INDIANA UNIVERSITY
COMPUTER SCIENCE DEPARTMENT

TECHNICAL REPORT NO. 375

Toward a Basis for Protocol Specification and Process Decomposition

Kamlesh Rath and Steven D. Johnson

JUNE 1993

To appear in the proceedings of the *1993 IFIP Conference on Hardware Description Languages and their Applications* (*CHDL '93*), Ottawa, Canada, April, 1993, Elsevier.

Toward a Basis for Protocol Specification and Process Decomposition †

Kamlesh Rath* and Steven D. Johnson

Computer Science Dept., Indiana University, Bloomington, Indiana, USA.
*Email: rathk@cs.indiana.edu

Abstract

In a formalism of top-down design, we consider the decomposition of behavioral specifications into interacting sequential components. The higher level of description specifies the operations to be performed in a major computation step. The goal is to incorporate a given interface specification in a lower-level specification that accounts for interactions with and among sequential components. This construction generalizes the earlier formalism of *system factorization* [14] to include interface protocols. It expands on the objectives of *high-level synthesis* by considering control-synchronization loops in scheduling. This paper presents a specification language for sequential process interaction and develops an interpretation based on finite-state-machines. Operations of minimization, composition and complementation are defined; the last of these being the key to top-down decomposition. A small example is used to illustrate the ideas.

Keyword Codes: B.4.3; B.4.4; F.3.1
Keywords: Input/Output and Data Communications, Interconnections (subsystems); Performance Analysis and Design Aids; Specifying and Verifying and Reasoning about Programs

1. Introduction

Design derivation is a branch of formal verification that deals with "correct by construction" reasoning. [12, 14, 15, 13]. A system of equivalence preserving transformations are used to derive an implementation from a specification. We can view such a derivation as a formal proof reflecting a top-down reasoning style. In this respect it should not be viewed as an alternative for deductive (i.e., conventional theorem-prover based) verification but as an alternate mode of reasoning in design [16, 26]. We can also view derivation as a formalization of synthesis, but as a formalization it is more centrally concerned with correctness in reasoning than with automated design.

A specification can have many implementations and a particular derivation chooses one.

†This research was supported in part by the National Science Foundation under grants numbered MIP 8921842 and MIP 9208745.

Hence, the transformations themselves add information to the (accumulating) implementation. If we interpret terms structurally, then rewriting

$$ab + ac \implies a(b + c)$$

selects an implementation with two operations rather than three. We have used the same kind of algebra at a higher level to deal with data abstraction in structural descriptions. A general transformation called *system factorization* [14] is applied to encapsulate abstract values and operations as simple sequential processes.

These transformations are strong enough to replace abstract memory *values* in a design, with black-box SRAMs, for example [14], but they need to be further generalized to replace memories with DRAMs. Such a transformation would have to incorporate the read/write protocol and account for refresh cycles. Another example of the problem is described in [16], where the derivation of a microprocessor implementation required moving a naive functional model of memory to a process model with indefinite *wait* states.

We think that this is also one of the key problems in raising synthesis to the system level. In a related discussion of *interface specification*, Boriello points out that, "the interface component has received limited attention even though it is crucial to integrating the circuit into an environment that will put it to use" [1]. However, while Boriello develops external interface specifications as a means to guide synthesis, our goal is to use them to guide design decomposition. *Both* sides of the protocol are involved in factoring nontrivial sequential components.

Most formal treatments of sequential decomposition are bottom-up in the sense that they are oriented toward post-design verification. This would include most of the recent research in finite-state machine verification [3]; extensions of FSM models (e.g. [25, 24]) and Petri net theories (e.g. [2]); and model-theoretic work involving process formalisms (e.g. [20, 9]). It is typical that an area of verification research would have this orientation, and also that the top-down view would be better represented in synthesis research. In addition to Boriello's work (cited above), approaches to scheduling by Ku, Micheli [17] and Nestor et.al [22] have considered protocol-like constraints. Dill et.al have used a Büchi automata based model to verify safety and liveness properties using language containment [7]. McMillan and Dill have also modelled timing constraints as min/max constraints and used a generalized branch and bound algorithm to verify the timing specification of connected components [19]. Drusinsky and Harel have used state-charts for hierarchical description for hardware and synthesis of component machines [8]. Holzmann formulated search heuristics to reduce the search space and time for validation of communication protocols [10]. Kurshan uses L-automata with language and process homomorphism [18] to verify reactive systems by stepwise reduction and refinement. He uses a bottom-up model with registers and controllers as processes at the lowest level and constructs complex systems by composing them.

Davie [4] takes a top-down approach to design using verification between specification and implementation steps in CIRCAL [21]. Design partitioning is done by description of components and composing them for verification with respect to the specification. Contextual constraints, restrictions imposed by a device on its environment, are introduced to write partial specifications of a component's environment. The constraints are also used to restrict the target architecture to reduce the complexity of verification [5]. A CIRCAL

based transformation to partition a design is mentioned. The designer specifies a component and an algorithm is used to generate the specification of the other component(s) in the design. Davie suggests that this transformation is of "limited usefulness" due to restrictions on it in their formalism. This is similar to our transformational approach in which the complement operation constructs the environment in a finite state machine based formalism. Our complement operation has no restrictions and is central to our decomposition exercise.

This paper presents a language for protocol specification that can be used to describe the interaction of a process with its environment, orthogonal to its functional behavior. Finite state machines are used as the semantic model for the language. Minimization, composition and complementation operations, and implementation and equivalence relations are defined on the machines.

A factorial machine is used as an example to illustrate our approach to process decomposition. We start with a state machine description of the factorial machine (Figure 6). A multiplication procedure in the factorial machine is factored out of the description and a sequential multiplier (Figure 2) is used. An implementation of the complement of the multiplier machine is incorporated into the factorial machine. The multipler machine and the new factorial machine form the decomposed components of the original factorial machine, which can then be synthesized independently.

