. The value tuple V used in the formal argument list of $\mathcal{M}_m$ is the starting state of module M. In the following, let Pred be the meaning of $pred^N$ and Fn be the meaning of f^N.

$$\mathcal{M}_m(M\ as\ (P,N),V,W,t) = \mathcal{M}_n(\mathsf{n_0}(M),V,W,t)$$

$$\mathcal{M}_n(n\ as\ \lambda D.(PO,\ \{(g_i,n_i,E_i)\ \mid\ i \in ntr(n)\}),V,W,t) =$$
$$\textbf{let}\ (PO',\ \{(g_i',n_i',E_i')\ \mid\ i \in ntr(n)\}) = (n\ V)$$
$$\textbf{in}$$
$$\wedge_{(p_i,e_i)\in PO'}\ W(p_i,t) = (\mathcal{M}_0\ W\ t)(e_i)$$
$$\wedge$$
$$\wedge_{i\in ntr(n)}\ (\mathcal{M}_0\ W\ t)(g_i') \Rightarrow \mathcal{M}_n(n_i',(\mathcal{M}_0\ W\ t)(E_i'),W,t+1)$$
$$\textbf{end}$$

$$(\mathcal{M}_0\ W\ t)(\neg literal) = not((\mathcal{M}_0\ W\ t)(literal))$$
$$(\mathcal{M}_0\ W\ t)(pred^N(term_i\ \mid\ i \in N)) = \mathsf{Pred}(map\ (\mathcal{M}_0\ W\ t)\ (term_i\ \mid\ i \in N))$$
$$(\mathcal{M}_0\ W\ t)(literal \wedge fmla) = and((\mathcal{M}_0\ W\ t)(literal),(\mathcal{M}_0\ W\ t)(fmla))$$
$$(\mathcal{M}_0\ W\ t)(f^N(term_i\ \mid\ i \in N)) = \mathsf{Fn}(map\ (\mathcal{M}_0\ W\ t)\ (term_i\ \mid\ i \in N))$$
$$(\mathcal{M}_0\ W\ t)(port) = W(port,t)$$

The meaning of a module M starting at node instance $(\mathsf{n_0}\ V)$ is defined with the help of the $\mathcal{M}_n$ function. The construct

$$\textbf{let}\ (PO',\ \{(g_i', n_i', E_i')\ \mid\ i \in ntr(n)\}) = (n\ V)$$
$$\ldots$$

specifies that $(n\ V)$ is to be pattern-matched with the pattern specified to the left of the equals sign.

The behavior at node n is specified by $\mathcal{M}_n$ in two parts: for every port-expression pair $(p_i, e_i) \in PO'$, the value of e_i is asserted on port p_i. Then, for every transition guarded by g_i', if this guard is true, the meaning of the next state is asserted, again using the $\mathcal{M}_n$ function. Notice that the E_i' are evaluated at time t. Thus, the next state is determined as a function of the *current* state and the *current* inputs.

Finally, the meaning function $(\mathcal{M}_0\ W\ t)$ is specified through structural induction over the expression syntax.

5.1 Adequacy of Well-formedness Conditions

Are the well-formedness checks WF1 through WF10 adequate in an intuitive sense? In this section, we examine this issue, using the notion of a *functional execution sequence* for hardware modules. A functional execution sequence specifies that the next-state, current outputs, and the present port-directions are all functionally determined by the current state and the current inputs. In most formulations, hardware modules denote stream functions [6] where each stream is associated with either an input port or an output port. Since ports retain their directions over time in these models, many of the well-formedness checks we prescribe do not apply in their models. A functional execution sequence is a generalization of the idea of stream functions because different ports are used as inputs and outputs at different time steps.

We show that a module M started at node instance $(\mathsf{n}_0(M)\ V_0)$ denotes the *functional execution sequence*

$$[(n_j, V_j, u_j, v_j)\ \mid\ j \geq 0],$$

if and only if M is well-formed according to WF1 through WF10. Here,

- n_j is a node, V_j are the values bound to D^{n_j}, u_j are the input values fed through the input ports $inp(n_j)$, and v_j are the values produced on the ports $outp(n_j)$.
- $n_0 = \mathsf{n}_0(M)$, $n_{j+1} = f_n(n_j, V_j, u_j)$, $V_{j+1} = f_V(n_j, V_j, u_j)$, and $v_j = f_v(n_j, V_j, u_j)$.

  Functions f_n, f_V, and f_v are (respectively) the *next-node*, *next-state*, and *present-output* functions. These functions are specified by the lambda expression n_j itself, as follows: the PO^{n_j} component of n_j specifies the *present-output* function, and each $(g_i^{n_j}, n_i^{n_j}, E_i^{n_j})$ specifies the *next-node* and *next-state* functions.

- The next set of input and output ports are also functionally determined from the present state and present inputs.
- For $j \geq 0$ if

  $$(n_j, V_j, u_j, v_j), \text{ and}$$
  $$(n_{j+1}, V_{j+1}, u_{j+1}, v_{j+1})$$

  are such that $u_j = u_{j+1}$, then $V_{j+1} = V_{j+2}$ and $n_{j+1} = n_{j+2}$. In other words, if the inputs u_j that cause node n_{j+1} to be reached are held steady ($u_{j+1} = u_j$), the self-loop is taken, holding the node, the data state, and the outputs also steady.
- Further, if $u_{j+1} = u_{j+2}$ then $v_{j+1} = v_{j+2}$ also. This is because v_j is a function f_v of (n_j, V_j, u_j).

The definition of a *functional execution sequence* captures many intuitive properties of synchronous system behavior that are well known. Our remarks about synchronous system behavior are based on the denotational semantics presented earlier. We now provide arguments to support the necessity and sufficience of the well-formedness conditions.

Not Well-formed $\Rightarrow$ Non-functional Execution

If WF1 is violated, there are dead-end nodes, which truncate the execution sequence. If WF2 is violated, *i.e.* if there are no "self-loop" transitions for a node n, a module will be forced to exit n the moment it is entered, thus making n a transient state, even though the same inputs are held. WF3 and WF4 capture the syntactic well-formedness of the lambda expressions. If WF5 is violated, the same port can end up being driven by both the module and the environment, as described in section 4. If WF6 is violated, two different output ports can produce "clashing" values. If WF7 is violated, a state transition can inhibit itself; then, the *next-state* function is not defined. If WF8 is violated, a module cannot stay at the same node despite the same inputs being provided from the external world. If WF9 is violated, the behavior is non-deterministic. If WF10 is violated, the behavior is only partially specified. Thus, if any of WF1 through WF10 are violated, a functional execution sequence will not result.

Well-formed $\Rightarrow$ Functional Execution

If all the well-formedness checks are passed, and a module is started at a node instance $(n\ V)$, its current outputs are determined by the port outputs PO^n functionally, without causing any input and output port values to clash (WF5) or two output port values to clash (WF6); there will be always one transition (WF1) and exactly one true guard (WF9 and WF10) – say the ith guard; the module will evaluate E_i^n and bind the values to n_i^n (WF4), and successfully complete its transition (WF7). While in the new state, if the same inputs are held, the module continues to retain the same node, and data state (WF8). Thus, given that all the well-formedness conditions are satisfied, a module exhibits a functional execution sequence.

