A PARTITIONED COMPUTATION MACHINE

by

Kenneth Brett Streeter

SUBMITTED TO THE DEPARTMENT OF ELECTRICAL
ENGINEERING AND COMPUTER SCIENCE IN
PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREES OF

BACHELOR OF SCIENCE
and
MASTER OF SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

August 1990

Signature of Author _____
Department of Electrical Engineering and Computer Science
August 17, 1990

Certified by _____
Nancy A. Lynch
Academic Thesis Supervisor

Certified by _____
Paul C. Brown
Company Supervisor (General Electric, Corporate Research and Development)

Accepted by _____
Arthur C. Smith
Chairman, Departmental Committee on Graduate Students

A PARTITIONED COMPUTATION MACHINE

by

Kenneth Brett Streeter

Submitted to the Department of Electrical Engineering and Computer Science on August 17, 1990 in partial fulfillment of the requirements for the degrees of Bachelor of Scienceand Master of Science.

Abstract

In software design and specification, a formal, easy-to-use specification language is desirable. Formal specification languages and easy-to-use languages already exist, but none combine these qualities with a visual representation. The Partitioned Computation Machine (PCM) is intended to serve as such a specification tool. It includes both a formal representation for a visual model extending Harel's statecharts, and a language which adds visual state representation and hierarchy to Lynch's I/O Automata. This thesis proposes a formal model for a PCM and explains how the PCM model relates to statecharts and I/O Automata.

Thesis Supervisor: Nancy A. Lynch
Title: Professor of Electrical Engineering and Computer Science
Thesis Supervisor: Paul C. Brown
Title: Computer Scientist, General Electric CRD

Acknowledgements

I would like to express my sincere gratitude to Paul Brown. His patient guidance of this work from start to finish has made it possible. I would also like to specially thank Prof. Nancy Lynch for her advice and supervision.

There are so many people at General Electric Corporate Research and Development that have helped me in some way during this project that to name each one would be impossible. I simply extend my gratitude to them all. I feel truly privileged to have worked with such an exceptional group of people.

I would especially like to thank my wonderful wife, Nancy, for the continual encouragement thoughout the work leading to the completion of this thesis. I would also like to thank my son, Benjamin, for providing an outlet for childishness when work reached feverish proportions, and my parents for the understanding, support, and love that they have always shown me.

Table of Contents

List of Figures

Chapter 1

The Motivations for the Partitioned Computational Model

1.1 Introduction

The Partitioned Computational Model (PCM) is a specification tool intended to permit description and reasoning about systems which can be characterized by discrete events. This thesis introduces the model, the reasons for its development and examples of its use. The text is organized as follows: The remainder of this first chapter presents the goals the PCM seeks to fulfill. Chapter two discusses the other forms of specification from which the PCM has been incrementally developed. Chapter three contains an informal overview of the model, and chapter four presents the model formally with examples of partitioned computation machines. Lastly, the fifth chapter discusses the strengths and weaknesses of the PCM as well as some possible directions for further work with the model.

1.2 Motivations for the PCM

The Partitioned Computation Model seeks to fulfill a number of goals that were determined before the project was commenced. These goals all share a common interest -- to make a specification language that is usable in real industrial examples.

One of the primary goals of the PCM is to form a primarily visual, state-based specification language similar in appearance to a finite state machine. These are really two separate but related goals: a visual language, and a state-based specification. First of all, the concept of a classical finite state machine is a common, well-known form of behavioral specification. Its chief advantages are that behavior is fully specified, and the state of the

machine can be encapsulated visually. The syntax of the language to describe finite state machines is simple and easy to understand, so a layman can quickly understand the specification's meaning. The primary advantage of the graphical language for a state machine is that the structure of the control flow with respect to transitions, loops, and decision branches is immediately evident. It is not necessary to track through pages of written code to determine from where a procedure is called or to see where a loop ends. Simply put, the visual language of the classical state machine makes its control flow easy to view. The ease of perceiving the control flow in a state machine is a desirable goal in any specification language. For this reason, the goal of a state-based visual specification is sought.

Another primary goal of the PCM is that the specification be not just a set of heuristics or guidelines for forming the program, but rather that the language have a specific, well-defined formal semantics. The model should have a formal meaning and a formal representation. There should be a single formal representation for each possible diagram, and there should be a manner to formally represent the semantics of the diagram so that the meaning is entirely preserved in the representation.

A third goal for the PCM is that it be designed in such a manner that incremental changes to a partially-completed specification are easy to make. The PCM is intended to be used as a working model for developing a specification. Some formal specification techniques are excellent for precisely indicating what is desired in a completed design. However, modifying such a specification during development of the system is often a difficult task. The PCM is intended to permit small incremental changes to be made easily, without requiring massive restructuring of the entire specification to make small changes to the behavior that is being specified.

A fourth goal is that the specification language should preserve the specifier's problem structure throughout the specification development process. This goal essentially

means that the formal model should not disregard any information provided by the specifier. For instance, if a designer wishes to indicate that a certain event always be handled in a specific way, the underlying formal model should represent that condition in the same manner, without having to separately indicate in each possible state that the specific event is handled in a specific way. It should be possible to specify behavior upon a group of states without having to consider such behavior as being a summation of many individual behaviors. On the other hand, there are occasions when group behavior is not intended, but is simply a ''coincidence'' of many individual behaviors. In this instance, the specification language should not automatically abstract group behavior, but should preserve the specifier's intent of separate, but coincidentally the same, behaviors. However, it may be appropriate for an implementation of the partitioned computation machine to suggest such an abstraction to the user when many states share common behavior behavior arises in case the system being specified really contains group behavior, but the specifier hasn't yet realized this.

A fifth goal of the partitioned computation machine model is that it have fully general computational power. The expressive ability of the model should not be limited to behavior that can be defined by only classical finite state machines, context-free grammars, or recursive functions. Instead, the partitioned computational model should be able to express any type of behavior that can be implemented by a fully general Turing machine -- that is, anything that can be computed. In fact, it is possible for the PCM to describe even a larger class of behaviors. At the same time, however, it is desirable for the partitioned computation machine model to permit the behavior to be readily examined to see if it can be represented solely by a finite state machine, computational grammar, or augmented regular expressions so that automated proving techniques could be used upon these aspects of the model's behavior in such a case.

To permit abstraction and modularity is another goal for the PCM. It is generally

extremely advantageous to reason about the behavior of a system as being a combination of other types of behavior. Modularity permits the overall behavior to be broken into modules and to reason about the behavior of each module separately. Abstraction permits one to determine the behavior of a module and then concern oneself only with its behavior, and not the internal details that provide that behavior. In addition to increasing the comprehensibility of a specification or program, the concepts of modularity and abstraction also permit modules that are generally useful to be implemented once and then re-used in many different applications. For both the improvement in ease of understanding and the possibility of reusing code, modularity and abstraction are desirable goals in the PCM.

Lastly, a final goal for the PCM is that the specification model should readily permit simulation or execution. It would be highly desirable for a simulator of the specified behavior to be built directly from the PCM model. If a system were to be fully specified by a PCM, the entire system could be simulated to test the behavioral specification to ensure that it is the desired specification. Then, as the system is implemented in a logical, modular fashion within the overall specification, each implemented model could replace the simulated specification, leaving the remainder of the system with just the simulation. In this manner, each implementation piece can be tested easily in the entire system, permitting both isolation and integration testing to occur naturally.

Chapter 2

Background of the PCM

2.1 Building Blocks of the PCM

The PCM is built primarily from ideas of four existing specification techniques. The first of these specification languages is that used to describe a classical finite state machine, a mechanism that is familiar to all computer scientists, making it an excellent starting point for a new specification technique. The PCM also uses Harel's statecharts [Harel 87] for some aspects of the visual syntax and some of the features relating to the specification of transitions. The PCM draws upon Rumbaugh's state trees [Rumbaugh 88] for the notion of formal inheritance within a hierarchical tree-like structure of states. Lastly, the PCM relies upon Lynch's Input/Output automata [Lynch 88] as a basis for the formal representation for the visual language. Additionally, Lynch's model has the full generality that the PCM desires; the fact that I/O automata can express the full generality of the PCM permits the same behavior to be expressed by each. Furthermore, since they can express the same behavior and have similar formal models, the PCM can theoretically be translated into an equivalent I/O automaton which can then be used for theorem-proving -- such a process essentially utilizes the PCM as a visual front-end for the I/O automata.

This section will examine, in greater detail, the aspects of the PCM that have directly evolved from each of these other specification models.

2.2 The Classical Finite State Machine

There are two primary ideas in the Partitioned Computation Machine that have evolved from ideas in the classical finite state machine. The first of these ideas is the notion of describing the behavior of the system as separate modes of behavior that can occur, depending on the current "state" of the system. The second is the concept of representing the states, input events, and output actions in a visual format.

The idea of describing the behavior of a system by first specifying the behavior in a number of modes and then specifying the transition behavior between the possible modes of the system is a very important concept in the state machine. The primary importance of this is that the only thing that matters to determine current or future behavior is knowing the current state. The specific sequence of events that have led to the current state is not important in determining behavior. This idea permits the entire concept of partitioning the behavioral specification of a system into equivalence classes (states), where each class shares common behavior.

The second idea of describing the state, input events, and output actions visually is also important. The primary advantage of a visual language is that the control flow of the system is more readily evident than in most textual languages. In a typical computer language, state changes are made haphazardly throughout the code, without a special way to indicate when a significant change in the behavior of the system will result. In a state machine, however, transitions that make significant changes in behavior are easily recognized. As an example of this, consider Figures 2-1 and 2-2.

The state machine in the figure and in the code both represent similar behavior for a simple text editor with the ability to handle text characters and two special function keys, escape and insert. The text characters are added to the document by the Process_Keypress procedure. The escape key is used to exit the program, and the insert key is used to toggle

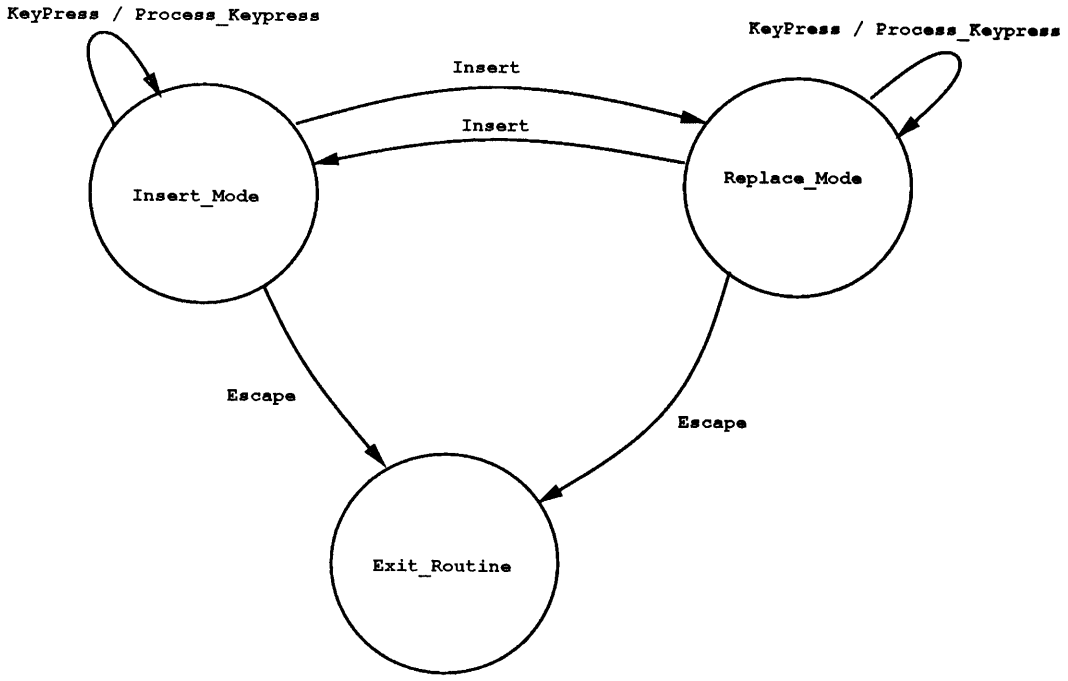


Figure 2-1: A State Machine for a Portion of a Text Editor

```
Get_Input(Standard_Input, Input_Character);

While  (Input_Character != Escape) Do
 {If (Insert_Mode) then
     {If (Input_Character == Insert) then
         {
           Insert_Mode := False;
           Replace_Mode := True;
         }
      else  Process_Keypress;}
  elseIf (Replace_Mode) then
     {If (Input_Character == Insert) then
         {
           Insert_Mode := True;
           Replace_Mode := False;
         }
      else Process_Keypress;}
  };

Exit_Routine;
```

Figure 2-2: C Code for a Portion of a Text Editor

between insert and replace modes. Although both specifications give similar behavior, in the state machine it is readily apparent what input events cause the transition to the new state (and new resultant behavior.) The equivalent C code, however does not make this obvious. The partitioned computation machine is built under the presumption that the input events that cause primary changes in behavior should be as evident from the PCM graphic specification as they are in the state machine specification of this example.

2.3 Statecharts

Statecharts are an extension of standard state-transition diagrams but are still isomorphic to finite state machines. [Harel 88] The primary advantage of statecharts is that they give notational shorthands for expressing traditional state machines. The innovative forms of shorthand employed in statecharts include a hierarchical depth structure and a cross-product notation to address the exponential growth in the number of states for linear system growth. (The exponential growth problem comes from needing a separate state for each possible combination of conditions represented. For example, if there are n conditions for which state information needs to be maintained, and each has a binary result, then there will be 2^n states needed. [Davis 88, p. 1102]) Additionally, statecharts begin exploration of state-based techniques for modeling a concurrent system with the cross-product notation. These primary statechart extensions to the visual language used for state machine description are also adopted in the partitioned computation machine.

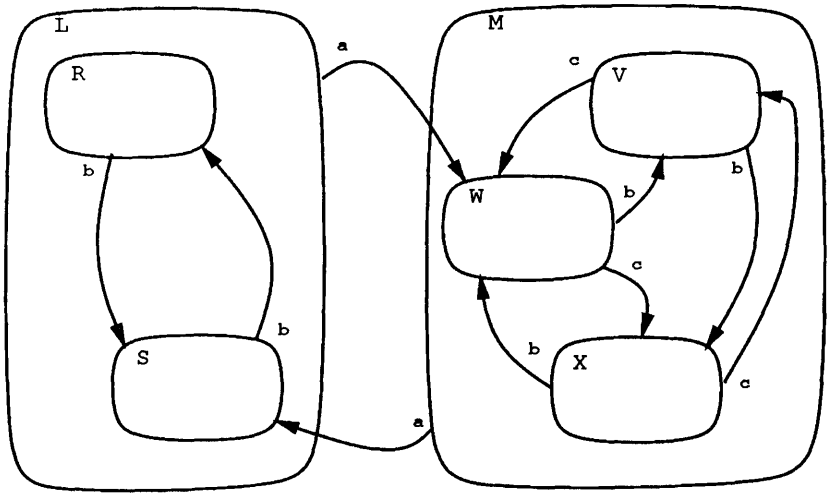


Figure 2-3: A Statechart Using a State Hierarchy

The first significant feature of statecharts that is utilized in the partitioned computation machine is the notation for representing a state hierarchy. The statechart

represents the hierarchy of states by graphically placing substates physically within superstates as is shown in Figure 2-3. Transitions may originate or finish at either a substate or a superstate. A transition to a superstate has the identical semantics as a transition to the "default" substate of that superstate. A transition originating from a superstate has exactly the same semantics as arcs originating from each and every one of the substates of that superstate. In this way, a single arc drawn from a superstate demonstrates that the behavior indicated on that arc is commonly shared among all of its substates. The ability to represent common behavior by parent states reduces the number of transitions which need to be drawn in large specifications. For example, in Figure 2-3, if transitions from parent states were not permitted, each of the states *V*, *W*, and *X* would have transitions going to state *R* for input *a*. Also, states *R* and *S* would both need transitions going to state *W* for input *a*.

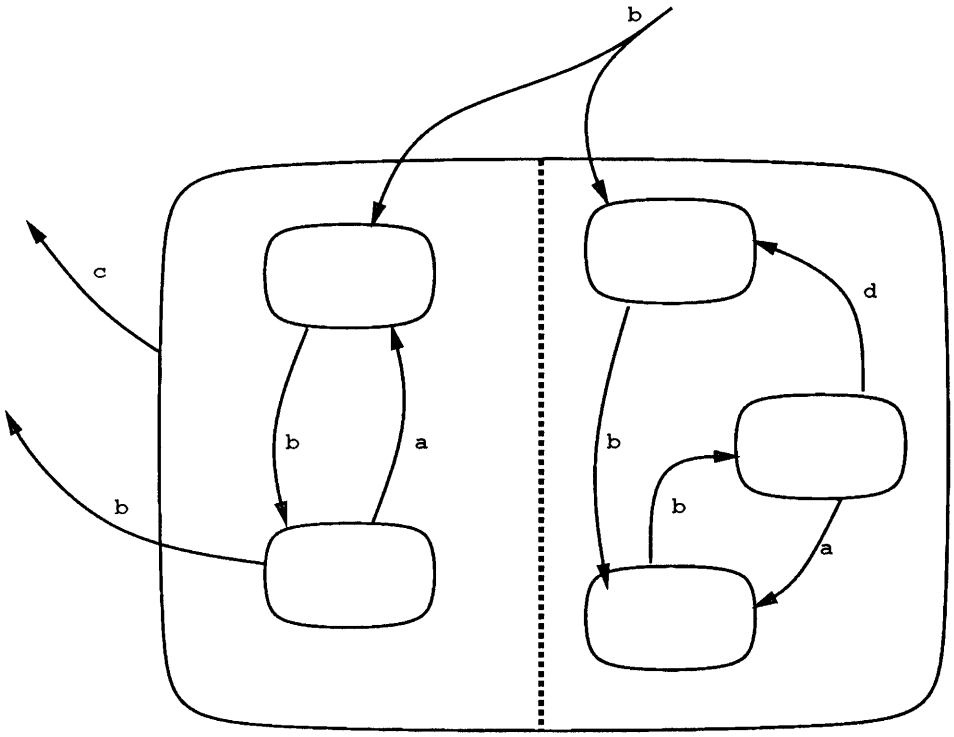


Figure 2-4: A Statechart Featuring Cross-Product Notation

The second major feature of statecharts is the manner in which a large number of

states can be represented with a cross-product notation. Figure 2-4 gives an example of a statechart that has concurrently operating substates. The two halves of the statechart divided by the dotted line indicate concurrently-operating portions of the statechart. The current state within the superstate is the cross-product of the states of each concurrent substate. Transitions in response to an event occur simultaneously in all concurrent substates. This type of notation helps the user understand a large system more easily as the number of states which are represented in the visual language is much less then the number of states into which the cross-product expands. The complexity in the number of states is still present in the underlying model, but is hidden from the user.

2.4 State Trees

Rumbaugh's state trees are a state-based specification tool intended for the design and specification of user interfaces. The model presents a number of innovations, including entry and exit routines with states and inheritance of behavior from parent states to their descendants. Of these innovations, the only one which is used in the partitioned computation model is the ability to inherit behavior from parent states.

Inheritance of behavior from a parent state is permitted in Harel's statecharts, however in a statechart, such inheritance requires consistency between transitions defined upon parents and their children. State trees, however, permit the children to take exception to the behavior which is defined at the level of the parents. In this way, the parent provides a default behavior that the child may use without change, or modify to suit its own needs.

The behavior present at the level of the parent may be viewed in one of two ways: as a requirement for the child to have a certain form of behavior, or as a default behavior that the child may follow or may override. Statecharts use the former view, while state trees view the behavior in the latter manner. Both languages can describe the same behavior, but the abstraction methods differ. In the partitioned computational model, the type of

inheritance defined in the statechart has been used because it provides a simpler model since child and parent behavior is required to be consistent.

2.5 Input/Output Automata

The Input/Output Automaton is a modeling tool intended for describing and reasoning about concurrent and distributed discrete event systems. The notable strengths of the I/O automaton are that the model is defined formally, that the model is executable and simulable (largely due to its formality), and that the model readily lends itself to proving fairness and liveness properties for a system. In addition, the model of I/O Automata permits distinct I/O Automata to be composed into a single automaton to express concurrent interaction between the separate component automata. All of these ideas are used in the partitioned computational model.

2.5.1 The I/O automata formal model

The formal model for I/O automata gives the specification technique two significant advantages. First, the precision of meaning which is provided by the formal model permits the execution of I/O Automata by a computer simulator. In this way, the behavior that is specified by the I/O Automata can be tested by running a computer simulation. Such a simulation technique provides an important method for checking that a specification may provide the desired results -- the results can be demonstrated directly.

A second advantage of the I/O automata formal model is that it has been built with the intention of carrying out algorithm correctness proofs. In fact, one of the primary goals for the construction of the I/O automata is to produce correctness proofs for large, complex concurrent algorithms. Without a formal model upon which to build such proofs, creating valid proofs would be nearly impossible. An added bonus for using the formal model to prove properties about the specified behavior is that many of the proofs could be machine-checked when automatic proof technology develops.