State space explosion is often a problem in bottom-up verification methods using finite state machine models. This occurs in the construction of the product machine. Our top-down approach avoids state space explosion using decomposition. We promote the use of top-down design as an alternative to the bottom-up techniques which require the designer to model a system as a network of processes with predefined interactions. We start from an initial abstract description of the system and add the interactions between components to the description in decomposition steps. Also, our composition algorithm only creates reachable states and transitions starting inductively from the start state, thereby avoiding state space explosion.

2. A Language for Protocol Specification

The word "protocol" has been used in many contexts by different authors. In this paper, "protocol" refers to the synchronous interaction of a process with its environment. The protocol specification of a process has two components, data interaction and control interaction. Data interactions occur over input/output data ports and control interactions occur over input/output control ports.

The language described here can be used to specify the input/output behavior of a sequential machine in its environment. The temporal ordering of input/output actions on the control and data ports is specified with reference to a synchronization signal that is used by the machine and its environment.

We begin with an informal discussion of the syntax and give a semantics in sections 3, and 4.

2.1. Syntax

The syntax of the language for interface specification is defined over the following sets: Input control ports (CI), Output control ports (CO), Input data ports (DI), Output data ports (DO) and Data values (V). As a convention, output ports have an "overline", which is not to be confused with boolean negation. The control ports are of type Boolean and can range over values TRUE (1), FALSE (0), Don't Care (#). The data ports can range over a set of values (V) which includes the value Don't Care (#).

An informal discussion about actions and expressions in the interface language is given below. We begin with actions that occur in a single step.

1. A data action of the form x/v denotes the data port x and its associated value v. A control action p means the control port p has a value TRUE associated with it, and a control action $not(q)$ denotes a control port q with a value FALSE associated with it.

2. Multiple control actions $(p, q, ..., r)$ and multiple data actions $(x/v_1, y/v_2, ..., z/v_3)$ can occur simultaneously.

3. Atomic actions are of the form $(x : y)$, where x is a set of control actions and y is a set of data actions. Control actions can occur as guards for data actions. An *input action* consists of a set of input control actions together with any set of data actions. Similarly an *output action* consists of data and output control actions.

4. The actions described above occur in a single step. The following actions span one or more steps.

   (a) (*compute* $\overline{y}$) means output action $\overline{y}$ is performed after some steps.

   (b) (*await* y) means a wait loop for input action y.

   (c) (x *before* $\overline{y}$), means that the input action x occurs in one or more consecutive steps before the output action $\overline{y}$ is performed.

   (d) ($\overline{x}$ *until* y), means output action $\overline{x}$ is performed until input action y occurs.

5. Compound actions of the form (A, A) can be formed by combining actions described in 3, 4 above. This can be used to specify independent input and output actions in the same step.

6. The expression $(e_1; e_2)$ means that the set of actions e_1 is followed by the set of actions e_2. The unary operator $(;)$ in the expression $(; e)$ is used to denote the initial state of the machine.

7. The expression $(e)^*$ means the finite repetition of the sequence of actions e.

8. The expression $(e_1 [] e_2)$ means the choice of actions e_1 or e_2.

The precedence of the operators in descending order is :

/ , : { *before await until compute* } ; []

$$
\begin{array}{lll}
\text{Action} & A ::= C : D \mid A, A & \text{where} \\
& \mid \mathit{compute}\ A_o \mid \mathit{await}\ A_i & C = c_1, .., c_m \\
& \mid A_o\ \mathit{until}\ A_i \mid A_i\ \mathit{before}\ A_o & c_j \in \{c, \mathit{not}(c) \mid c \in CI \cup CO\} \\
& & D = d_1/v_1, .., d_n/v_n \\
& & d_k \in DI \cup DO,\ v_k \in V \\
\text{Expression} & E ::= ; A \mid E; A \mid E \,[]\, E \mid E^* & \\
\text{Definition} & M ::= \text{Process}\,(CI, CO, DI, DO) \triangleq E &
\end{array}
$$

Figure 1. BNF-style syntax description

The BNF-style syntax description of the language is given in Figure 1. A process is parameterized on the port names occurring in the expression in its definition. Input and output actions are denoted by A_i and A_o respectively.

2.1.1. Example

A multiplier machine with timing diagram as shown in Figure 2a can be expressed as the definition *mult* (Figure 2b) and as a state machine (Figure 2c). This machine holds the control output $\overline{done}$ until the control input *start* occurs along with the value x on input data port p. The value y occurs on input data port p in the next state. After an indeterminate but finite number of steps the control output $\overline{done}$ is asserted with value $x * y$ on the data output port $\overline{o}$. This sequence of events repeats indefinitely until the machine is turned off or reset.

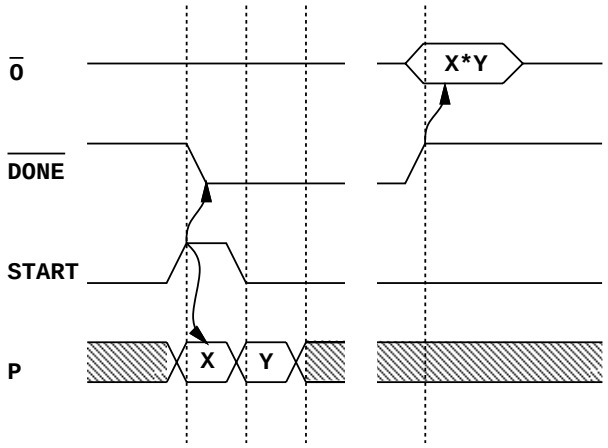
The semantics of the interface specification language is described in terms of state diagrams which are based on a finite-state machine model discussed in the next section. Further discussion about the interpretation of the language and a method to construct finite state machines from interface specifications are presented in Section 4.

3. Finite State Machine Model

A machine is a sextuple $M = \langle S, T, s, f, P, V \rangle$, where S is the set of states, T is a non-empty set of transitions, s is the start state, f is the final state, P is the set of ports, and V is the domain of values. The set of ports is a union of the sets of control inputs, control outputs, data inputs, and data outputs ($P = CI \cup CO \cup DI \cup DO$).

