A Hybrid Approach to Formal Verification Applied to an ATM Switching System *

Duncan Clarke and Insup Lee
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104-6389

December 14, 1995

Abstract

Verifying the correctness of real-time system models by traditional approaches that depend on the exploration of the entire system state space is impractical for large systems. In contrast, testing allows the search for violations of a property to be narrowed to a relatively small portion of the overall state space based on assumptions regarding the structure of an implementation. We present a hybrid approach that exploits formal methods to verify subcomponents of a system and testing to gain confidence in the correctness of the assembled system. The feasibility of the approach is demonstrated by application of the method to a process algebra model of the Sunshine ATM switching network.

*This research was supported in part by ONR N00014-89-J-1131S1, NSF CCR93-11622, and DARPA/NSF CCR90-14621.

Contents

1 Introduction

In the fifty years since the birth of electronic digital computing, computers have evolved from room-sized machines requiring constant attention from a staff of human servants into omnipresent servants of the masses. Having proved their worth in their roles as passive assistants in non-critical tasks, computers are increasingly being applied in safety-critical areas where the programming errors that cause annoyance and lost productivity in more mundane applications become a threat to property and life.

The spread of computers into safety critical applications has been propelled by a rapid growth in our ability to efficiently describe and implement complex systems. Unfortunately there has not been a corresponding rapid growth in our ability to effectively verify the correctness of such systems. To date the best developed methods for improving system reliability are 1) verification using formal methods; and 2) software testing.

Formal methods use abstract system models such as process algebras or formal logic to represent systems and their requirements. Myriad techniques have been developed for demonstrating the equivalence of syntactically different formulations, deriving proofs of system properties, and demonstrating that a system model satisfies its requirements. Formal approaches to systems analysis are valuable because they offer absolute proof of system correctness with respect to requirements. However they have the distinct disadvantage that present techniques are applicable only to relatively small systems.

In contrast to formal methods, software testing is an analysis and validation technique that is applied to the end-product of development—the executable program—rather than a model. There are a great many techniques that explore system behaviors based on simplifying assumptions about their internal structure or use a randomized method to test as many behaviors as possible in a fixed amount of time. Software testing is valuable because it provides insight into the reliability of software systems of any size. However, software testing only provides qualitative assurances of reliability. Only in trivial instances does it provide an absolute proof of correctness.

In this paper we propose a method that brings together formal methods and testing techniques to assess the correctness of system models that are too large for analysis by formal methods alone. Specifically, a process algebra is used to model a large system and key subcomponents of the system are verified using traditional formal methods techniques. For other subcomponents and the assembled system we use testing techniques to sample behaviors and check their correctness. The end result of analysis is a system model that has been rigorously verified at its lowest level and tested to varying degrees at higher levels.

Modeling and analysis will be carried out using the Algebra of Communicating Shared Resources with Value Passing (ACSR-VP) and the Verification, Execution, and Rewrite System for ACSR (VERSA), a toolkit for constructing and analyzing ACSR processes. The analysis techniques are made tractable by two novel features of ACSR-VP and the VERSA toolkit. 1) The built-in priority mechanism of ACSR-VP can be used to impose a strict ordering on otherwise unordered events. This results in an asymptotic reduction in the size of the state space that is constructed during testing. 2) VERSA uses a top-down technique to construct the LTS corresponding to a system's state space. During testing this allows us to derive state machines that correspond exactly to the state space that is explored by the tests. This is in contrast to bottom-up state space derivation techniques that result in so many unreachable states being computed that the computation is intractable.

To demonstrate the feasibility of the approach we model and analyze the Sunshine Asynchronous Transfer Mode (ATM) packet switch architecture. ATM is a packet-switched networking technology that exploits small packet sizes and high data link speeds to support a wide range of network applications. Correct operation of the packet switch is essential to insuring correct and timely routing of packets. The Sunshine packet switch architecture was chosen because its modular design made it an ideal candidate for our hybrid analysis techniques, and its large state space made it impossible to verify with a pure formal methods approach.

The remainder of this paper is organized as follows. Section 2 describes the process algebra that we use to model real-time systems and the toolkit that we have built to support it. Section 3 presents the architecture of the Sunshine ATM switch that will be the vehicle for presenting our method. Section 4 derives an ACSR-VP model of a Sunshine ATM switch. Section 5 describes our methodology and presents the results of applying our hybrid verification techniques to the ATM switch model. Section 6 presents concluding remarks.

2 Algebra of Communicating Shared Resources with Value Passing and its Toolkit

In this section we describe the Algebra of Communicating Shared Resources with Value Passing (ACSR-VP). We also describe a toolkit called Verification, Execution, and Rewrite System for ACSR (VERSA) that we have implemented for analyzing processes written in the Algebra of Communicating Shared Resources (ACSR), a precursor of ACSR-VP. ACSR-VP is an enhanced version of ACSR that has an explicit notion of value passing between concurrent processes. ACSR-VP processes can be translated into ACSR processes by a method that will be presented, allowing VERSA to be used to analyze ACSR-VP processes.

2.1 ACSR-VP

The formal model of real-time systems that we are using to develop our analysis techniques is the Algebra of Communicating Shared Resources with Value-Passing, a timed process algebra that includes features for representing synchronization, value passing communication, time, temporal scopes, resource requirements, and priorities. This section presents a summary of ACSR-VP's syntax and semantics necessary to understand the descriptions of models to be analyzed. For a complete exposition of the algebra see [LBGG94, BG94, CLX95]. The description of ACSR-VP presented here is based on a discrete-time model. The analysis techniques presented in this paper are equally applicable to a dense-time ACSR-VP that could be derived from [BG94]. We omit further consideration of the dense-time model due to space limitations.

Event and Action Model. ACSR-VP is based on the synchronization model of CCS[Mil89], with the addition of (1) value passing communication; and (2) synchronous timed steps to model time consuming tasks and their resource requirements. Synchronization takes place through untimed *events* denoted by a pair (l, p), where l is the *label* of the event, and p is its *priority*. We use $\mathcal{D}_E$ to denote the domain of events, and let e, f and g range over $\mathcal{D}_E$. We use $l(e)$ and $\pi(e)$ to represent the label and priority, respectively, of the event e. If an event transmits values a "!" operator and value vector will be appended to it, as in $(l, p)!\langle 3, 4, 5\rangle$. If an event receives values a "?" operator and variable vector will be appended to it, as in $(l, p)?\langle a, b, c\rangle$.

$$
\begin{array}{rcl}
P & ::= & \mathrm{NIL} \mid A{:}P \mid e\,.\,P \mid be \rightarrow P \mid P_1 + P_2 \mid P_1 \| P_2 \\
 & & \mid\ P \backslash F \mid P \backslash\backslash I \mid P[R_e, R_a] \mid rec\ X.P \mid \mathsf{C}(\underline{x}) \\
e & ::= & (\tau, ve) \mid (l, ve)?\langle \underline{x} \rangle \mid (l, ve)!\langle \underline{ve_1} \rangle \\
A & ::= & \emptyset \mid \{S\} \\
S & ::= & (r, ve) \mid (r, ve), S
\end{array}
$$

Figure 1: ACSR-VP Syntax

An *action* that consumes resources from a set $\mathcal{R}$ at priorities from $\mathbb{N}$ for one unit of time is drawn from the domain $\mathbb{P}(\mathcal{R} \times \mathbb{N})$, with the restriction that each resource be represented at most once. As an example, the singleton action $\{(r, p)\}$ denotes the use of some resource $r \in \mathcal{R}$ running at the priority level p for one unit of discrete time. The action $\emptyset$ represents idling for one time unit, since all resources are inactive. We use $\mathcal{D}_R$ to denote the domain of timed actions, and we let A, B and C range over $\mathcal{D}_R$. We define $\rho(A)$ to be the set of resources used by the action A; e.g., $\rho(\{(r_1, p_1), (r_2, p_2)\}) = \{r_1, r_2\}$. We also use $\pi_r(A)$ to denote the priority level of the resource r in the action A; e.g., $\pi_{r_1}(\{(r_1, p_1), (r_2, p_2)\}) = p_1$. By convention, if r is not in $\rho(A)$, then $\pi_r(A) = 0$.

Syntax and Semantics. We use x, y for value variables, ve, ve_1 for value expressions and be, be_1 for boolean expressions. We denote a vector of value variables, value expressions and boolean variables as $\underline{x}$, $\underline{ve}$ and $\underline{be}$, respectively. Also, we use l, l_1 to denote event labels and r, r_1 to denote resources. F and I represent sets of event labels and resources, respectively. R_e and R_a are event label and resource relabeling functions, respectively. We assume that a process name, denoted by C, is associated with the parameters it takes. ACSR-VP's syntax is defined by the BNF grammar shown in Figure 1. Parenthesis may be introduced to clarify the grouping of operators. Process expressions may be bound to names and names may be referenced in process expressions.

NIL is a process that executes no action (i.e., it is deadlocked). The two prefix operators correspond to the two types of actions. The first, $A : P$, executes a timed, resource-consuming action A, consumes one unit of discrete time, and proceeds to the process P. The second prefix operator, $e.P$, executes the instantaneous event e, and proceeds to P. The conditional operator $be \rightarrow P$ continues as P if be evaluates to *true*, otherwise it executes no action, like NIL. The Choice operator $P_1 + P_2$ represents nondeterminism – either of the processes may be chosen to execute, subject to the event offerings and resource limitations of the environment. The operator $P_1 \| P_2$ is the concurrent execution of P_1 and P_2.

The restriction operator, $P \backslash F$, limits the behavior of P by disallowing any externally observable events with labels in F. The resource hiding operator, $P \backslash\backslash I$, hides the resources consumed by P that appear in $I \subseteq \mathcal{R}$ by eliding them from any observable resource consuming steps they may appear in. The relabeling operator, $P[R_e, R_a]$, relabels the externally observable events of P according to the relabeling function R_e and the resources of P according to the relabeling function R_a.

The process $rec\ X.P$ denotes standard recursion, allowing the specification of infinite behaviors. A process variable X is *free* in a process expression if it is not contained within the scope of a $rec\ X$ operator. An *agent* is a syntactically valid process expression with no free process variables.

In [CLX95] the formal semantics of ACSR-VP is given in two steps. The first is an operational semantics that describes the translation of ACSR-VP agent expressions into the unprioritized labeled transition system (LTS) "$\longrightarrow$". The second step uses a preemption relation defined for events and actions to incorporate notions of preemption and priority, producing the prioritized transition system "$\longrightarrow_\pi$".

The following rules describing the Choice operator are typical of the rules that describe the translation to the unprioritized transition system:

$$\textbf{ChoiceL} \quad \frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'} \qquad\qquad \textbf{ChoiceR} \quad \frac{Q \xrightarrow{\alpha} Q'}{P + Q \xrightarrow{\alpha} Q'}$$

These rules state that if an agent P is capable of executing α (an event or action) and proceeding as P', then $P+Q$ is capable of doing the same. The case for events and actions of Q is symmetric. Thus the choice operator "+" is given meaning in terms of transitions in $\longrightarrow$.