2.5.2 Composition of automata

Another special feature of the I/O automata is that many automata can be composed to yield other I/O automata. This composition permits one to describe a algorithm or behavioral specification in a modular manner. The process of composition maintains some properties of component behavior for the result of the composition. The modularity permits complex behavior to be represented easily as well as permitting the user to reason about the behavior of the whole as the sum of its parts, instead of attempting to understand an entire complex system at once.

One shortcoming of the I/O automata composition, however, is that the formal model does not maintain the modularity of the component specifications in its formal model. The composition process for I/O automata compresses a number of automata into a single one, but does not maintain the distinction between the component automata within the formal model. The only difficulty with this is that the specifier's view of the composition is organized around the individual components, but the structure of the components disappears in the composition process. This restructuring is acceptable for the primary applications of I/O Automata, proving algorithm correctness, since the I/O automata manner of composition guarantees that some types of properties proven for the component parts of a composition are still guaranteed for the whole. However, for the application domain of the partitioned computation machine, program specification, preservation of the specifier's intent at each level of specification is important.

Chapter 3

The Partitioned Computational Model

This chapter presents the partitioned computational model informally. First the basic building block, the finite state machine, is presented. Then, the expansions to the basic state machine, namely, a state hierarchy, variables associated with states, and the ability to represent concurrent substates are presented. Finally, the concepts of additive and multiplicative composition of partitioned computation machines are discussed.

3.1 The Simplest Form -- The Finite State Machine

The partitioned computation machine has evolved by adding a number of extensions to the basic finite state machine. In its simplest form, a partitioned computation machine has no more complexity than a simple state machine. The classical finite state machine has a set of states and transitions between states as its primary elements. Each transition between states is enabled when the event associated with the transition occurs. The classical FSM model requires that every input must permit some transition, even if it is only a self-transition back to the current state. Such a simple state machine is considered to be a partitioned computation machine, albeit a very simple one.

The Mealy modification to the language for the state machine is one of the oldest state machine modifications. It augments the state machine to generate output events by placing output events on the transition arcs between states. The Mealy model permits the state machine to respond to input by performing a transition to a new state, and also to produce output in response to input. As an example of a Mealy finite state machine, see Figure 2-1 on page 13. The extensions made by the Mealy FSM are the first extensions that make the state machine a useful computer specification tool, as it can describe the input /

output behavior of a computer system or any other type of machine. This simple model is sufficient to describe many types of behavior; in fact, it can describe any type of behavior by a "real" computer, since all computers have, in reality, a finite number of states. However, such a description is unwieldy for specifying large systems.

3.2 Basic Notation

The basic notation for the partitioned computation machine gives a unique name to every state. The pictoral representation for a state consists of a rectangle with rounded corners with a "flag" on one of the upper corners that indicates the name of the state. (See Figure 3-1.) In the figure are two states, state *A* and state *B*. Transitions from one state to another state are indicated with an arrow. All transitions are enabled by an input event; the input event enabling a transition is indicated next to the arrow for the transition in the pictoral notation. Output events, if any, are indicated next to the arc, preceded by a slash (/). In the figure are transitions from *A* to *A* for input *m* and from *B* to *A* for input *m*, producing an output of *n*.

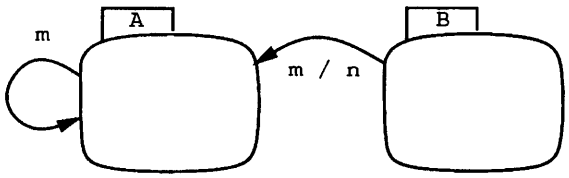


Figure 3-1: The basic notation of a PCM

The partitioned computation machine permits a single arc to contain any number of output events. All of the output events on a single arc are considered to occur simultaneously. As an example of this, see Figure 3-2. In the example there is a transition from state D to state C for input *g* that produces outputs of *x* and *w*. The PCM will produce output events for *x* and *w* in some order, determined non-deterministically. The ordering of these events affects the externally observable behavior of the machine. Furthermore, if these outputs are also inputs to the partitioned computation machine, the order in which

they are generated could affect the future behavior of the machine. The implications of the different possible orderings will be discussed further in section 3.8 on page 37.

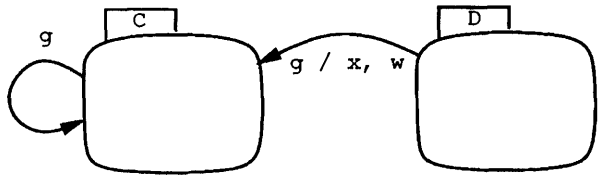


Figure 3-2: Simultaneous output events in a PCM

3.3 The Inclusion of a State Hierarchy

The next extension to the visual language for the Mealy state machine that is utilized in the partitioned computation machine is Harel's extension of adding a state hierarchy to the machine. The basic notation for representing a hierarchy of states in the partitioned computation machine is adopted from Harel's notation for statecharts. [Harel 87] In this extension, a tree hierarchy of states may be built, with each state, excepting a root state, being given a unique parent. The tree of states is represented by nesting contours representing the states. The contours for child states are drawn within the contour of their parent as in Figure 3-3. Parent states differ from leaf states in one manner: they may not be considered the current state of the machine. The current state of the machine must always be a leaf state of the tree. However, transitions may be defined as beginning or ending at a parent state.

3.3.1 Transitions from a parent state

The semantics of transition arcs from a parent state is that an arc emanating from a parent is exactly the same as if the arc were a set of arcs emanating directly from each of the child states of the parent. For an example of this, let us consider a simple hierarchical PCM with three child states, *A*, *B*, and *C*, and one parent state, *Y*. (See Figure 3-3a.) In this simple PCM, there is a transition for input *x* that starts from parent *Y* and ends at child *C*.

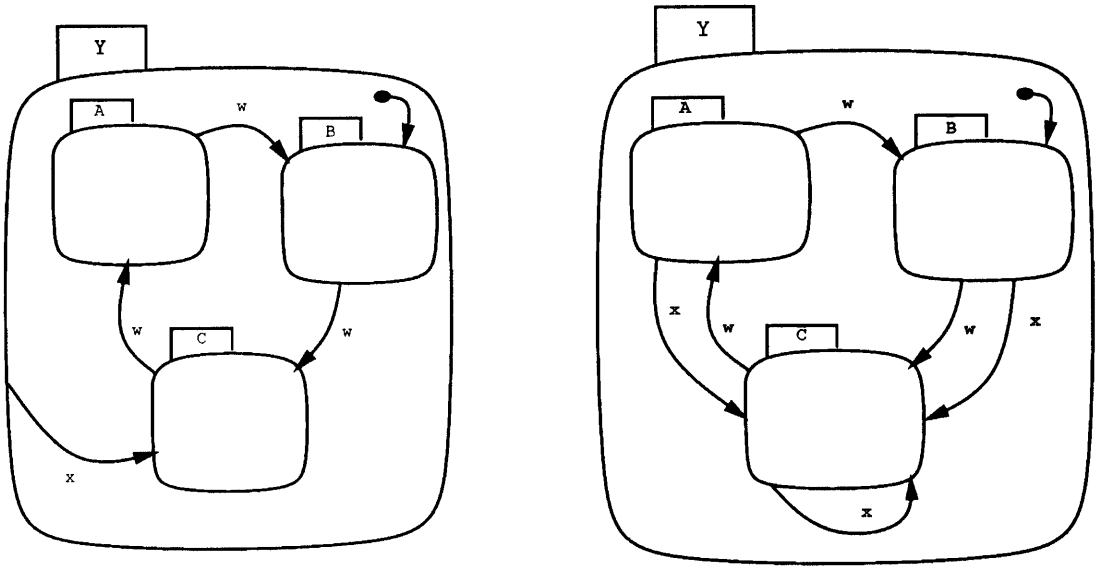


Figure 3-3: Transitions from parents in a PCM hierarchy

The transition from the parent *Y* is considered to have exactly the same meaning as three separate transition arcs for input *x* -- one that traverses from *A* to *C*, one that traverses from *B* to *C*, and one that traverses from *C* to *C*. (See Figure 3-3b.) In this manner, the arc defined at parent *Y* expresses the desired behavior that the input *x* always results in a transition to state *C*. Such notation permits behavior that is common to a number of states to be represented at a single parent state. Such an abstraction makes the behavior of the system easier to reason about, and also reduces the complexity of the pictoral representation of the machine.

3.3.2 Transitions to a parent state

The other form of transition involving a parent state is a transition to a parent state. Each parent state has a unique child state which is considered to be the default state for that parent. A parent's default state is indicated in the PCM notation by a transition emanating from a small black dot within the parent state. A transition arc which ends at a parent state is interpreted as being a transition to the default child of the parent state. The semantics

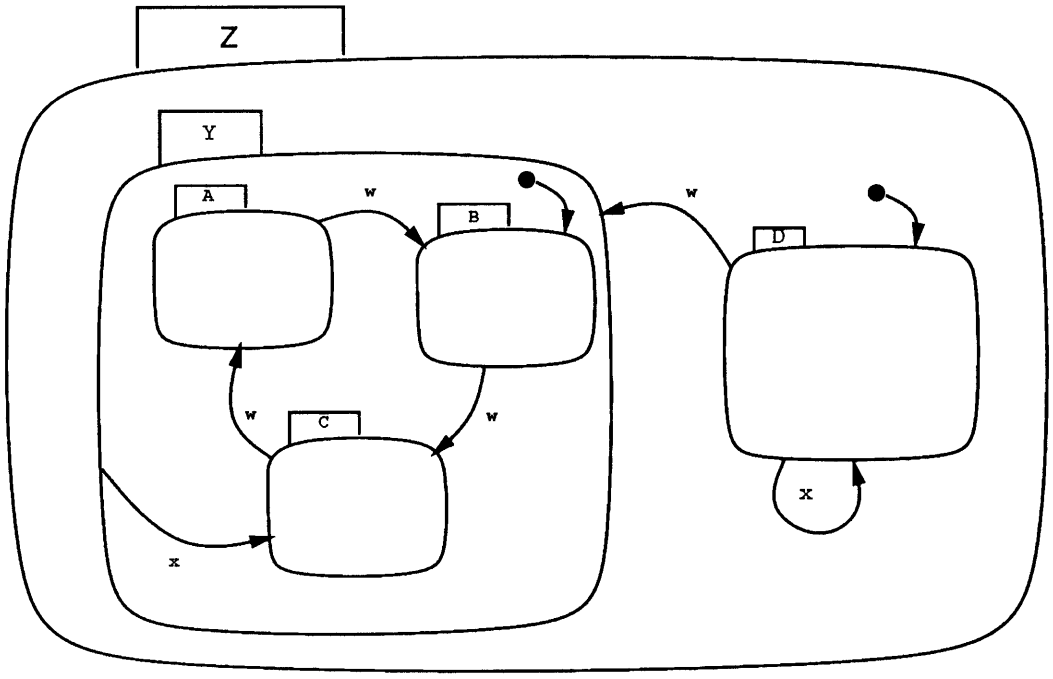


Figure 3-4: Transitions to a parent state in a PCM hierarchy

provided by this notation is that a transition which has a parent state as its destination is exactly the same as a transition directly to the default state of the parent. The reason for permitting transitions to a parent state when they are exactly the same as transitions to the default child of the parent state is that such a transition permits the parent state to encapsulate the default state -- this allows the state from which the transition emanates to abstract away from the internal state of the transition destination. As an example of such a transition, see Figure 3-4. This figure is an extension of Figure 3-3a with the additions of another level in the hierarchy and a transition from state *D* to parent state *Y* for input *w*. The diagram also indicates that state *B* is the default child of state *Y*. In this diagram, the transition from *D* to *Y* for input *w* gives the same behavior as a direct transition from *D* to *B* for input *w*. The distinction, however, is important to maintain in case future changes are made to the diagram.

Default states of a parent are also used to determine the starting state of the partitioned computation machine. When a machine first begins execution, it begins as if

there were a transition to the root state. This root state would provide a default state which would actually be the starting state. (Of course, the default state of the root state could itself be a parent state with a default state, and etc.)

3.3.3 Consistency requirements with parent transitions

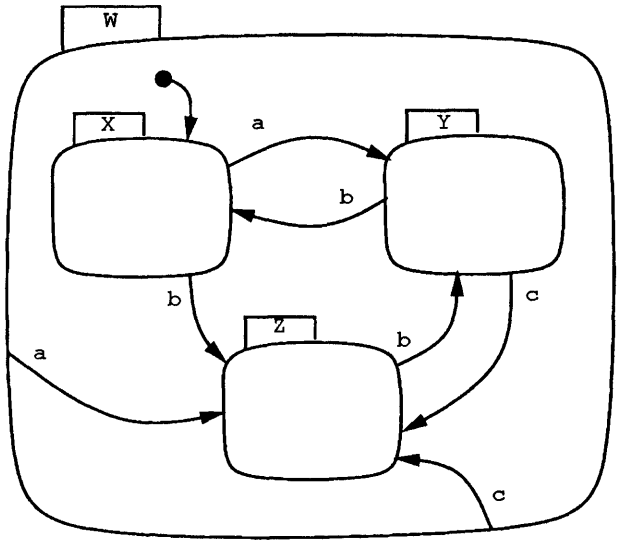


Figure 3-5: Inconsistencies between parent and child transitions

The ability to define transitions at the level of parent states as well as at the level of the child states permits inconsistencies to arise if transitions are defined at both levels for the same inputs. Figure 3-5 shows a case where the transition *a* is defined both at the child state *X* and the parent state *W*. The two transitions for *a* from *X* and *W* give different destination states. The primitive set elements of the PCM formal model are capable of representing such an inconsistency (see section 4.1); however, in order to provide deterministic behavior, the PCM imposes constraints to prohibit these inconsistencies. (Ensuring consistency for transitions defined for the same inputs is discussed further in section 4.2. If the transitions have the same destinations, having the redundant definition of the transition would be acceptable. An example of such a redundant definition exists with the transitions labeled *c* in the same figure.

In an implementation of the PCM, inconsistencies of this type can be easily detected by examining the ancestors and descendants of the state for other transitions defined for the same input; an error message indicating that such an inconsistency is present could be provided to the user. (See section 4.2 for a constraint forbidding this type of inconsistency.) Some specification languages permit such inconsistencies, either by non-deterministically choosing which of the transitions to consider [Lynch 88], or by having some deterministic technique for determining which transition overrides the other [Rumbaugh 88]. In the partitioned computation machine, we choose to avoid such inconsistencies, however, to permit deterministic behavior and also to view behavior defined at the level of a parent as a required behavior for all children, as discussed in section 2.4, beginning on page 16.

3.4 Variables Associated with States

The next extension to the hierarchical partitioned computation machine is to associate variables with the states. This extension is similar to an extension for parameterizing states in statecharts suggested by Harel. [Harel 87] However, the partitioned computation machine includes this feature as part of the language and formal model, using an entirely different syntax than the one Harel suggests.

Each state can have variables associated with it. These variables maintain state information that the designer has chosen not to represent as a grouping of separate states. The use of variables associated with states can greatly simplify the state diagram if most of the variable values result in the same control flow characteristics. Each variable has a scope limiting the visibility of the variable to the state to which it is associated and any descendants of that state. In the pictorial representation for partitioned computation machines, variables are listed in the state below the flag giving the name for the state, a range or list of values which each variable can assume, and an initial value for the variable. Variables that have not yet been explicitly assigned will have their initial value. Variables

are accessed and modified by the transition arcs. In addition to responding to input events and possibly generating output events, each transition arc can have a predicate clause which checks the value of any visible variables, enabling the transition only when the predicate is true. Each arc can also have variable assignments that are performed when the transition is followed during execution. The only requirements upon transition arcs is that they have a source state, an input event, and a destination state.

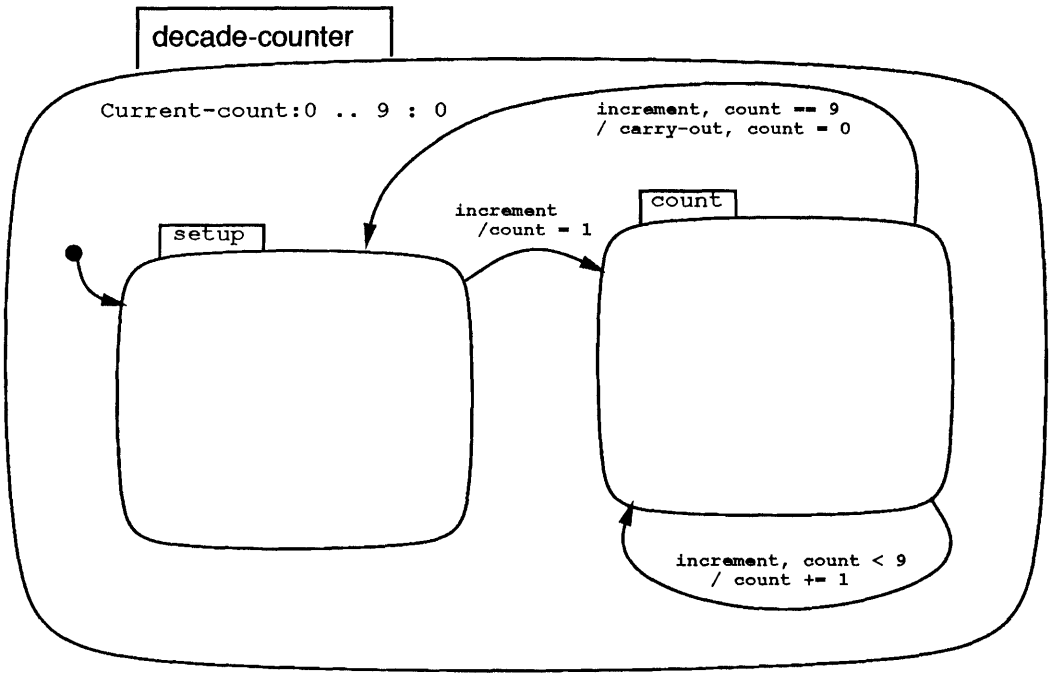


Figure 3-6: A simple decade counter

Figure 3-6 provides an example where a simple hierarchical partitioned computation machine represents a counter which repeatedly increments the value of the *count* variable upon each occurrence of the *increment* input event. The counter commences at the initial value of zero, and whenever the count increments from a value of nine, a *carry-out* output event is generated and the counter restarts at the initial value of zero. This example demonstrates how the use of variables can greatly simplify the state diagram. In this case, although the variable *count* can take on ten different values, the only criterion affecting the control flow of the partitoned computation machine is whether or not the variable has a

value of exactly nine. If this condensed form of variable representation were not available, it would be necessary to pictorally depict ten different states for each of the ten possible values of the counter. The many states required to represent variable values would serve to clutter the diagram and obscure the control flow which the partitioned computation machine seeks to make evident; the condensed representation of variables, however, permits the specification to depict the control-affecting aspects of the variable pictorally without necessitating a distinct pictoral state for each possible value.