3.1. States

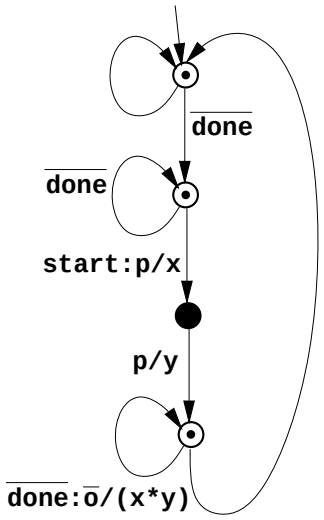
The start state represents the state of the machine after a reset. There are no visible storage elements (registers) in this model. There are two kinds of states, *transit* states and *wait* states.



(a)

$mult\ (\text{start}, \overline{\text{done}}, p, \overline{o}) \triangleq$
$[; \overline{\text{done}}\ \textit{until}\ \text{start} : p/x\ ;\ p/y\ ;\ \textit{compute}\ \overline{\text{done}} : \overline{o}/(x * y)]^*$

(b)



(c)

Figure 2. Three depictions of a multiplier interface

Definition 3.1: A state is called a *transit* state if there are no transitions from the state to itself. A machine can stay in a transit state for only one step.

Definition 3.2: A state is called a *wait* state if there is at least one transition from the state to itself. The machine takes a transition out of the state based on external conditions (control inputs) or internal conditions (control outputs).

3.2. Transitions

In this model transitions are defined as: $s_1 \xrightarrow{\mathcal{L}} s_2$, where s_1 is the source state, s_2 is the target state, and $\mathcal{L}$ is a label. A label $\mathcal{L}$ is the sum of assignment functions $\mathcal{L}_c$ and $\mathcal{L}_d$, where $\mathcal{L}_c : CI \cup CO \mapsto \{0, 1, \#\}$ and $\mathcal{L}_d : DI \cup DO \mapsto V$. The set $care_{\mathcal{L}} = \{p \mid \mathcal{L}(p) \neq \#\}$ denotes the set of ports that do not have don't care values according to the assignment function $\mathcal{L}$.

3.3. State Diagrams

A state diagram is a graphical representation of a finite state machine with nodes representing states and edges representing transitions. The different types of states are represented using different node symbols : *Transit* - ●, and *Wait* - ⊙. A state of unknown type is represented as ○. Transitions are represented as labeled edges in the state diagram. The start state is indicated as an incoming edge without any source state. The state diagram for the multiplier machine is shown in Figure 2 c.

3.4. Synchronization

In a synchronous system, we use three synchronization primitives, *lock-step, 1-way*, and *2-way*, to model communication between a machine and its environment.

Lock-step: The machine and its environment are synchronized. No waiting is involved, and there is no need for control synchronization.

1-way: One of the machines (component or its environment) is computing while the other is waiting. This may take an indeterminate but finite number of steps. The machine that is computing needs control outputs to synchronize, and the machine that is waiting needs control inputs to synchronize. Interactions of the form *compute* A_o and *await* A_i are used to specify 1-way synchronization.

2-way: Both the machines (component and its environment) are computing and both have to finish to synchronize. One of them sends a sequence of signals to the other until it receives an acknowledgment signal. Both machines need at least one control input and at least one control output to synchronize. Interactions of the form A_o *until* A_i and A_i *before* A_o are used to specify 2-way synchronization.

4. Interpretation

The interpretation of an expression in the interface specification language is the state diagram it constructs. The state diagram represents a machine that takes a transition on every step. The construction of the state diagram is described below.

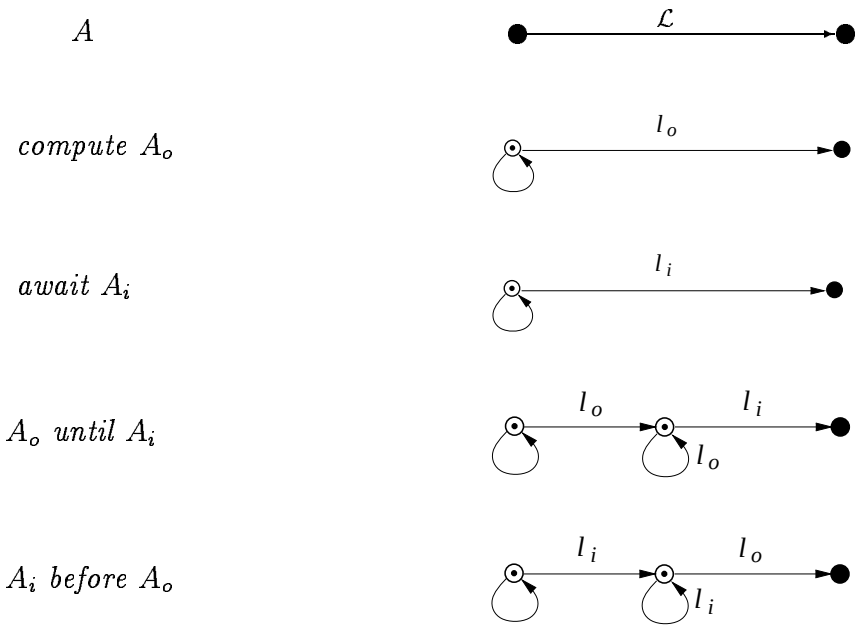


Figure 3. Action semantics

4.1. Actions

An atomic action of the form $A = \{c_1, .., c_m : d_1/v_1, .., d_n/v_n\}$ denotes a state diagram with a transition from one transit state to another transit state with the label $\mathcal{L}$ such that

$$\begin{array}{rcll} \mathcal{L}(p) & = & 1 & \text{if } p \in A \ \wedge \ p \in (CI \cup CO) \\ & & 0 & \text{if } not(p) \in A \ \wedge \ p \in (CI \cup CO) \\ & & v & \text{if } p/v \in A \ \wedge \ p \in (DI \cup DO) \\ & & \# & \text{otherwise} \end{array}$$

This transition is taken if $\bigwedge_{i=1}^{m} c_i \in care_{\mathcal{L}} = \mathsf{TRUE}$. The data port d_j has the corresponding value v_j.