Preemption and Prioritized Transitions. The prioritized transition system is based on *preemption*, which incorporates ACSR-VP's treatment of synchronization, resource-sharing, and priority. The definition of preemption is straightforward. Let "$\prec$", called the *preemption relation*, be a transitive, irreflexive, binary relation on actions. Then for two actions α and β, if $\alpha \prec \beta$, we can say that "α is preempted by β." This means that in any real-time system, if there is a choice between executing either α or β, it will always execute β.

Definition 2.1 (Preemption Relation) *For two actions, α, β, we say that β preempts α ($\alpha \prec \beta$), if one of the following cases holds:*

1. Both α and β are timed actions in $\mathcal{D}_R$, where
$$(\rho(\beta) \subseteq \rho(\alpha)) \wedge (\forall r \in \rho(\alpha).\pi_r(\alpha) \leq \pi_r(\beta)) \wedge (\exists r \in \rho(\beta).\pi_r(\alpha) < \pi_r(\beta))$$
2. Both α and β are events in $\mathcal{D}_E$, where $\pi(\alpha) < \pi(\beta) \wedge l(\alpha) = l(\beta)$
3. $\alpha \in \mathcal{D}_R$ and $\beta \in \mathcal{D}_E$, with $l(\beta) = \tau$ and $\pi(\beta) > 0$. □

Case (1) shows that the two timed actions, α and β, compete for common resources, and in fact, the preempted action α may use a superset of β's resources. However, β uses all of the resources that have at least the same priority level as α (recall that $\pi_r(B)$ is, by convention, 0 when r is not in B). Also, β uses at least one resource at a higher level.

Case (2) shows that an event may be preempted by another event having the same label but a higher priority.

Finally, case (3) shows the only case in which an event and a timed action are comparable under "$\prec$." That is, if $n > 0$ in an event (τ, n), we let the event preempt any timed action.

The following examples show some comparisons made by the preemption relation, "$\prec$."

•$\{(r_1,2),(r_2,5)\} \prec \{(r_1,7),(r_2,5)\}$	•$\{(r_1,2),(r_2,5)\} \not\prec \{(r_1,7),(r_2,3)\}$
•$\{(r_1,2),(r_2,0)\} \prec \{(r_1,7)\}$	•$\{(r_1,2),(r_2,1)\} \not\prec \{(r_1,7)\}$
•$(\tau,1) \prec (\tau,2)$	•$(a,1) \not\prec (b,2)$
•$(a,2) \prec (a,5)$	•$\{(r_1,2),(r_2,5)\} \prec (\tau,2)$

We define the prioritized transition system "$\rightarrow_\pi$," which refines "$\rightarrow$" to account for preemption.

$$
\begin{aligned}
XBIn_{port} \quad = \quad & (XIn_{port}, 1)?\langle key, pay\rangle.\{(Path_{port}, 3), (Idler_{1-port}, 3)\} : XBSort_{port}(key, pay) \\
& + \{(Path_{port}, 1), (Idler_{1-port}, 1)\} : XBSort_{port}(KeyRange, 0) \\
& + \{(Path_{port}, 2), (Idler_{port}, 2)\} : \emptyset : XBIn_{port}
\end{aligned}
$$

$$
\begin{aligned}
XBSort_0(key, pay) \quad = \quad & (swap, 1)!\langle key, pay\rangle.(swap, 1)?\langle key2, pay2\rangle. \\
& \qquad (key \leq key2) \rightarrow XBOut_0(key, pay) \\
& \qquad + (key2 < key) \rightarrow XBOut_0(key2, pay2) \\
XBSort_1(key, pay) \quad = \quad & (swap, 1)?\langle key2, pay2\rangle.(swap, 1)!\langle key, pay\rangle. \\
& \qquad (key \geq key2) \rightarrow XBOut_1(key, pay) \\
& \qquad + (key2 > key) \rightarrow XBOut_1(key2, pay2)
\end{aligned}
$$

$$
\begin{aligned}
XBOut_{port}(key, pay) \quad = \quad & (key < KeyRange) \rightarrow (XOut_{port}, 1)!\langle key, pay\rangle, 1).\emptyset : XBIn_{port} \\
& + (key = KeyRange) \rightarrow \emptyset : XBIn_{port}
\end{aligned}
$$

$$
XBar \quad = \quad (XBIn_0 || XBIn_1) \backslash\backslash \{Path_0, Path_1, Idler_0, Idler_1\} \backslash \{swap\}
$$

Figure 2: ACSR-VP model of a 2×2 Switching Element

Definition 2.2 *The labeled transition system "$\rightarrow_\pi$" is defined as follows: $P \xrightarrow{\alpha}_\pi P'$ if and only if*

1. *$P \xrightarrow{\alpha} P'$ is an unprioritized transition, and*

2. *There is no unprioritized transition $P \xrightarrow{\beta} P''$ such that $\alpha \prec \beta$.* □

Example: A 2×2 Crossbar Switching Element. We demonstrate the main features of the ACSR-VP paradigm by presenting two approaches to the specification of a 2×2 crossbar switching element. A 2×2 switching element sorts a pair of inputs according to key field values embedded in the inputs. Since only two input values are involved, sorting reduces to reversing the order of the inputs when the input keys are out of order. 2×2 switching elements are the basic building block of the Sunshine network.

The processes shown in Figure 2 implement the switch as process *XBar*. Concurrent port processes $XBIn_0$ and $XBIn_1$ receive the data for ports 0 and 1 via the indexed input event labels XIn_{port} with parameters *key* and *pay*. The *port* subscript represents the port number the data is addressed to (0 or 1), and the *key* variable represents the key value of the input (from the range $[0, KeyRange)$), and the *pay* variable represents the payload, or data content of the input (from the range $[0, PayRange)$). The processed data is output via the indexed output event labels $XOut_{port}$ with parameters *key* and *pay*.

Figure 3 illustrates the structure and operation of the *XBar* process. Process *XBar* uses two concurrent port processes that (1) autonomously receive inputs; (2) exchange their input values so each port process can make a comparison internally; and (3) output the smaller (port 0) or larger (port 1) value. The dashed lines indicate the point at which one unit of discrete time passes.

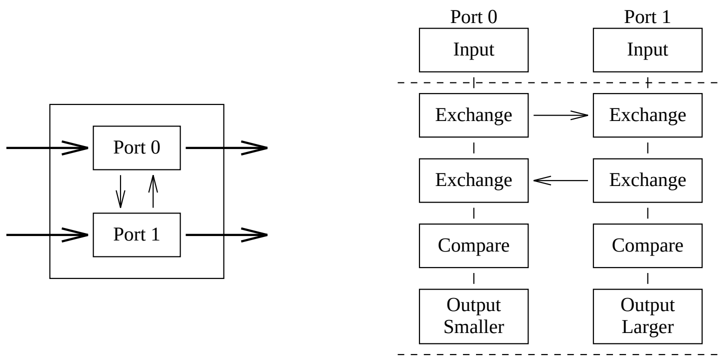

Figure 3: Structure and Operation of 2×2 Crossbar Model

When an input value is received on port *port* the $XBIn_{port}$ process consumes the $Path_{port}$ and $Idler_{1-port}$ resources for one time unit and continues as $XBSort_{port}$. The *XBSort* processes share the key values between port processes and proceed to the output processes $XBOut_{port}$. If each port detects that the key values are incorrectly ordered they autonomously swap their key and payload values for the values communicated by the other port during the sort step. Otherwise key and payload values are passed through unchanged. The *XBOut* processes output the key and payload values received from the sort step, consume one unit of time idling and restart the *XBIn* processes.

The second and third lines of the the $XBIn_{port}$ definition provide alternative actions that are executed when one input is inactive, and both inputs are inactive, respectively. For example, if port 0 receives an input value and port 1 does not, then port 0 will execute a $\{(Path_0, 3), (Idler_1, 3)\}$ action. Port 0's use of the $Idler_1$ resource preempts port 1's attempt to use it (in line three of the definition of $XBIn_{port}$) so port 1 must execute the other timed action step $\{(Path_1, 1), (Idler_0, 1)\}$ that passes the out-of-range key value *KeyRange* to $XBSort_1$. This value is later filtered by the $XBOut_1$ process and it is not output as part of an event. The use of a key value larger than any valid input key insures that any singleton inputs will be forwarded to port 0. If neither port receives input then both ports execute the timed action step in which $Path_{port}$ has a priority of 2, preempting the lower priority possibility. This *Path* consumes two units of time and restarts the *XBIn* processes without performing comparisons.

The *XBar* process hides its use of the *Path* and *Idler* resources to avoid conflict between concurrent $XBar's$. Restriction of the *swap* event labels insures that exchanges within *XBar* use local values.

The *XBar* process of Figure 2 has the virtue that it is a relatively compact and concise description of the switching element. However, because of the complex internal actions that are used in this implementation its correctness is not readily apparent. In contrast, the processes of Figure 4 present a more direct implementation of the switch requirements at the expense of a larger and more cumbersome ACSR-VP representation.

The *XBarSeq* process of Figure 4 is a switching element implementation that is equivalent (in a sense that will be defined below) to *XBar* of Figure 2. However *XBarSeq* uses no parallel composition. The sorting of input and output values is implicit in the the events that *XBarSeq* offers at any given time. Figure 5 illustrates the operation of *XBarSeq*.

Initially *XBarSeq* accepts any input event. If an input event is received on port p then *XBarSeq* proceeds to a state that allows any input key and payload to be accepted on the complementary

$$
\begin{aligned}
\mathit{XBarSeq} \;=\;& (\mathit{XIn}_0,1)?\langle \mathit{key0}, \mathit{pay0}\rangle. \\
&\qquad (\mathit{XIn}_1,1)?\langle \mathit{key1}, \mathit{pay1}\rangle.\mathit{XBarSeq'}(\mathit{key0}, \mathit{pay0}, \mathit{key1}, \mathit{pay1}) \\
&\qquad + \emptyset : (\tau,2).(\tau,2).(\mathit{XOut}_0,1)!\langle \mathit{key0}, \mathit{pay0}\rangle.\emptyset : \mathit{XBarSeq} \\
&+ (\mathit{XIn}_1,1)?\langle \mathit{key1}, \mathit{pay1}\rangle. \\
&\qquad (\mathit{XIn}_0,1)?\langle \mathit{key0}, \mathit{pay0}\rangle.\mathit{XBarSeq'}(\mathit{key0}, \mathit{pay0}, \mathit{key1}, \mathit{pay1}) \\
&\qquad + \emptyset : (\tau,2).(\tau,2).(\mathit{XOut}_0,1)!\langle \mathit{key1}, \mathit{pay1}\rangle.\emptyset : \mathit{XBarSeq} \\
&+ \emptyset : \emptyset : \mathit{XBarSeq}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{XBarSeq}'(\mathit{key0}, \mathit{pay0}, \mathit{key1}, \mathit{pay1}) \;=\;& \emptyset : (\tau,2).(\tau,2). \\
&\quad (key0 \leq key1) \rightarrow \quad (\mathit{XOut}_0,1)!\langle \mathit{key0}, \mathit{pay0}\rangle. \\
&\qquad\qquad (\mathit{XOut}_1,1)!\langle \mathit{key1}, \mathit{pay1}\rangle.\emptyset : \mathit{XBarSeq} \\
&\qquad\qquad + (\mathit{XOut}_1,1)!\langle \mathit{key1}, \mathit{pay1}\rangle. \\
&\qquad\qquad (\mathit{XOut}_0,1)!\langle \mathit{key0}, \mathit{pay0}\rangle.\emptyset : \mathit{XBarSeq} \\
&\quad + (key0 > key1) \rightarrow \quad (\mathit{XOut}_0,1)!\langle \mathit{key1}, \mathit{pay1}\rangle. \\
&\qquad\qquad (\mathit{XOut}_1,1)!\langle \mathit{key0}, \mathit{pay0}\rangle.\emptyset : \mathit{XBarSeq} \\
&\qquad\qquad + (\mathit{XOut}_1,1)!\langle \mathit{key0}, \mathit{pay0}\rangle. \\
&\qquad\qquad (\mathit{XOut}_0,1)!\langle \mathit{key1}, \mathit{pay1}\rangle.\emptyset : \mathit{XBarSeq}
\end{aligned}
$$