3.5 Concurrent States in the Hierarchy

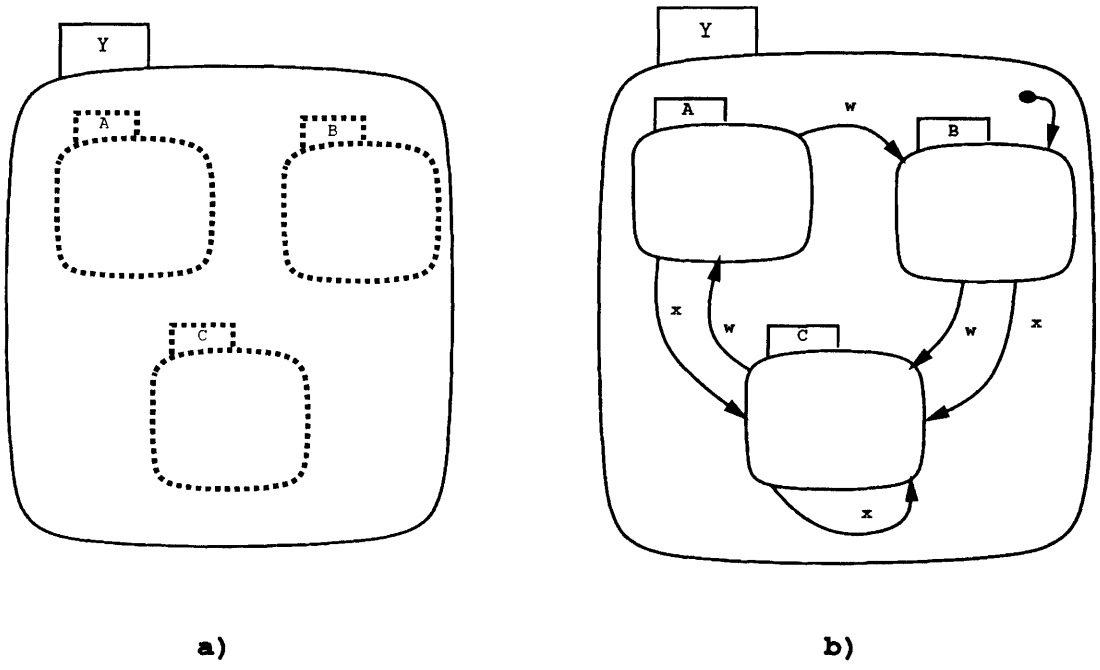


Figure 3-7: An example of PCM concurrent notation

Another significant feature in the partitioned computation machine is the ability to express the concurrent operation of states in a manner similar to Harel's cross-product notation. The partitioned computation machine permits the children of a parent state to be operating concurrently. The notation for describing concurrency is to outline the concurrent child states with dashed lines as in Figure 3-7a instead of enclosing them in the

normal solid lines as would be the case for the non-concurrent children in Figure 3-7b. This notation is used instead of that of statecharts to permit the parent and each of the concurrent children to have local variables without requiring both a dashed line separating children and a box around each child as in figure 3-8. [Harel 87]

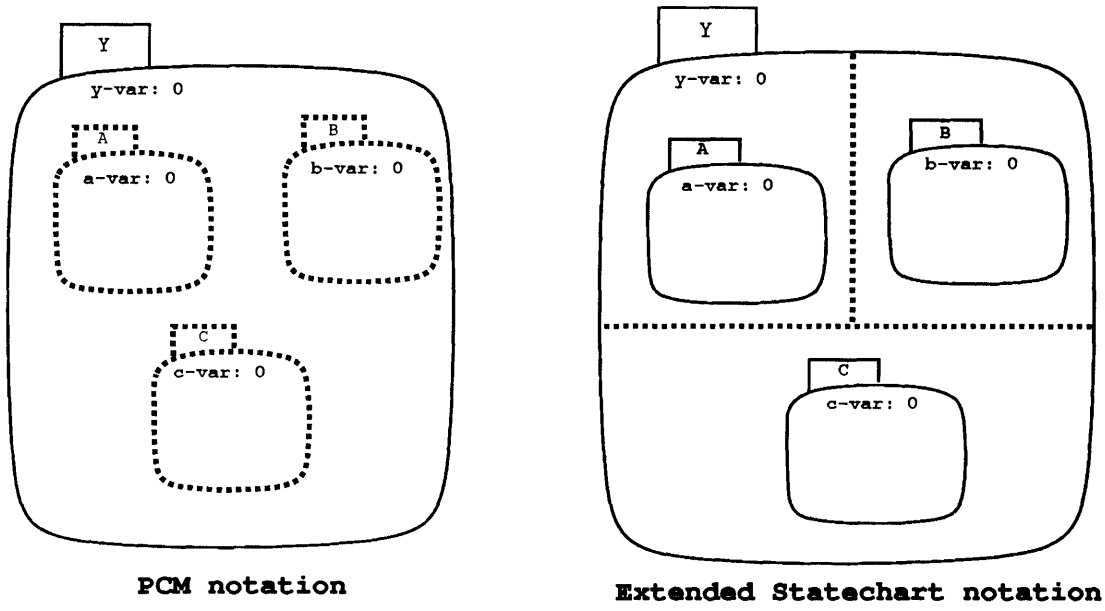


Figure 3-8: PCM vs. Statechart's notation extended to include variables

In a state with concurrent children, the current state of the system actually consists of a set of states that are active concurrently. In figure 3-9, the current state would be considered to be a set of states such as $\{A, M, X\}$, or $\{B, N, X\}$. The semantics of transition behavior when the current state is a tuple of states is that an arc from each element of the tuple may be enabled. Transitions in all concurrent states for the same input events appear to occur simultaneously. The simultaneous execution of such transitions will lead from one tuple of states to another tuple of states. As an example of this behavior, consider a case where the current state of Figure 3-9 is $\{A, M, X\}$. When an input event f is received, the next state will be $\{B, M, Y\}$ because the transitions from A to B, M to M, and X to Y will all occur simultaneously when input event f is received.

One advantage of the concurrent notation is that many fewer states need to be

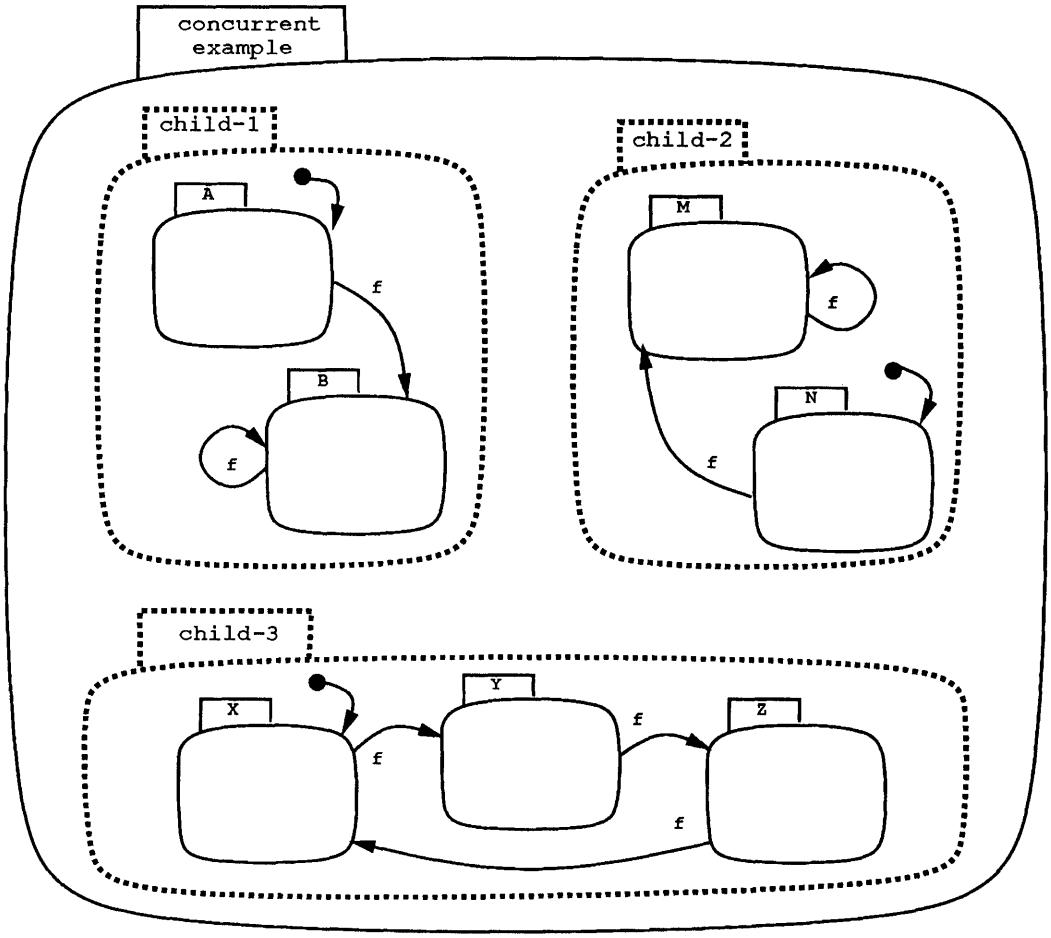


Figure 3-9: An example of PCM concurrent notation

depicted in order to describe the actual control flow than would be depicted in a fully-expanded notation. In addition, the concurrency aspect of the notation permits the designer to use the natural idea of parallel execution to describe the system. This is especially useful if a system has multiple modes or values that could be either on or off simultaneously. Consider an editor which has a insert mode, a caps-lock mode, a control-character mode, and an automatic justification mode. Depending on whether or not these modes are active, different behavior will be provided by the system. The cross-product notation permits the user to think about the modes as being separate, but active concurrently. This is an advantage in reasoning about the systems, and in preventing the enumeration of all sixteen possible combinations of the modes.

3.6 Consistency Requirements with Concurrent Children

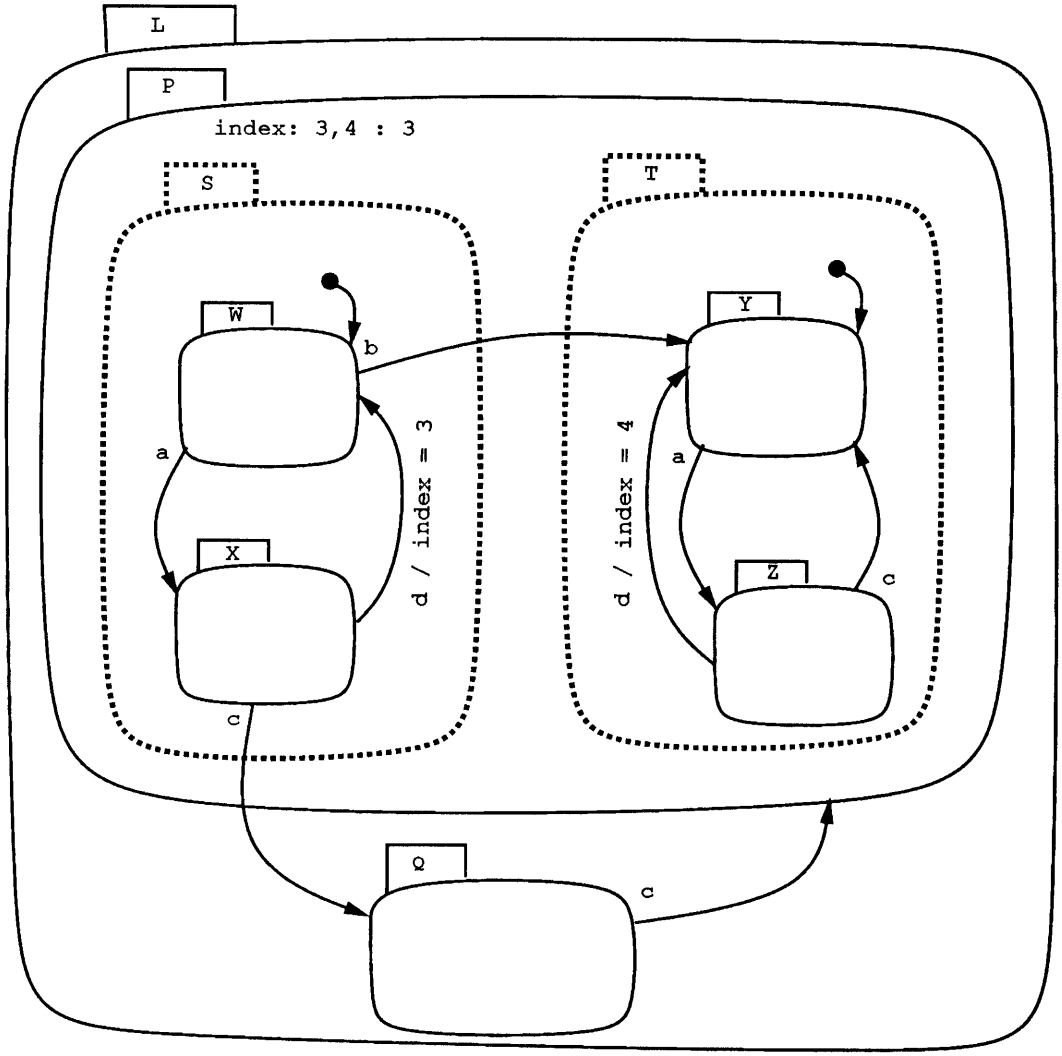


Figure 3-10: Inconsistencies between concurrent transitions

The ability to define transitions for the same inputs on each of the concurrent children in a partitioned compuation machine permits inconsistencies to arise if such transitions have different destination states or modify the same variables. Figure 3-10 shows a case where the transition *a* is defined on two concurrent states, *W* and *Y*. These two transitions are consistent because they both give transitions that maintain the concurrency. Assuming the current state is (*W*, *Y*) and *a* is recieved as input, the next state will be (*X*, *Z*). However, the transition for *b* from the state *W* is disallowed as it violates the concurrency of the states *S*

and T, by starting and finishing in states which are concurrent. Such transitions are prohibited in the PCM. As another example of an inconsistent transition, the two transitions for c from X and Z conflict because the transition from X indicates that the concurrent states should be exited, with the new state being simply Q, while the transition from Z would maintain the concurrent pairings with the next state being (*, Y). As with inconsistencies with parent and child states mentioned in section 3.3.3 on page 24, the sets used in the formal model for the PCM are capable of representing such an inconsistency, but constraints are imposed on the model to forbid this case.

An implementation for the PCM will prohibit the existence of the inconsistencies outline above. A more subtle inconsistency exists in the same figure where the transitions from a current state of (X, Z) for input d give conflicting variable assignments for the *index* variable. One way to require this form of consistency is to prohibit arcs which could be simultaneously enabled from making assignments that reference the same variable. The more general problem of ensuring that multiple assignments resolve to the same value could be very difficult. The possibility of including such a consistency check is discussed in section 5.1.1.1. None of these inconsistencies are allowed in a well-formed partitioned computation machine. In an implementation of the PCM, all of these inconsistencies could be detected by examining the other concurrent states for conflicting transitions that are simultaneously enabled; an error message indicating that such an inconsistency is present could be provided to the user. (See section 4.2 for a constraint forbidding this type of inconsistency.) As with inconsistencies between parents and children, the partitioned computation machine chooses to prohibit such potential problems and require deterministic behavior.

3.7 Composition

A primary strength of the partitioned computation machine model is the ability to combine several partitioned computation machines into a single new machine. The resulting machine permits the behavior of the composition to be reasoned about as a combination of separate parts instead of requiring the entire machine to be considered as a whole. Additionally, composition permits general-purpose PCMs to be developed for use as building blocks for the construction of other PCMs. For example, a specification for data input from a keyboard device could be developed once, to be used by any application which needed to specify behavior for data input.

The partitioned computation machine permits two types of composition, additive composition and multiplicative composition. Additive composition takes two or more partitioned computation machines and produces a new PCM by giving all of the components a common parent. This form of composition permits the behavior of the resultant machine to take on the behavior of any of the composed elements one at a time. Multiplicative composition produces a new PCM from two or more PCMs by making each component a concurrent child of a common parent. The essential difference between the two composition techniques is in the number of control threads maintained in the composition. Additive composition keeps one control thread, permitting transitions from each component machine to the others, but permitting only one component to be active at any time. Multiplicative composition, however, lets each component child operate concurrently, permitting multiple control threads. This corresponds to grouping the component PCMs as concurrent children of a common parent.

3.7.1 Additive composition

Additive composition is a function that accepts some number of partitioned computation machines and produces a new PCM from them. The resultant PCM consists of a parent state with each of the given machines as a child state. In addition to requiring some number of PCMs as arguments, the additive composition function also requires a name for the new parent state and an indication as to which of the argument PCMs will be the default child for the composition. Furthermore, the composition function can accept optional arguments which associate variables with the new parent state, new input events, new output events, and new transitions describing how the current state can change from one of the children to another. The only restrictions upon the composition is that all the new events are disjoint from ones already defined in the components, and that the names of states and variables are unique across all the components. Further constraints can be

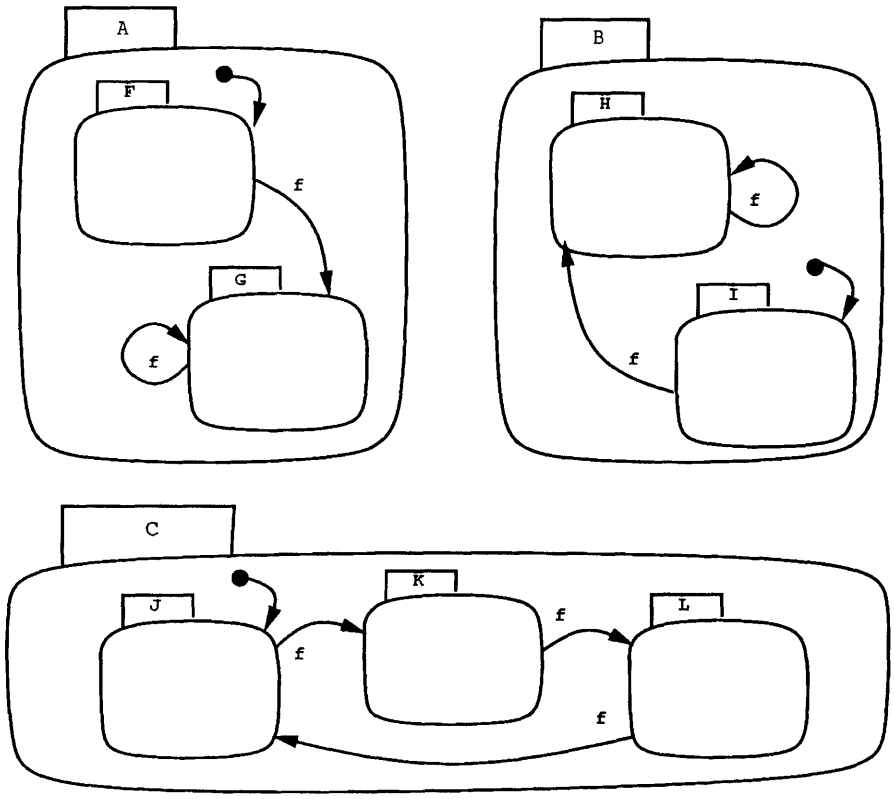


Figure 3-11: Three PCMs to be additively composed as an example

imposed to prohibit the possibility of inconsistencies arising between child and parent transitions as discussed in section 3.3.3 on page 24. These further constraints are also discussed in section 4.2.

As an example of composition, consider a composition of the three partitioned computation machines shown in Figure 3-11. These three machines will be given as arguments to the additive composition function, along with *Additive-Composition-Example* as the name for the new parent state. Also, PCM *A* is chosen to be the default child for the composition. No new variables are indicated in this composition, but new transitions are given for input *h* from *A* to *B*, from *B* to *C*, and from *C* to *A*. The resultant PCM from this composition is shown in Figure 3-12.

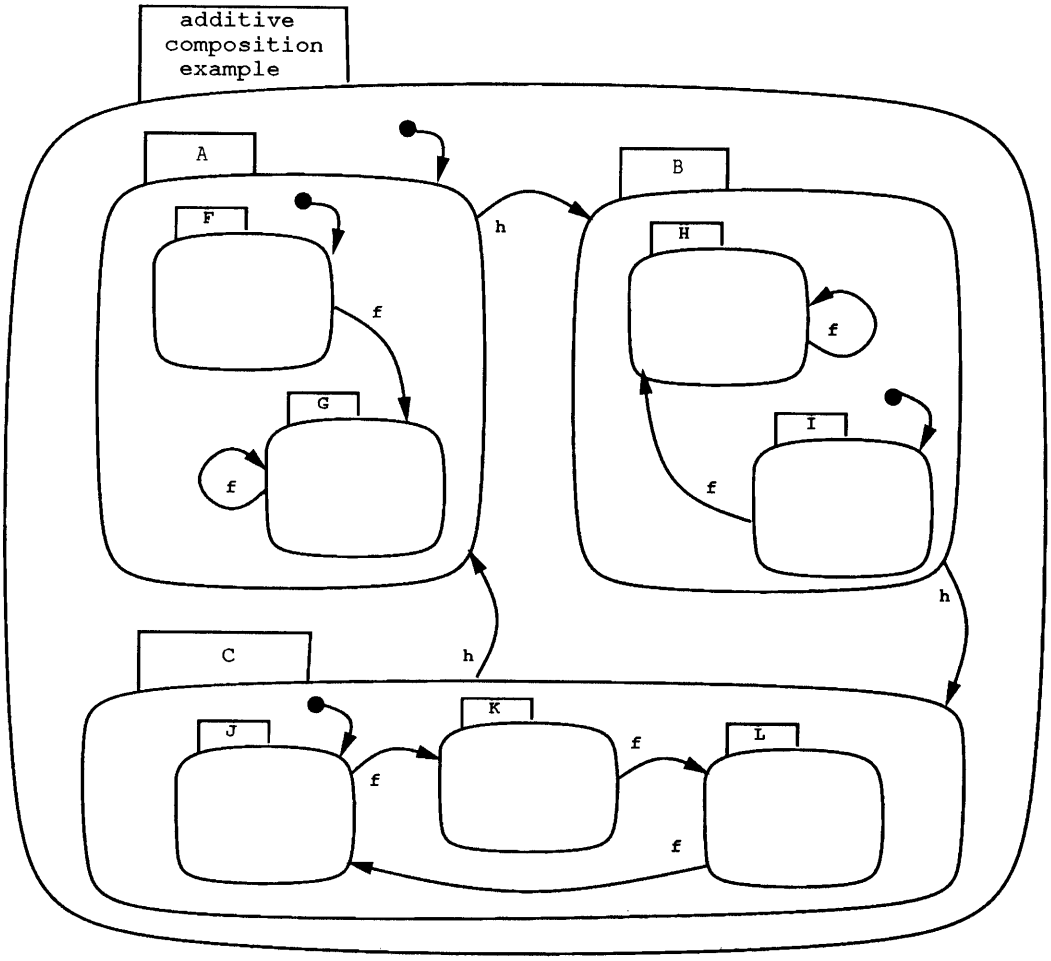


Figure 3-12: The additive composition of the three PCMs of Figure 3-11

3.7.2 Multiplicative composition

Multiplicative composition is a function that accepts some number of partitioned computation machines and produces a new machine from them. The resultant PCM consists of a parent state with each of the given PCMs as a concurrent child state. In addition to requiring some number of PCMs as arguments, the multiplicative composition function also requires a name for the new parent state. Furthermore, the composition function can accept optional arguments which associate variables with the new parent state, new input events, new output events, and new transitions that handle behavior at level of the new parent. The only required restriction upon the composition is that all the variables, and statenames for the components and new definitions be unique. As with additive composition, additional constraints may be placed on the arguments of the multiplicative