Let l_o be the label for A_o, l_i be the label for A_i. Let $A_o = \{c_{o1}, .., c_{om} : d_{o1}/v_1, .., d_{on}/v_n\}$ and $A_i = \{c_{i1}, .., c_{im} : d_{i1}/v_1, .., d_{in}/v_n\}$. The interpretation of atomic and multi-step actions is shown in Figure 3.

The interpretation of compound actions of the form A_1, A_2 is shown in Figure 4. If one of the actions denotes a state diagram with a transition from a transit state to another and the other a multi-step action, then A_1, A_2 denotes a compound action with the label of A_1 combined with the label on each transition of A_2. If both A_1 and A_2 denote multi-step actions and T_1 and T_2 are the state diagrams of A_1 and A_2 without the final transitions (l_1 and l_2), the state diagram for A_1, A_2 can be constructed by adding the transition with label l_1, l_2 to the product $T_1 \times T_2$ as shown in Figure 4. A label of the form l_1, l_2 is denoted as the sum of the assignment functions.

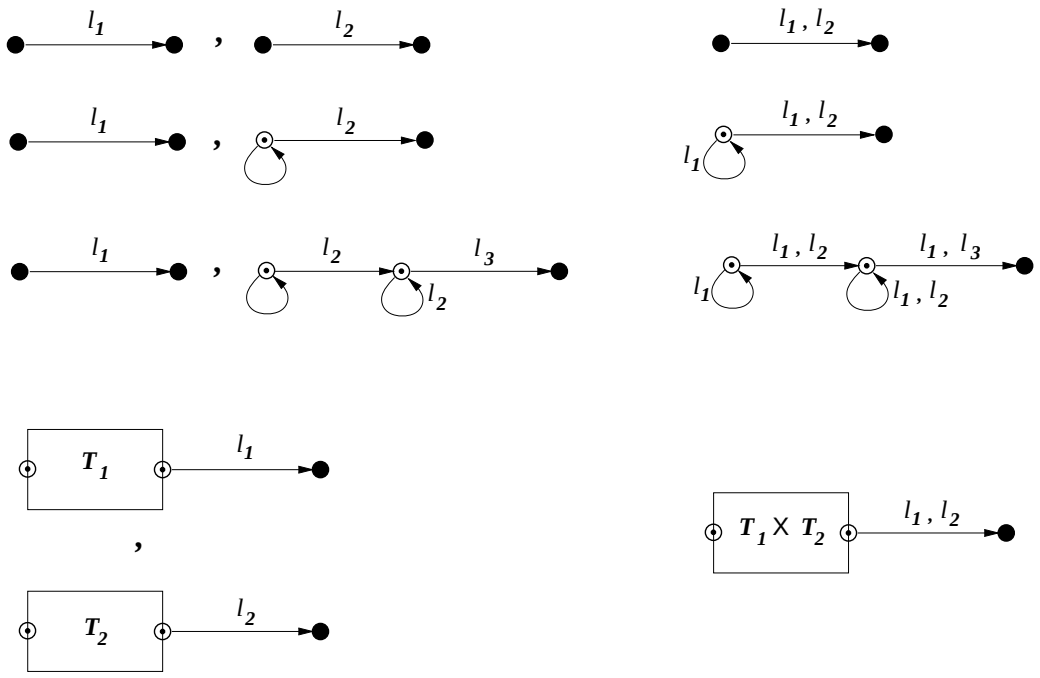


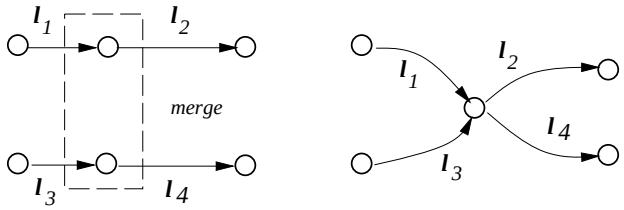
Figure 4. Interpretation of Compound Actions

Unlabeled transitions represent hidden actions. All the primitive forms described in Figure 3 construct a graph with a start and a final state. The start state does not have any incoming transitions from any other state. The final state does not have any outgoing transitions to any other state. All the primitive forms construct state diagrams that have a transit final state.

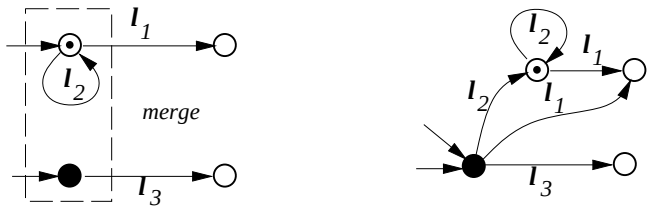
4.2. Expressions

A *merge* operation is defined on states and is used in constructing state diagrams from expressions. States have incoming and outgoing transitions, and can be of the same type or of different types.

Case 1: If both states are of the same type, then this operation creates a single new state of the same type in place of both. All incoming and outgoing transitions of both states are assigned to this state.

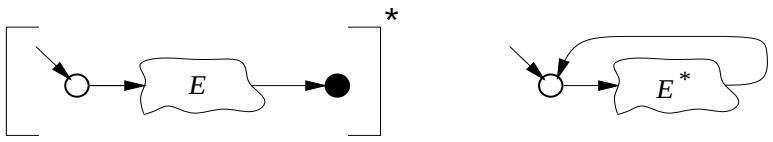


Case 2: If one state is transit and the other wait, then for each outgoing transition from the wait state, a transition with the same label and same target state is added with the transit state as the source. Since there is always a transition from the wait state to itself, a transition is created from the transit state to the wait state.



An expression in the language begins with ";". This unary operator denotes the start state. In the state diagram this is represented by an unlabeled incoming transition with the start state as the target state and no source state.

Repetition: The state diagram for the expression (E^*) can be constructed from the state diagram for E by merging its start and final states.



Follow: The binary operator $(;)$ in the expression $(E;A)$ concatenates the state diagram constructed by E with the diagram constructed by A. This construction is performed by merging the start state of the second state diagram with the final state of the first state diagram.