Figure 4: Sequential ACSR-VP model of a 2×2 Switching Element

port (1-p). Once both input values have been received computation advances through an idle step and two τ steps. Then a state is reached that outputs the keys and payloads in the correct order on the appropriate ports. The sorting of key values is implicit in the conditional operators of process *XBarSeq'*—the only *XOut* events that can occur are the events that produce the keys and payloads in the correct order. Once the outputs have been delivered *XBarSeq* executes another idle step and returns to the initial state.

Bisimulation and Strong Equivalence. Our formal analysis techniques are based on process equivalence, where we attempt to prove that a process P is equivalent to a process Q. Typically, P is an abstract operational specification of the problem, while Q is a more detailed implementation. The objective is to show that the two processes are operationally equivalent. Equivalence between two ACSR-VP processes is based on the concept of *bisimulation*[Par81], which compares the computation trees of the two processes.

Definition 2.3 *For a given transition system "$\longrightarrow$", any binary relation r is a* strong bisimulation *if, for $(P, Q) \in r$ and $\alpha \in \mathcal{D}$,*

1. *if $P \xrightarrow{\alpha} P'$ then, for some Q', $Q \xrightarrow{\alpha} Q'$ and $(P', Q') \in r$, and*
2. *if $Q \xrightarrow{\alpha} Q'$ then, for some P', $P \xrightarrow{\alpha} P'$ and $(P', Q') \in r$.* □

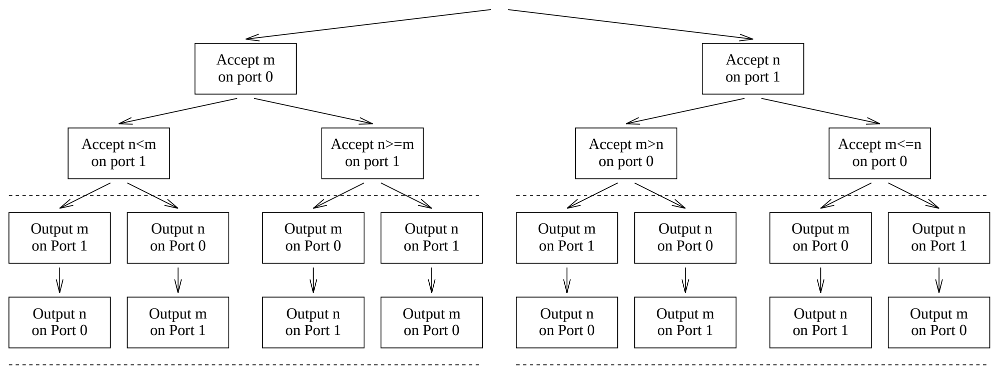

Figure 5: Operation of Sequential 2×2 Crossbar Model

In other words, if P (or Q) can take a step on α, then Q (or P) must also be able to take a step on α with both of the next states also bisimilar. There are some very obvious bisimulation relations, e.g. $\emptyset$ (which certainly adheres to the above rules) or syntactic identity. However, using the theory found in [Mil80], it is straightforward to show that there exists a largest such bisimulation over "$\rightarrow$," which we denote as "$\sim$." This relation is an equivalence relation, and is a congruence with respect to ACSR-VP's operators[Ger91]. Similarly, "$\sim_\pi$" is the largest strong bisimulation over "$\rightarrow_\pi$," and we call it *prioritized strong equivalence.*

That *XBar* $\sim_\pi$ *XBarSeq* will be proved mechanically in Section 2.2.

Emulating Value Passing. As we mentioned previously, we intend to use the VERSA toolkit developed for the process algebra ACSR to analyze ACSR-VP processes. However, ACSR does not include an explicit mechanism for passing values in communication. Instead, value passing is implemented as subscripts on event labels, as suggested by Milner[Mil89]. For example, $e_{3,7,1}$ is the event e passing the tuple of values $\langle 3, 7, 1\rangle$. To facilitate the description of value passing processes, which often include choice between a larger number of alternative input events, we allow the use of summation notation to compactly represent choice between several alternatives. For example, $\sum_{i=0}^{n} P_i$ is equivalent to $P_1 + P_2 + \cdots + P_n$.

The following example illustrates the use of summation over indexed events to emulate value passing. Consider a thermostat process responsible for sensing the temperature using the event *temp* and continuing as process *Heat* if the temperature is below 68 degrees, *Idle* if the temperature is 68 to 78 degrees, or *Cool* if the temperature rises above 78 degrees. ACSR-VP, with its built-in notion of value passing, might represent this process as follows:

$$
\begin{array}{rcl}
\mathit{Therm} & = & (\mathit{temp}, 1)?\langle T\rangle.\mathit{Therm'}(T) \\
\mathit{Therm'}(T) & = & \quad (T < 68) \rightarrow \mathit{Heat} \\
 & & + \, (T > 78) \rightarrow \mathit{Cool} \\
 & & + \, ((T \geq 68) \wedge (T \leq 78)) \rightarrow \mathit{Idle}
\end{array}
$$

In this implementation a value is received by communication event *temp* and stored in a variable T, and a conditional operator "$\rightarrow$" selects the next process based on T's value.

To achieve similar behavior using ACSR we would create the following process:

$$\begin{aligned} Therm &= \sum_{i=0}^{67}(temp_i, 1).Heat + \\ & \sum_{i=68}^{78}(temp_i, 1).Idle + \\ & \sum_{i=79}^{120}(temp_i, 1).Cool \end{aligned}$$

In contrast to the pure value-passing implementation, here the event that is received determines the next process. There is no explicit conditional operator since the decision is implicit in the process structure. Note also that the range of values for the input temperature must be explicitly enumerated in this scheme.

2.2 VERSA

The Verification, Execution, and Rewrite System for ACSR (VERSA) is an integrated set of tools for developing system models using ACSR. We present here an overview of the features that are pertinent to the development of the ATM switching system example. For a more complete presentation of VERSA's capabilities see [CLX94].

VERSA is an interactive system with a text based interface. Users enter their process descriptions with a syntax that is as close to the pure ACSR syntax as the keyboard will allow. Readability of processes and the ability to describe systems with repetitive structure is enhanced by a macro facility that allows manifest constants and parameterized macros to be used in process descriptions. Description of modular systems is facilitated by a file inclusion facility that allows system descriptions to be broken up into manageable components and re-combined during analysis.

System Features. Description and analysis of system models in ACSR is facilitated by the following features and capabilities:

- *Syntax Checking.* Processes are checked for syntactic and semantic correctness as they are entered. When errors are discovered they are fed back to the user immediately.

- *Compilation.* Finite state ACSR agent expressions (processes in which all process variables referenced are bound) can be translated into their underlying LTS representation. Translation is carried out by a top-down technique that builds the LTS by applying the ACSR semantic rules. The translation is top-down in the sense that it begins with the initial process term and computes the process terms that derive from it by one transition. This is in contrast to bottom-up techniques that recursively form the LTS from the components of the initial process term.

- *State Space Analysis.* Once an LTS has been constructed it can be explored to determine various key properties including (1) state and edge count; (2) presence of deadlocks; (3) presence of Zeno states (states involved in cycles of untimed events); and (4) presence of states that can "stop the clock" (states that are deadlocked if synchronization on external events does not occur).

Pure ACSR	VERSA Syntax
$a, \overline{a}, a', a_i$	`a, 'a, a', a[i]`
τ	`t` or `tau`
NIL	Any capitalization of `NIL`
$\|\|$ (composition)	`||` or `|`
$P\Delta_t^a(Q,R,S)$	`scope(P,a,t,Q,R,S)`
∞	`inf` or `infinite` or `infinity`
$[P]_I$	`[P]I`
$P\backslash\backslash I$	`P\\I`
$P[R_e, R_a]$	`P%[`R_e`,`R_a`]`
R_e and R_a	`{l1'/l1,l2'/l2,`$\cdots$`,ln'/ln}`
$\emptyset$	`{}` or `idle`
Definition of index i	{*i,begin,end,step,predicate*}
$\sum_{i=0}^{N} P$	`Choice[P {i,1,N}]`
$\prod_{i=0}^{N} P$ (indexed composition)	`Parallel[P {i,1,N}]`
$\{a_i \| i = 1,\cdots,N \wedge pred(i)\}$	`Set[a[i]{i,1,N,1,pred(i)}]`

Table 1: Pure ACSR vs. VERSA Syntax

- *Bisimulation Checking.* VERSA can test whether two finite state ACSR agent expressions are bisimilar. Bisimulation checking is implemented using a state space minimization based algorithm[KS90].

This list includes only those features of VERSA that will be useful in analyzing the ATM switching system. For a complete description of VERSA's capabilities see [CLX94].

Process Syntax. Wherever possible the standard ACSR syntax has been retained in VERSA. The changes made to accommodate the capabilities of most keyboards are outlined in Table 1.

An identifier (process variable names, resource names, event labels, etc.) can be any length string of alphanumerics or the underscore character that begins with a letter. An identifier preceded by an apostrophe (e.g., `'id`) is an output event label, as in $\overline{id}$. The types of identifiers (process variable vs. resource names, etc.) are inferred from their first use.

Process expressions are bound to process names by the following syntax:

<process name> `=` *<process expression>*`;`

Process expressions can be bound to a sequence of process names by the following syntax:

<indexed process name> `=` *<process expression>* *<index definitions>*`;`

Definitions of indices take the form shown in Table 1, where i is the index being defined; *begin* is an integer expression defining the initial value for i; *end* is an integer expression for the largest allowed value of i; *step* is an integer expression for the value i is incremented by; and *predicate* is a boolean predicate that is tested for each value of i. If *predicate* is *false*, the index value is not used.