6 Structural Descriptions, and Parallel Composition

Two modules M^0 and M^1 interconnected at a subset of their boundary ports forms a new module $par_m(M^0, M^1)$. This function is defined with the help of function par which composes two nodes:

$$
\begin{aligned}
& par_m(M^0, M^1) = (\mathsf{P}(M_0) \cup \mathsf{P}(M_1), N) \\
& where \\
& n^0 \in \mathsf{N}(M_0) \wedge n^1 \in \mathsf{N}(M_1) \Rightarrow par(n_0, n_1) \in N \\
& N\ is\ the\ least\ such\ set
\end{aligned}
$$

$$
\begin{aligned}
& par(n^0\ as\ \lambda D^0.(PO^0, \{(g_i^0, n_i^0, E_i^0)\ \mid\ i \in ntr(n^0)\}), \\
& \qquad n^1\ as\ \lambda D^1.(PO^1, \{(g_j^0, n_j^1, E_j^1)\ \mid\ j \in ntr(n^1)\})) \\
& = \lambda(D^0 \cup D^1). \\
& \quad (\{(p, G(e))\ \mid\ (p, e) \in PO^0 \cup PO^1\}, \\
& \qquad\qquad \{G(g_i^0 \wedge g_j^1), (par(n_i^0, n_j^1), G(E_i^0 \cup E_j^1))\ \mid\ (i, j) \in (ntr(n^0), ntr(n^1))\})
\end{aligned}
$$

$$
\begin{aligned}
& G\ is\ undefined\ if\ clashing(PO^0 \cup PO^1)\ \vee\ cyclic(PO^0 \cup PO^1) \\
& \quad else\ G\ is\ as\ defined\ below \\
& clashing(PO) = (p_i, e_i) \in PO \wedge \exists (p_i, e_j) \in PO\ s.t.\ e_j \neq e_i \\
& cyclic(PO) = \exists (p, e) \in PO\ s.t.\ p \in ports(e)
\end{aligned}
$$

The definition of par_m specifies that the ports are unioned and N is inductively defined in terms of $\mathsf{N}(M_0)$ and $\mathsf{N}(M_1)$ via par. par takes the two nodes n^0 and n^1 and returns the resulting node (denoted by a lambda expression).

The main step performed by par is to determine a function G and then apply G to the expression component e of port-expression pairs, to the conjunction of the individual guards,

$g_i^0 \wedge g_j^1$, and to the concatenation of the actual parameters, $E_i^0 \cup E_j^1$ corresponding to the next-state.

G is defined through the following algorithm:

$$\begin{aligned}
&G = flatten(PO^0 \cup PO^1, \#(PO^0 \cup PO^1)) \\
&flatten(PO\ as\ \{(p_i, e_i)\ \mid\ i \in \#PO\}, j) \\
&\quad = PO,\ if\ j = 0 \\
&\quad = undefined,\ if\ p_{j-1} \in ports(e_{j-1})\ or\ clashing(PO) \\
&\quad = flatten(\{(p_i, e_i[\![\ e_{j-1}/p_{j-1}\ ]\!])\ \mid\ i \in \#PO\}, j-1)
\end{aligned}$$

Algorithm *flatten* successively eliminates each occurrence of port p for which there is a (p, e) pair in PO, from every e such that $(p, e) \in PO$. It terminates in as many steps as there are elements in PO, and returns *undefined* only if *cyclic* or *clashing* are violated.

Applying G to $g_i^0 \wedge g_j^1$ and $E_i^0 \cup E_j^1$ is tantamount to *symbolically propagating port values into the places where these values get used.* This step has been found to be quite valuable in hardware verification, as explained in [20, 31]. Some of the benefits are that the inferred behavioral description becomes intuitively clearer and the errors in the specification (combinational loops for instance) are revealed during *flatten*.

7 *par* Preserves Well-formedness

Given well-formed modules, *par* produces a well-formed module, as the following steps argue:

WF1,WF2,WF3,WF4: Trivially true.

WF5: *par* first of all unites the input ports and the output ports from the individual nodes. Since the individual modules are well-formed, the result of uniting the ports is also well-formed as per WF5. The substitution performed using G only eliminates zero or more input ports from $g_i^0 \wedge g_j^1$ and $E_i^0 \cup E_j^1$, and this cannot violate WF5.

WF6: The check *clashing* done during *par* assures this.

WF7: If the substitution referred to in WF7 were to be $[\![\ (E_i^0 \cup E_j^1)/(D^0 \cup D^1)\ ]\!]$, for the composite module, then the composite module satisfies WF7. However, the actual substitution applied during composition is $[\![\ G(E_i^0 \cup E_j^1)/(D^0 \cup D^1)\ ]\!]$.

As G merely replaces each occurrence of a port in $(E_i^0 \cup E_j^1)$ with the expression associated with this port in PO, substitution using G does not violate WF7. (A port p

occurring in an expression E indicates that any value that may come through p can be substituted for the occurrence of p in E. Thus, the application of G merely specializes the expression $(E_i^0 \cup E_j^1)$.) Note: The check during *par* that PO is not *cyclic* makes sure that G is not *undefined*.

WF8, WF9, WF10: Since they are true for the individual modules, they will end up being true for the composite module, as can be easily shown.

8 The Meaning of *par*

We argue that par_m denotes conjunction; in other words,

$$\begin{aligned}&\mathcal{M}_m(par(M^0, M^1), V^0 \cup V^1, W, t) = \\ &\quad \mathcal{M}_m(M^0\ as\ (P^0, N^0), V^0, W, t)\ \wedge\ \mathcal{M}_m(M^1\ as\ (P^1, N^1), V^1, W, t) \\ &\ \ \textit{where } W \textit{ maps ports over } (P^0 \cup P^1) \textit{ to } (time \mapsto values)\end{aligned}$$

The following lemma will be used in our proof of the above:

$$\wedge_{i \in N}\ (g_i \Rightarrow C_i)\ \equiv\ \vee_{i \in N}\ (g_i \wedge C_i)$$

where $N > 0$ is an index set, and g_i, $i \in N$ are *mutually exclusive* and *exhaustive*, and C_i are any boolean formulae. The proof is straightforward from the definition of implication ($\Rightarrow$) and the conditions *mutually exclusive* and *exhaustive*.