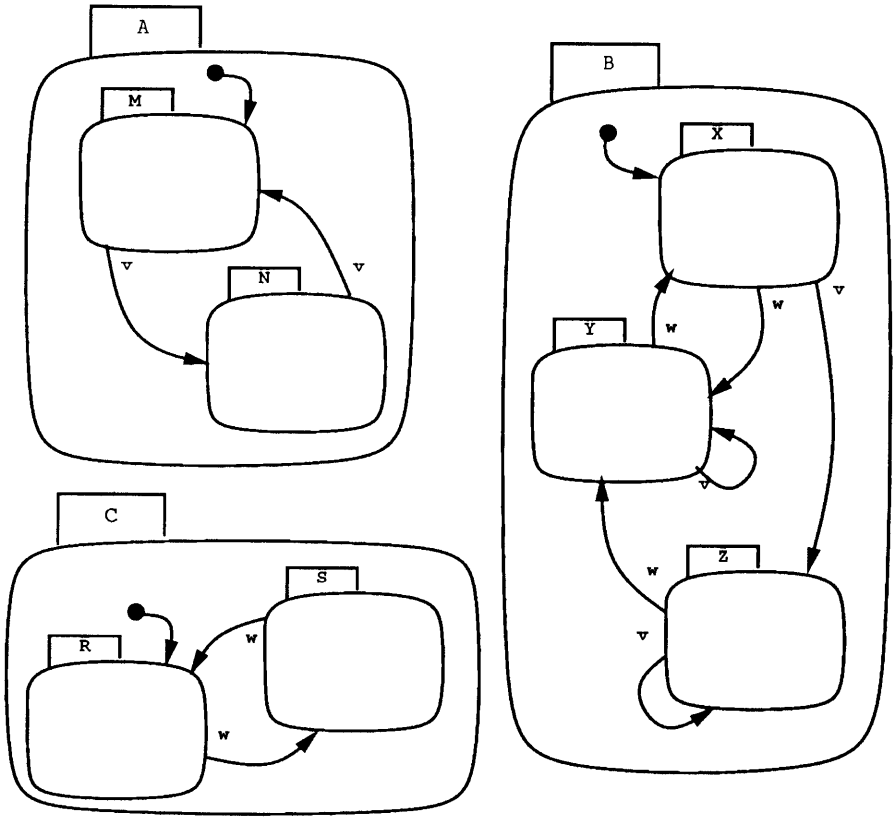


Figure 3-13: Three PCMs to be multiplicatively composed as an example

composition to prevent inconsistent transition behavior both between a parent and children and between concurrent childen. These constraints are discussed in section 4.2. Another potential problem with multiplicative composition is that the composition may produce simultaneous outputs for some inputs where the original components do not. The consistency problem which may result from such simultaneous outputs is discussed in section 3.8 on page 37. Resolution of this consistency problem is left as a topic for future work in section 5.1.2.

As an example of composition, consider a composition of the three partitioned

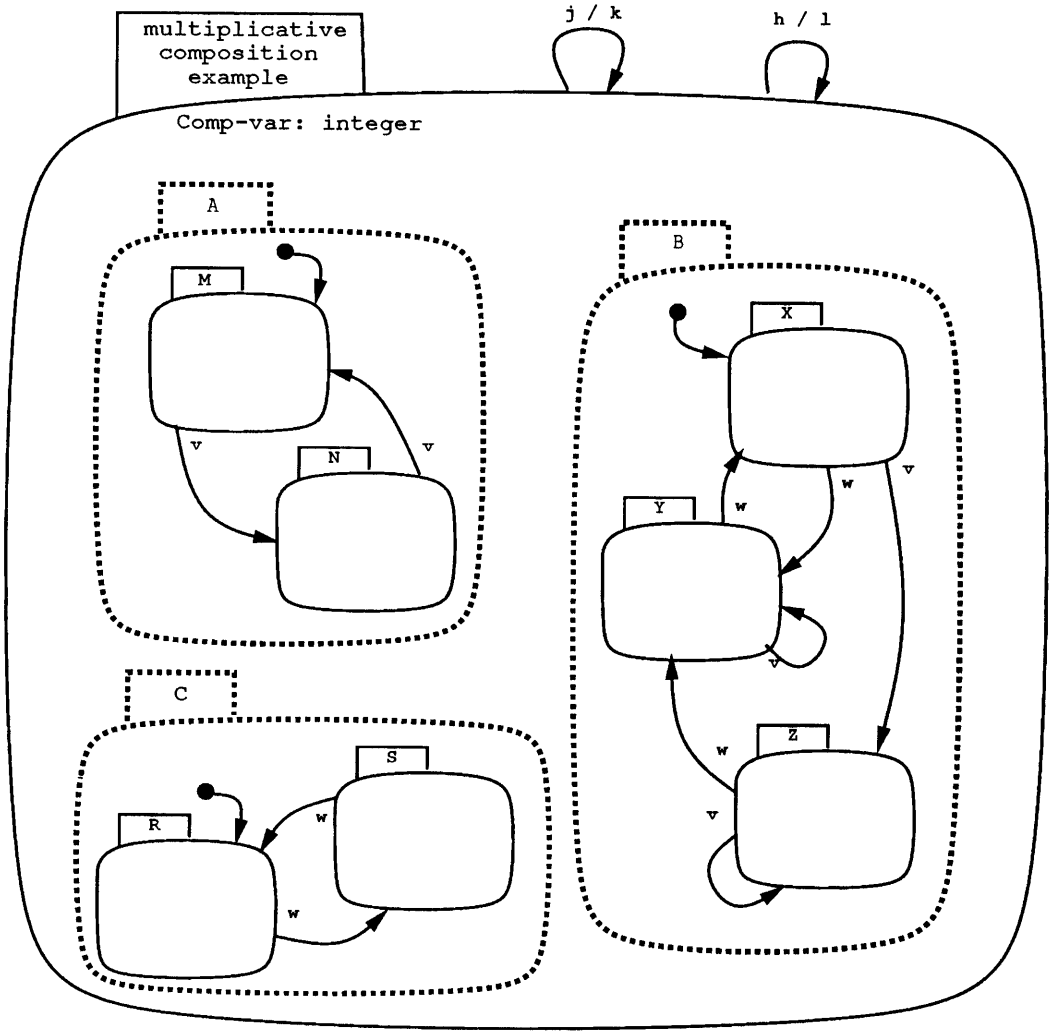


Figure 3-14: The multiplicative composition of the three PCMs of Figure 3-13

computation machines shown in Figure 3-13. These three machines will be given as arguments to the multiplicative composition function, along with *Multiplicative-Composition-Example* as the name for the new parent state. Also, a new variable, *comp-var*, which can contain any integer value, is added to the parent state. New transitions are also given to the composition function indicating transitions just on the parent state for input *j* producing *k* as an output, and for input *h* producing *l* as an output. The resultant PCM from this composition is shown in Figure 3-14.

3.8 Consistency Issues with Simultaneously Generated Events

The partitioned computation machine generates simultaneous events in a number of cases, as has been pointed out in sections 3.2 and 3.7.2. If a partitioned computation machine is responding to simultaneous events, however, the events are considered in some serial order, chosen non-deterministically. It is possible that the chosen order of serialization could affect future behavior of the machine.

To demonstrate this point, consider the example PCM in Figure 3-15. Assuming that the current state is state *A*, and the transition *x* is received, what would be the sequence of states followed? Both possible input serializations of *y* and *z* generated by the transition from *A* to *B* for input *x* must be considered. If the non-deterministic serialization is the ordering (*y*, *z*), then the transition from *B* to *A* will be taken for input *y* and then the transition from *A* to *D* will be taken for input *z*, resulting in *D* being the current state after the transitions are considered.

Considering the other case where the input serialization in this example is the ordering (*z*, *y*), the transitions from *B* to *C* for input *z* will be taken, and then the transition from *C* to *C* will be followed for input *y*, resulting in *C* being the current state after the transitions are considered. In this case, the external input of *x* results in two different states for the partitioned computation machine after both possible input orderings of the resultant

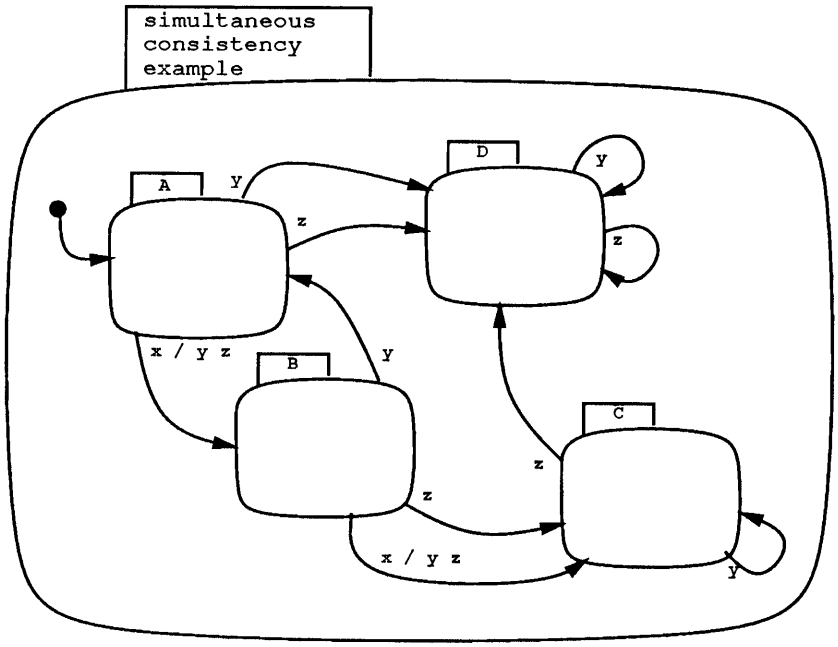


Figure 3-15: Serialization of simultaneous events could affect future behavior

events y and z are considered. A discrepancy such as this could lead to non-deterministic results for a fixed simultaneous input, something which would be highly undesirable in an environment requiring deterministic results. The non-deterministic ordering of simultaneous events does not always result in non-deterministic results, however. Consider the situation if the current state is B when the external input x is received. In such a case, state D will be reached after y and z are handled, regardless of the chosen order of serialization of the events. This second case would pose no problems for a deterministic system, provided that the ordering of y and z is insignificant outside the machine. However, if this machine were to be multiplicatively composed with another machine which used y and z as inputs, the other machine may provide inconsistent behavior for different orderings of the y and z events.

The fundamental question regarding the possibility of inconsistent behavior resulting from the generation of simultaneous events concerns the restrictions, if any, that should be placed on the partitioned computation machine to guarantee consistency. Preventing the model from generating simultaneous events would significantly decrease the usefulness of

multiplicative composition in the model, as machines which produced output depending on the same inputs would require the user to serialize all events, even if no inconsistency were to result. The proposed solution in the partitioned computation machine, however, is to permit the model to represent the fact that the possibility for inconsistency is present so that an implementation of the partitioned computation machine could warn the user of the potential problem. The user could then ignore the warning if consistency is unnecessary for his application, or take steps to correct the inconsistency. An implementation could search the partitioned computation machine to check if all possible orderings of simultaneously generated events result in the same state of the machine and perform the same operations on all variables. Such a search could consider only the handling of the initially generated simultaneous events, or could consider the effects of events generated in response to the initial events before requiring a consistent state for all possible serializations of the simultaneous events. A search of this form is made more difficult where predicates for variables exist. Different levels of consistency could be determined depending on the degree of consistency required by the user; this issue is discussed further in section 5.1.2 as a topic for future work.

Chapter 4

The Formal Partitioned Computational Model

The partitioned computational machine's most important quality is the fact that it is supported by a formal model. The visual representation has been designed with the goal of developing a corresponding formal model that provides a meaning for every visual construct. The formal representation of the PCM describes the structure of the state hierarchy and associated variables, groups the partitioned computation machine's events into input and output events, and defines the execution semantics for the partitioned computation machine. The first section of this chapter describes the formal representation of the pictoral information contained in the PCM diagram. The second section places constraints on the model to enforce consistency; the third section formally describes the execution semantics of the machine, and the fourth section provides the formal descriptions of composition.

4.1 The Formal Representation of Pictoral Information

The formal model for the partitioned computation machine's pictoral information consists of a collection of sets and mappings. Each set or mapping represents a portion of the information provided by a PCM diagram. The formal representation must maintain information regarding the state hierarchy, the input and output events, the usage of variables, and the transitions indicated in the diagram.

The first four elements in the formal model serve to encapsulate the hierarchy of states. The fifth and sixth elements group the possible events of the PCM as input and output events. Elements seven through nine represent the variables which are associated with states. The tenth and final element of the formal model describes the arcs that are present in the PCM diagram.

The representation of the hierarchy of states consists of sets and mappings which signify the states in the hierarchy, and the parent relationships among them. The events recognized by the PCM are simply grouped into events that can be used as input or as output. The usage of variables is slightly more complex, however. The model assumes that a language exists to describe the type, conditional predicate, and variable assignment of variables, yet does not restrict the language used for this description. To avoid restricting the PCM's expressive capabilities by a specific language, the general concepts of values, variable names, types (where a type is a set of values), conditionals, and variable assignments are used. For a partitioned computation machine, X, the following would be defined:

- V(X) : the set of variables used by the PCM X,
- Val(X) : the set of possible values for variables,
- Type(X) : $2^{Val(X)}$(the power set of Val(X)),
- Env(X) : V(X) $\rightarrow$ Val(X),
- Pred(X) : $2^{Env(X)}$ (the power set of Env(X)),
- VAsg(X) : Env(X) $\rightarrow$ Env(X).

The above definitions warrant some additional explanation. Each element of Type(X) is a valid type for a variable. In this context, a *type* is simply a subset of the values in Val(X) which a particular variable can assume. An element of Env(X) is an *environment*, which maps each of the variables of the PCM into a value. An element of Pred(X) is referred to as a *predicate* and consists of a set of environments. An environment that is an element of a predicate is said to *satisfy* the predicate. This set of environments contains all environments that satisfy the predicate. Finally, an element of VAsg(X) is a *variable assignment* which produces a new environment, given an initial environment. Conceptually, a variable assignment assigns new values to variables.

Also, to permit the usage of these sets, an additional function, *Variables*(asg) where

asg $\in$ VAsg(X), is defined as the set of variables in asg which either have a different new value assigned to them, or can affect the new values assigned to other variables. (If asg were a set of assignment statements in a language, Variables(asg) would be the set of variables referenced on either the right- or left-hand-sides of the equation.) Finally, for two predicates to be *Overlapping*($pred_1$, $pred_2$) means that [$pred_1 \cap pred_2 \neq \varnothing$]. To state this simply, two predicates are overlapping if there are any environments for which they are both satisfied.

The specific use of these sets and mappings for variable usage is defined as part of the PCM. The sets and mappings make the formal model more complex, but permit any language to be used to describe the use of variables. In the examples presented in this document, simple C-like syntax has been used for variable access and assignments, however, the model does not require such a syntax. Definitions of specific languages for variable manipulation and issues accompanied with their use are left as topics for future work as discussed in section 5.1.

Once the state hierarchy, events, and variable usage are defined, the only remaining element of the formal model is a transition relation which contains the information present on the transition arcs of the pictoral diagram.

The formal model for a partitioned computation machine X consists of the ten following elements (continued onto the next page):

1. A set of states, S(X), with the distinguished root element, $s_0(X) \in S(X)$
2. A parent mapping, $P(X) = S(X) - \{s_0(X)\} \rightarrow S(X)$ which configures the set of states into a tree, with $s_0(X)$ as the root.

   Define the set of leaf states, LS(X), the elements of S(X) that have no children as indicated by P(X).

   Define the set of internal states, IS(X), the elements of S(X) with both parents and children as indicated by P(X).

   Define the set of child states, $CS(X) = LS(X) \cup IS(X)$

   Define the set of parent states, $PS(X) = IS(X) \cup \{s_0(X)\}$

3. A partition of PS(X) into AP(X) and MP(X), the parent states with additively composed and multiplicatively composed children, respectively.

4. A default-child relation, $DC(X) \subseteq PS(X) \times CS(X)$, relating each additive parent state with one of its children and each multiplicative state to all of its children. (Each default child state must be one of the parent's children; formally, $\forall (p, c) \in DC(X), (c, p) \in P(X)$.)

   Define the set of default leaf states, DLS(X), the elements of LS(X) that are default children of some state as indicated by DC(X).

   Define Default-Leaves-of-Subtree(s) as the elements of DLS(X) that are descendants of the state $s \in S(X)$ including s if $s \in S(X)$.

5. A set of input events: IE(X)

6. A set of output events: OE(X)

7. A set of variables: V(X)

8. A variable-state mapping, $VS(X): V(X) \rightarrow S(X)$ (This indicates the state with which the variable is associated.)

9. A variable-type mapping, $VT(X): V(X) \rightarrow Type(X)$ (This indicates the type of the variable.)

10. A transition relation: $TR(X) \subseteq S(X) \times IE(X) \times Pred(X) \times 2^{OE(X)} \times VAsg(X) \times S(X)$

    (There is one element of the transition relation for each arc in the PCM diagram.)

The preceding ten elements serve to contain all the information present in the PCM pictoral representation. The formal model given above serves to represent all partitioned computation machines, including additively or multiplicatively composed machines.

Further notation is introduced to permit discussion of the elements of the transition relation of the PCM. For an element *t* of a transition relation TR(X), "dot" notation will be used to refer to the components of the transition. The starting state of a transition (position 1) is referred as t.s. The input event for a transition (position 2) is $t.\pi$. Similarly, the remaining components are t.pred, t.oe, t.asg, and $t.s'$.

4.2 Constraints Imposed on the Model

There are a number of constraints that must be imposed upon the PCM model in order to limit the forms of transitions that may be included in the diagram. These constraints serve to ensure that all transitions present in the PCM can be handled by the execution semantics, as will be defined in the next section. There are three forms of transitions that are avoided. These three forms of transitions are transitions which start and end at states that can be active concurrently, transitions that give conflicting destination states, and transitions that give conflicting variable assignments.

In order to define these constraints, some further definitions are needed:

- For two states, members of S(X), to be Parent-Related(s_1, s_2) means that $s_1 = s_2$, or s_1 is an ancestor of s_2, or s_2 is an ancestor of s_1 for P(X).
- For two states, members of S(X), to be Concurrent-Related(s_1, s_2) means that s_1 and s_2 are not parent-related and the least-common-ancestor of s_1 and s_2 in P(X) is a member of MP(X).

The first definition indicates that two states are considered to be parent-related if one is an ancestor of the other, or if they are the same state. (A state is always parent-related to itself.) The second definition indicates that states are concurrent-related if they are distinct states that can be active concurrently.

The formal definitions (further explanation to follow) for the constraints are:

Constraint #1:

If it is the case that $t \in TR(X)$ then it is not the case that Concurrent-Related(t.s, t.s′).

Constraint #2:

If it is the case that $t_1 \in TR(X)$, and $t_2 \in TR(X)$, such that $t_1.\pi = t_2.\pi$, and such that Overlapping(t_1.pred, t_2.pred), where either Parent-Related(t_1.s, t_2.s) or Concurrent-Related(t_1.s, t_2.s), then it must be the case that Concurrent-Related($t_1.s'$, $t_2.s'$) or ($t_1.s' = t_2.s'$).

Constraint #3:

If it is the case that $t_1 \in TR(X)$, and $t_2 \in TR(X)$, such that $t_1.\pi = t_2.\pi$, and such that

Overlapping($t._1$.pred, t_2.pred), where either Concurrent-Related(t_1.s, t_2.s), or Parent-Related(t_1.s, t_2.s), then it must be the case that [Variables (t_1.asg) $\cap$ Variables (t_2.asg) = $\varnothing$].

The first constraint is used to prohibit transitions among concurrent substates. The second constraint serves to prohibit transitions which are defined for the same state and input event with overlapping predicates from giving different non-concurrent destinations, while the third constraint prohibits such transitions from referencing the same variables.