Preprocessing facilities including file inclusion and macro definition are implemented with the standard C programming language syntax. That is, `#include` is used for file inclusion and `#define` is used for constant and parameterized macro definition. The `#ifdef` and `#ifndef` conditional compilation directives are also implemented. Comments can be added to process descriptions using the standard C `/*` and `*/` block delimiters or the C++ line delimiter, `//`.

Example: 2×2 **Crossbar Switching Element Revisited.** Section A of the Appendix presents a complete translation of the switching element of Figure 2 into VERSA syntax. It also augments the original process description with indices that allow multiple copies of the process to be created with priorities assigned to their events. Thus the original *XBar* specification is translated into `XBar[pty]` with `pty` taken from a range of priority values, one for each different crossbar that is created.

The VERSA description begins with header comments that describe the operation of the `XBar` process in detail. Processes `XBIn` and `XBSort` are translations of the *XBIn* and *XBSort* processes of Figure 2. The `pty` index is added to the process name to distinguish multiple copies running at different priority levels, and the `pty` index as used as the priority level in events.

Process `XBOut` is a similar translation of *XBOut* with the addition of the `XBarReversal` parameter. If the `XBarReversal` name is defined the sort order is reversed (the larger value is output on port 0). The description of `XBar` uses the shorthand notation `{*}` to hide all resources instead of naming them explicitly as they were in the original ACSR version of Figure 2.

To demonstrate the use of ACSR's equivalence testing features to verify the correctness of `XBar` we saved the description of `XBar` to the file `XBar.acsr`. We then performed a similar translation for *XBarSeq* of figure 4 and saved it as `XBarseq.acsr`. Then we carried out the steps shown in Figure 6.

In Figure 6 lines 1 through 3 set the priority parameters to generate exactly one process named `XBar[1]`. Lines 4 and 5 set the ranges for the key and payload values, respectively. Lines 6 and 7 read the specifications one at a time, performing syntax and semantic checking as the files are read. Line 8 shows the switch of Figure 2 (process `XBar[1]`) being compared to the purely sequential specification (process `XBarSeq`, adapted from Figure 4). The output indicates that the LTS's that were constructed for each process were equivalent according to our definition of prioritized strong equivalence.

With the aid of VERSA's equivalence testing features we have proved the correctness of `XBar[1]`, our basic building block for the ATM switching system, subject to the following assumptions:

- `XBar[1]` is an accurate encoding of *XBar*.
- `XBarSeq` is an accurate encoding of *XBarSeq*.
- *XBarSeq* is correct (which is apparent from inspection of its simple sequential structure).

Figure 6 also includes a sample of the output generated by entering the interactive interpreter and analyzing the state space of the LTS. Line 9 requests that VERSA create the LTS corresponding to `XBar[1]` and enter interactive interpretation mode. The lines of output that follow display the edges that can be traversed from the initial state of `XBar[1]`. Line 10 requests state space exploration, and the lines of output that follow are the result. According to the output, the LTS for `XBar[1]` contains 615 states and 1035 edges. None of the states are deadlocked, although there are 108 states that require external synchronization or a deadlock will occur.

```
ready> #define PtyMin 1                                                    (1)
ready> #define PtyMax 1                                                    (2)
ready> #define PtyStep 1                                                   (3)
ready> #define KeyRange 6                                                  (4)
ready> #define PayRange 2                                                  (5)
ready> #include "XBar.acsr"                                                (6)
ready> #include "XBarSeq.acsr"                                             (7)
ready> XBar[1] == XBarSeq?                                                 (8)
    true (by prioritized strong equivalence)
ready> XBar[1]!                                                            (9)
XBar[1] <1>  --(XIn[0,0,0],1)-->
        <2>  --(XIn[0,0,1],1)-->
        ...
        <25>  --{}-->  no-name
[XBar[1]] ready> show stats                                                (10)
State machine contains 615 reachable states (0 deadlocked), 1035 edges.
Edges represent 171 timed transitions,
                336 internal actions, and
                528 external untimed actions.
108 non-deadlocked states are capable of stopping the clock.
Time to compute LTS: 3 seconds user time; 0.13 seconds system time.
```

Figure 6: VERSA Verification of 2×2 Switching Element

3 The Sunshine ATM Switch Architecture

Sunshine[GHM+91, Zeg93] is a packet switch that uses sorting networks to route packets from input ports to output ports based on a destination address field. Packet loss due to output port contention is mitigated by a feedback loop that reroutes packets to the input ports rather than dropping them.

Figure 7 presents a simplified representation of the basic Sunshine architecture. Packets arriving at the N input ports contain an address field that indicates which of the N output ports the data should be routed to. At the same time T recycled packets are received from the upper feedback loop. These are packets that could not be routed on their first pass due to output port contention. (Contention occurs when two or more packets are destined for the same output port address simultaneously.) Once received at the input ports of the first sorting network the N new packets and T recycled packets are treated identically. Thus the main body of the switch is responsible for routing $N + T$ packets to N output ports and T overflow paths.

The first sorter uses a Batcher sorting network to sort packets by their output port address. The trap stage examines contiguous packets to determine when multiple packets are destined for the same output port address. Within each group of packets destined for the same output port one packet per port is marked for output (*output* bit set to 1) and the remainder for recirculation (*output* bit set to 0). The *output* bit is prepended as the high-order bit of the output port address

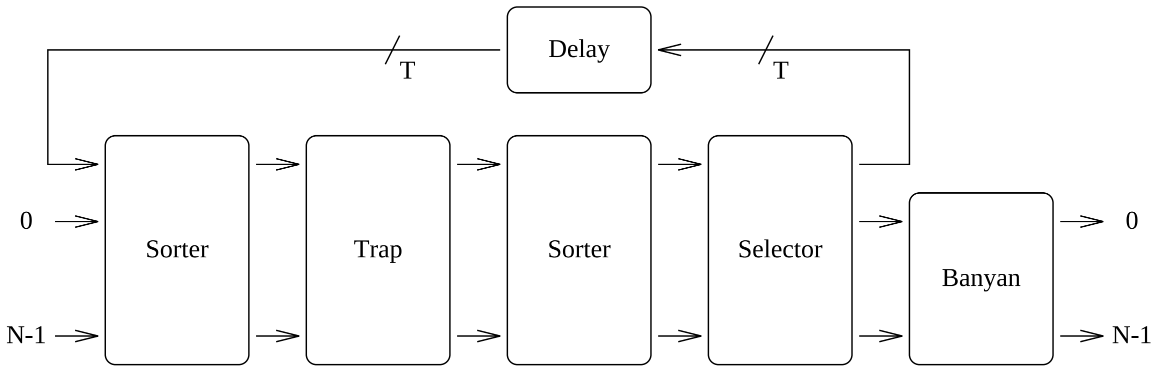

Figure 7: Sunshine Architecture

and the packets are re-sorted by the second sorter. The second sort separates the packets into two contiguous groups: one for recirculation and one for output. The selector then selects T packets for recirculation, drops the remainder of the packets marked for recirculation, and forwards the packets selected for output to the Banyan network. The Banyan network routes the packets to their designated output address. The Banyan network is guaranteed to be non-blocking and to route the packets to the correct output ports because its inputs arrive as a contiguous sorted group.

Modeling Parameters We adopt the following simplifications and parameter valuations for the Sunshine architecture in constructing our model:

1. There will be six input and output ports and two feedback paths. Using the notation of Figure 7, $N = 6$ and $T = 2$.

2. Data packets will be composed of a key value (the output port address) and a payload (a data value to be carried from the input port to the output port). The payload will be a one-bit value.

3. The actual Sunshine architecture uses priority values and a recirculation counter to determine which packets are selected for output, selected for recirculation, or dropped. In order to maintain a reasonable level of complexity in our models we omit these features altogether. Instead, output and recirculation priority is based on the value of the payload bit and the order in which packets are received at the input ports.

4. Output port queues have capacity one; concentrators are omitted.

4 Modeling a Sunshine Switch with ACSR-VP

Our model of the Sunshine switch is an attempt to emulate not just the functional requirements but also the actual architecture of the switch. Each of the modules shown in Figure 7 has been modeled by an individual process and the final switch process is formed by interconnecting these processes in the manner required by the switch architecture. The sections that follow describe in detail the modeling of each of the subassemblies of the switch and their final interconnection into a complete model of a Sunshine switch.

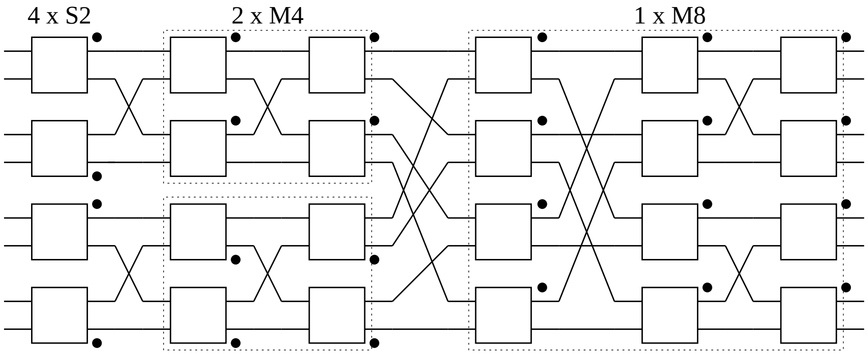

Figure 8: 8×8 Batcher Sorting Network

4.1 Sorting Networks

According to the parameters laid out in Section 3 there will be eight data paths through the switch: six for new inputs and two for recirculated inputs. Thus the two sorting networks will be of dimension 8×8 (eight inputs sorted to eight outputs). For the first sorting network the sort key will be an output port number in the range $[0, 5]$. For the second network the sort key will include an extra bit of data indicating whether the data has been tagged by the trap stage for output. Thus the sort key will be in the larger range of $[0, 11]$. Since the functional characteristics of the two sorting networks are otherwise identical we will use a parameter (`KeyRange`) to represent the range of key values.

Our model uses an 8×8 Batcher network constructed from 2×2 crossbar switching elements to realize the 8×8 sorter. Figure 8 presents the Batcher sorting network architecture. In the figure each square represents a 2×2 crossbar switching element and the lines between the crossbar switches represent the interconnections needed to realize an 8×8 sorter. The dots next to the output lines of the crossbar switches indicate the direction of the sort, with the marked line receiving the input value with smaller magnitude.

The Batcher network can be thought of as a three stage merge sort. The first stage uses four 2×2 crossbar switches to order each adjacent pair of inputs. Once the inputs have been reordered into sorted pairs the next stage uses a pair of 4×4 merge networks to merge the sorted input pairs into a pair of sorted sequences of length four. Finally, an 8×8 merge network is used to merge the two sorted subsequences of length four into a sorted sequence of length eight.

Appendix C presents the ACSR-VP implementation of the 4×4 merge network. Process `M4xbar[0][0]` is the crossbar switch in the upper-left corner of the 4×4 merge networks shown in Figure 8. It is a simple renaming of the inputs and outputs of the generic 2×2 crossbar process to conform to the naming and internal interconnection requirements of the merge network. Processes `M4xbar[0][1]`, `M4xbar[1][0]` and `M4xbar[1][1]` are defined similarly. Processes `M4Revxbar[*][*]` create the internal components for a 4×4 merge network with the sort order of internal crossbars reversed.