Using this lemma, we can rewrite the denotation of a module (defined in section 5) as given below:

$$\mathcal{M}_m(M\ as\ (P, N), V, W, t) = \mathcal{M}_n(\mathsf{n_0}(M), V, W, t)$$

$$\begin{aligned}&\mathcal{M}_n(n\ as\ \lambda D.(PO,\ \{(g_i, n_i, E_i)\ \mid\ i \in ntr(n)\}), V, W, t) = \\ &\mathbf{let}\ (PO',\ \{(g_i', n_i', E_i')\ \mid\ i \in ntr(n)\}) = (n\ V) \\ &\mathbf{in}\end{aligned}$$

$$\begin{aligned}&\bigvee_{i \in ntr(n)} \\ &\quad (\wedge_{(p_i,e_i) \in PO'}\ W(p_i, t) = (\mathcal{M}_0\ W\ t)(e_i) \\ &\quad \wedge\ (\mathcal{M}_0\ W\ t)(g_i')\end{aligned}$$

$$\wedge\ \mathcal{M}_n(n_i^{'}, (\mathcal{M}_0\ W\ t)(E_i^{'}), W, t+1)$$
$$)$$
end

We actually prove a more general fact: that the conjunction of the meanings of two nodes n^0 and n^1 is equivalent to the meaning of the node $par(n^0, n^1)$. We determine the values of the nodes at (D^0, W, t) and (D^1, W, t) instead of at (V^0, W, t) and (V^1, W, t). We first write the conjunction of the meanings of two nodes n^0 and n^1 (called LHS below), as

$$\begin{aligned}
&\mathcal{M}_n(n^0\ as\ \lambda D^0.(PO^0,\ \{(g_i^0, n_i^0, E_i^0)\ \mid\ i \in ntr(n^0)\}), D^0, W, t)\\
&\wedge\\
&\mathcal{M}_n(n^1\ as\ \lambda D^1.(PO^1,\ \{(g_i^1, n_i^1, E_i^1)\ \mid\ i \in ntr(n^1)\}), D^1, W, t)\\
&=\\
&\ (\bigvee_{i \in ntr(n^0)}\\
&\quad \wedge_{(p_k,e_k)\in PO^0}\ W(p_k, t) = (\mathcal{M}_0\ W\ t)(e_k)\\
&\quad \wedge\ \ (\mathcal{M}_0\ W\ t)(g_i^0)\\
&\quad \wedge\ \ \mathcal{M}_n(n_i^0, (\mathcal{M}_0\ W\ t)(E_i^0), W, t+1))\\
&\bigwedge\\
&\ (\bigvee_{j \in ntr(n^1)}\\
&\quad \wedge_{(p_l,e_l)\in PO^1}\ W(p_l, t) = (\mathcal{M}_0\ W\ t)(e_l)\\
&\quad \wedge\ \ (\mathcal{M}_0\ W\ t)(g_j^1)\\
&\quad \wedge\ \ \mathcal{M}_n(n_i^1, (\mathcal{M}_0\ W\ t)(E_i^1), W, t+1))
\end{aligned}$$

This can be rewritten (by taking the product of the two disjuncts) as

$$\begin{aligned}
&\bigvee_{i \in ntr(n^0),\, j \in ntr(n^1)}\\
&(\\
&\quad \wedge_{(p_k,e_k)\in PO^0}\ W(p_k, t) = (\mathcal{M}_0\ W\ t)(e_k)\\
&\quad \wedge\\
&\quad \wedge_{(p_l,e_l)\in PO^1}\ W(p_l, t) = (\mathcal{M}_0\ W\ t)(e_l) \qquad (3)\\
&\\
&\quad \bigwedge
\end{aligned}$$

$$
\begin{array}{l}
(\mathcal{M}_0\ W\ t)(g_i^0) \\
\wedge \\
(\mathcal{M}_0\ W\ t)(g_j^1)
\end{array}
\qquad (4)
$$

$$\bigwedge$$

$$
\begin{array}{l}
\mathcal{M}_n(n_i^0, (\mathcal{M}_0\ W\ t)(E_i^0), W, t+1) \\
\wedge \\
\mathcal{M}_n(n_i^1, (\mathcal{M}_0\ W\ t)(E_i^1), W, t+1)
\end{array}
\qquad (5)
$$

$$)$$

The meaning of $par(n^0, n^1)$ (called RHS, below) is

$$
\begin{array}{ll}
\mathcal{M}_n(par(n^0, n^1), (D^0 \cup D^1), W, t) = & \\
\bigvee_{(i,j)\in(ntr(n^0), ntr(n^1))} & \\
(& \\
\quad \wedge_{(p,e)\in(PO^0\cup PO^1)} W(p,t) = (\mathcal{M}_0\ W\ t)(G(e)) & (6) \\
\quad \wedge \quad (\mathcal{M}_0\ W\ t)(G(g_i^0 \wedge g_j^1)) & (7) \\
\quad \wedge \quad \mathcal{M}_n(par(n_i^0, n_j^1), G(E_i^0 \cup E_j^1), W, t+1) & (8) \\
) & \\
where & \\
\quad G = flatten(PO^0 \cup PO^1, \#(PO^0 \cup PO^1)) & (9)
\end{array}
$$

We will argue that the pairs of formulae 3, 4, and 5 are (respectively) equivalent to the formulae 6, 7, and 8. Consider the pair of formulae 3, and the formula 6:

$$
\begin{array}{l}
\wedge_{(p_k,e_k)\in PO^0} W(p_k,t) = (\mathcal{M}_0\ W\ t)(e_k) \\
\wedge \\
\wedge_{(p_l,e_l)\in PO^1} W(p_l,t) = (\mathcal{M}_0\ W\ t)(e_l) \qquad -- \ (Same\ as\ 3\ above)
\end{array}
$$

$$\wedge_{(p,e)\in(PO^0\cup PO^1)} W(p,t) = (\mathcal{M}_0\ W\ t)(G(e)) \quad -- \ (Same\ as\ 6\ above)$$

If, for every pair $(p_k, e_k) \in PO^0$ and a pair $(p_l, e_l) \in PO^1$, if $p_k \notin ports(e_l)$ and $p_l \notin ports(e_k)$, then G is the identity function, and then equations 3 and 6 are identical. Suppose

p_l occurs in e_k; then, we have

$$\begin{aligned}
&((\mathcal{M}_0\ W\ t)\ \ e_k)\\
&= \{\ \textit{By the definition of } \mathcal{M}_0\ \}\\
&((\mathcal{M}_0\ W\ t)\ \ e_k[\![\ W(p_l,t)/p_l\ ]\!])\\
&= \{\ \textit{By the definition of } W\ \}\\
&((\mathcal{M}_0\ W\ t)\ \ e_k[\![\ (\mathcal{M}_0\ e_l\ t)/p_l\ ]\!])\\
&= \{\ \textit{By the definition of } \mathcal{M}_0\ \}\\
&((\mathcal{M}_0\ W\ t)\ \ e_k[\![\ e_l/p_l\ ]\!])
\end{aligned}$$

Thus, we can see that the conjunction of definitions for W as captured by equation 3 can be captured by an equivalent definition where ports p_l occurring in e_k can be eliminated by means of the substitution $e_k[\![\ e_l/p_l\ ]\!]$. If we carry this process to the limit, eliminating every such occurrence of a port such as p_l from e_k, the effect would be the same as applying G to expression e, as done by equation 6. Thus, the equations 3 and 6 capture the same effect. A detailed proof based on induction is omitted.