4.2.1 Constraint #1: to prohibit transitions among concurrent states

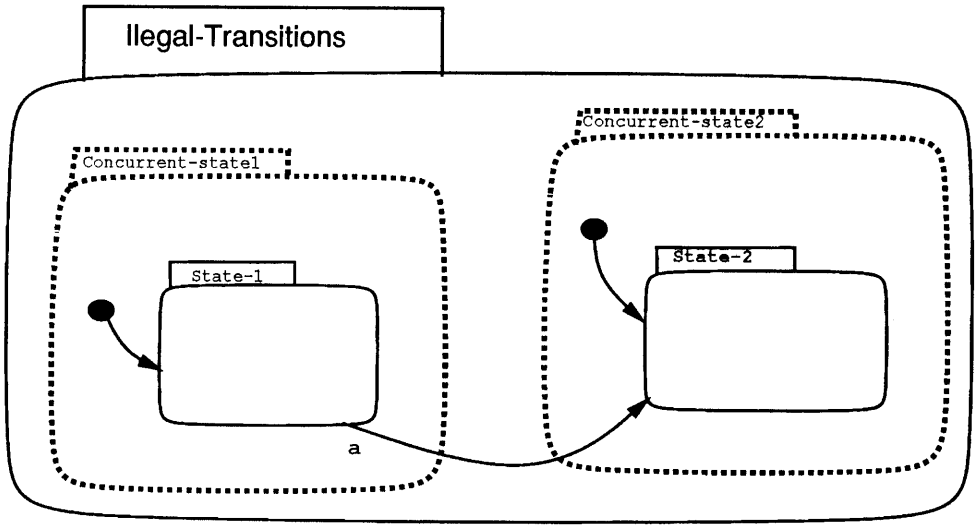


Figure 4-1: Example of an Illegal Transition Between Concurrent States

Constraint #1 simply prohibits transitions that have a source and destination state that could potentially be active concurrently. As an example of a transition that would be disallowed by this constraint, consider a transition as for input *a* from state *State-1* in Figure 4-1. This transition does not maintain the concurrency between the two components and would not be permitted.

4.2.2 Constraint #2: to require a deterministic choice of next state

In order to prevent some cases that give rise to a non-deterministic choice of a destination state during the execution of the PCM (see section 4.3.2), constraint #2 is imposed upon the formal model. Fulfilling such a determinism requirement would be desirable for implementations of the PCM that are used for real-world systems.

The first "or" case of constraint #2 prevents two states that are parent-related from having transitions with the same input and overlapping predicates but different non-concurrent destination states. The constraint does not rule out multiple arcs from a single

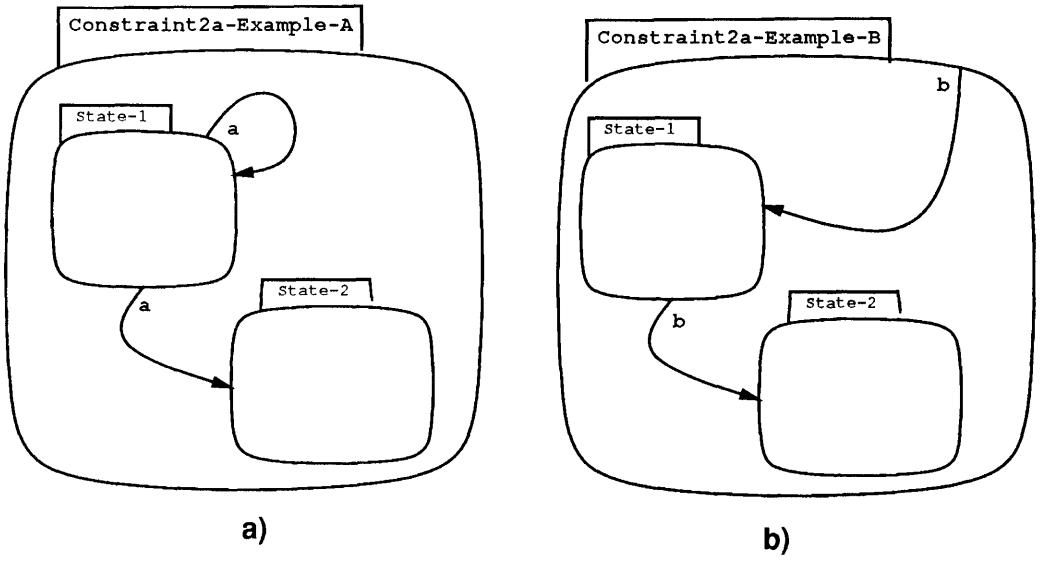


Figure 4-2: Examples of Syntactic Constraint Violations in a PCM

state for the same input and overlapping predicates, as long as the arcs have the same destination or concurrent destinations. As an example of an arc forbidden by this constraint, in Figures 4-2a and 4-2b, respectively, the pairs of transitions for inputs *a* and *b* both violate this constraint since each pair is enabled from *state-1*, giving conflicting destination states.

The second "or" case of constraint #2 prevents concurrent states from having transitions with the same input and overlapping predicates but distinct destination states that are non-concurrent. For example, in Figure 4-3, the pair of transitions for input *d*

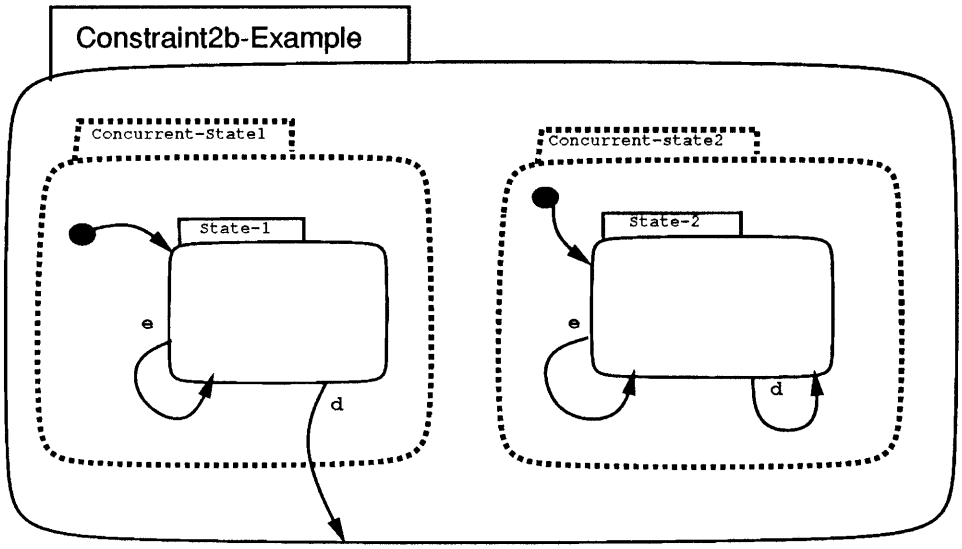


Figure 4-3: Examples of Syntactic Constraint Violations in a PCM

starting from *state-1* and *state-2* violate this constraint since the source states are concurrently active, but give different destinations which are not concurrently related. On the other hand, the pair of transitions for input *e* satisfies the constraint since the two destination states are still concurrent.

4.2.3 Constraint #3: to require determinism in variable assignments.

In order to prevent some cases that permit multiple assignments to a single variable or circular variable assignments, constraints #3 is imposed upon the formal model. , guarantee that multiple or circular variable assignments do not occur in the PCM.) As with the determinism requirement upon destination states, the requirement that variable assignments occur deterministically is desirable for implementations of the PCM that are used for real-world systems.

The first "or" clause of constraint #3 prevents two states which are parent-related from having transitions with the same input, overlapping predicates, and assignments referring to the same variables. The constraint permits multiple arcs from a single state for the same input and overlapping predicates, as long as all arcs have no assignments

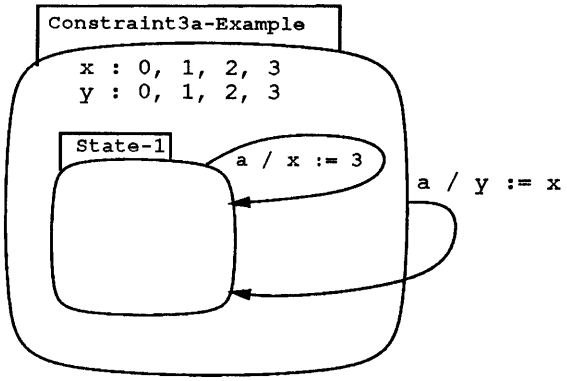


Figure 4-4: Examples of Conflicting Variable Assignments in a PCM

referencing the same variables. As an example of an arc forbidden by this constraint, in Figures 4-4a and 4-4b, the pairs of transitions for inputs *a* and *b* both violate this constraint since each pair is enabled from *state-1*, giving assignments referencing the same variables.

The second "or" clause of constraint #3 prevents concurrent states from having transitions with the same input, overlapping predicates, and assignments referring to the same variables. For example, in Figure 4-5, the pair of transitions for input *d* starting from

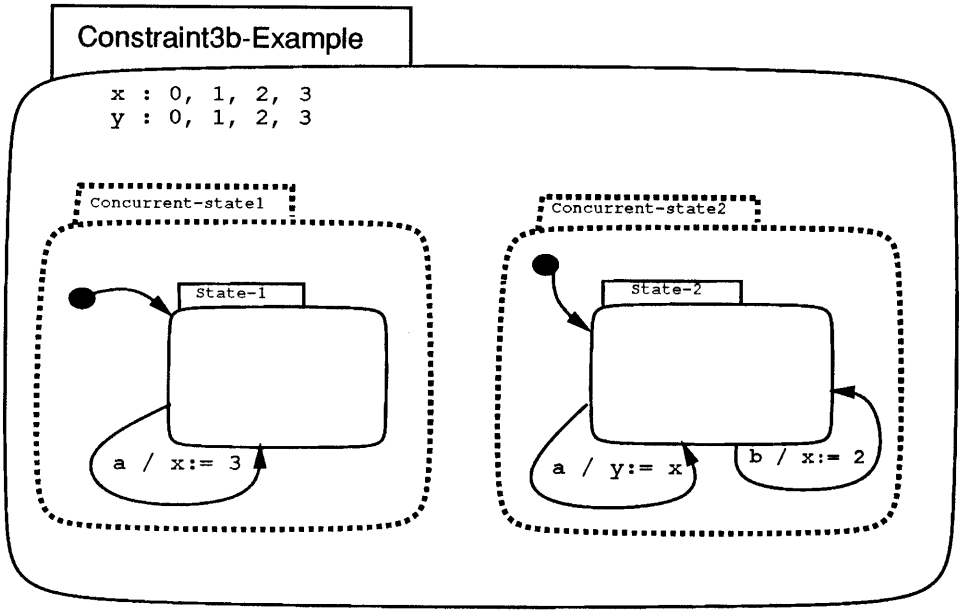


Figure 4-5: Examples of Conflicting Variable Assignments in a PCM

state-1 and *state-2* violate this constraint since the source states are concurrently active, but reference the same variables in their assignments.

4.2.4 A degenerate PCM

The correspondence between the pictoral representation of the PCM and the formal model is best described by an example. The simplest partitioned computation machine to consider is a degenerate PCM which consists of only one state. With such a machine, LS(X) is identical to $\{s_0(X)\}$. This specific case, however, is the only time that LS(X), IS(X), and $\{s_0(X)\}$ are not all disjoint.

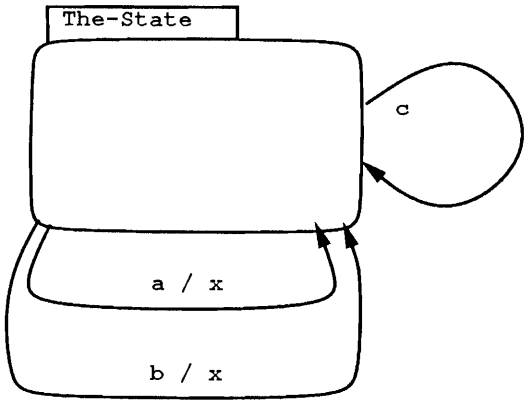


Figure 4-6: A Sample Degenerate Partitioned Computation Machine

A sample one-state PCM is shown in Figure 4-6. The formal representation for this PCM is given in Figure 4-7. This machine is intended to produce an output *x*, whenever the inputs *a*, or *b* are received. No output results from the input *c*. The set of states for the degenerate PCM contains only one element, which is also the distinguished root element: *The-State*. This degenerate PCM has no hierarchy or variables, so the sets and mappings describing these are all empty. The three transitions in this PCM are described in the transition relation. Each element in the transition relation corresponds exactly to one of the arcs in the diagram. For example, the arc for *a* in the diagram produces an element of the transition relation which starts and ends at *The-State*, has *a* as an input, and produces *x* as an output. This information is represented by building the element of the transition relation appropriately.

1. S(X) = {The-State}, s_0 = The-State
2. P(X) = {}
3. AP(X) = {}
   MP(X) = {}
4. DC(X) = {}
5. IE(X) = {a, b, c}
6. OE(X) = {x}
7. V(X) = {}
8. VS(X) = {}
9. VT(X) = {}
10. TR(X) = {(The-State, a, True, {x}, Ø, The-State),
    (The-State, b, True, {x}, Ø, The-State),
    (The-State, c, True, Ø, Ø, The-State)}

Figure 4-7: The Formal Representation for the One-State PCM of Figure 4-6

4.3 The Execution of the PCM

In addition to the formal representation for the pictoral information, the PCM formal model includes a formal semantics of execution. The execution semantics relies upon expanding the set of states, S(X), into a set of all possible maximal sets of concurrent states. This expansion makes an *execution state* of a partitioned computation machine just one of the members of the expanded state set; a single execution state consists of a set of states from S(X).

4.3.1 Expansion of concurrent cross-product states

The partitioned computation machine permits concurrency in states with multiplicative parents. The execution model for the partitioned computation machine, as will be presented in section 4.3.2, actually operates upon a set containing all possible concurrently active states. Before presenting the execution of the PCM, this expanded state set, ES(X), will be defined.

The *expanded state set*, ES(X), is defined by first defining a set, IES(X), then creating ES(X) to include only the elements of IES(X) which have no proper superset in IES(X). IES(X) is conceptually all sets consisting of leaf states of S(X) where all elements of the set are concurrent-related with one another.

$$IES(X) = \{ ls \mid ls \in 2^{LS(X)} \text{ and } \forall s_1, s_2 \in ls, \text{Concurrent-Related}(s_1, s_2)\}$$

$$ES(X) = \{ es \mid es \in IES(X) \text{ and } \forall ls \in IES(X), \neg(es \subset ls)\}$$

Conceptually, ES(X) is a set containing all maximal sets of concurrently active states. The elements of ES(X) are the possible execution states of the PCM X.

In order to formally determine the semantics of transition arrows, it is necessary to define two more constructs:

- *Expanded-Parent-Related*(es, s_1), for an expanded state es $\in$ ES(X), and a state $s_1 \in$ S(X), is defined as $\exists$ s $\in$ es where Parent-Related(s, s_1).
- *Concurrent-Default-Related*(s_1, s_2), for states $s_1 \in$ DLS(X) and $s_2 \in$ S(X) is defined as Concurrent-Related(s_1, s_2) where the least common ancestor of s_1 and s_2 according to P(X) is also an ancestor of s_1 according to DC(X).

Finally, the set of *Compatible-States*(S_1, S_2), where S_1 and S_2 are sets of states, is the set of states in S_1 that are Concurrent-Related to all elements of S_2.

4.3.2 Execution semantics

The PCM always has an execution state which is one of the members of the expanded state set, ES(X). Since concurrent children have been expanded into a single state in ES(X), a single element of this set can represent the concurrent operation of many states.

Each element in the formal model's transition relation represents an arc in the pictoral representation for the PCM. The partitioned computation machine operates under the assumption that only one input event is received at any time. This is an acceptable assumption to make as "simultaneous" input events can simply be serialized in some order before being received by the partitioned computation machine.

An element t of TR(X) is said to be *enabled* with respect to an expanded state es $\in$ ES(X), an input event $\pi \in$ IE(X), and a environment env mapping variables to values, if Expanded-Parent-Related(es, t.s), and $\pi = t\pi$, such that env satisfies t.pred. The *enabled-transition-set*(X, es, π, env) is defined for a given state es $\in$ ES(X), an input event $\pi \in$ IE(X) and an environment env, as the set of all t $\in$ TR(X) that are enabled with respect to es, π, and v. Lastly, before defining a step in the execution of the PCM, it is convenient to define the *Default-Leaves-of-Subtree(s)* as the set of states which are leaves of the subtree determined by DC(X) with *s* as the root of the subtree.

The basic unit of execution in a partitioned computation machine is a *step*. A step takes a machine from a current expanded state set and environment to a "next" expanded state set and environment when a specific input event is enabled. In this way, the complete execution of a PCM is composed of many individual steps. A step of the execution of a PCM X has a definition contingent upon the contents of the enabled-transition-set(X, es, π, v), and as such will be written step(X, es, π, v). A single step is a six-tuple of the form (es_1, env_1, π_2, oes_2, s_2, env_2). The formal definition of a step(X, es, π, v) is divided into two primary cases as follows:

- If enabled-transition-set(X, es, π, env) = $\varnothing$ then step(X, es π, env) is defined as (es, env, π, { }, es, env).

  (Conceptually, this means that if the transition for input π is not explicitly handled, assume that a self-transition with no output events or variable changes should take place.)

- If enabled-transition-set(X, es, π, env) $\neq \varnothing$, then step(X, es, π, env) is defined as (es, env, π, oes, es′, env′) where

  oes = Union (t.oe) for all t $\in$ enabled-transition-set(X, es, π, env), and

  Define explicit-destinations = Union (Default-Leaves-of-Subtree(t.s′))))) for all t $\in$ enabled-transition-set(X, es, π, env).

  Define unchanged-destinations = compatible-states(es, explicit-destinations).

  Define implicit-destinations = the elements of Compatible(DLS(X), explicit-destinations $\cup$ unchanged-destinations) that are Concurrent-Default-Related to at least one element of es.

es′ = explicit-destinations ∪ unchanged-destinations ∪ implicit-destinations

env′ = the functional composition of Variable-Assignments$_i$ applied to the argument env where Variable-Assignments$_i$ ∈ Union({t.asg}) for all t ∈ enabled-transition-set(X, es, π, env).

Essentially, the first three elements of a step are composed of the starting expanded state and environment with the appropriate input event. The output event set in a step is the union of the output events from all enabled transitions. The new expanded state consists of the union of states to which arcs are explicity given, with any of the original states that are still active, and any states which are entered implicitly. Lastly, the new environment is simply the application of the composition of any variable assignments to the original environment.

The set *steps*(X) is defined as the set of all possible values for step(X, es, π, env). An element of steps(X) is said to be a *step* of X.

An *execution fragment* of X is a finite sequence s_1, env_1, π_2, oes_2, s_2, env_2, ..., s_n, env_n, or an infinite sequence s_1, env_1, π_2, oes_2, s_2, env_2, ... such that (s_i, env_i, π_{i+1}, oes_{i+1}, s_{i+1}, env_{i+1}) is a step of X. The set of all possible execution fragments of X is called *efrags*(X). An execution fragment of X beginning with the expanded state Default-Leaves-of-Subtree(s_0(X)) is called an *execution* of X. The set of all possible executions of X is called *execs*(X). A state is said to be *reachable* if it is the final state of a finite execution. The special case of a sequence with no input events, (s_1, env_1), is also considered a valid exection and will be referred to as a *null execution.*

An execution string represents a ''run'' of the partitioned computation machine. In most situations, one is only concerned with the sequence of input and output events that occur during the execution and not with the sequence of states involved with the computation. A behavior of an execution fragment for X is the subsequence consisting of the events defined for X, denoted by behav(X). The set of all possible behaviors of X is denoted by behavs(X).