The `M4` process that represents the complete 4×4 merge network is formed by executing the four crossbar switches concurrently and hiding their internal connections. The resulting process is a 4×4 merge network with its internal structure hidden, its external inputs named `M4In`, and its external outputs named `M4Out`. `M4Rev`, the 4×4 merge network with reversed internal sort order

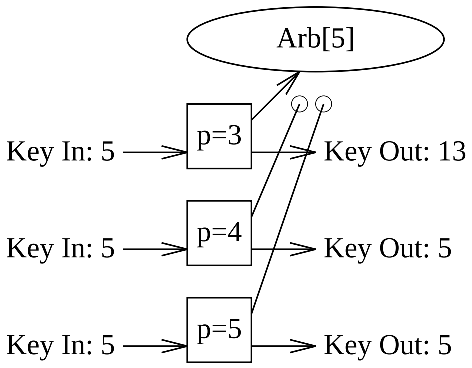

Figure 9: Example of Trap Stage Operation

is defined analogously.

Appendix D presents the ACSR-VP implementation of the 8×8 merge network. Twelve crossbar switch elements are used to construct the merge network according to the interconnection pattern shown in Figure 8. The process is similar to the 4×4 merge network, with renaming being used to create interconnections and the final `M8` process formed from the parallel composition of the twelve crossbar switch elements with their internal structure hidden. The resulting process is an 8×8 merge network with its internal structure hidden, its external inputs named `M8In`, and its external outputs named `M8Out`.

Appendix E presents the ACSR-VP implementation of the complete 8×8 Batcher sorting network. The first stage is formed by including four 2×2 crossbar switches, renaming their inputs to reflect the naming of the inputs for the Batcher network and renaming their outputs to begin the interconnection with the merge networks. Note the use of some reversed crossbar switches to achieve correct output ordering for each switch element. The next stage includes two 4×4 merge networks and renames their inputs and outputs to create interconnections with the first stage switch elements and the next merge network. Then, one 8×8 merge network is included.

Once all of the component processes have been created the sorter process is built in three steps. First, the upper crossbar pair of Figure 8 and the first 4×4 merge network are combined by parallel composition and their interconnection via `Internal` events is hidden from external processes. Then a similar process is created for the lower crossbar pair and the second 4×4 merge network. Finally the complete 8×8 Batcher sorting network is created by executing these two subassemblies in parallel with the 8×8 merge network and hiding their internal communications. The final product is an 8×8 sorting network with inputs `BatchIn` and outputs `BatchOut`. The sort key will be in the range [0, `KeyRange`) and the payload value will be in the range [0, `PayRange`). (`KeyRange` and `PayRange` are constants that will be `#define`*d* in the environment in which the sorting network is created.)

4.2 The Trap Stage

The trap stage is responsible for marking one of the packets destined for each output port for output and leaving the rest to recirculate. The ACSR-VP implementation of a trap stage shown in Appendix F uses ACSR-VP's priority arbitration mechanism to select the input that is routed for output.

The basic operation of the trap stage is illustrated in Figure 9. The squares represent data paths 3 through 5 (a subset of the complete set of eight data paths) and the oval represents the process responsible for selecting the data value that will be marked for output. The three data paths with key value 5 attempt to synchronize with the `Arb[5]` process through event `arbitrate[5]`, but only one will be allowed to succeed. The selection takes place through priority arbitration of the `arbitrate[5]` events, with payload value and path number being mapped through a function that assigns priority. Packets with payload 1 have a higher priority than packets with payload 0, and the lowest numbered path will have a higher event priority than all of the higher numbered paths attempting to communicate with the same event. As the example shows, only data path 3 is successful in its attempt to synchronize with `Arb[5]`, so its key value is increased and the other key values remain the same. (Increasing the key value by the total number of paths is how a data path's value is marked as selected for output.)

Appendix F shows the trap stage being constructed directly; no crossbar switching elements are used. The `TrapPath` processes represent the eight paths through the trap, and the `Arbitrator` process creates `Arb` processes that use priority arbitration to select one output value for each of the six output ports. The `Trap` process is the parallel composition of the data paths and the selection process with the internal arbitration event labels hidden. The final product is an 8×8 trap stage with inputs `TrapIn` and outputs `TrapOut`. The output port addresses received on the `TrapIn` inputs will be in the range $[0, \texttt{KeyRange})$. The output port addresses and selection flag that is sent to the `TrapOut` outputs will be in the range $[0, 2 \times \texttt{KeyRange})$. Payload values are copied from input to output without change.

4.3 The Selector

The selector stage described in Appendix G separates packets and forwards them to the recirculation area or the output ports depending on the key value received from the trap stage. Input packets are received on inputs named `SelIn`. If the key value received is in the range $[0, \texttt{KeyRange})$ the data is copied to a `SelDel` output port where it will be received by the delay stage for recirculation. If the key value received on `SelIn` is in the range $[\texttt{KeyRange}, 2 \times \texttt{KeyRange})$ the data is copied to a `SelBan` output port where it continues on its way to its output port address. Thus the selector performs its required function of forwarding packets marked for output to the output port area, forwarding some or all of the recirculating packets to the delay stage for recirculation, and dropping any packets that exceed the capacity of the recirculation paths.

The ACSR-VP process description shown in Appendix G uses the payload value and path number to determine whether to copy a packet to a `SelDel` port, a `SelBan` port, or drop the packet altogether. The entire process is carried out through input event priority arbitration and use of the choice operator. The packets destined for output are shifted down to path 0 using the lowest output path number determined by the `MinBan` process.

4.4 Banyan Output Network

The packets received by the Banyan stage in Figure 7 have been sorted into the correct order by the first sorter, and the selector insures that there will be at most one packet received for each output port. The task of the Banyan network is to distribute the compressed sequence of incoming packets to their respective output ports. For example, if two packets are received, one destined for output port 3 on input port 0 and the other destined for output port 5 on input port 1, the

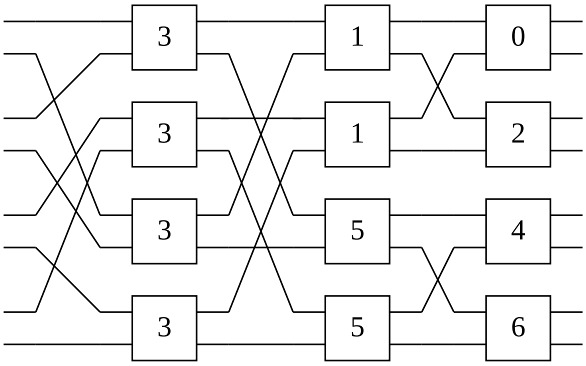

Figure 10: Banyan Network

Banyan network has to route the first packet from input port 0 to output port 3, and the second from input port 1 to output port 5.

The structure of the Banyan network is shown in Figure 10. Twelve 2×2 crossbar switching elements are divided into three, four switch stages to create the network. Inputs are shuffled to insure proper operation of the network. The crossbar switching elements require two inputs, so in cases where fewer than two inputs are available a default input value is built into each switch. (Each switching element in Figure 10 is labeled with its default value.) The default values are used only for internal comparisons by each switching element. They are never output by any switching element.

The following example illustrates the operation of the Banyan network. Consider a set of packets destined for output ports 0, 1, 3, 5 and 6. The Banyan network will receive the vector $\langle 0, 1, 3, 5, 6, -, -, -\rangle$. The initial shuffle indicated by the interconnections to the first stage of switching elements will reorder the packets so that the first stage receives $\langle 0, 6, 1, -, 3, -, 5, -\rangle$. The first stage sorts its incoming packet pairs (with three default values mixed in) and produces the output vector $\langle 0, 6, 1, -, 3, -, -, 5\rangle$. The shuffle between stages one and two results in the vector $\langle 0, 3, 1, -, 6, -, -, 5\rangle$ being received by the second stage. The second stage sorts these value pairs and generates the output vector $\langle 0, 3, 1, -, -, 6, 5, -\rangle$. The shuffle between the second and third stages results in the third stage receiving $\langle 0, 1, 3, -, -, 5, 6, -\rangle$, which is pair-wise sorted to produce the final output vector $\langle 0, 1, -, 3, -, 5, 6, -\rangle$. Thus the Banyan network has correctly routed the input packets from the five low-order inputs to their proper output ports.

The ACSR-VP process to realize an 8×8 Banyan network is shown in Appendix H. The first process, `DummyIO`, is the process that will be integrated with each crossbar to produce and consume the default inputs that are used when less than two real inputs are received. The dummy values can be recognized and discarded because they have a payload value that is out of range. The `DXbar` processes package each of the twelve crossbar switches with its respective `DummyIO` generator. Since the crossbar switches produce a stable sort, several of the `RDXBar` processes modify the `DXBar`'s outputs slightly to insure that sorting one dummy input and one real input within a crossbar switch always results in the real value being copied to the low-order output port.

Processes `B8[0]`, `B8[1]` and `B8[2]` rename the input and output ports of the crossbar switches to realize the interconnections specified in Figure 10. Finally, Process `Banyan8` creates an 8×8 Banyan network by executing the twelve crossbar switch processes in parallel and hiding all internal structure. Inputs are received by `B8In` events and outputs by `B8Out` events.

4.5 The Delay Module

The delay module process shown in Appendix I simply latches inputs it receives from the selector and makes them available to the first sorter stage.

4.6 Final Assembly

The process that realizes the Sunshine ATM switch from its components is shown in Appendix J. The process is constructed as follows. First, each of the components is created and the input and output events of each individual component are renamed to implement the required interconnections. Then the switch process is built up by combining the trap and second sorter stage, adding the first sorter stage, then the selector, Banyan, and delay stages. As each stage is added the internal communication between stages is restricted so that it will not be visible to processes that are not directly involved in the communication.

The model is bound to the `Sunshine` process variable. It has inputs `in[port]` with parameters `key` and `payload` and outputs `out[port]` with parameters `key` and `payload`, where `port` and `key` are drawn from $[0, 6)$ and `payload` from $[0, 2)$.

5 Analysis of the Sunshine Switch Model

With an algebraic model of the Sunshine switch in hand the next steps of a pure formal methods analysis of the model would be to (1) define a set of logical properties that the model should satisfy; (2) build a labeled transition system representation of the state space of the model; and (3) explore the state space to verify that it satisfies all of the logical properties of a correct implementation. (This process is generally referred to as *model checking*[CES86].) For "large" models like our ATM switch this process breaks down when we try to construct the state space.

A rough estimate of the size of the state space of the model is derived as follows. The sorter and Banyan stages are formed from a total of 60 concurrently executing copies of the crossbar switch. For the given data domain (six addresses and two payload values) the crossbar has a minimal state machine representation consisting of 410 states. Taking into account only the contribution of the state space of the crossbar switches, for the given data domain the Sunshine switch has a state space of $\mathcal{O}(410^{60}) \cong \mathcal{O}(10^{156})$ states. As a point of contrast, taking into account memory and time limitations, the largest state space ever constructed by the VERSA system is $\mathcal{O}(10^5)$. State of the art symbolic model checking techniques (using binary decision diagrams rather than a direct labeled transition system representation) claim $\mathcal{O}(10^{20})$ states[BCD$^+$90] for certain classes of problems.