Now consider the proof that the conjunction of the meaning of the guards (equation 4) is equivalent to the meaning of the conjunction of the guards to which function G is applied (equation 7). From the definition of $\mathcal{M}_0$, it is clear that the pair of conjuncts 4 is equivalent to

$$(\mathcal{M}_0\ W\ t)(g_i^0 \wedge g_j^1).$$

Now, notice that the above pair of conjuncts is conjoined with the pair of conjuncts 3; the pair of conjuncts 3 is equivalent to

$$\wedge_{(p,e)\in(PO^0\cup PO^1)}\ W(p,t) = (\mathcal{M}_0\ W\ t)(G(e)).$$

Thus, for every occurrence of a port p in $g_i^0 \wedge g_j^1$, the value of the port from W can be substituted. This is tantamount to applying function G to $g_i^0 \wedge g_j^1$ before its meaning is computed, as the formula 7 indicates.

The arguments for the equivalence between 5 and 8 are similar, and hence are omitted.

9 Illustration of HOP

In this section we specify an assortment of examples, each illustrating a different aspect of the HOP model.

```
MODULE nand()
PORT
  INPUT  in1, in2 : bit;
  OUTPUT out      : bit
FUNCTION
  fun nand(x,y) = not(x andalso y);
BEHAVIOR
  nand(); {out=nand(in1,in2)} = true -> nand()
END nand
```

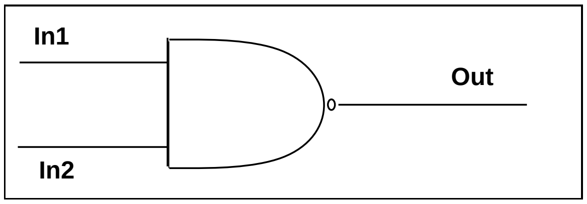

Figure 4: The Specification of a Nand Gate

9.1 The Specification of a Nand Gate

The HOP description of a Nand gate is given in Figure 4. Process `nand` starts in state `nand()` and makes a transition back to itself. Its output `out` is combinationally generated from `in1` and `in2` according to the expression `nand(in1,in2)`.

9.2 Specification of Large Sequential Systems

The specification of traditional gate-style circuits that use components with unidirectional ports, single or multiple phase clocks, and (possibly) large amounts of internal state is not a challenge at all (*e.g.* see [31]). For example, a *least recently used* (LRU) unit can be specified as shown in Figure 5, using the user defined abstract data type "lru type" to model the internal data state, `LS`. Single-phase clocking is assumed.

In state `lru`, when the clock is high and a suitable combination of inputs is applied to trigger an operation, the operation is carried out and the circuit goes to state `lru1`, where it awaits the clock to go low before reverting back to state `lru`. The line

```
| ck /\ least /\ ~reset /\ ~use   -> lru1(lru-use(LS,lru-least(LS)), lru-least(LS))
```

is now explained, and all the other lines are similar. When the combination `ck` and `least`

```
lru(LS,dleast); { leastout = dleast }
=
{ ck /\ ~reset /\ ~use /\ ~least  -> lru1(LS, undefined)
| ck /\ reset /\ ~use /\ ~least   -> lru1(lru-reset(LS), undefined)
| ck /\ use   /\ ~reset /\ ~least -> lru1(lru-use(LS,a), undefined)
| ck /\ least /\ ~reset /\ ~use   -> lru1(lru-use(LS,lru-least(LS)), lru-least(LS))
}

lru1(LS, dleast); { leastout = dleast }
=
{ ck  -> lru1(LS,dleast)
| ~ck -> lru(LS,dleast)
}
```

Figure 5: An LRU unit

are true and the rest of the control inputs are false, the system goes into control state `lru1` with data state modified to `lru-use(LS,lru-least(LS))` (where `lru-use` is a constructor and `lru-least` is an observer on lru type), and the output `least` set to `lru-least(LS)`.

In the rest of this section, we examine how more un-conventional synchronous circuits – circuits that use bidirectional ports, multiphase clocks, *etc.* – are specified using HOP.

9.3 Specifying Transistor/Switch-level Circuits

The HOP model is suitable for modeling many kinds of switch-level circuits. In this section, we discuss how circuits such as a "stack multiplexor" shown in Figure 7 (which is instantiated as SM in Figure 8 – adapted from [43]) can be described. The stack multiplexor multiplexes the data inputs and data outputs of the stack cell described in [43] onto a common wire.

Though transistors can often be viewed as *idealized switches* (having zero resistance, threshold drop, and gate capacitance), many VLSI circuits actually rely, for their correct operation, on transistors having non-zero *on* resistance, threshold drop, gate capacitance, and/or delay. Examples of these circuits are ratioed designs, interlock elements [43, Chapter 7], and dynamic memories. All existing abstract transistor models that are employed in high level simulators or in verification systems are designed to specify only a proper subset of the circuits that can be realized using transistors. Likewise, the model that we use in HOP can only specify a small subset of transistor circuits.

One attractive feature of our transistor model is that it is based on one notion—that of *signal drive* (inspired by [39]). Associated with each port p are two propositional variables:

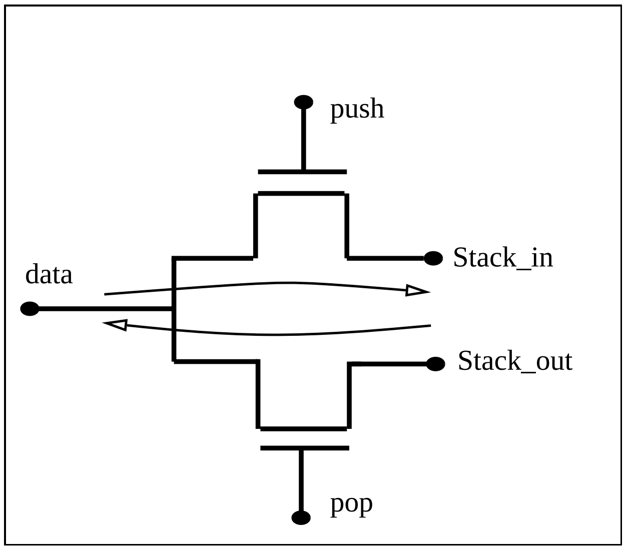

Figure 6: Multiplexor used in a Stack

$p!$, and $p?$ called *drive variables*. In a state, if port p is treated as an output, the propositional variable $p!$ is assumed to be *true*; else it is false. This means: "the module M to which p belongs is driving p as an output in the current state". Variable $p?$ has the interpretation: "the module connected to M via p is driving p". Variable $p?$ is typically used in Boolean guards. It is assumed that the HOP preprocessor will automatically add *drive variables* of the form $p!$ and $p?$ variables to user-given specifications, depending upon which ports are being treated as outputs and which as inputs at each moment (*i.e.* in each state). See Figure 7 for examples.