4.3.3 An example specifying a soda machine

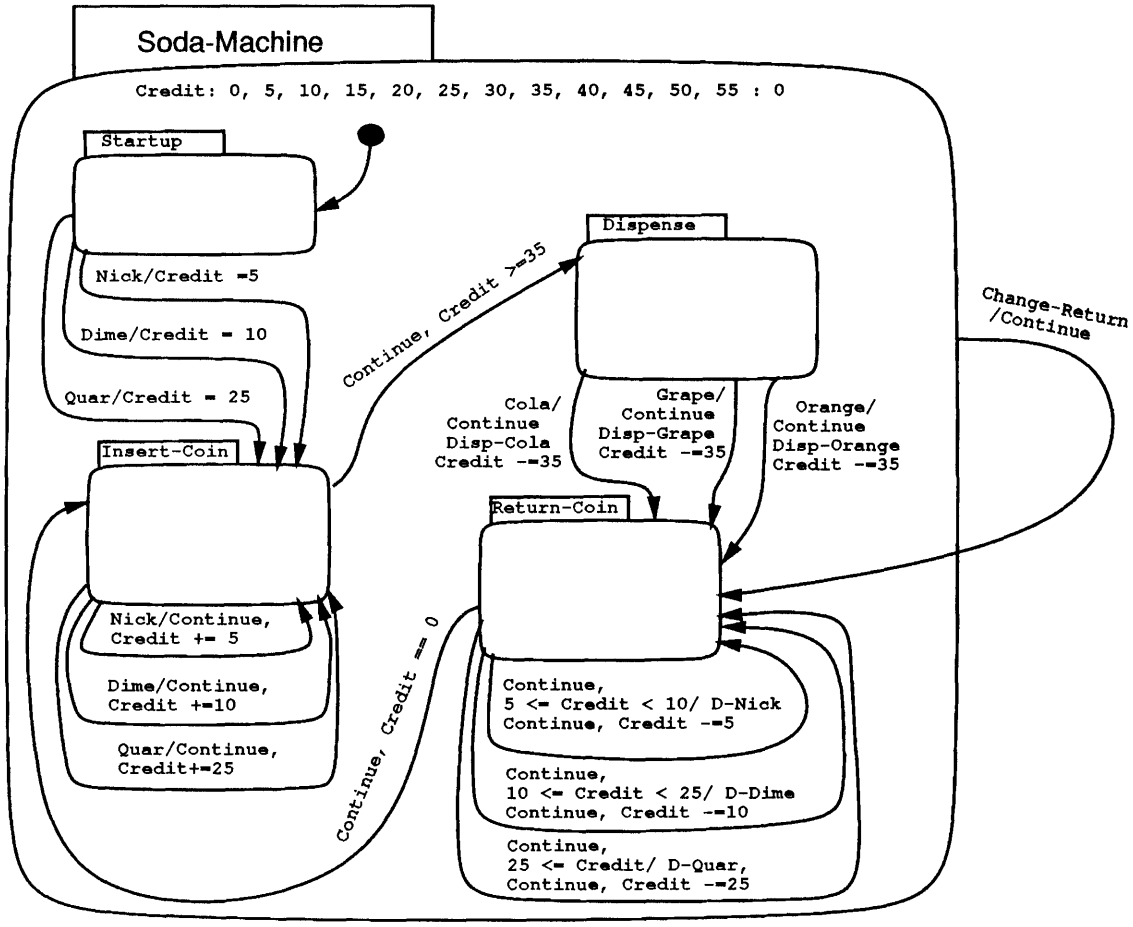


Figure 4-8: A Soda Machine as a Partitioned Computation Machine

Figure 4-8 provides the PCM pictoral representation for a soda machine. The soda machine accepts nickels, dimes and quarters as input, and dispenses cola, grape, and orange sodas after thirty-five cents worth of change have been inserted. The soda machine has four primary states under its root state. Each state represents the primary modes of the *soda machine*: a *startup* state, a state where it allows a person to *insert coins*, a state where it will *dispense* soda, and a state where it will *return coins* as change.

The soda machine starts in the *startup* state where the machine accepts inputs of *nick*, *dime*, and *quar*. When one of these inputs is received, the credit is set appropriately and the *insert-coin* state is made the new current state. The *insert-coin* state accepts inputs of *nick*,

```
1. S(X) = {Startup, Insert-Coin, Dispense, Return-Coin,
           Soda-Machine}, s0(X) = Soda-Machine

2. P(X) = {(Startup, Soda-Machine),
           (Insert-Coin, Soda-Machine),
           (Dispense, Soda-Machine),
           (Return-Coin, Soda-Machine)}

3. AP(X) = {Soda-Machine}

   MP(X) = {}

4. DC(X) = {(Soda-Machine, Startup)}

5. IE(X) = {Nick, Dime, Quar, Cola, Grape, Orange,
            Change-Return, Continue}


6. OE(X) = {Disp-Cola, Disp-Grape, Disp-Orange, D-Nick, D-Dime,
            D-Quar, Continue}

7. V(X) = {Credit}

8. VS(X) = {(Credit, Soda-Machine)}

9. VT(X) = {(Credit, (0, 5, 10, 15, 20, 25, 30,
                      35, 40, 45, 50, 55))}

10. TR(X) = {
  (Startup, Nick, True, {}, Credit = 5, Insert-Coin),
  (Startup, Dime, True, {}, Credit = 10, Insert-Coin),
  (Startup, Quar, True, {}, Credit = 25, Insert-Coin),
  (Insert-Coin, Nick, True, {Continue},
                            Credit += 5,  Insert-Coin),
  (Insert-Coin, Dime, True, {Continue},
                            Credit += 10, Insert-Coin),
  (Insert-Coin, Quar, True, {Continue},
                            Credit += 25, Insert-Coin),
  (Insert-Coin, Continue, Credit >= 35, {}, Ø, Dispense),
  (Dispense, Cola, True, {Disp-Cola, Continue},
                         Credit -= 35, Return-Coin),
  (Dispense, Grape, True, {Disp-Grape, Continue},
                          Credit -= 35, Return-Coin),
  (Dispense, Orange, True, {Disp-Orange, Continue},
                           Credit -= 35, Return-Coin),
  (Return-Coin, Continue, 5 <= Credit < 10, {D-Nick, Continue},
                          Credit -= 5, Return-Coin),
  (Return-Coin, Continue, 10 <= Credit < 25, {D-Dime, Continue},
                          Credit -= 10, Return-Coin),
  (Return-Coin, Continue, 25 <= Credit, {D-Quar, Continue},
                          Credit -= 25, Return-Coin),
  (Return-Coin, Continue, Credit == 0, {}, Ø, Insert-Coin),
  (Soda-Machine, Change-Return, True, {Continue}, Ø,
                                                  Return-Coin)}
```

Figure 4-9: The Formal Representation for the Soda Machine of Figure 4-8

dime, and *quar*, representing nickels, dimes, and quarters until thirty-five or more cents have been inserted. When sufficient funds have been inserted, the current state will become the *dispense* state. In the *dispense* state, the machine responds to inputs corresponding to a selection of the type of soda desired, generates the appropriate output to dispense the soda, and subtracts thirty-five cents from the credit inserted before proceeding to the *return-coin* state. The *return-coin* state returns all remaining credit and then returns to the *insert-coin* state to begin the cycle again. If the *change-return* input is received at any point duing this process, the *return-coin* state is entered and any remaining credit is returned. Note that the *continue* event is used both as an input and an output to permit a sequence of transitions to be enabled in response to an external input.

The formal representation of the pictoral information appears in figure 4-9. This example demonstrates the use of variable predicates, variable assignments, and multiple events all on the same arc.

4.3.4 Examples of execution of the PCM

To illustrate the definition of execution, this section presents sample executions for the soda machine presented in Figures 4-8 and 4-9 in section 4.3.3. The execution of this example is presented for a variety of different inputs.

4.3.4.1 Execution of a soda machine

Let us consider the execution of the soda machine of Figure 4-8 for the following sequence of inputs: Nick, Grape, Nick, Orange, Quar, Orange, Dime, Nick, Nick, Change-Return

To simplify the listing of the execution string, the starting state of a step will always be listed at the beginning of the line; the current value of the credit variable is given as the second element on the line. The third element on the line is the input event for the transition, and the fourth element is the set of output events from the transition. The fifth

and sixth elements of a step (the destination state and environment) are listed on the next line as the first two elements, as they will be the starting state and environment for the next transition. Also, to increase the readability of an execution, the first letters of any pending input events are shown on the extreme right of the line.

When examining the execution for the soda machine, note that the *Continue* event appears both as an output and as an input of the machine. Each occurrence of the *Continue* event appears twice in the execution -- once when it is an output, and once when it is used as input.

One possible execution for the machine given the above inputs would be:

```
Startup, 0, Nick, {},                                  G
Insert-Coin, 5, Grape,{},                              N
Insert-Coin, 5, Nick, {Continue},                      C O
Insert-Coin, 10, Continue, {},                         O
Insert-Coin, 10, Orange, {},                           Q
Insert-Coin, 10, Quar, {Continue} ,                    C
Insert-Coin, 35, Continue, {},                         O
Dispense, 35, Orange, {Disp-Orange, Continue},         C
Return-Coin, 0, Continue, {},                          D
Insert-Coin, 0, Dime, {Continue},                      C
Insert-Coin, 10, Continue, {},                         N
Insert-Coin, 10, Nick, {Continue},                     C
Insert-Coin, 15, Continue, {},                         N
Insert-Coin, 15, Nick, {Continue},                     C
Insert-Coin, 20, Continue, {},                         CR
Insert-Coin, 20, Change-Return, {Continue},            C
Return-Coin, 20, Continue, {D-Dime, Continue},         C
Return-Coin, 10, Continue, {D-Dime, Continue},         C
Return-Coin, 0, Continue, {},
Insert-Coin, 0
```

The above execution does essentially what one would expect; one inserts a total of 55 cents, and receives one orange soda, and twenty cents in change by the end of the execution. The above execution assumes that the PCM has an opportunity to consider the transitions with no input event before the next input event has been received. However, the model does not require that the PCM has these transitions before new input is received. Consider the case when the entire above input sequence reached the machine in the exact order above, before the machine had an opportunity to handle even one of its own output events. This would give the following behavior:

```
Startup, 0, Nick, {},                                    N G N O Q O D N N CR
Insert-Coin, 5, Grape, {},                               G N O Q O D N N CR
Insert-Coin, 10, Nick, {Continue},                       N O Q O D N N CR C
Insert-Coin, 10, Orange, {},                             Q O D N N CR C
Insert-Coin, 35, Quar, {Continue},                       O D N N CR C C
Insert-Coin, 35, Orange, {},                             D N N CR C C
Insert-Coin, 45, Dime, {Continue},                       N N CR C C C
Insert-Coin, 50  Nick, {Continue},                       N CR C C C C
Insert-Coin, 55, Nick, {Continue},                       CR C C C C C
Insert-Coin, 55, Change-Return, {Continue},              C C C C C C
Return-Coin, 55, Continue, {D-Quar, Continue},           C C C C C C
Return-Coin, 20, Continue, {D-Quar, Continue},           C C C C C C
Return-Coin, 10, Continue, {D-Nick, Continue},           C C C C C C
Return-Coin, 0, Continue, {},                            C C C C C
Insert-Coin, 0, Continue, {},                            C C C C
Insert-Coin, 0, Continue, {},                            C C C
Insert-Coin, 0, Continue, {},                            C C
Insert-Coin, 0, Continue, {},                            C
Insert-Coin, 0, Continue, {},
Insert-Coin, 0
```

In the second execution, the PCM is never given an opportunity to perform any of its *Continue* transitions until the input sequence has ended. It still ends up processing the same nine *Continue* events it did in the first scenario. The net effect here, however, is that 55 cents were inserted, and 55 cents were given back in change.

This example demonstrates that if inputs are coming into the machine in a manner that is not permitting the machine to handle its own generated events, the behavior received by the specification may not be exactly what the specifier may have expected. The above examples are both valid executions for the system. An interesting possibility for future work, however, would be to modify the model to permit the user to specify a response to an input, and then handle the generated events before additional input is processed. Such a technique would essentially permit multiple transitions in the machine to be treated as a single, atomic transition. Exploring this direction of research could be an area of future work. (See section 5.1.5.)

4.4 Composition of PCMs

The formal model for the PCM also includes a formal mechanism for composing a number of machines. Composition is a valuable operation as it permits a number of machines, each providing a distinct behavior, to be combined into a machine which behaves in a manner which can be predicted by examining only the behaviors of the component machines. (See Section 4.4.5.) Partitioned computation machines may be composed in one of two ways: additive composition or multiplicative composition. The primary distinction between the two forms of composition is that an additive composition maintains one control thread (i.e. the machines interact sequentially) and a multiplicative composition maintains multiple control threads (i.e. the machines appear to interact concurrently). The manner of each type of composition is discussed in further detail below.

4.4.1 Additive composition

As described informally in section 3.7.1, additive composition produces a new PCM from a number of other partitioned computation machines. A new PCM, Y, is produced out of *n* existing PCMs. The parameters of the additive composition process are:

1. Each of the *n* PCMs to be composed. (Call these X_i).
2. Default-X_i: The name for the X_i which is to be the default child of the composition.
3. New-Root: The name for the root state of the composition.
4. New-Variables: Set of variables to be associated with New-Root (optional).
5. New-Variable-Types: Mapping of New-Variables $\rightarrow$ Types(Y) (optional).
6. New-Input-Events: Set of new input events (optional).
7. New-Output-Events: Set of new output events (optional).
8. New-Transitions: Set of new relations for the Transition Relation (optional).

The parameters provided for the additive composition are restricted syntactically to prevent the composition of the machines from creating duplicate names for states or

variables. The first restriction is that all of the state names in the components and the new root name must be unique. An actual implementation of the partitioned computation machine could generate unique names for the component states which overlap, but the formal model simply requires that the all state names are unique. Similarly, all variables must also have unique identifiers.

In order to guarantee that the result of an additive composition satisfies the constraints given in section 4.2, it is sufficient that all component PCMs satisfy the constraints on a PCM, and furthermore, that the parameters to the composition satisfy the following constraints: (the multiplicative composition will have the same constraints upon its parameters)

Composition Constraint #1:

If it is the case that $t \in \text{Union}_{i=1}^{n} (TR(X_i) \cup \text{New-Transitions})$, then it must not be the case that Concurrent-Related(t.s, t.s′)

Composition Constraint #2:

If it is the case that $t_1 \in (\text{Union}_{i=1}^{n} (TR(X_i)) \cup \text{New-Transitions})$, and $t_2 \in$ New-Transitions, such that $t_1.\pi = t_2.\pi$, and such that Overlapping(t_1.pred, t_2.pred), where either Parent-Related(t_1.s, t_2.s) or Concurrent-Related(t_1.s, t_2.s), then it must be the case that Concurrent-Related($t_1.s'$, $t_2.s'$) or ($t_1.s' = t_2.s'$).

Composition Constraint #3:

If it is the case that $t_1 \in (\text{Union}_{i=1}^{n} (TR(X_i)) \cup \text{New-Transitions})$, and $t_2 \in$ New-Transitions, such that $t_1.\pi = t_2.\pi$, and such that Overlapping(t_1.pred, t_2.pred), where either Concurrent-Related(t_1.s, t_2.s), or Parent-Related(t_1.s, t_2.s), then it must be the case that [Variables (t_1.asg) $\cap$ Variables (t_2.asg) = $\varnothing$].

Parameters which satisfy the above constraints are used to construct a new PCM. Additive composition creates a new PCM, Y, from the above parameters:

1. $S(Y) = \text{Union}_{i=1}^{n} S(X_i) \cup \{\text{New-Root}\}$, $s_0(X) = \text{New-Root}$
2. $CP(Y) = \text{Union}_{i=1}^{n} CP(X_i) \cup \text{Union}_{i=1}^{n} \{(RS(X_i), \text{New-Root})\}$
3. $AP(Y) = \text{Union}_{i=1}^{n} AP(X_i) \cup \{\text{New-Root}\}$

   $MP(Y) = \text{Union}_{i=1}^{n} MP(X_i)$
4. $DC(Y) = \text{Union}_{i=1}^{n} DC(X_i) \cup \{(\text{New-Root}, \text{Default-}X_i)\}$
5. $IE(Y) = \text{Union}_{i=1}^{n} IE(X_i) \cup \text{New-Input-Events}$
6. $OE(Y) = \text{Union}_{i=1}^{n} OE(X_i) \cup \text{New-Output-Events}$
7. $V(Y) = \text{Union}_{i=1}^{n} V(X_i) \cup \text{New-Variables}$
8. $VS(Y) = \text{Union}_{i=1}^{n} VS(X_i) \cup \{(\text{New-Variable}_j, \text{New-Root}) \mid \text{New-Variable}_j \in \text{New-Variables}\}$
9. $VT(Y) = \text{Union}_{i=1}^{n} VT(X_i) \cup \text{New-Variable-Types}$
10. $TR(Y) = \text{Union}_{i=1}^{n} TR(X_i) \cup \text{New-Transitions}$

LS(Y), R(Y), and IS(Y) are still defined as they are in the basic formal model presented in section 4.1.

Clearly, the result of the additive composition is also a partitioned computation machine, by construction. The satisfaction of the three composition constraints guarantees that the resultant machine satisfies the three constraints upon all PCMs given in section 4.2.

Conceptually, additive composition takes all of the components and gives them a common parent to form a single partitioned computation machine. The new PCM is constructed from the components by defining its elements to be the union of the elements of the components, with the addition of new set elements indicated in the parameters of the composition. The set of new states, S, becomes the union of the state sets of the components and the New-Root for the composition. The tree structure, CP, is amended to indicate that the root states of the components, X_i, now have the New-Root as a parent. The New-Root is made a member of the additive-parent set, PA to indicate that its children have been additively composed. The default child mapping, DC, is expanded to indicate the default child for the New-Root. New events may be added to the sets of input and output events; new variables can be associated with the new root state. The set V, and mappings

VS and VT are all updated appropriately if New-Variables are defined. Finally, new transitions may be added to the composition as well.

4.4.2 Example of additive composition to build a simple user interface

This section provides an example to demonstrate the procedure of additive composition. The machines to be composed each specify simple menus to map buttonpresses generating the events *button-1*, *button-2*, and *button-3* into output events that would execute the correct program for that menu choice. The composition takes two such machines and additively combines them; the desired behavior of the composition is that the *escape* event toggles between the two menus. Figures 4-10 and 4-11 present the graphical and formal representations of the component partitioned computation machines. The arguments to the additive composition are listed at the top of page 65. Figures 4-12 and 4-13, respectively, provide the graphical and formal representations of the result of the composition.

1. S(X) = {Menu-One}

2. P(X) = {}

3. AP(X) = {}

   MP(X) = {}

4. DC(X) = {}

5. IE(X) = {Button-1, Button-2, Button-3}

6. OE(X) = {Move, Iconify, Resize}

7. V(X) = {}

8. VS(X) = {}

9. VT(X) = {}

10. TR(X) = {(Menu-One, Button-1, True, {Move}, Ø, Menu-One),
    (Menu-One, Button-2, True, {Iconify}, Ø, Menu-One),
    (Menu-One, Button-3, True, {Resize}, Ø, Menu-One)}

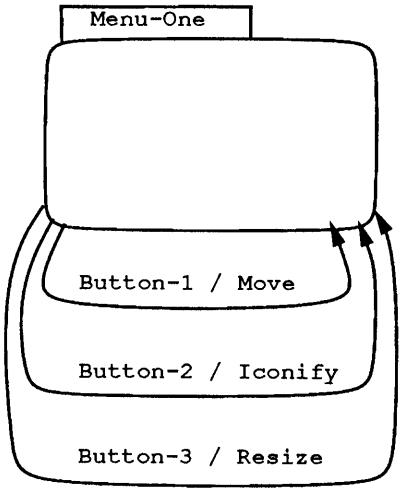


Figure 4-10: Example of PCM Formal Model for a Mouse Menu (Menu-One)

1. S(X) = {Menu-Two}

2. P(X) = {}

3. AP(X) = {}

   MP(X) = {}

4. DC(X) = {}

5. IE(X) = {Button-1, Button-2, Button-3}

6. OE(X) = {Kill, Raise, Lower}

7. V(X) = {}

8. VS(X) = {}

9. VT(X) = {}

10. TR(X) = {(Menu-Two, Button-1, True, {Kill}, Ø, Menu-Two),
(Menu-Two, Button-2, True, {Raise}, Ø, Menu-Two),
(Menu-Two, Button-3, True, {Lower}, Ø, Menu-Two)}

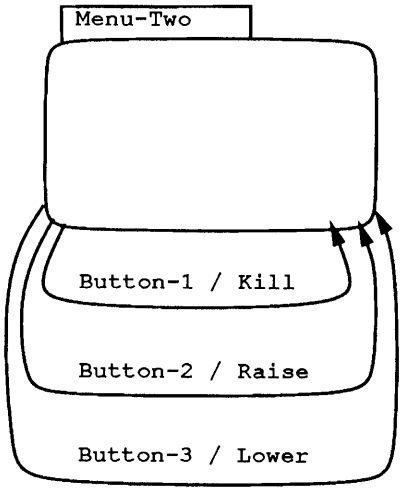


Figure 4-11: Example of PCM Formal Model for a Mouse Menu (Menu-Two)

The arguments to the additive composition are:

1. Components:	PCM(Menu-One), PCM(Menu-Two)
2. Default-X_i:	Menu-One
3. New-Root:	Window-Menus
6. New-Input-Events:	{Escape}
8. New-Transitions:	{(Menu-One, Escape, True, { }, ∅, Menu-Two), (Menu-Two, Escape, True, { }, ∅, Menu-One)}

The resulting Window-Menus is represented by:

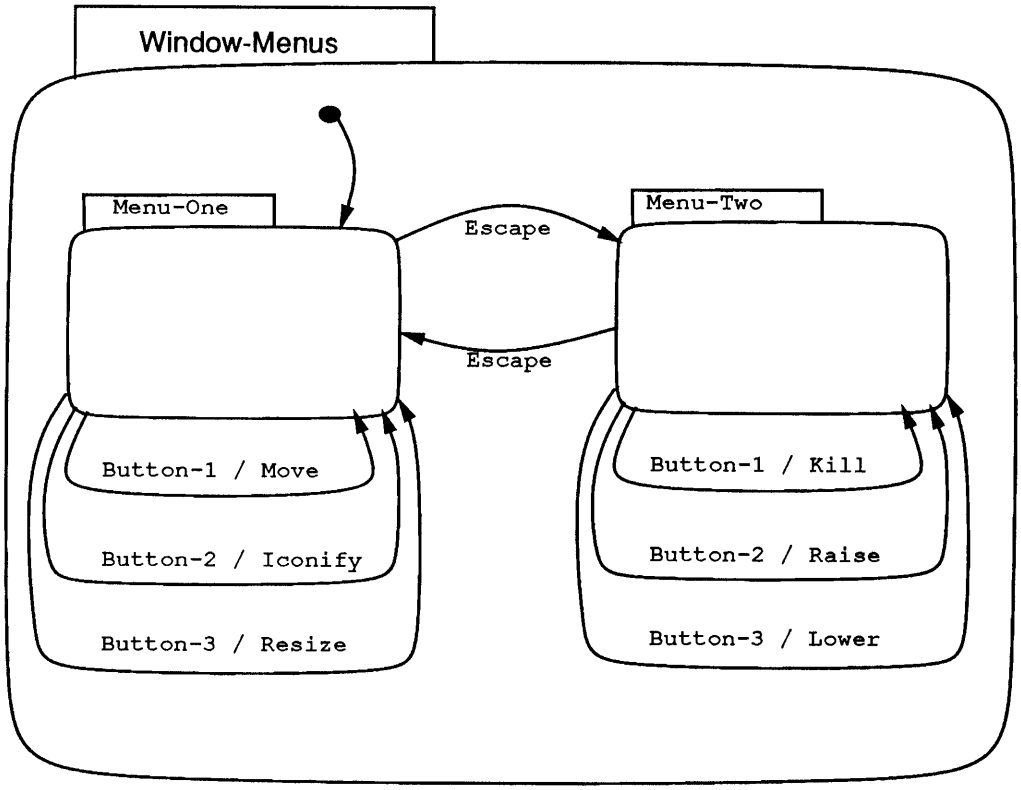


Figure 4-12: Example of PCM Additive Composition for Window-Menus

```
 1. S(X) = {Menu-One, Menu-Two, Window-Menus}

 2. P(X) = { (Menu-One, Window-Menus),
             (Menu-Two, Window-Menus) }

 3. AP(X) = {Window-Menus}

    MP(X) = { }

 4. DC(X) = { (Window-Menus, Menu-One) }

 5. IE(X) = {Button-1, Button-2, Button-3, Escape}

 6. OE(X) = {Move, Iconify, Resize, Kill, Raise, Lower}

 7. V(X) = { }

 8. VS(X) = { }

 9. VT(X) = { }

10. TR(X) = { (Menu-One, Button-1, True, {Move}, Ø, Menu-One),
          (Menu-One, Button-2, True, {Iconify}, Ø, Menu-One),
          (Menu-One, Button-3, True, {Resize}, Ø, Menu-One),
          (Menu-Two, Button-1, True, {Kill}, Ø, Menu-Two),
          (Menu-Two, Button-2, True, {Raise}, Ø, Menu-Two),
          (Menu-Two, Button-3, True, {Lower}, Ø, Menu-Two),
          (Menu-One, Escape, True, {}, Ø, Menu-Two),
          (Menu-Two, Escape, True, {}, Ø, Menu-One) }
```

Figure 4-13: Formal Model for Window-Menus of Figure 4-12.