Choice: The state diagram for the expression $(E_1 [] E_2)$ can be constructed from the state diagrams for E_1 and E_2 by merging both the start states to form the new start state and both the final states for the new final state.

5. Relations and Operations on Machines

This section presents some operations on machines. The minimization operation results in a minimal machine by removing the redundant transitions and collapsing equivalent states. The hiding operation is used to internalize ports in a machine and the restriction operation is used to restrict access of certain ports. The composition operation constructs a product machine with reachable states and transitions for a given set of port connections. The complement operation is used to create the environment machine.

5.1. Minimization

A machine can be minimized by going through iterations of collapsing equivalent states and removing redundant transitions until no states or transitions can be removed. The minimization algorithm for finite state automata described in [11] can be used here.

Two states are equivalent if all the transitions from one state have the same label and equivalent target as the transitions from the other state. Both the equivalent states are removed and a new state is created with all the transitions associated with both the states. The new state is of type wait if either of the equivalent states is of type wait, otherwise it is of type transit.

Two transitions are equivalent if they have the same source and target states and equivalent labels. Only one of the equivalent transitions is kept in the machine, the other is removed.

5.2. Hiding and Restriction

The hiding operation can be used to hide certain ports in a machine. This operation is used to internalize ports. A list of ports to be hidden $P_h = CI_h \cup CO_h \cup DI_h \cup DO_h$ and a machine $M = \langle S, T, s, f, P, V \rangle$, where $P = CI \cup CO \cup DI \cup DO$, must be provided. *Hide* (M, P_h) is a machine with the ports $(CI - CI_h) \cup (CO - CO_h) \cup (DI - DI_h) \cup (DO - DO_h)$. All control ports and data ports that are hidden are removed from the transition label, the transitions themselves are not removed. The new transition labels in *Hide* (M, P_h) are $\mathcal{L}'(p) = \mathcal{L}(p)$ if $p \in (DI - DI_h) \cup (DO - DO_h) \cup (CI - CI_h) \cup (CO - CO_h)$, otherwise $\mathcal{L}'_d(p)$ is undefined.

A machine is restricted by eliminating transitions based on restricted control ports. The transitions with label $\mathcal{L}$ where $(care_{\mathcal{L}} \cap (CI_h \cup CO_h) \neq \phi)$ are removed from the set of transitions, otherwise $\mathcal{L}'_c(p) = \mathcal{L}_c(p)$ if $p \in (CI - CI_h) \cup (CO - CO_h)$. The restricted data ports are removed from the transition labels, the transitions themselves are not removed. The new transition labels in *Restrict* (M, P_h) are $\mathcal{L}'_d(p) = \mathcal{L}_d(p)$ if $p \in (DI - DI_h) \cup (DO - DO_h)$.

5.3. Composition

The composition operation on machines with respect to a particular connection of their ports is used to construct a product machine that behaves as the constituent machines executing synchronously. The connection of the ports of two machines is formally described below.

Given machines $M_1 = \langle S_1, T_1, s_1, f_1, P_1, V_1 \rangle$ and $M_2 = \langle S_2, T_2, s_2, f_2, P_2, V_2 \rangle$ with ports $P_1 = CI_1 \cup CO_1 \cup DI_1 \cup DO_1$ and $P_2 = CI_2 \cup CO_2 \cup DI_2 \cup DO_2$, we define the binary relation $\mathcal{N}$ on $P_1 \cup P_2$, as follows:

Definition 5.3: $p_1 \mathcal{N} p_2$ iff p_1, p_2 are either both control or both data ports and p_1, p_2 are connected (equipotential).

$\mathcal{N}$ is an equivalence relation. Each equivalence class of $\mathcal{N}$ is called a net. Connecting input ports together creates an input net. All other combinations of port connections create output nets.

Definition 5.4: Given machines $M_1 = \langle S_1, T_1, s_1, f_1, P_1, V_1 \rangle$, $M_2 = \langle S_2, T_2, s_2, f_2, P_2, V_2 \rangle$ and net-list $\mathcal{N}$ between P_1 and P_2, the composed machine $(M_1 \parallel M_2)_{\mathcal{N}} = \langle S, T, s, f, P, V \rangle$ where $P = CI \cup CO \cup DI \cup DO$.
Each equivalence class N of $\mathcal{N}$ forms a port in the composed machine. It is represented as $[p]$ where $p \in N$.

$CI = \{N \mid N$ contains no control output ports$\}$
$CO = \{N \mid N$ contains at least one control output port$\}$

$DI = \{N \mid N \text{ contains no data output ports}\}$
$DO = \{N \mid N \text{ contains at least one data output port}\}$

The set of states S and transitions T are constructed by the following induction schema:

1. The start state of the composed machine $s = \langle s_1, s_2 \rangle$.
2. The transition $\langle q, r \rangle \xrightarrow{\mathcal{L}} \langle q', r' \rangle \in T$ and $\langle q', r' \rangle \in S$ if:
   - i. $q \xrightarrow{\mathcal{L}_1} q' \in T_1$ and $r \xrightarrow{\mathcal{L}_2} r' \in T_2$
   - ii. $\langle q, r \rangle \in S$
   - iii. $\exists . j_1 . j_2 \ldots j_k$ such that
     - *a*) $care_{\mathcal{L}_1} \cup care_{\mathcal{L}_2} \subseteq \bigcup_{i=1 \text{ to } k} [p_{j_i}]$
     - *b*) $\forall p, p' \in [p_{j_i}] \,.\, \mathcal{L}_{12}(p) = \mathcal{L}_{12}(p') \vee \mathcal{L}_{12}(p) = \# \vee \mathcal{L}_{12}(p') = \#$
       where $\mathcal{L}_{12}(p) = \mathcal{L}_1(p)$ if $p \in P_1$
       $\mathcal{L}_2(p)$ if $p \in P_2$

   where $\mathcal{L}([p]) = \mathcal{L}_{12}(p)$ if $p \in care_{\mathcal{L}_1} \cup care_{\mathcal{L}_2}$, # otherwise
3. The state $\langle q, r \rangle$ is of type transit if either q or r are of type transit, otherwise it is of type wait.