Drive variables are used to select a direction of conduction for bilateral devices such as pass-transistors. This is illustrated through the specification given in figures 6 and 7

9.3.1 Explanation of the Stack Multiplexor

Process `stack_mux` responds to the `push` command conveyed through the encoding `push/\~pop`, or the `pop` command conveyed through the encoding `~push/\pop`. It decides to treat port `data` either as input or as output, depending on the values on `push` and `pop`, and also how the modules connected to `stack_mux` try to use it. If it finds a drive matching its guard literal `data?`, namely `data!`, coming from the environment, it selects the first branch. The selection of the `pop` branch is similar. When neither the `push` nor the `pop` combination arrives, `stack_mux` essentially performs a 'no operation'. Other combinations will result in an error state being entered. Notice that `stack_mux` changes over to control state `stack_mux1` when it drives `stack_in` and to `stack_mux2` when it drives `data`. This is in accordance with

```
MODULE stack_mux()

PORT INPUT  stack_out, push, pop : bit;
     OUTPUT stack_in : bit;
     BIDIR  data : bit

BEHAVIOR
 stack_mux()
      =      {  push /\ ~pop /\ data?         -> stack_mux1(data)
              | ~push /\ {  pop /\ stack_out? -> stack_mux2(stack_out)
                          | ~pop              -> stack_mux()
                         }
             }
 stack_mux1(D); {stack_in = D, stack_in! = 1} (* stack_in! added by preprocessor *)

      =      {  push /\ ~pop /\ data?         -> stack_mux1(data)
              | ~push /\ ~pop                 -> stack_mux()
             }
 stack_mux2(D); {data = D, data! = 1} (* data! added by preprocessor *)

      =         ~push /\ {  pop /\ stack_out? -> stack_mux2(stack_out)
                          | ~pop              -> stack_mux()
                         }
END stack_mux
```

Figure 7: Illustration of “Drive”s on the Stack Multiplexor

```
A stack composed of SM, the stack multiplexor, two instances of SC,
the stack cell, and SC, the stack controller.
```

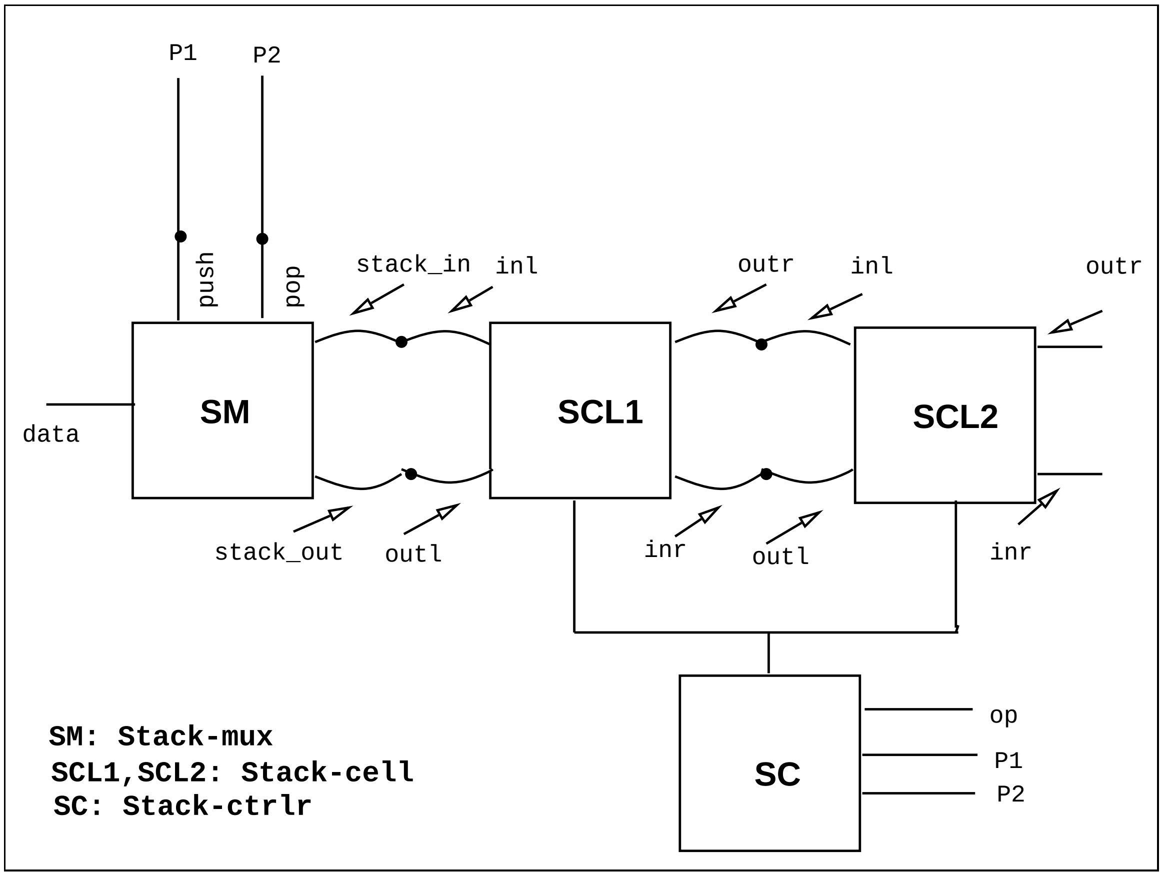

Figure 8: A Stack

```
 stack_mux()
      =      {  push /\ ~pop /\ data?        -> stack_mux1(data)
              | ~push /\ {  pop /\ stack_out? -> stack_mux2(stack_out)
                          | ~pop             -> stack_mux()
                         }
             }
 stack_mux1(D); {stack_in = D, stack_in! = 1} (* stack_in! added by preprocessor *)

      =      {  push /\ ~pop /\ data?        -> stack_mux1(data)
              | ~push /\ {  pop /\ stack_out? -> stack_mux2(stack_out)
                          | ~pop             -> stack_mux()
                         }
             }
 stack_mux2(D); {data = D, data! = 1} (* data! added by preprocessor *)

      =      {  push /\ ~pop /\ data?        -> stack_mux1(data)
              | ~push /\ {  pop /\ stack_out? -> stack_mux2(stack_out)
                          | ~pop             -> stack_mux()
                         }
             }
END stack_mux
```

Figure 9: A Specification of the Stack Multiplexor Violating Well-formedness

our convention that the set of ports configured to be outputs is determined by the state.

9.3.2 Illustration of Well-formedness Conditions

We can illustrate some of the well-formedness conditions on this example. Suppose the specification of the "stack multiplexor" were (incorrectly) written as shown in Figure 9. Consider the first guard in state `stack_mux2`. In this guard, port `data` is used as an input. However, since `data` is used as an output in state `stack_mux2`, the first clause of WF5 is violated.

While in state `stack_mux1`, port `data` is being treated as an input. Now consider the second guard in state `stack_mux1`, which is `~push/\pop/\stack_out?`. It makes the system go to state `stack_mux2` where the system drives port `data`. This again violates the first clause of WF5 because an input port "turns around" and drives the external world before the external world had a chance to turn off *its* drive. If we remove the choice of a move through `~push/\pop/\stack_out?`, we force the system to make a transition from `push/\~pop/\...` to `~push/\~pop/\...` which is a "neutral state"; changing over from `push/\~pop/\...` to `~push/\pop/\...` can be seen to be in violation of WF5 – in fact, the requirement of *non-overlap* between `push` and `pop` is what is getting violated!

The specification in Figure 7 satisfies all the well-formedness checks.