5.1 Methodology

Given the size of the state space for our ATM switch model, model checking is not a viable approach for verifying the model's correctness. In its stead we will apply a combination of (1) state space reduction techniques during modeling; (2) algebra to LTS translation techniques optimized for testing; and (3) testing techniques. Although the results of our analysis will lack the certainty guaranteed my traditional model checking, arbitrarily high confidence levels can be obtained by adjusting the number of tests that are executed.

State Space Reduction through Priority Arbitration. Formalisms for describing concurrent systems typically represent simultaneous events through an *interleaved* semantics. That is, whenever events can occur concurrently, they are assumed to take place sequentially at the same time instant. For example, the process expression $(e_1, 1).NIL||(e_2, 1).NIL$ describes a process in which e_1 and e_2 occur simultaneously. According to the interleaved semantics this is equivalent to either $(e_1, 1).(e_2, 1).NIL$ or $(e_2, 1).(e_1, 1).NIL$.

Because the state space of a process has to represent all of the possible executions of the system, interleaved semantics results in all of the possible interleavings of simultaneous events being represented explicitly in a process term's labeled transition system. When a system is built out of several independent components the representation of all of the possible interleavings of the concurrent behaviors of the system's components is a major contributor to the "state space explosion" problem.

In our model, ACSR-VP's priority arbitration mechanism is used to mitigate the state explosion problem by imposing a strict ordering on the simultaneous execution of events. Each event of each crossbar that is included in the model is assigned a fixed priority. When multiple crossbars offer simultaneous events the event order is determined by priority arbitration rather than nondeterministic choice. For example if all 40 crossbars in the Sunshine model simultaneously offer one τ step, an unprioritized model would include all 40! potential paths, arriving at the same end state. However, assigning a distinct priority to the events executed by each crossbar allows us to construct a single path based on the event order imposed by the relative priorities of the offered events.

Of course there are limitations to the application of this technique. Specifically, using priority in the model to enforce a strict ordering on simultaneous events can hide race conditions that may exist in the system being modeled. In our model, where the internal actions of the crossbar switches are truly independent, this problem does not arise.

Testing Process Algebraic Models. Our testing technique uses parallel composition to apply tests written in ACSR to components of the Sunshine model. A test is the process $R = (P \parallel T)\backslash E$, where P is the process being tested, T is the sequence of events and actions making up the test, E is the set of event labels on which P and T interact, and R is the result of applying T to P subject to the restriction of E. Tests are defined in such a way that the success or failure of P tested by T can be determined by exploring the state space of R. If R is deadlock free, P passes T. If R has one or more deadlocked states, P fails T.

For example, the specification of the crossbar switching element requires that inputs be ordered on their key and output after one time unit. One possible test of a crossbar switching element is as follows:

$$T \quad = \quad (\overline{in}_{0,4,0}, 1).(\overline{in}_{1,3,1}, 1).\emptyset : (out_{0,3,1}, 1).(out_{1,4,0}, 1).rec\ X.\emptyset : X.$$

If P is a correct implementation of the crossbar specification for this input vector then the state space of $R = (P \parallel T)\backslash\{in_{*,*,*}, out_{*,*,*}\}$ will be deadlock free. If P accepts the input vector but produces incorrect output, then R will deadlock when T is unable to progress due to a lack of synchronization on $out_{0,3,1}$ or $out_{1,4,0}$.

Testing individual process behaviors (rather than constructing a single monolithic state machine representing all possible behaviors) also has practical advantages. When testing individual behaviors, the verification process can be broken into manageable pieces representing small numbers of tests tailored to fit within limited computing resources. Further, being able to easily divide the verification process into pieces simplifies the tasks of checkpointing long computations to guard against loss of results in a system failure, and distributing large computations across several hosts.

Efficient Translation for Testing. Our ability to construct the labeled transition system corresponding to the result R of a test depends on efficient translation of ACSR process terms into their state space representation. Two approaches to this problem are common: (1) a bottom-up technique that computes the LTS for each of the sub-terms of an expression, and then combines them to create the LTS for the expression; and (2) a top-down technique that uses the algebra's semantic rules to interpret the process expression one event or action step at a time. Because top-down translation is based on interpreting the process to derive its behaviors, it follows that only the reachable portion of the process state space will be constructed.

Bottom-up techniques are not applicable to our process testing technique because the computation of the LTS for $(P \parallel T)\backslash E$ will begin by computing the LTS for P. We have already shown that if P is the complete Sunshine switch model, this problem is intractable. VERSA's top-down translation makes testing possible because a single test process will exercises a small subset of the state space of the process being tested.

5.2 Empirical Results for the Sunshine Switch Model

The Sunshine switch model has been verified or tested at three levels: (1) VERSA has been used to verify that two of the smaller components conform to correctness specifications formulated as simple, sequential ACSR-VP processes; (2) exhaustive tests have been applied to the remaining components; and (3) a set of functional tests have been applied to the top-level process that models the Sunshine switch. The verification of the crossbar model using VERSA was demonstrated in Section 2.2. In this section we present the results of analysis of the remaining components and the Sunshine switch itself.

Batcher Sorting Network. Given our data domain of six output port addresses, two payload values, and the possibility of missing inputs, the 8×8 Batcher sorting network is required to sort $(6 \times (2+1))^8 = 1,679,616$ different input vectors. (An input vector consists of a list of packets.) Of course there are an infinite number of sequences of these vectors. But if we assume that the Banyan network implementation treats packets identically no matter where in a timed sequence of packets they are received then we can exhaustively test the Banyan network by executing 1,679,616 tests.

Executing nearly 1.7 million tests is tractable, but unnecessary if we perform some additional analysis. The *0-1 Sorting Lemma*[Knu73] states that if an *oblivious* comparison-exchange algorithm sorts all input sets consisting solely of 0's and 1's, then it sorts all input sets with arbitrary values. A comparison-exchange algorithm is *oblivious* if the key comparisons that are made never depend on the results of previous comparison-exchange operations. That this property holds for the Batcher sorting network follows from the static interconnection topology of the Batcher network. Further, the correct routing of payload values with keys can be verified by inspection of the ACSR-VP process descriptions, since keys and payloads are always exchanged as atomic pairs. Therefore exhaustive testing of the 8×8 Batcher sorting network requires only $3^8 = 6561$ different input vectors of 0's and 1's with arbitrary payloads.

Table 2: Computing Resources Used by Trap Model Tests

Host	Architecture	Memory (MB)	Tests	CPU Time (secs)
A	70 MHz SPARCstation 5	56	124,754	335,852.67
B	60 MHz SPARCserver 1000	256	114,643	265,336.66
C	60 MHz SPARCserver 690MP	320	57,753	114,706.08
D	75 MHz SPARCstation 20	224	78,707	109,429.31
E	60 MHz SPARCstation 10	224	96,533	187,394.19
Total			472,390	

Executing the 6561 tests of the Batcher implementation shown in Appendix E as a background task on a moderately loaded 70 MHz Sun SPARCstation 5 with 56 MB of memory required a total of 3135.90 seconds of CPU time (an average of 0.48 seconds per test). The tests confirmed the correctness of the Batcher network model subject to the assumptions that (1) input vectors are treated identically no matter where in a sequence they arrive; and (2) there are no faults in the ACSR-VP specification that would cause the Batcher model to implement a non-oblivious sort.

Trap. Trap input vectors consist of a sorted list of key/payload pairs. Since payload values are a factor in determining the result computed by the trap, all 471,041 unique input vectors are significant. (The number of possible input vectors was determined by combinatorial analysis and verified by enumerating all sorted key/payload vectors of zero to eight inputs with arbitrary payloads.)

Executing the 471,041 tests of the trap implementation shown in Appendix F required more computing resources than a single host was able to provide. The computation of the test outcomes was distributed across five different hosts as shown in Table 2. (Note that the total number of tests executed was slightly larger than 471,041 because 1349 test executions were duplicated.) The tests confirmed the correctness of the trap model subject to the assumption that input vectors are treated identically no matter where in a sequence they arrive.

Selector. Selector inputs consist of a vector made up of two sorted lists of packets. The first list represents packets destined for the delay circuit, and the second list represents at most one packet destined for each output port. Further, the destination port addresses of delay-bound packets will be a subset of the port addresses of output bound packets, and the payload value of the outbound packet for each output port must be greater than or equal to all delay-bound packet payloads for that port. Taking into account all these restrictions we can identify 203,991 unique, valid input packet vectors by enumerating and examining all possible input vectors. The input space can be further reduced by noting that payload values are not used in determining how inputs map to outputs, so only the unique vectors of port addresses are significant. This reduces the input space to a much more manageable 3003 unique, valid input packet vectors.

Executing the 3003 tests of the selector implementation shown in Appendix G as a background task on a moderately loaded 70 MHz Sun SPARCstation 5 with 56 MB of memory required a total of 1689.84 seconds of CPU time (an average of 0.56 seconds per test). The tests confirmed the correctness of the selector model subject to the assumption that input vectors are treated identically no matter where in a sequence they arrive. Note that the computation of the final output priority values for the delay-bound packets (which depend on the payload values of the input packets) was not checked. Since there is no way to directly confirm externally visible priority values in our testing scheme, these values will have to be verified at the system-level.

Banyan Network. Given our data domain of six output port addresses and two payload values the Banyan network is required to route $3^6 = 729$ different input vectors. (An input vector consists of a sorted and compressed list of packets destined for a subset of the six output ports. Since there can be a packet with payload one or two, or no packet, destined for each of the six ports, there are $3 \times 3 \times 3 \times 3 \times 3 \times 3$ possible input vectors.) Of course there are an infinite number of sequences of these vectors. But if we assume that the Banyan network implementation treats packets identically no matter where in a timed sequence of packets they are received then we can exhaustively test the Banyan network by executing 729 tests. Executing the 729 tests of the Banyan implementation shown in Appendix H as a background task on a moderately loaded 70 MHz Sun SPARCstation 5 with 56 MB of memory required a total of 4607.11 seconds of CPU time (an average of 6.32 seconds per test). The tests confirmed the correctness of the Banyan network model subject to the assumption that input vectors are treated identically no matter where in a sequence they arrive.

Delay. For a fixed number of paths and fixed data domains the delay component has a relatively small state space that can be verified formally by constructing a sequential correctness specification and demonstrating a bisimulation between the concurrent and sequential models. Section K describes `DelaySeq`, an ACSR-VP implementation of the delay component that uses only sequential operators. The sequential delay component model has the advantage that its structure is simple enough that its correctness is apparent from the syntax. Its disadvantage is that it does not scale as easily as the `Delay` model of Section I. However, for fixed parameter values this is not an issue. VERSA was used to demonstrate the existence of a strong bisimulation between `Delay` and `DelaySeq`, thus proving the correctness of `Delay` for two delay paths, keys in the range $[0, 6)$ and payloads in the range $[0, 2)$.