4.4.3 Multiplicative composition -- multiple control threads

As described informally in section 3.7.2, multiplicative composition produces a new PCM from a number of other partitioned computation machines. The formal procedure for multiplicative composition creates a new PCM, Y, out of *n* existing PCMs by supplying the necessary parameters for multiplicative compositions. The parameters are:

1. Each of the *n* PCMs to be composed. (Call these X_i).
2. New-Root: The name for the root state of the Composition.
3. New-Variables: Set of variables to be associated with New-Root (optional).
4. New-Variable-Types: Map of New-Variables $\rightarrow$ Types(Y) (optional).
5. New-Input-Events: Set of new input events (optional).
6. New-Output-Events: Set of new output events (optional).
7. New-Transitions: Set of new relations for the Transition Relation (optional).

As with the additive composition, there are some syntactic restrictions on these parameters to prevent the composition of the machines from creating duplicate names for states or variables. The syntactic restrictions are the same as those for additive composition, namely, that all of the state names and new root name be unique, in addition to all variables having unique identifiers.

The composition constraints which are placed upon the multiplicative composition parameters in order to guarentee that the result of the composition satisfies the constraints for a PCM are the same as the constraints placed upon the additive composition parameters as given in section 4.4.1.

Multiplicative composition constructs a single new PCM from the components in much the same manner as the additive composition. As with the additive composition, the new PCM is constructed from the components by having its elements be the union of the components' elements, with the addition of new set elements indicated as parameters to the composition. The only differences between the representations for an additive and multiplicative composition is that the multiplicative composition makes the New-Root an element of the multiplicative-parent set, PM, instead of PA, and that the multiplicative composition does not expand the default child mapping, as multiplicative compositions do not have a default child -- all children appear to act concurrently.

The formal construction of the multiplicative composition of n PCMs, X_i, into a single PCM, Y, is the same as the additive composition presented in section 4.4.1 on page 60 with the following exceptions:

3. $AP(Y) = Union_{i=1}^{n} AP(X_i)$

   $MP(Y) = Union_{i=1}^{n} MP(X_i) \cup \{New\text{-}Root\}$

4. $DC(Y) = Union_{i=1}^{n} DC(X_i) \cup Union_{i=1}^{n} \{(New\text{-}Root, R(X_i))\}$

LS(Y), R(Y), and IS(Y) are still defined as they are in the basic formal model presented in section 4.1.

4.4.4 Example of multiplicative composition to build a combination clock-odometer

This section provides an example of multiplicative composition to demonstrate the procedure. The multiplicative components to be composed consists of simple controllers for a clock, odometer, and a visual display. These components are being combined to create a monolithic system which provides the functions of a clock and odometer on a simple display. Figures 4-14, 4-15, 4-16 present the graphical and formal representations of the component partitioned computation machines. The arguments to the multiplicative composition are listed at the top of page 72. Figures 4-17 and 4-18, respectively, provide the graphical and formal representations of the result of the composition.

1. S(X) = {Clock-Run, Clock-Stop, Clock}
2. P(X) = {(Clock-Run, Clock), (Clock-Stop, Clock)}
3. AP(X) = {Clock}

   MP(X) = {}
4. DC(X) = {(Clock, Clock-Stop)}
5. IE(X) = {clock-start, clock-stop, clock-tick}
6. OE(X) = {}
7. V(X) = {Time}
8. VS(X) = {(Time, Clock)}
9. VT(X) = {(Time, *Integer*)}
10. TR(X) = {(Clock-Run, clock-stop, True, {},
                Ø, Clock-Stop),
            (Clock-Run, clock-tick, True, {},
                Time +=1, Clock-Run),
            (Clock-Stop, clock-start, True, {},
                Ø, Clock-Run)}

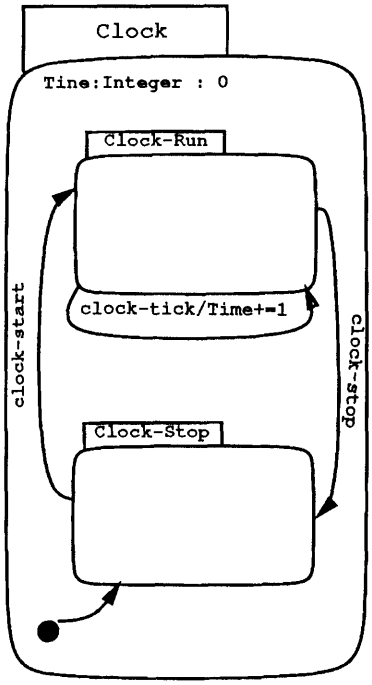


Figure 4-14: Example of PCM Formal Model for a Clock

1. S(X) = {Odo-Run, Odo-Stop, Odo}
2. P(X) = {(Odo-Run, Odo), (Odo-Stop, Odo)}
3. AP(X) = {Odo}

   MP(X) = {}
4. DC(X) = {(Odo, Odo-Stop)}
5. IE(X) = {odo-start, odo-stop, odo-tick}
6. OE(X) = {}
7. V(X) = {Dist}
8. VS(X) = {(Dist, Odo)}
9. VT(X) = {(Dist, *Integer*)}
10. TR(X) = {(Odo-Run, odo-stop, True, {}, ∅, Odo-Stop),
    (Odo-Run, odo-tick, True, {}, Dist += 1, Odo-Run),
    (Odo-Stop, odo-start, True, {}, ∅, Odo-Run)}

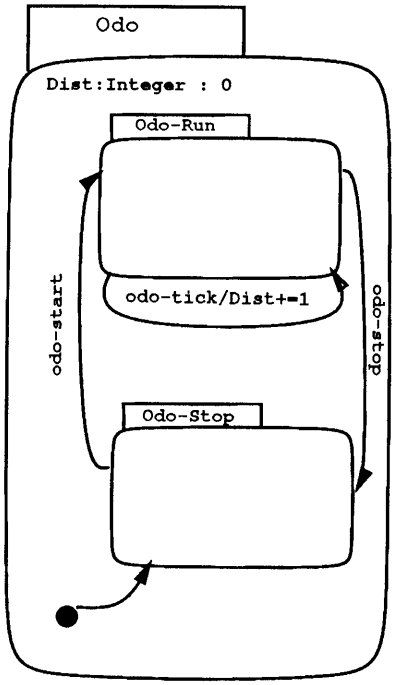


Figure 4-15: Example of PCM Formal Model for Odometer

1. S(X) = {Disp-Latched, Disp-Unlatched, Display-Control}

2. P(X) = {(Disp-Latched, Display-Control),
(Disp-Unlatched, Display-Control)}

3. AP(X) = {}

MP(X) = {Display-Control}

4. DC(X) = {(Display-Control, Disp-Unlatched)}

5. IE(X) = {lap-on, lap-off}

6. OE(X) = {latch-display, unlatch-display}

7. V(X) = {}

8. VS(X) = {}

9. VT(X) = {}

10. TR(X) = {(Disp-Latched, lap-off, True, {unlatch-display},
Ø, Disp-Unlatched),
(Disp-Unlatched, lap-on, True, {latch-display},
Ø, Disp-Latched)}

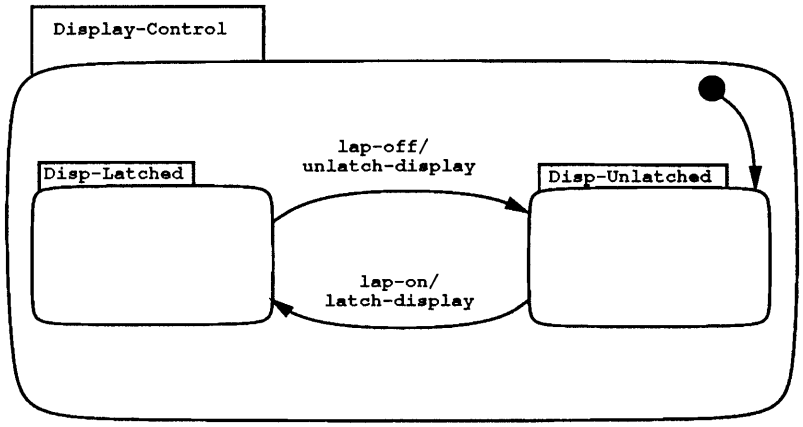


Figure 4-16: Example of PCM Formal Model for Visual Display

The arguments to the multiplicative composition are:

1. Components: PCM(Clock), PCM(Odo), PCM(Display-Control)
2. New-Root: Clock-Odometer

The resulting Clock-Odometer is represented by:

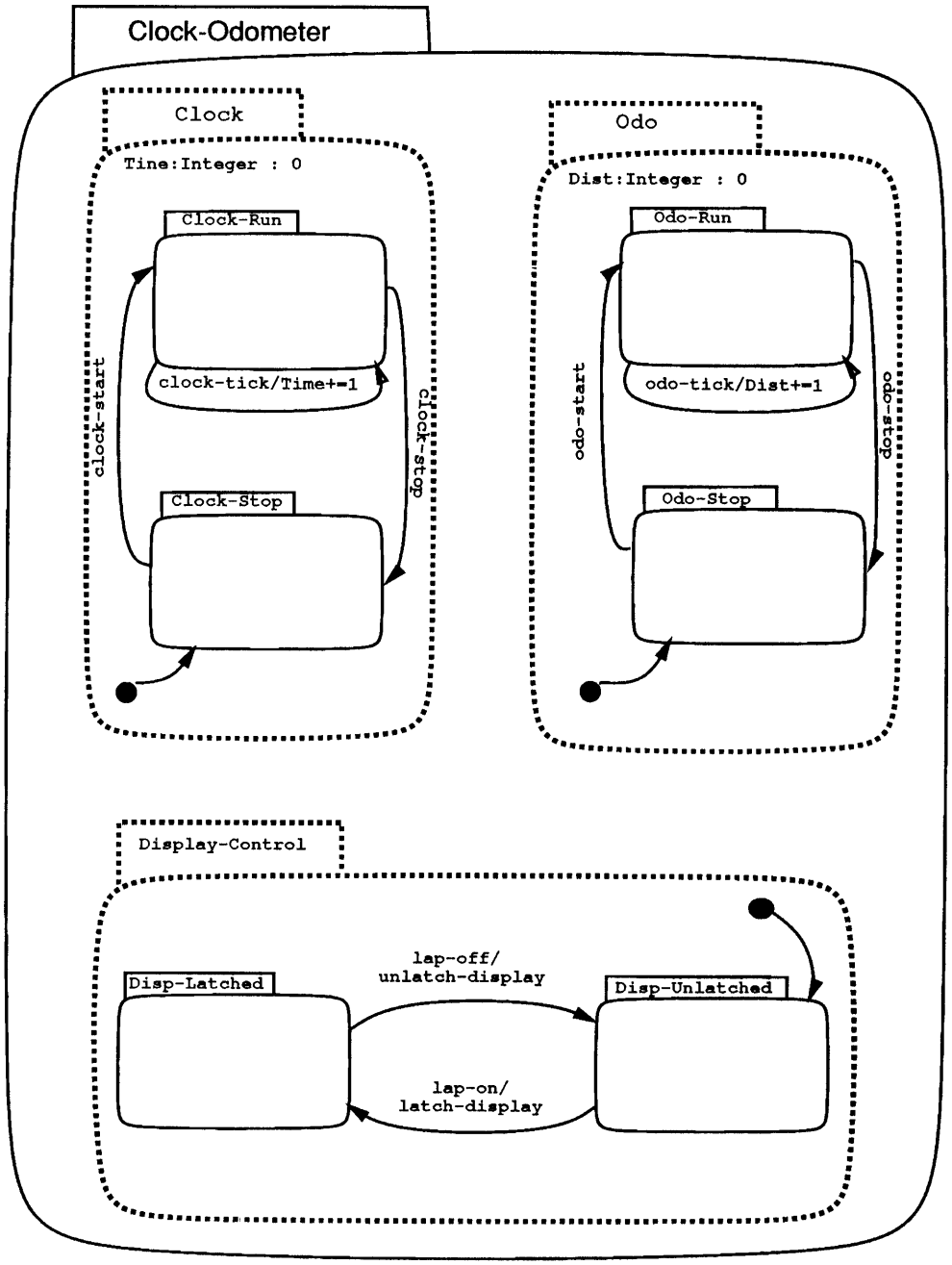


Figure 4-17: Example of PCM Multiplicative Composition for Clock-Odometer

```
1. S(X)= {Clock-Run, Clock-Stop, Odo-Run,
          Odo-Stop, Disp-Latched, Disp-Unlatched,
          Clock, Odo, Display-Control,
          Clock-Odometer}

2. P(X) = {(Clock, Clock-Odometer),
           (Odo, Clock-Odometer),
           (Display-Control, Clock-Odometer),
           (Clock-Run, Clock),
           (Clock-Stop, Clock),
           (Odo-Run, Odo),
           (Odo-Stop, Odo),
           (Disp-Latched, Display-Control),
           (Disp-Unlatched, Display-Control)}

 3. AP(X) = {Clock, Odo, Display-Control}

    MP(X) = {Clock-Odometer}

 4. DC(X) = {(Clock-Odometer, Clock),
             (Clock-Odometer, Odo),
             (Clock-Odometer, Display-Control),
             (Clock, Clock-Stop),
             (Odo, Odo-Stop),
             (Display-Control, Disp-Unlatched)}

 5. IE(X) = {clock-start, clock-stop, clock-tick, odo-start,
             odo-stop, odo-tick, lap-on, lap-off}

 6. OE(X) = {latch-display, unlatch-display}

 7. V(X) = {Time, Dist}

 8. VS(X) = {(Time, Clock),
             (Dist, Odo)}

 9. VT(X) = {(Time, Integer),
             (Dist, Integer)}

10. TR(X) = (Clock-Run, clock-stop, True, {}, Ø, Clock-Stop),
            (Clock-Run, clock-tick, True, {},
                                Time +=1, Clock-Run),
            (Clock-Stop, clock-start, True, {}, Ø, Clock-Run),
            (Odo-Run, odo-stop, True, {}, Ø, Odo-Stop),
            (Odo-Run, odo-tick, True, {},
                                Dist += 1, Odo-Run),
            (Odo-Stop, odo-start, True, {}, Ø, Odo-Run),
            (Disp-Latched, lap-off, True, {unlatch-display},
                                Ø, Disp-Unlatched),
            (Disp-Unlatched, lap-on, True, {latch-display},
                                Ø, Disp-Latched)}
```

Figure 4-18: Formal Model for Combination Clock-Odometer of Figure 4-17.

4.4.5 Executions of composed machines

With certain restrictions imposed upon the parameters for composition, additive and multiplicative composition, the possible behaviors of a composed machine can be built simply from the executions of the component machines. These restrictions can greatly simplify proofs regarding the executions of partitioned computation machines. Even when these restrictions do not apply, however, the structure of the composed machine makes the resultant behavior easir to understand.

The restrictions that are imposed serve to prevent inconsistencies between the transitions among the component machines and their internal behavior as discussed in section 3.3.3 on page 24. Furthermore, the restrictions prevent the component machines from interfering with each other's behavior.

4.4.5.1 Claims regarding additive composition executions

Additive Composition Claim #1:

Given that the following constraints are imposed on the parameters of an additive composition, (see section 4.4.1.)

- $[(\text{New-Input-Events} \cup \text{New-Output-Events}) \cap (\text{Union}_{i=1}^{n}\ IE(X_i) \cup \text{Union}_{i=1}^{n}\ OE(X_i))] = \varnothing$

  (This constraint requires that the composition parameters are restricted so that all new input or output events are disjoint from events used in any of the components.)

- For all $t \in$ New-Transitions, $t.\pi \in$ New-Input-Events

  (This constraint requires that all new transitions are enabled by new inputs.)

- For all $t \in$ New-Transitions, $t.oe \subseteq$ New-Output-Events

  (This constraint requires that all new transitions only produce new outputs.)

then the following assertion is claimed to be true:

An execution fragment of the additive composition consists of execution fragments of one of the component machines, separated by new input events and output events defined in the composition.

Formally, the execution fragment, ε_y of the PCM Y, produced from the additive composition of machines X_i can be expressed as a sequence ε_a, π_a, oe_a, ε_b, π_b, oe_b, ... where each $\varepsilon \in$ efrags(TR$(X)_i$), $\pi \in$ New-Input-Events, and oe $\subseteq$ New-Output-Events.

Proof: Let Y = the additive composition of X_i, and suppose that $\varepsilon_y = s_1,env_1,\pi_2,oe_2,s_2,env_2...$ By the definition of an execution fragment, s_1 is an element of ES(Y), and every six-tuple $(s_{k-1},env_{k-1},\pi_k,oe_k,s_k,env_k)$ is a step of Y. Two facts follow from the definition of the additive composition Y. First, $s_1 \in ES(X_i)$ for some i. Second, if s_{k-1} is an expanded state of $ES(X_i)$ and $\pi_k \notin$ New-Input-Events, then $(s_{k-1},env_{k-1},\pi_k,oe_k,s_k,env_k)$ is a step of the same X_i. If s_{k-1} is an expanded state of $ES(X_i)$ and $\pi_k \in$ New-Input-Events, then the six-tuple $(s_{k-1},env_{k-1},\pi_k,oe_k,s_k,env_k)$ can be broken into four portions, where s_{k-1},env_{k-1} is a null execution of an X_i, $\pi_k \in$ New-Input-Events,$oe_k \subseteq$ New-Output-Events, and s_k,env_k is a null execution of an X_i. Therefore, an execution fragment, ε_y of the PCM Y, produced from the additive composition of machines X_i can be expressed as a sequence ε_a, π_a, oe_a, ε_b, π_b, oe_b, ... where each $\varepsilon \in$ efrags(TR$(X)_i$), $\pi \in$ New-Input-Events, and oe $\subseteq$ New-Output-Events.

4.4.5.2 Claims regarding multiplicative composition executions

As with the additive composition, a number of claims regarding the execution of multiplicatively composed machines can be made. The results of these claims can be used to reason about the execution of a composed machine as the "sum" of the executions of the component machines. When examining the executions of multiplicatively composed machines, the concept of a *projection* is useful.

A *projection* of an execution fragment, ε (or a portion of an execution fragment) for a partitioned computation machine, X, is defined as the same sequence as ε (or the portion of the fragment), but including only states in S(X), input events in IE(X), and output events that are in OE(X). The projection of an execution fragment ε for a PCM X can be written as $\varepsilon|X$.