9.3.3 Preprocessings Before *par*

Before we can compose two modules M_0 and M_1 using *par*, where M_0 and M_1 have drive variables, the drive variables must be preprocessed as described below, to take into account their intended meaning. The desired semantics of $p?$ is to specify that p must be driven from outside. Thus, if $p \in \mathsf{P}(M_0)$ and $q \in \mathsf{P}(M_1)$ are connected (by being renamed to a common name $r_0(p) = r = r_1(q)$, and if this connection is *not exported* from $par(M_0, M_1)$, we know that $p?$ means $q!$, and $q?$ means $p!$. Thus, for every such use of $p?$, we substitute $q!$, and for every such use of $q?$, we substitute $p!$.

If, however, p and q are connected, *i.e.*, if $r_0(p) = r = r_1(q)$, and if r *is exported* from $par(M_0, M_1)$, then $p?$ means $q! \vee r?$, and likewise $q?$ means $p! \vee r?$, meaning that p can be driven from "outside" by either M_1 driving p via q, or the external world driving p via r. These preprocessings are also performed.

Occurrences of port outputs of the form "$p!$" are also subject to preprocessing. If p and q are connected and the connection is not exported, every occurrence of $p! = 1$ in M_0 is replaced with two port assertions, $p! = 1$ and $q! = 0$ – this says "since p is being driven by M_0, q may not be driven by M_1". Likewise, every occurrence of $q! = 1$ in M_1 is replaced with $q! = 1$ and $p! = 0$. On the other hand, if the connection is exported via r, $r! = 1$ is also asserted. The purpose of asserting $q! = 0$ in M_0 and $p! = 0$ in M_1 is to check for the consistency of drives during *par*: each node must be subject to at most one drive; hence we are adding extra drive assertions that prevent the "other port" from driving the same node. After performing the above transformations, *par* can be invoked as described before.

9.4 The Stack Cell, SC

The operations of the stack cell are described in Figure 10.

The additions by the preprocessor are `outl!  = 1` and `outr!  = 1`. Also, wherever ports "`p`" are used, the drive variables "`p?`" will be added by the preprocessor in the guard expression. They are not shown for brevity. For example, the line

```
(* do push *) | shr /\ ~trl /\ ~shl /\ ~trr -> scl(inl,S2)
```

would actually look like

```
(* do push *) | shr /\ shr? /\ ~trl /\ trl? /\ ~shl /\ shl?
                   /\ ~trr /\ trr? /\ inl?               -> scl(inl,S2)
```

```
scl(S1,S2); {outl = ~S1, outr = ~S2, outl! = 1, outr! = 1}
{
(* isolate *)  ~shr /\ ~trl /\ ~shl /\ ~trr -> scl(S1,S2)

(* do push *) | shr /\ ~trl /\ ~shl /\ ~trr -> scl(inl,S2)

(*replenish*) | ~shr /\ ~trl /\ ~shl /\ trr -> scl(S1,inv(S1))

(* do pop  *) | ~shr /\ ~trl /\ shl /\ ~trr -> scl(S1,inr)

(*replenish*) | ~shr /\ trl /\ ~shl /\ ~trr -> scl(inv(S2),S2)
}
```

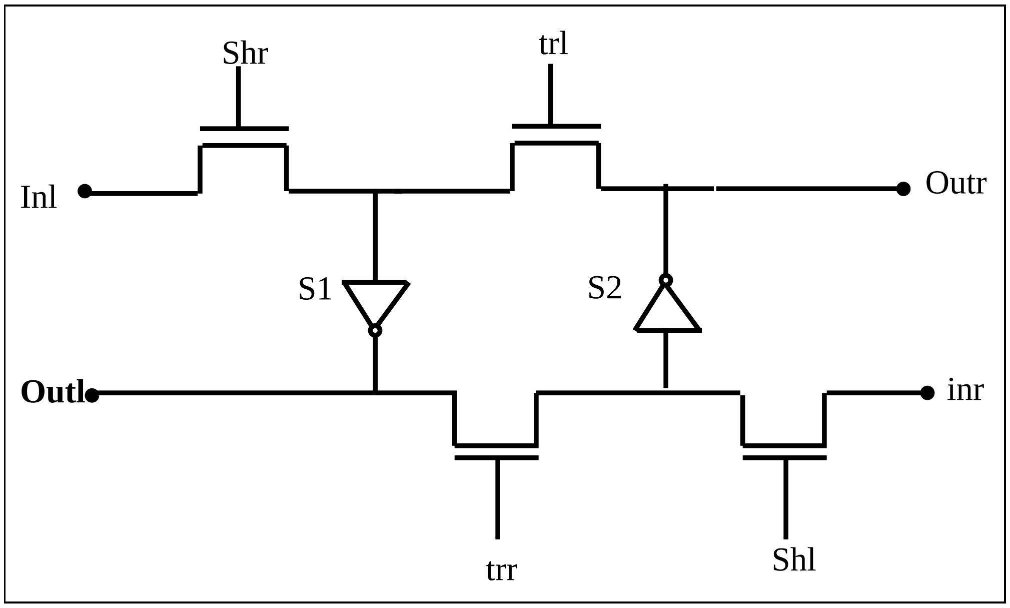

Figure 10: A Stack Cell

```
sc0(s1,s2, dtrr,dshl,dtrl,dshr); {trr=dtrr,shl=dshl,trl=dtrl,shr=dshr}
 =
  { ~p1 /\ ~p2 -> sc0(s1,s2, 0,0,0,0)
   | p1 /\ ~p2 -> sc1(op,s2, 0,0,~s2,s2)
  }

sc1(s1,s2, dtrr,dshl,dtrl,dshr); {trr=dtrr,shl=dshl,trl=dtrl,shr=dshr}
 =
  {  ~p1 /\ ~p2 -> sc1(s1,s2, 0,0,0,0)
   | ~p1 /\ p2  -> sc0(s1,op, ~s1,s2,0,0)
  }
```

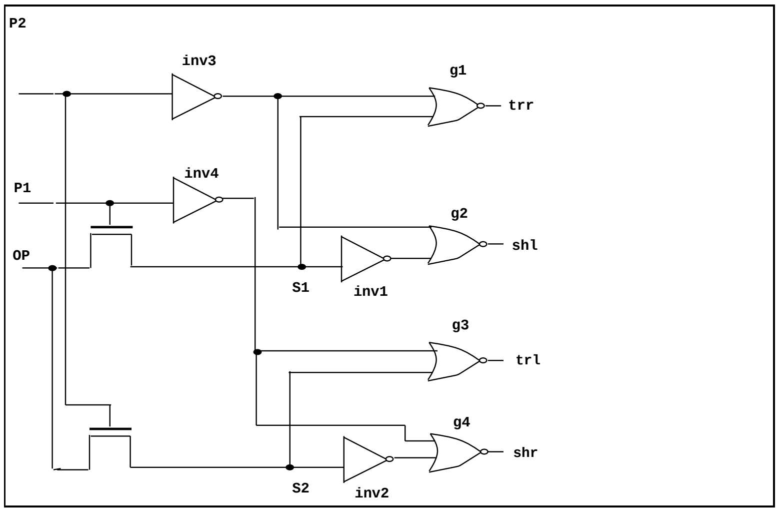

Figure 11: A Stack Controller

9.5 The Stack Controller, SC

The last component type used in the stack, the stack controller, is described in Figure 11. The controller is driven by a two-phase clock brought in through the inputs p1 and p2. Depending on the clock combinations and the starting state of the system (the values of s1,s2, dtrr,dshl,dtrl,dshr), the stack controller generates the sequence described in [43, Page 73].