If $\langle f_1, f_2 \rangle \notin S$ then $(M_1 \parallel M_2)_{\mathcal{N}}$ is considered *unsafe*.

The construction of the composed machine above is similar to the "lock-step cartesian product" in HOP[9].

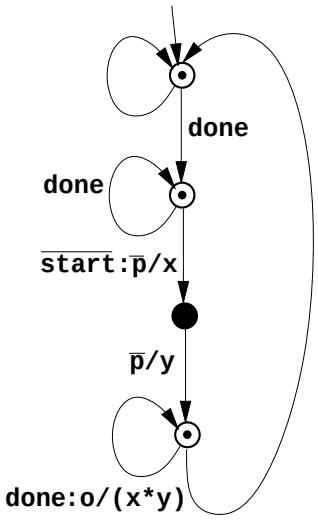


Figure 5. Complement of Multiplier Machine

5.4. Complementation

The complement of a machine M defines its environment machine $\overline{M}$. The complement of a machine can also be constructed from its state diagram. This is done by changing all input ports to output ports and vice versa.

Definition 5.5: Given a machine $M = \langle S, T, s, f, P, V \rangle$ where $P = CI \cup CO \cup DI \cup DO$, the complement machine $\overline{M} = \langle S, \overline{T}, s, f, \overline{P}, V \rangle$ where $\overline{P} = \{\text{rename}(p) \mid p \in P\}$, and rename is a function that generates a new input(output) port name for an output(input) port. The set of transitions

$$\overline{T} = \{s_1 \xrightarrow{\overline{\mathcal{L}}} s_2 \mid s_1 \xrightarrow{\mathcal{L}} s_2 \in T \text{ and } \overline{\mathcal{L}} = \text{Rename}(\mathcal{L})\}$$

where $\text{Rename}(\mathcal{L})(p') = \mathcal{L}(p)$ if $p' = \text{rename}(p)$.

The state diagram of the complement of the multiplier machine is shown in Figure 5.

5.5. Implementation and Equivalence

A machine M_1 is implemented by ($\sqsubseteq$) machine M_2 if every state in M_1 is simulated by some state in M_2, and the start state of M_1 is simulated by the start state of M_2.

A one-to-one mapping of the ports in M_1 and M_2 is the basis for the *inclusion* relation over transition labels. Let p_1 and p_2 be ports in M_1 and M_2 respectively. A map of the form $p_1 \mapsto p_2$ means p_1 corresponds to p_2 and the values on these ports can be compared.

Definition 5.6: The *inclusion* relation over transition labels with respect to a port map is defined as:

$$\mathcal{L}_1 \preceq \mathcal{L}_2 \iff \forall p_1 \in care_{\mathcal{L}_1} . \, p_2 \in care_{\mathcal{L}_2} . \, p_1 \mapsto p_2 \wedge \mathcal{L}_1(p_1) = \mathcal{L}_2(p_2)$$

The binary relation "simulated by" ($\sqsubset$) is a maximal relation over states, $\mathcal{S} \subseteq S_1 \times S_2$, where S_1, S_2 are the sets of states in M_1, M_2.

Definition 5.7: A relation over states is a *simulation* relation if $s_1 \sqsubset s_2$ implies:

$$\forall s_1 \xrightarrow{\mathcal{L}_1} s_1' \, . \, \exists s_2 \xrightarrow{\mathcal{L}_2} s_2' \, . \, (\mathcal{L}_1 \preceq \mathcal{L}_2) \wedge (s_1' \sqsubset s_2')$$

Let r_1, r_2 be the start states of M_1, M_2.

Definition 5.8: The relation M_1 *implementated by* M_2 ($M_1 \sqsubseteq M_2$) holds iff:

1. $r_1 \sqsubset r_2$
2. $\forall s_1 \in S_1 \, . \, \exists s_2 \in S_2 \, . \, s_1 \sqsubset s_2$

Two machines are equivalent if each machine is implemented by the other.

Definition 5.9: The *equivalence* relation on machines is defined as

$$M_1 \equiv M_2 \quad \stackrel{\text{def}}{=} \quad (M_1 \sqsubseteq M_2) \text{ and } (M_2 \sqsubseteq M_1)$$

5.5.1. Path Implementation

Each path from the start state to the final state in the complement machine represents a valid sequence of interactions to complete a protocol. A machine can interact with an implementation of any interaction path of its complement machine. Decomposition of a component from a system is accomplished by incorporating the appropriate *path implementation* of the complement of the component into the description of the rest of the system.

Path implementation of a process implements one of many protocols of the process. It is a weaker relation than *implementation* which implements all protocols in a process. In case of a component process with different protocols for different operations, only one of which is used by its environment in a procedure, the *path implementation* of the complement of the component, that performs the required procedure, is incorporated into the environment.

To define the *path implementation* relation we must first define the relation *path simulates* over states. The binary relation *path simulates* is a maximal relation over states, $\mathcal{S}_p \subseteq S_1 \times S_2$, where S_1, S_2 are the sets of states in the two machines. $s_1 \sqsubset_{\mathrm{p}} s_2$ implies that there are possible transitions from s_1 and s_2, such that the transition from s_1 is included by the transition from s_2, and they lead to states which satisfy the same relation. Also, for all transitions with active control inputs from s_1, there is a corresponding transition from s_2, which includes the transition from s_1, and these transitions lead to states which satisfy the same relation. It also implies that, if s_1 is a wait state for a control input, then s_2 must also be a wait state for the same control input, or s_2 must lead to a wait state for the same control input.