Multiplicative Composition Claim #1:

For a multiplicative composition producing a PCM, Y, if New-Transitions = $\varnothing$ and all $OE(X_i)$ are disjoint, the following assertion is claimed to be true:

If $\varepsilon_y \in$ efrags(Y) where Y is the multiplicative composition of some X_i, $\varepsilon_y|X_i \in$ efrags(X_i).

Proof: Let Y = the multiplicative composition of X_i, and suppose that $\varepsilon_y = s_1,env_1,\pi_2,oe_2,s_2,env_2...$ By the definition of an execution fragment, s_1 is an element of ES(Y), and every six-tuple $(s_{k-1},env_{k-1},\pi_k,oe_k,s_k,env_k)$ is a step of Y. Two facts follow from the definition of the multiplicative composition Y. First, $s_1|X_i \in ES(X_i)$ for some i. Second, $(s_{k-1},env_{k-1},\pi_k,oe_k,s_k,env_k)|X_i$ is a step of the same X_i. Thus, for $\varepsilon_y|X_i$, every six-tuple $(s_{k-1},env_{k-1},\pi_k,oe_k,s_k,env_k)$ is a step of X_i. Therefore, $\varepsilon_y|X_i \in$ efrags(X_i).

Multiplicative Composition Claim #2:

Furthermore, if the following constraints are imposed on the parameters of the multiplicative composition,

- [(New-Input-Events $\cup$ New-Output-Events) $\cap$ (Union$_{i=1}^{n}$ IE(X_i) $\cup$ Union$_{i=1}^{n}$ OE(X_i))] = $\varnothing$

  (This constraint requires that the composition parameters are restricted so that all new input or output events are disjoint from events used in any of the components.)

- For all j, $OE(X_j) \cap$ Union$_{i=1,i\neq j}^{n}$ $OE(X_i) = \varnothing$

  (This constraint requires that the composition parameters are restricted so that the output events of each component are disjoint from output events of the other components.)

- For all t $\in$ New-Transitions, t.$\pi \in$ New-Input-Events

  (This constraint requires that all new transitions are enabled by new inputs.)

- For all t $\in$ New-Transitions, t.oe $\subseteq$ New-Output-Events

  (This constraint requires that all new transitions only produce new outputs.)

then, the following assertion is claimed to be true:

The projection of an execution fragment of the multiplicative composition for a component X_i consists of execution fragments of X_i, separated by new input events and output events defined in the composition.

Formally, if $\varepsilon_y \in$ efrags(Y) where Y is the multiplicative composition of some X_i, $\varepsilon_y|X_i$ can be expressed as a sequence ε_a, π_a, oe_a, ε_b, π_b, oe_b, ... where each $\varepsilon \in$ efrags(TR(X)$_i$), $\pi \in$ New-Input-Events, and oe $\subseteq$ New-Output-Events.

Proof: Let Y = the multiplicative composition of X_i, and suppose that $\varepsilon_y = s_1,env_1,\pi_2,oe_2,s_2,env_2...$ By the definition of an execution fragment, s_1 is an element of ES(Y), and every six-tuple $(s_{k-1},env_{k-1},\pi_k,oe_k,s_k,env_k)$ is a step of Y. Two facts follow from the definition of the multiplicative composition Y. First, $s_1|X_i \in ES(X_i)$ for some i. Second, if $\pi \notin$ New-Input-Events, $(s_{k-1},env_{k-1},\pi_k,oe_k,s_k,env_k)|X_i$ is a step of the same X_i. If $\pi \in$ New-Input-Events, then the six-tuple $(s_{k-1},env_{k-1},\pi_k,oe_k,s_k,env_k)$ can be broken into four portions, where $s_{k-1},env_{k-1}|X_i$ is a null execution of X_i, $\pi_k \in$ New-Input-Events, $oe_k \subseteq$ New-Output-Events, and $s_k,env_k|X_i$ is a null execution of X_i. Therefore, if $\varepsilon_y \in$ efrags(Y) where Y is the multiplicative composition of some X_i, $\varepsilon_y|X_i$ can be expressed as a sequence ε_a, π_a, oe_a, ε_b, π_b, oe_b, ... where each $\varepsilon \in$ efrags(TR(X)$_i$), $\pi \in$ New-Input-Events, and oe $\subseteq$ New-Output-Events.

Chapter 5

Future Work and Conclusions

The visual language and formal representation of the partitioned computation machine presented in this thesis reach the goals that were set for this work. The partitioned computation machine uses a primarily visual, state-based langauge and a formal representation for describing the execution of this language. The representation permits incremental changes to be made and describes the formal semantics without losing information provided by the specifier. Furthermore, modularity and abstraction are encouraged by the PCM without restricting the computational power of the model to less than that of a Turing machine. However, there are some aspects of the partitioned computation machine model and its usage that would benefit from further research.

5.1 Future Work

Future work with the formal model of the partitioned computation machine could provide additional formality to the issues of variable usage and consistency with simultaneous events. Additional future work could also implement both the visual language and the formal representation in a single system to permit easy, rapid development of PCM specifications. Finally, the model could be extended to incorporate the ideas of multiple transitions occurring atomically.

5.1.1 Variable usage

5.1.1.1 Languages to describe variables

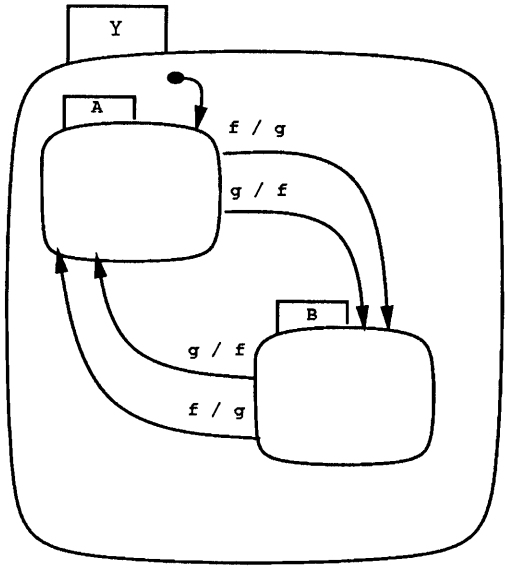
One area of future work with the partitioned computation machine is to formalize the language used to describe variable usage. The elements of the formal model which are used

to describe the type, conditional predicate, and variable assignment of variables are extremely formal and need a language to describe them to make the model readily useful. A specific language should be developed to describe the different types a variable can assume and the valid predicates and assignments for that type. This language would need to provide a way for translating expressions in the language into all the sets used for variable description as described in section 4.3.2.

A significant area for research with variable usage concerns issues which arise with the use of arbitrary predicates to determine the enabling of arcs in a language. The general problem of determining if two arcs are ever enabled for the same input (in specific, this is the determination of overlapping($pred_1$, $pred_2$) at the language level) is potentially undecidable with arbitrary predicates on infinite variables. In order to guarantee the satisfaction of the constraints given in section 4.2, restrictions on the language of predicates could be imposed. In addition, it may be desirable to relax the requirment that all assignments reference different variables to the less strict requirement that all variable assignments occuring on simultaneous transitions provide the same final values for the variables in all sequentializations of simultaneous assignments. Exploring the issues involved with relaxing the constraints on the PCM by use of a specific variable description language is an interesting future area of work.

5.1.2 Simultaneous event issues

A second area of potential future work with the partitioned computation machine is the development of algorithms for detecting the inconsistencies introduced by simultaneous events. As discussed in section 3.8, the model requires that valid specifications provide consistent behavior from the generation of simultaneous output events. The required consistency can be at varying levels of strictness. The strictest sense of consistency requirements is that the machine must have the exact same state and assignments to



b)

Figure 5-1: A PCM generating infinite events

variables immediately after performing the simultaneous output events in any order. This strictest form of consistency is the one suggested here, as it seems to be less difficult to verify than slightly looser requirements. Looser consistency could be permitted by instead requiring that the machine have the exact same state and variable assignments after performing the simultaneous output events in any order and handling any events that are generated by transitions resulting from the original simultaneous events. This type of consistency, however, could be much more difficult to verify as the original simultaneous transitions may generate events which generate other transitions, etc. Indeed, a machine could be envisioned which has a single input event which generates another event, which then generates another event, etc., as in Figure 5-1. An execution of such a machine could never process all of the pending events. Alternatively, consistency does not have to be required by the computational model at all. It is mentioned as a requirement in this thesis because the partitioned computation machine is intended to serve as an application for

program development. In a program development environment, determinism of executions is desirable so that bugs in specifications or implementations can be detected readily. Non-deterministic executions such as PCM executions that depend on ordering of simultaneous events are exceedingly difficult to debug.

5.1.3 Translation to input/output automata

It is desirable to consider the possibility of translating the PCM into an I/O automata. Such a translation would permit the PCM to take advantage of existing proof techniques developed for I/O automata. This translation involves building two input/output automata which interact to simulate the PCM. This section proposes one way that such automata could be built. Determining the details of the construction would be a future research topic.

5.1.3.1 The high level translation

The partitioned computation machine cannot be directly translated into an equivalent input/output automaton. One reason for the inability to perform a direct translation is the fact that the PCM and I/O automaton have different restrictions on the input and output events which they accept. The PCM permits the same event to be both an input and output of the machine. The I/O automaton, however, requires that the input and output action sets be disjoint for a single automaton. A second distinction between the two specification models is that the PCM permits an input and possibly multiple outputs to appear on a single transition arc while the I/O automaton requires that each step have a single action associated with it. Thus, the transitions taken by the partitioned computation machine in a single step may require multiple steps in an I/O automaton. Providing exactly the same behavior as the PCM requires that an I/O Automaton takes several steps atomically. Since the I/O automaton is always input-enabled, it is necessary to have a second I/O automaton act as a queue automaton to permit the first automaton to take multiple steps consisting of locally-controlled actions without being interuppted by new input. Finally, the I/O

automaton acting as a queue automaton can also serialize the input events as the PCM expects. This serialization is important so that the I/O automaton does not make non-deterministic choices about which input event to handle -- the PCM considers all input events serially.

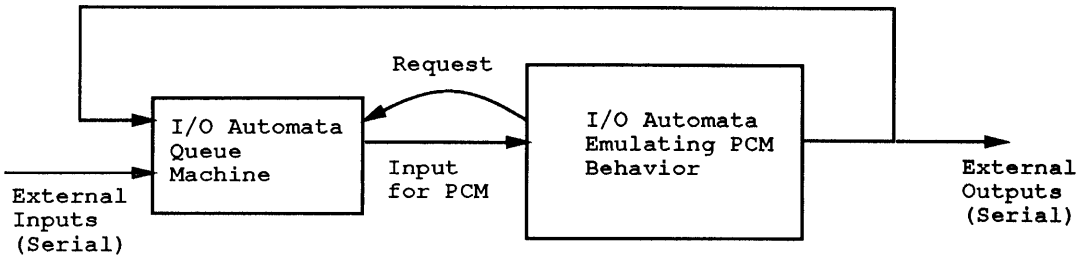


Figure 5-2: The High Level Model for I/O Automata Emulation of a PCM

Figure 5-2 shows how two I/O Automata can interact to simulate a PCM. The left I/O Automaton acts as the queue automaton, while the right I/O Automaton directly corresponds to the PCM which is being emulated. The queue automaton receives all external inputs, and also receives all output events from the primary automaton. The primary automaton only receives inputs from the queue automaton when it requests them. Additionally, the primary automaton can indicate if its output events are to be considered as simultaneous.

5.1.3.2 The interaction of the automata

The way that the pair of automata work at a high level is as follows:

1. The primary automaton requests an input from the queue automaton.
2. The queue automaton provides the primary automaton with the input from the front of the queue. Meanwhile, the queue automaton may be queueing external inputs that are received. If the queue is currently empty, the queue automaton will send the next input received to the primary automaton.
3. When the queue automaton provides the primary automaton with an input, the primary automaton performs a sequence of steps corresponding to what the emulated PCM would do with such an input in one of its steps. Any outputs

that are produced by the primary automaton during this sequence of steps are queued by the queue automaton with no external inputs separating them. This queueing behavior occurs since any outputs produced would be simultaneous in the emulated PCM since they are the result of a single PCM step.

4. After the equivalent of a single PCM step, the primary automaton requests another input. The queue automaton queues up any external inputs received since the last request, and then queues up any feedback inputs from the primary automaton. This process then returns to step 2 above.

The actual translation of a partitioned computation machine into input/output automata requires the construction of the appropriate input/output automata to serve as the queue automaton and as the primary automaton. The two automata are then composed in the manner of input/output automata to emulate the partitioned computation machine. The definition of the queue automaton should be relatively straightforward as it is similar to other work already done with I/O Automata. [Lynch 88]

5.1.3.3 The definition of the primary automaton

The input/output automaton specifying the primary automaton contains the real behavioral information present in the partitioned computation machine being emulated. There are a number of issues to be considered in specifying the primary automaton. First, there is no explicit tree hierarchy dividing states into leaf, internal, and root states in an I/O automata. Thus, the hierarchy of the partitioned computation machine must be "flattened" in the construction of an equivalent input/output automaton. The PCM formal model already creates such a set, ES(X) for the formal execution. However, the existence of multiple outputs and variable assignments occuring on a single transiton arc in the PCM will require such transitions to be broken into a short sequence of input/output automaton steps. The major work to be done here would specify what modifications to the extended set of states, ES(X), are needed and to determine how the transition relation for the primary automaton is defined from the PCM.

5.1.4 Implementation of the PCM

A large area for future work with the partitioned computation machine would be an implementation of both the visual language and the execution of the formal representation. A graphical interface specifically intended for the development of PCMs would make the visual specification of a machine more precise than a pencil and paper drawing. The pencil and paper drawings, however, would still have usefulness as a preliminary method of designing a specification due to the ease of which they can be drawn while a specifier is in a creative frame of mind. An implementation of the visual language would permit the formal representation to be created directly from the graphical representation. The formal representation can be executed directly. Thus, a user could create a specification of an entire system in the partitioned computation machine's visual language. The computer could then simulate the execution of the PCM, producing an execution string. This execution string could be compared to the specifier's desired behavior of the system. This process would give immediate feedback to the specifier as to whether the actual behavior corresponds to what is desired.

5.1.5 Atomic multiple-step transitions

The partitioned computation machine relies on a continual stream of inputs to continue processing. This stream of inputs can either come from outside the machine, or be generated by the machine itself. A current limitation of the machine is that it handles input events and output events with a strict queueing mechanism. It may be desirable at times to have a multiple-step transition within the machine take place before any external input is examined. An extension to provide this capability could be done by defining multiple transitions to occur as a atomic transition. The definition of such atomic transitions may be an area for future work.

5.1.6 Complexity of concurrent machines

The partitioned computation machine makes concurrent computation easier to consider through the use of concurrent children in the hierarchy. However, the complexity of simulating concurrent machines is not addressed by the formal model for the PCM. The formal model represents concurrency as the cross-product of all possible states in concurrent children. The notation reduces the conceptual complexity of the concurrent computation, but the formal model attacks the issue by brute-force cross-product expansion.

Future research with the PCM could consider the possibility of permitting the simulation of a formal model with concurrent children by using concurrent processes in a machine. The model may need to be extended to deal with temporal issues in a concurrent system.

5.2 Conclusion

The partitioned computation machine does, to a significant extent, satisfy the goals for which it has been constructed. The PCM definitely provides a visual language that helps to make apparent the control flow of the specified system. The hierarchy and modularity of the visual language make the language easier to understand as common behavior can be viewed as such. The representation for the visual languge is sufficiently formal to permit a precise semantics for the execution of a partitioned computation machine and also maintains all of the information regarding the visual language excepting the physical placement of graphical items. The information regarding the state hierarchy, interconnections of transition arcs, and variable types, predicates and assignments are all maintained in the formal representation. The PCM also maintains at least the computational power to model all computable behaviors.

References

[Allworth 81] S.T. Allworth.
The Process Virtual Machine.
Introduction to Real-Time Software Design.
Springer-Verlag, 1981, pages 95, 100-109, Chapter 7, "The Process Virtual Machine".

[AlpernSchneider 85]
B. Alpern and F.B. Schneider.
Defining Liveness.
Information Processing Letters 21(4):181-186, October, 1985.

[Buhr 84] R.J.A. Buhr.
System Design with Ada.
Prentice-Hall, Englewood Cliffs, NJ, 1984.

[Buhr 88] R.J.A. Buhr.
Machine Charts for Visual Prototyping in System Design.
SCE-88 2, Carleton University, August, 1988.

[Cardelli 84] L.A. Cardelli.
Semantics of Multiple Inheritance in Semantics of Data Types.
Lecture Notes in Computer Science. Volume 173. *Semantics of Data Types.*
Springer-Verlag, 1984, pages 51-67.

[ChandyMisra 88] K.M. Chandy and J. Misra.
Parallel Program Design: A Foundation.
Addison-Wesley, 1988.

[Davis 88] A.M. Davis.
A Comparison of Techniques for the Specification of External System Behavior.
Communications of the ACM 31(9):1098-1115, Sept, 1988.

[Green 86] M. Green.
A Survey of Three Dialogue Models.
ACM Transactions on Graphics 5(3):244-275, July, 1986.

[Harel 87] D. Harel.
Statecharts: A Visual Formalism for Complex Systems.
Science of Computer Programming 8:231-274, 1987.

[Harel 88] D. Harel.
On Visual Formalisms.
Communications of the ACM 31(5):514-530, May, 1988.

[Harel 89] D. Harel.
The Semantics of Statecharts.
Technical Report, i-Logix, August, 1989.

[Harel et al 87] D. Harel, A. Pnueli, J.P. Schmidt, and R. Shwerman.
On the Formal Semantics of Statecharts.
In *Proceedings of the 2nd IEEE Symposium on Logic in Computer Science*, pages 54-64. Ithaca, NY, 1987.

[Harel et al 88] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, and A. Shtul-Trauring.
STATEMATE: A Working Environment for the Development of Complex Reactive Systems.
In *Proceedings of the 10th IEEE International Conference on Software Engineering*, pages 396-406. Singapore, April, 1988.

[Hoare 78] C.A.R. Hoare.
Communicating Sequential Processes.
Communications of the ACM 21(8):666-677, August, 1978.

[Jacob 83] R.J.K. Jacob.
Using Formal Specifications in the Design of a Human-Computer Interface.
Communications of the ACM 26(4):259-264, April, 1983.

[KaramBuhr 87a] G.M. Karam and R.J.A. Buhr.
A Temporal Logic-Based Operational Specification Language, Interpreter, and Deadlock Analyzer for Ada.
SCE-87 7, Carleton University, August, 1987.

[KaramBuhr 87b] G.M. Karam and R.J.A. Buhr.
Starvation and Critical Race Analyzers for Ada.
SCE-87 8, Carleton University, August, 1987.

[Lamport 89] L. Lamport.
A Simple Approach to Specifying Concurrent Systems.
Communications of the ACM 32(1):32-45, January, 1989.

[LewisPapa 81] H.R. Lewis and C.H. Papadimitriou.
Elements of the Theory of Computation.
Prentice-Hall, Englewood Cliffs, NJ, 1981.

[Lynch 88] N.A. Lynch.
I/O Automata: A Model for Discrete Event Systems.
MIT/LCS/TM 351, MIT, March, 1988.

[LynchTuttle 88] N.A. Lynch and M.R. Tuttle.
An Introduction to Input/Output Automata.
MIT/LCS/TM 373, MIT, November, 1988.
(TM-351 Revised).

[Pnueli 86] A. Pnueli.
Applications of Temporal Logic to the Specification and Verification of Reactive Systems: A Survey of Current Trends.
Lecture Notes in Computer Science. Volume 224.*Current Trends in Concurrency*.
Springer-Verlag, 1986, pages 510-584.

[Reiss 86] S.P. Reiss.
Visual Languages and the GARDEN System.
Lecture Notes in Computer Science. Volume 282.*Visualization in Programming*.
Springer-Verlag, 1986, pages 178-198.

[Rumbaugh 88] J.E. Rumbaugh.
State Trees as Structured Finite State Machines for User Interfaces.
In *ACM SIGGRAPH Symposium on User Interface Software*. Banff, Alberta, 1988.

[Ward 86] P. Ward.
The Transformation Schema: An Extension of the Data Flow Diagram to Represent Control and Timing.
IEEE Transactions of Software Engineering 12:198-210, 1986.

[WardMellor 86] P. Ward and S.J. Mellor.
Structured Deleopment for Real-Time Systems.
Specifying Control Transformations.
Yourdon Press, New York, NY, 1986, pages 64-80, Chapter 7.

[Wasserman 85] A.I. Wasserman.
Extending State Transition Diagrams for the Specification of Human-Computer Interactions.
IEEE Transactions on Software Engineering SE-11(8):699-713, August, 1985.

[Yourdon 89] E. Yourdon.
State-Transition Diagrams.
Modern Structured Analysis.
Yourdon Press, Englewood Cliffs, NJ, 1989, pages 259-274, Chapter 13.

[Zave 85] P.A Zave.
A Distributed Alternative, to Finite-State-Machine Specifications.
ACM Transactions on Programming Languages and Systems 7(1):10-36, January, 1985.