9.6 A Dynamic Multiphase Clocked Circuit

Interactions among multiphase clocked systems as well as dynamic storage are illustrated on the example in Figure 12. Shown in this figure are two 'half-latches' that are typically

```
(* First specify the type latch; then create two instances l1 and l2 *)
(* Instance creation will be shown later *)
MODULE latch
PORT
 INPUT  in, phi
 OUTPUT out

BEHAVIOR
 latch(D); {out=D}
        = {  phi  /\ {  in?  -> latch(~in)
                      | ~in? -> latch(D)
                     }
           | ~phi -> latch(D)
          }
END latch
```

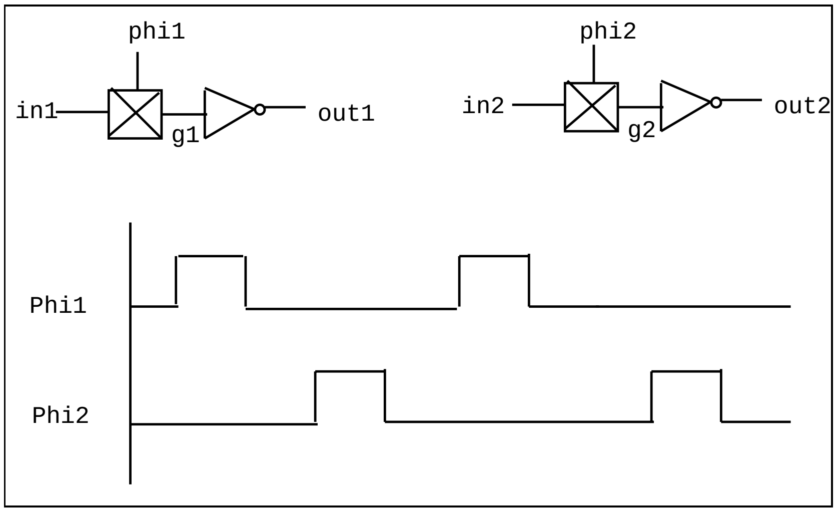

Figure 12: Multiphase interactions

cascaded to form a full latch. We will describe module `latch` that describes how one of these half-latches works. We omit the subscripts `1` and `2` when presenting `latch`.

Process `latch` starts in state `latch(D)`, with its output having value `D`, its present state. If `phi` is found true, and if `in?` (`in` is driven from outside), the next steady state is one where `out=~in`; else if `~in` or `~phi`, the old state `D` is held `out`. Notice that the inverter delay is not captured. This is because the system is assumed to attain steady state—however long it takes—before the next excitation is applied. If `in` is not driven from outside (`~?in`) then the charge on the gate node is retained, as evidenced by the execution going back to state `latch(D)`.

If `l1` and `l2` are cascaded, and the clock sequence `~phi/\~phi2`, `phi/\~phi2`, `~phi1/\~phi2`, `~phi1/\phi2`, ..., is applied, l1 and l2 respond to clock wires `cw` of their own concern (`phi1` and `phi2`, respectively). For example, while `l2` is making a `phi2` move, `l1` would be making a `~phi1` move. In this manner, in a multi-phase clocked system described in HOP, a subset of the modules can be making 'useful' moves, whereas the others are making 'useless' (idling, or "self loop") moves.

10 Summary, and Conclusions

We have described the HOP automaton model and presented its syntax directed semantics. We have attempted to clarify many of the notions underlying formal modeling of synchronous hardware, and make a coherent presentation of the formal semantics of synchronous hardware, stating well-formedness conditions, as well as the adequacy thereof. This work can open up new opportunities to build synchronous hardware design and debugging tools that check for as many errors as possible at the syntactic level, infer the behavior of a structural description, and assist in the formal verification of system descriptions.

HOP was first implemented using Common Lisp and FROBS [21]. *par* was realized as an algorithm PARCOMP that has since been used with great success in debugging several designs (see [31] for example). Earlier versions of PARCOMP suffered from the following problems: (1) the simplification of guard expressions was *ad hoc*; (2) well-formedness checks were not incorporated into PARCOMP; (3) the implementation was based on a notion of global clocking. We plan to re-implement PARCOMP based on the new HOP model. A related implementation effort [33] proposes the use of extended BDD models to make synchronous system verification more efficient. These ideas will be incorporated into the HOP system.

Acknowledgements: Helpful discussions with Professor Eduard Cerny of the University of Montreal and Professor Steven Johnson of the University of Indiana are gratefully acknowl-

edged.

References

1. Zvi Kohavi. *Switching and Automata Theory.* Tata McGraw Hill, 1978.

2. David Ku and Giovanni De Micheli. HardwareC - A Language for Hardware Design, Version 2.0. Technical Report CSL-TR-90-419, Computer Science Laboratory, Stanford University, April 1990.

3. Mario R. Barbacci. Instruction set processor specifications (isps): The notation and its applications. *IEEE Transactions on Computers*, C-30(1):24–40, January 1981.

4. Vhdl language reference manual, August 1985. *Intermetrics Report IR-MD-045-2; See also IEEE Design and Test, April 1986.*

5. Donald E. Thomas and Philip Moorby. *The Verilog Hardware Description Language.* Kluwer Academic Publishers, 1991. ISBN 0-7923-9126-8.

6. Steven D. Johnson. *Synthesis of Digital Designs from Recursion Equations.* The MIT Press, 1984. An ACM Distinguished Dissertation-1983.

7. Mary Sheeran. mufp, a language for vlsi design. In *Proceedings of the ACM Symposium on Lisp and Functional Programming*, pages 104–112, 1984.

8. Raymond Boute. System semantics: Principles, applications, and implementation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 10(1):118–155, January 1988.

9. P. Caspi, D.Pilaud, N.Halbwachs, and J.A.Plaice. LUSTRE: A declarative language for programming synchronous systems. In *Proceedings of the 14th Annual Symposium on Principles of Programming Languages*, pages 178–188. ACM, 1987.

10. Gerard Berry and Laurent Cousserat. The ESTEREL synchronous programming language and its mathematical semantics. In S.D.Brookes, A.W.Roscoe, and G.Winskel, editors, *Seminar on Concurrency, LNCS 197*, pages 389–448. Springer-Verlag, 1984.

11. C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, Englewood Cliffs, New Jersey, 1985.

12. Occam programming manual, 1983.

13. Vincenza Carchiolo, Alberto Faro, Orazio Mirabella, Giuseppe Pappalardo, and Giuseppe Scollo. A LOTOS specification of the PROWAY highway service. *IEEE Transactions on Computers*, C-35(11):949–968, November 1986.