Definition 5.10: A relation over states is a *path simulation* relation if $s_1 \sqsubset_{\mathrm{p}} s_2$ implies :

1. $\exists s_1 \xrightarrow{\mathcal{L}_1} s_1', s_2 \xrightarrow{\mathcal{L}_2} s_2' \,.\, \mathcal{L}_1 \preceq \mathcal{L}_2 \ \wedge \ s_1' \sqsubset_{\mathrm{p}} s_2'$
2. $\forall s_1 \xrightarrow{\mathcal{L}_1} s_1' \,.\, (care_{\mathcal{L}_1} \cap CI_1) \neq \phi \ \Rightarrow \ \exists s_2 \xrightarrow{\mathcal{L}_2} s_2' \ \wedge \ \mathcal{L}_1 \preceq \mathcal{L}_2 \ \wedge \ s_1' \sqsubset_p s_2'$
3. $\exists s_1 \xrightarrow{\mathcal{L}_{1_1}} s_1, s_1 \xrightarrow{\mathcal{L}_{1_2}} s_1' \,.\, s_1 \neq s_1' \ \wedge \ (care_{\mathcal{L}_{1_2}} \cap CI_1) \neq \phi \ \Rightarrow$
   $(\exists s_2 \xrightarrow{\mathcal{L}_{2_2}} s_2' \,.\, s_2 \neq s_2' \ \wedge \ \mathcal{L}_{1_2} \preceq \mathcal{L}_{2_2} \ \wedge \ s_1' \sqsubset_{\mathrm{p}} s_2') \wedge$
   $((\exists s_2 \xrightarrow{\mathcal{L}_{2_1}} s_2 \,.\, \mathcal{L}_{1_1} \preceq \mathcal{L}_{2_1}) \ \vee \ (\exists s_2 \xrightarrow{\mathcal{L}_{2_3}} s_k \,.\, \mathcal{L}_{1_1} \preceq \mathcal{L}_{2_3} \ \wedge \ s_1 \sqsubset_{\mathrm{p}} s_k))$

Definition 5.11: A machine M_1 *path implements* machine M_2 ($M_1 \sqsubseteq_{\mathrm{p}} M_2$) if :

1. The start state of M_1 is path simulated by the start state of M_2.
2. The final state of M_1 is path simulated by the final state of M_2.

6. An Example of Process Decomposition

Given a specification of a sequential process, one would like to decompose the process in two or more sub-processes. A procedure F in the specification can be instantiated as a subprocess P_F. The goal here is to incorporate the complementary process $\overline{P_F}$ as an interaction "stub" into the original process.

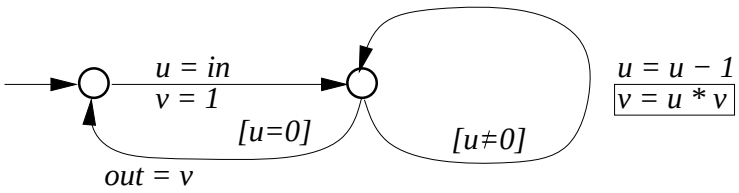


Figure 6. Factorial Machine State Diagram

Consider the example of a factorial machine *fac* (Figure 6). This specification shows the internal facet of *fac*, e.g. internal registers u, v, and internal conditions $[u = 0], [u \neq 0]$.

To abstract the procedure $v = u * v$ and use the multiplier machine in Figure 2, the factorial machine should incorporate an *implementation* of the complement machine $\overline{mult}$ (Figure 5). The assumption here is that *mult* performs multiplication. The following steps are involved in modifying *fac*:

1. Add all the control and data ports in $\overline{mult}$, $(done\ , \overline{start}\ , \overline{p}\ , o)$ to *fac*.

2. Replace the state transition containing $v = u * v$ with an *implementation* of $\overline{mult}$, by merging the start state with the source state of the transition and the final state with the target state of the transition (Figure 7).

3. The previous step does not take into account the other procedure $u = u - 1$ in the replaced transition in the original *fac*. This procedure is scheduled in the earliest possible transition that does not go from a state to itself, such that the register value dependencies in the original *fac* are preserved. For simple expressions this is done by textual comparison. In general this involves verification of equivalence of logical and arithmetic expressions, and is a heuristic task [6].

Let $\hat{u}$, $\hat{v}$ be the values in registers u, v before the procedures $u = u - 1$ and $v = u * v$ in the original *fac*. The values in the registers after the procedures in the original *fac* and by scheduling $u = u - 1$ with t_1 and t_2 in the modified *fac* (Figure 7) are shown in the table below. t_2 is the earliest transition where the procedure $u = u - 1$ gives the same values as the original *fac*.

	Register Values after Procedures	
	u	v
Original	$\hat{u} - 1$	$\hat{u} * \hat{v}$
$u = u - 1$ with t_1	$\hat{u} - 1$	$(\hat{u} - 1) * \hat{v}$
$u = u - 1$ with t_2	$\hat{u} - 1$	$\hat{u} * \hat{v}$

The modified *fac* can be turned into a state diagram with only the interface specification by hiding all the names in the original *fac* except the ones added in step 1 above, and minimizing the resulting state diagram. The modified *fac* and *mult* are subprocesses that together implement the original *fac*.

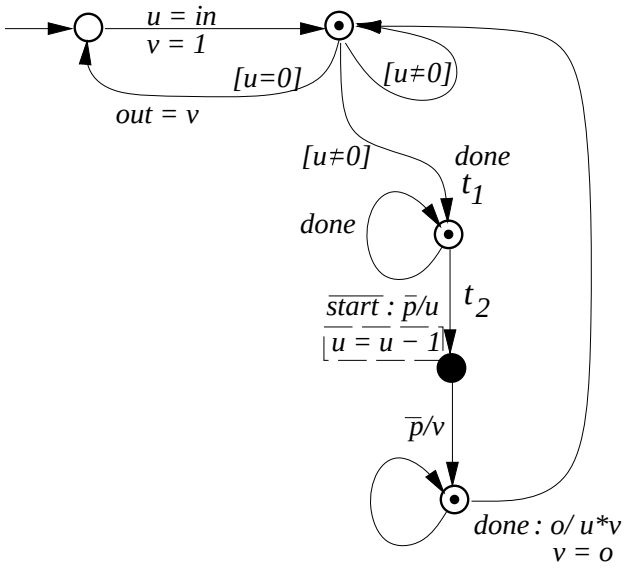


Figure 7. Factorial Machine with Multiplier Factored

7. Observations and Conclusion

In this paper we described a language for interface specification and a finite state machine based semantic model. This language can be used to describe the interaction among components in a synchronous hardware system. A definition in this language describes the input/output behavior of a machine. A set of operations and relations are defined on machines. The complement operation is used to construct the environment of a machine. Decomposition is done by factoring an internal procedure to a sequential machine that performs the procedure. An implementation of the complement of this machine is incorporated in the original machine. Successive decomposition steps result in an implementation of a network of machines that implement the high-level specification. We have used process decomposition to derive a DRAM memory sub-system for an implementation of the FM9001 microprocessor [23].