Sunshine Switch. The complete Sunshine switch has a large number of possible input vectors: each input will accept no packet, or a packet addressed to any of six outputs, and packets contain one of two possible payload values, for a total of $(6 \times 2 + 1)^6$ =4,826,809 different input vectors. Further, the recirculation path within the Sunshine switch insures that the relative ordering of packet vectors within a sequence of inputs is significant. Consequently exhaustive testing of the Sunshine switch model is intractable. Instead we executed a small test suite made up of input sequences of one to four input vectors that exercised the major functional characteristics of the switch. A total of 14 tests were selected that exercised routing, recirculation, and dropping of packets. Executing the 14 tests of the Sunshine implementation shown in Appendix J as a foreground task on a lightly loaded 70 MHz Sun SPARCstation 5 with 56 MB of memory required a total of 11,549.27 seconds of CPU time (an average of 769.95 seconds per test). The tests all completed successfully, providing a level of assurance of the correctness of the Sunshine implementation.

6 Summary

We have presented a process algebraic model of a realistic large-scale system and demonstrated techniques for checking its correctness. Two low-level subcomponent of the model were completely verified using traditional formal methods techniques. Testing techniques that approximate verification for larger components were applied to the remaining components and the complete switch model.

Verification of the larger components of the model using traditional formal methods techniques would have been intractable due to state space explosion. To avoid this problem our approach exploited the following facts: (1) a judicious assignment of priorities to concurrent but unrelated events to introduce a strict ordering on their execution mitigates the state space explosion problem by eliminating large numbers of alternative orderings of interleaved concurrent events; (2) the state count of an LTS corresponding to a test of a system is vastly smaller than the LTS for all possible executions of the system; and (3) a top-down translation of a process algebra expression into an LTS allows the LTS corresponding to a test to be constructed in time proportional to the complexity of the test and the portion of the system it interacts with, independent of the complexity of the system under test taken as a whole.

The test and verification technique for large models presented in this paper demonstrates the feasibility of applying process algebras in real-world contexts. In addition to traditional specification languages, system designers can also consider process algebras as a vehicle for early prototyping. Applying process algebras in this context includes all of the advantages of traditional specification languages, along with the added benefit of a rigorous formal semantics which gives precise meaning to models, and verification and analysis techniques not possible in specification languages with less precise semantics.

A number of potential improvements to the test and verification framework for ACSR presented in this paper are the focus of ongoing research. For example, our initial goal was to perform rigorous random testing[DN84] of the Sunshine model by sampling a significant portion of the input space. The speed of translation of large process algebra terms to their LTS representation made this impossible. We expect optimization of the ACSR to LTS translation for large process terms to significantly reduce the time required to evaluate each test, making randomized testing possible. Another change that would net a significant increase in translation speed would be to represent value passing explicitly in the algebra and the LTS, instead of using indexed event labels as the ACSR-VP to ACSR translation required. The VERSA system is currently being modified to operate directly on ACSR-VP processes. Finally, it would be helpful to augment VERSA with built-in test generation and evaluation features to automate the testing process. The tests demonstrated in this paper were generated by C programs and applied and evaluated manually, but most of these tasks could be automated and incorporated into the ACSR framework.

References

[BCD+90] J. Burch, E. Clarke, D. Dill, L. Hwang, and K. McMillan. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the 5th IEEE Symposium on Logic in Computer Science*, 1990.

[BG94] P. Brémond-Grégoire. *A Process Algebra of Communicating Shared Resources With Dense Time and Priorities.* PhD thesis, Department of Computer and Information Science, University of Pennsylvania, May 1994.

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems*, 8(2), April 1986.

[CLX94] D. Clarke, I. Lee, and H. Xie. VERSA: A tool for the specification and analysis of resource-bound real-time systems. *Journal of Computer and Software Engineering*, 1994. to appear.

[CLX95] J-Y. Choi, I. Lee, and H-L Xie. The Specification and Schedulability Analysis of Real-Time Systems using ACSR. In *Proc. of IEEE Real-Time Systems Symposium*, December 1995.

[DN84] J. Duran and S. Ntafos. An evaluation of random testing. *IEEE Transactions on Software Engineering*, SE-10:438–444, July 1984.

[Ger91] R. Gerber. *Communicating Shared Resources: A Model for Distributed Real-Time Systems.* PhD thesis, Department of Computer and Information Science, University of Pennsylvania, 1991.

[GHM+91] J. Giacopelli, J. Hickey, W. Marcus, W. Sincoskie, and M. Littlewood. Sunshine: A high-performance self-routing broadband packet switch architecture. *IEEE Journal on Selected Areas in Communications*, 9(8), October 1991.

[Knu73] D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming, 2nd edition.* Addison-Wesley, 1973.

[KS90] P. C. Kanellakis and S. A. Smolka. CCS Expressions, Finite State Processes, and Three Problems of Equivalence. *Information and Computation*, 86:43–68, 1990.

[LBGG94] I. Lee, P. Brémond-Grégoire, and R. Gerber. A Process Algebraic Approach to the Specification and Analysis of Resource-Bound Real-Time Systems. *Proceedings of the IEEE*, 82(1):158–171, January 1994.

[Mil80] R. Milner. *A Calculus for Communicating Systems.* LNCS 92, Springer-Verlag, 1980.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

[Par81] D. Park. Concurrency and Automata on Infinite Sequences. In *Proc. of 5th GI Conference.* LNCS 104, Springer Verlag, 1981.

[Zeg93] E. Zegura. Architectures for atm switching systems. *IEEE Communications Magazine*, February 1993.

A 2x2 Crossbar Switching Element (ACSR)

```
// 2x2 Crossbar switch.  Sorts inputs in range [0...KeyRange-1] into
// ascending order.  Requires two time steps--one for input; one to
// sort and output.  Sort is stable--if the same key is received on
// both inputs, the payload orders will not be reversed.
//
// Input on XBIn[0 or 1][key][payload] at times: 0,2,4,6,...
// Output on XBOut[0 or 1][key][payload] at times: 1,3,5,7,...
//
// If exactly one input is received a dummy input with key=KeyRange
// (a key larger than any valid input) is created so that the real input
// is always switched to the low-order output port.  The dummy value is
// never output.  If no inputs are received two {}: actions are executed.
//
// Normally inputs are sorted so that the lower valued key appears on output
// port XBOut[0] and the larger on XBOut[1].  #define'ing the macro
// XbarReversal will reverse this order.
//
// Priority values allow interleaved internal actions to be strictly
// ordered when multiple concurrent crossbars are included in a single
// agent.  This dramatically reduces state count of the resulting LTS.
//
// Range of priority indices [PtyMin,...,PtyMax] by PtyStep
// KeyRange -- Range of key values [0,...,KeyRange-1]
// PayRange -- Range of payload values [0,...,PayRange-1]
// XbarReversal -- Reverses order of outputs

XBIn[pty][port] =
        Choice[(XIn[port][key][pay],pty).{(Path[port],3),(Idler[1-port],3)}:
                                  XBSort[pty][port][key][pay] ,
               {key,0,KeyRange-1},{pay,0,PayRange-1}] +
        {(Path[port],1),(Idler[1-port],1)}:XBSort[pty][port][KeyRange][0] +
        {(Path[port],2),(Idler[port],2)}:{}:XBIn[pty][port]
        {pty,PtyMin,PtyMax,PtyStep},
        {port,0,1};

XBSort[pty][0][key][pay] = ('swap[key][pay],pty).
        (Choice[(swap[key2][pay2],pty).XBOut[pty][0][key][pay] ,
                {key2,key,KeyRange},{pay2,0,PayRange-1}] +
         Choice[(swap[key2][pay2],pty).XBOut[pty][0][key2][pay2] ,
                {key2,0,key-1},{pay2,0,PayRange-1}])
        {pty,PtyMin,PtyMax,PtyStep},
        {key,0,KeyRange},{pay,0,PayRange-1};
XBSort[pty][1][key][pay] =
        Choice[(swap[key2][pay2],pty).('swap[key][pay],pty).XBOut[pty][1][key][pay] ,
               {key2,0,key},{pay2,0,PayRange-1}] +
        Choice[(swap[key2][pay2],pty).('swap[key][pay],pty).XBOut[pty][1][key2][pay2] ,
               {key2,key+1,KeyRange},{pay2,0,PayRange-1}]
        {pty,PtyMin,PtyMax,PtyStep},
        {key,0,KeyRange},{pay,0,PayRange-1};
```

```
XBOut[pty][port][key][pay] =
#ifdef XbarReversal
                        ('XOut[1-port][key][pay],pty).{}:XBIn[pty][port]
#else
                        ('XOut[port][key][pay],pty).{}:XBIn[pty][port]
#endif
        {pty,PtyMin,PtyMax,PtyStep},
        {port,0,1},{key,0,KeyRange-1},{pay,0,PayRange-1};
XBOut[pty][port][KeyRange][pay] = {}:XBIn[pty][port]
        {pty,PtyMin,PtyMax,PtyStep},
        {port,0,1},{pay,0,PayRange-1};

XBar[pty] =
            (XBIn[pty][0] || XBIn[pty][1]) \\ {*}
            \ Set[swap[key][pay] , {key,0,KeyRange},{pay,0,PayRange-1}]
        {pty,PtyMin,PtyMax,PtyStep};
```

B 2x2 Crossbar Switching Element (ACSR-VP)

```
// 2x2 Crossbar switch.  Sorts inputs in range [0...KeyRange-1] into
// ascending order.  Accepts inputs every other cycle.  Latency is
// one cycle.  Sort is stable--if the same key is received on
// both inputs, the payload order will not be reversed.
//
// Input on XIn[0 or 1] <key,payload> at times: 0,2,4,6,...
// Output on XOut[0 or 1] <key,payload> at times: 1,3,5,7,...
//
// If exactly one input is received a dummy input with key=KeyRange
// (a key larger than any valid input) is created so that the real input
// is always switched to the low-order output port.  The dummy value is
// never output.  If no inputs are received two idle actions are executed.
//
// Priority values allow interleaved internal actions to be strictly
// ordered when multiple concurrent crossbars are included in a single
// agent.  This dramatically reduces state count of the resulting LTS.
//
// Process:
//   XBar(pty) -- Smaller value output on port 0; pty is priority of events
//   XBarRev(pty) -- Smaller value output on port 1; pty is priority of events
// Constants:
//   KeyRange -- Range of key values is [0,...,KeyRange-1]
//   XbarReversal -- Reverses order of outputs

// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
// 1. Input step
// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

XBIn[port](pty) =
    (XIn[port],pty)?<key,pay>.
      {(Path[port],3),(Idler[1-port],3)}:XBSort[port](pty,key,pay)
  + {(Path[port],1),(Idler[1-port],1)}:XBSort[port](pty,KeyRange,0)
  + {(Path[port],2),(Idler[port],2)}:{}:XBIn[port](pty)
{port,0,1};

// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
// 2. Sorting step
// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

XBSort[0](pty,key,pay) =
  (swap,pty)!<key,pay>.(swap,pty)?<key2,pay2>.
    ((key <= key2) -> XBOut[0](pty,key,pay) else XBOut[0](pty,key2,pay2));
XBSort[1](pty,key,pay) =
  (swap,pty)?<key2,pay2>.(swap,pty)!<key,pay>.
    ((key >  key2) -> XBOut[1](pty,key,pay) else XBOut[1](pty,key2,pay2));


// Continued...
```

```
// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
// 3. Output Step
// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

XBOut[port](pty,key,pay) =
    (key <  KeyRange)  ->  (XOut[port],pty)!<key,pay>.{}:XBIn[port](pty)
  +                    else {}:XBIn[port](pty)
{port,0,1};

// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
// 4. XBar is constructed from parallel composition of two paths
// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

XBar(pty) = (XBIn[0](pty) || XBIn[1](pty)) \\ {*} \ {swap};

// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
// 5. XBarRev is constructed by reversing outputs of XBar
// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

XBarRev(pty) = XBar(pty) % [{XOut[1]/XOut[0],XOut[0]/XOut[1]},{}];
```