14. Michael Gordon. HOL: A proof generating system for Higher Order Logic. In Graham Birtwistle and P.A.Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 73–128. Kluwer Academic Publishers, Boston, 1988. ISBN-0-89838-246-7.

15. Boyer and Moore. *A Computational Logic.* Academic Press, 1979.

16. Warren A. Hunt Jr. The mechanical verification of a microprocessor design. In D. Borrione, editor, *From HDL Descriptions to Guaranted Correct Circuit Designs.* Elsevier Science Publishers B.V. (North Holland), 1987. (Proc of the IFIP WG 10.2 Working Conference with the same title.).

17. I.S.Dhingra. Formal verification of a design style. In Graham Birtwistle and P.A.Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 293–321. Kluwer Academic Publishers, Boston, 1988. ISBN-0-89838-246-7.

18. Michael J.C. Gordon. Mechanizing programming logics in higher order logic. In G.Birtwistle and P.A.Subrahmanyam, editors, *1988 Banff Hardware Verification Workshop, Banff, June 1988*, 1988. *Invited Paper, to appear as a chapter in a forthcoming Springer-Verlag book.*

19. Albert John Camilleri. Mechanizing the failures model of csp in hol. *IEEE Transactions on Software Engineering*, (9), September 1990.

20. Ganesh C. Gopalakrishnan, Richard Fujimoto, Venkatesh Akella, and Narayana Mani. HOP: A process model for synchronous hardware. semantics, and experiments in process composition. *Integration: The VLSI Journal*, pages 209–247, August 1989.

21. Narayana Mani. Behavioral simulation from high level specifications. Master's thesis, Dept. of Computer Science, University of Utah, Salt Lake City, UT 84112, May 1989.

22. Prabhat Jain, Ganesh Gopalakrishnan, and Prabhakar Kudva. Verification of regular arrays by symbolic simulation. Technical Report UUCS-TR-91-022, University of Utah, Department of Computer Science, October 1991. *Submitted to the Brown/MIT Advanced VLSI Workshop.*

23. Prabhakar Kudva. PCA: An algorithm for the Parallel Composition of regular Arrays, 1991. *Implementation in Standard ML, plus software documentation.*

24. Ganesh C. Gopalakrishnan, Richard M. Fujimoto, Venkatesh Akella, N.S. Mani, and Kevin N. Smith. Specification-driven design of custom architectures in hop. In G.Birtwistle and P.A.Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, chapter 3, pages 128–170. Springer-Verlag, 1989.

25. Ganesh C. Gopalakrishnan, Narayana Mani, and Venkatesh Akella. A design validation system for synchronous hardware based on a process model: A case study. In *Proceedings of the IMEC-IFIP Workshop on Applied Formal Methods for Correct VLSI Design, Leuven, Belgium*, pages 721–740, November 1989.

26. Ganesh C. Gopalakrishnan. Specification and verification of pipelined hardware in HOP. In *Proc. Ninth International Symposium on Computer Hardware Description Languages*, pages 117–131, 1989.

27. Ganesh C. Gopalakrishnan, Narayana S. Mani, and Venkatesh Akella. Parallel composition of lock-step synchronous processes for hardware validation: Divide-and-conquer composition. In J.Sifakis, editor, *Springer Verlag Lecture Notes in Computer Science, No.407 (Automatic Verification Methods for Finite-State Systems, International Workshop, Grenoble, France, June 1989)*, pages 374–382. Springer-Verlag, 1989.

28. Ganesh Gopalakrishnan, Prabhat Jain, Venkatesh Akella, Luli Josephson, and Wen-Yan Kuo. Combining verification and simulation. In Carlo Sequin, editor, *Advanced Research in VLSI : Proceedings of the 1991 University of California Santa Cruz Conference.* The MIT Press, 1991. *ISBN 0-262-19308-6.*

29. Venkatesh Akella and Ganesh Gopalakrishnan. High level test generation via process composition. In *Designing Correct Circuits, Oxford, 1990*, pages 99–119. Springer Verlag, 1990. *Proceedings of the DCC Workshop, Oxford, September, 1990, published in Springer's new series 'Workshops in Computing'.*

30. Prabhat Jain and Ganesh Gopalakrishnan. Towards making symbolic simulation based verification efficient, November 1991. *Submitted to the IEEE Transactions on CAD. An earlier version has been submitted to the Design Automation Conference, 1992.*

31. Ganesh Gopalakrishnan and Richard Fujimoto. Design and verification of the rollback chip using hop: A case study of formal methods applied to hardware design. Technical Report UU-CS-TR-91-015, University of Utah, Department of Computer Science, 1991. *Submitted to the ACM Transactions on Computer Systems.*

32. Wayne Wolf, Andres Takach, and Tien-Chien Lee. Architectural optimization methods for control-dominated machines. In *High-Level VLSI Synthesis*, chapter 10. Kluwer Academic Press, 1991.

33. Michel Langevin and Eduard Cerny. Comparing generic state machines. In *Proc. of the Workshop on Computer-Aided Verification, Aalborg, Denmark*. July 1991.

34. Ghislaine Thuau and Daniel Pilaud. Using the declarative language lustre for circuit verification. In *Designing Correct Circuits, Oxford, 1990*, pages 313–331. Springer Verlag, 1990. *Proceedings of the DCC Workshop, Oxford, September, 1990, published in Springer's new series 'Workshops in Computing'.*

35. Michael Gordon. Register transfer systems and their behavior. In *Proc. of the 5th International Conference on Computer Hardware Description Languages*, 1981.

36. Randal E. Bryant and Carl-Johan Seger. Formal verification of digital circuits using ternary system models. Technical Report CMU-CS-90-131, School of Computer Science, Carnegie Mellon University, May 1990. *Also in the Proceedings of the Workshop on Computer-Aided Verification*, Rutgers University, June, 1990.

37. Carl-Johan Seger and Jeffrey Joyce. A two-level formal verification methodology using HOL and COSMOS. Technical Report 91-10, Dept. of Computer Science, University of British Columbia, Vancouver, B.C., June 1991.

38. Randal E. Bryant, Derek L. Beatty, Karl Brace, Kyeongsoon Cho, and T. Sheffler. Cosmos: A compiled simulator for mos circuits. In *Proc. ACM/IEEE 24th Design Automation Conference*, pages 9–16, June 1987.

39. Zhou Chaochen and C.A.R. Hoare. A model for synchronous switching circuits and its theory of correctness, 1990. *Proceedings of the DCC Workshop, Oxford, September, 1990, published in Springer's new series 'Workshops in Computing'.*

40. Glynn Winskel. A compositional model of mos circuits. In Graham Birtwistle and P.A.Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 323–348. Kluwer Academic Publishers, Boston, 1988. ISBN-0-89838-246-7.

41. David Musser, Paliath Narendran, and William Premerlani. Bids: A method for specifying and verifying bidirectional hardware. In Graham Birtwistle and P.A.Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 217–233. Kluwer Academic Publishers, Boston, 1988. ISBN-0-89838-246-7.

42. John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, 1979.

43. C. A. Mead and L. Conway. *An Introduction to VLSI Systems*. Addison Wesley, 1980.