The language described here needs to be extended to allow symbolic values on control ports. The syntax is restricted in order to maintain a simple semantics. There is no mechanism to quantify time in the language. Our approach can provide a framework for both top-down and bottom-up reasoning by integrating with a verification system. Liveness and safety issues in system design will also be explored within our framework. The language, the FSM model, and the operations presented here will be automated and integrated with our design derivation system.

8. Acknowledgements

We are grateful to Venkatesh Choppella for his many helpful suggestions in revising this paper.

REFERENCES

1. G. Borriello. Specification and synthesis of interface logic. *High-Level VLSI Synthesis*, pages 153–176, 1991.
2. Tam Anh Chu. Synthesis of self timed VLSI circuits from graph theoritic specifications. In *Intl. Workshop on Petri Nets and Performance Models*, August 1987.
3. Ed Clarke, D. Dill, J. Burch, K. L. McMillan, and L. J. Hwang. Symbolic model checking: 10**20 states and beyond. In *International Workshop on Formal Methods in VLSI Design*. ACM-SIGDA, January 1991.
4. Bruce S. Davie. *A Formal, Hierarchical Design and Validation Methodology for VLSI*. PhD thesis, University of Edinburgh, 1988.
5. Bruce S. Davie and George J. Milne. Contextual constraints for design and verification. In Birtwistle and Subramanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 257–265. Kluwer, 1988.
6. S. Devadas and K. Keutzer. An Automata-Theoretic Approach to Behavioral Equivalence. In *Proceedings of the International Conference on Computer-Aided Design*, pages 30–33, November 1990.
7. David L. Dill, Alan J. Hu, and Howard Wong-Toi. Checking for language inclusion using simulation preorders. In Larsen and Skou, editors, *Proceedings of Computer Aided Verification*, pages 255–265. Springer, July 1991. LNCS 575.
8. Doron Drusinsky and David Harel. Using statecharts for hardware description and synthesis. *Transactions on CAD*, 8(7):798–807, July 1989.
9. Ganesh C. Gopalakrishnan, Richard M. Fujimoto, Venkatesh Akella, and Narayana S. Mani. HOP: A process model for synchronous hardware; semantics and experiments in process composition. *Integration, the VLSI journal*, 8:209–247, 1989.
10. Gerard J. Holzmann. Tracing protocols. In Yemini, editor, *Current Advances in Distributed Computing and Communications*, pages 189–207. Computer Science Press Inc, 1987.
11. J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison Wesley, 1979.
12. Steven D. Johnson. Applicative programming and digital design. In *Proceedings of 11th Annual SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 218–227, 1984.
13. Steven D. Johnson. *Synthesis of Digital Designs from Recursion Equations*. MIT Press, Cambridge, 1984. ACM Distinguished Dissertation 1984.
14. Steven D. Johnson. Manipulating logical organization with system factorizations. In Leeser and Brown, editors, *Hardware Specification, Verification and Synthesis: Mathematical Aspects*, volume 408 of *LNCS*, pages 260–281. Springer, July 1989. Proceedings of Mathematical Sciences Institute Workshop, Cornell University, 1989.
15. Steven D. Johnson and Bhaskar Bose. A system for mechanized digital design derivation. In Subramanyam, editor, *Proceedings of ACM International Workshop on Formal Methods in VLSI Design*, January 1991.
16. Steven D. Johnson, R.M. Wehrmeister, and B. Bose. On the interplay of synthesis and verification: Experiments with the FM8501 processor description. In Claesen, editor, *Applied Formal Methods for Correct VLSI Design*, pages 385–404. Elsevier,

1989. IMEC 1989.
17. David Ku and Giovanni De Micheli. Relative scheduling under timing constraints. In *Proceedings of ACM/IEEE Design Automation Conference*, June 1990.
18. R. P. Kurshan. Analysis of discrete event simulation. In Bakker, Roever, and Rozenberg, editors, *Stepwise Refinement of Distributed Systems*, pages 414–453. Springer-Verlag, July 1989. LNCS 430.
19. Kenneth L. McMillan and David L. Dill. Algorithms for interface timing verification. In *Proceedings of IEEE International Conference on Computer Design*, pages 48–51. IEEE Computer Society, November 1992.
20. George J. Milne. CIRCAL: A calculus for circuit description. *Integration*, 1:121–160, 1983.
21. George J. Milne. Design for verifiability. In Leeser and Brown, editors, *Hardware Specification, Verification and Synthesis: Mathematical Aspects*, pages 1–13. Springer, July 1989. LNCS 408.
22. J. A. Nestor and D. Thomas. Behavioral synthesis with interfaces. In *Proceedings of ICCAD*, November 1986.
23. Kamlesh Rath, Bhaskar Bose, and Steven D. Johnson. Derivation of a DRAM memory interface by sequential decomposition. to appear in ICCD, 1993.
24. Andrés R. Takach and Wayne Wolf. Behavior FSMs for high-level synthesis and verification. Technical Report CE-W91-13, Dept. of Electrical Engineering, Princeton University, July 1991.
25. Wayne Wolf, Andrés Takach, and Tien-Chien Lee. Architectural optimization methods for control-dominated machines. *High-Level VLSI Synthesis*, pages 231–254, 1991.
26. Zheng Zhu and Steven D. Johnson. An example of digital design transformation in an algebraic framework. In Subramanyam, editor, *Proceedings of ACM International Workshop on Formal Methods in VLSI Design*, January 1991.