C 4x4 Merge Network

```
// Four-input merge network built out of 2x2 crossbars.  Merges inputs
// in range [0...KeyRange-1].  Accepts input every other cycle.  Latency
// is two cycles.
//
// Input on M4In[0...3] <key,payload> at times: 0,2,4,6,...
// Output on M4Out[0...3] <key,payload> at times: 2,4,6,8,...
//
// Process:
//   M4(pty) -- [pty,pty+3] is priority of events for crossbars
//   M4Rev(pty) -- M4 made of reversed crossbars
// Constants:
//   None

// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
// 1. Crossbar interconnections
// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

M4xbar[0][j](pty) =
    XBar(pty+j)
  % [Set[M4In[i+2*j]/XIn[i],Internal[i+2*j]/XOut[i] {i,0,1}],{}]
{j,0,1};
M4xbar[1][j](pty) =
    XBar(pty+j+2)
  % [Set[Internal[2*i+j]/XIn[i],M4Out[i+2*j]/XOut[i] {i,0,1}],{}]
{j,0,1};

M4Revxbar[0][j](pty) =
    XBarRev(pty+j)
  % [Set[M4In[i+2*j]/XIn[i],Internal[i+2*j]/XOut[i] {i,0,1}],{}]
{j,0,1};
M4Revxbar[1][j](pty) =
    XBarRev(pty+j+2)
  % [Set[Internal[2*i+j]/XIn[i],M4Out[i+2*j]/XOut[i] {i,0,1}],{}]
{j,0,1};

// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
// 2. M4 is constructed from parallel composition of interconnected X-bar's
// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

M4(pty) =
    (        M4xbar[0][0](pty) || M4xbar[0][1](pty)
     || ({}:(M4xbar[1][0](pty) || M4xbar[1][1](pty))))
  \ Set[Internal[l] {l,0,3}];
M4Rev(pty) =
    (        M4Revxbar[0][0](pty) || M4Revxbar[0][1](pty)
     || ({}:(M4Revxbar[1][0](pty) || M4Revxbar[1][1](pty))))
  \ Set[Internal[l] {l,0,3}];
```

D 8x8 Merge Network

```
// Eight-input merge network built out of 2x2 crossbars.  Merges inputs
// in range [0...KeyRange-1].  Accepts input every other cycle.  Latency
// is three cycles.
//
// Input on M8In[0...7] <key,payload> at times: 0,2,4,6,...
// Output on M8Out[0...7] <key,payload> at times: 3,5,7,9,...
//
// Process:
//   M8(pty) -- [pty,pty+11] is priority of events for crossbars
// Constants:
//   None

// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
// 1. Crossbar interconnections
// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

M8[0][j](pty) =
    XBar(pty+j)
  % [Set[M8In[i+j*2]/XIn[i],Internal[i+j*2]/XOut[i] {i,0,1}],{}]
{j,0,3};
M8[1][j](pty) =
    XBar(pty+j+4)
  % [Set[Internal[2*j-3*(j/2)+4*i]/XIn[i],Internal[8+i+2*j]/XOut[i] {i,0,1}],{}]
{j,0,3};
M8[2][j](pty) =
    XBar(pty+j+8]
  % [Set[Internal[8+2*i+j%2+4*(j/2)]/XIn[i],M8Out[i+2*j]/XOut[i] {i,0,1}],{}]
{j,0,3};

// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
// 2. M8 is constructed from parallel composition of interconnected X-bar's
// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

M8(pty) =
  (   ((      Parallel[M8[0][j] {j,0,3}]
        || {}:Parallel[M8[1][j] {j,0,3}]) \ Set[Internal[l] {l,0, 7}])
   ||   {}:{}:Parallel[M8[2][j] {j,0,3}]) \ Set[Internal[l] {l,8,15}];
```

E 8x8 Batcher Sorting Network

```
// Sorts eight items using Batcher's sorting network using
// 2x2 Crossbar switches.  Sorts inputs in range [0...KeyRange-1] into
// ascending order.  Accepts input every other cycle.  Latency is
// six cycles.  Sort is stable--if the same key is received on
// two inputs, the payload order will not be reversed.
//
// Input on BatchIn[0...7] <key,payload> at times: 0,2,4,6,...
// Output on BatchOut[0...7] <key,payload> at times: 6,8,10,12,...
//
// Process:
//   B8(pty) -- [pty,pty+23] is priority of events for crossbars
// Constants:
//   None

// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
// 1. Load 4x4 and 8x8 merge network models
// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

#include <atm/M4.acsr>
#include <atm/M8.acsr>

// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
// 1. Four 2x2 sorting networks; XBar(1) and XBar(3) must have their output
//    sequencing reversed.
// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

B8S2[j](pty) =
    XBar(pty+j)
  % [Set[BatchIn[i+j*2]/XIn[i],Internal[i+j*2]/XOut[i] {i,0,1}],{}]
{j,0,3,2};
B8S2[j](pty) =
    XBarRev(pty+j)
  % [Set[BatchIn[i+j*2]/XIn[i],Internal[i+j*2]/XOut[i] {i,0,1}],{}]
{j,1,3,2};
```

// Continued...

```
// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
// 2. Two 4x4 merge networks; second must have all of its crossbars
//    sort sequencing reversed.
// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

B8M4[0](pty) =
    M4(pty)
  % [Set[Internal[(i-(i/2))+(i%2)]/M4In[i],
         Internal[i+8]/M4Out[i] {i,0,3}],{}];
B8M4[1](pty) =
    M4Rev(pty+4)
  % [Set[Internal[(i-(i/2))+(i%2)+4]/M4In[i],
         Internal[i+4+8]/M4Out[i] {i,0,3}],{}];

// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
// 3. One 8x8 merge network.
// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

B8M8(pty) =
    M8(pty)
  % [Set[Internal[4*(i%2)+(i%4)/2+2*(i/4)+8]/M8In[i],
         BatchOut[i]/M8Out[i] {i,0,7}],{}];

// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
// 4. Interconnect the first two crossbars with the first M4.
// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

B8UPaths(pty) =
    (B8S2[0](pty) || B8S2[1](pty+1) || {}:B8M4[0](pty+2))
  \ Set[Internal[i] {i,0,3}];

// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
// 5. Interconnect the third and fourth crossbars with the second M4.
// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

B8LPaths(pty) =
    (B8S2[2](pty) || B8S2[3](pty+1) || {}:B8M4[1](pty+2))
  \ Set[Internal[i] {i,4,7}];

// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
// 6. Connect the 4x4 paths to the 8x8 merge network
// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

B8(pty) =
  (            (B8UPaths(pty+0) || B8LPaths(pty+6))
   || {}:{}:{}:B8M8(pty+12))                        \ Set[Internal[i] {i,8,15}];
```

F Trap

```
// Trap.  Selects one of up to PathCount identical key values to pass through
// with its key value increased by KeyRange.  The remainder are passed through
// with their key values unchanged.  Accepts input every other cycle.  Latency
// is two cycles.
//
// Input on TrapIn[port] <key,payload> at times: 0,2,4,6,...
// Output on TrapOut[port] <key,payload> at times: 2,4,6,8,...
//
// Process:
//   Trap
// Constants:
//   Range of port indices [0...PathCount-1]
//   KeyRange -- Range of key values [0...KeyRange-1]

// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
// 1. Paths.  All are defined identically, with the exception of the port
//    number.
// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

TrapPath[p] =
    {}:{}:TrapPath[p]
  + (TrapIn[p],1)?<key,pay>.{}:
      (  (Choice[(key == keyVal) ->
                   ((arbitrate[keyVal],(PathCount-p)+(pay*PathCount))!
                     {}:(TrapOut[p],1)!<key+KeyRange,pay>.TrapPath[p]
                 {keyVal,0,KeyRange-1}]
       + {}:(TrapOut[p],1)!<key,pay>.TrapPath[p]))
{p,0,PathCount-1};

// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
// 2. Conflict arbitrator; Allows the paths to use priorities to figure out
//    which of the keys to pass and which to trap.
// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Arb[key] = (arbitrate[key],1)?{}:Arb[key] + {}:Arb[key] {key,0,KeyRange-1};
Arbitrator = Parallel[Arb[key] {key,0,KeyRange-1}];

// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
// 3. Trap system
// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Trap =
    (Parallel[TrapPath[p] {p,0,PathCount-1}] || Arbitrator)
  \ Set[arbitrate[key] {key,0,KeyRange-1}];
```

G Selector

```
// Selector.  Routes packets either to the output banyan networks or
// to the delay area, depending on whether the key value has been
// increased by KeyRange to indicate a packet selected by the trap,
// or left alone, indicating that it should be routed to the delay area.
// Accepts input every other cycle.  Latency is two cycles.
//
// Input on SelIn[port] <key,payload> at times 0,2,4,6,...
// Output on SelDel[port] <key,payload> and SelBan[port] <key,payload> at
// times 2,4,6,8,...
//
// Process:
//   Selector(pty): {pty,pty+1,pty+18...pty+18+PathCount} are priority values
//   for communication on SelIn/SelDel
// Constants:
//   Range of port indices [0...PathCount-1]
//   Range of delay path inputs [0...DelayPathCount-1]
//   KeyRange -- Range of key values [0...KeyRange-1]

// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
// 1. Paths.  All are defined identically, with the exception of the port
//    number.  Keys in the range [0...KeyRange-1] are routed to the delay.
//    Keys in the range [KeyRange...2*KeyRange-1] are routed for output.
//    Keys routed to the delay may be dropped, since it is assumed there are
//    fewer delay paths than through paths.
// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

SelPath[p](pty) =
    (SelIn[p],pty+24)?<key,pay>.
        (key < KeyRange) -> {}:
          (  Choice[(SelDel[dp],PathCount-p+PathCount*pay)!<key,pay>.
                      {}:SelPath[p](pty)
                    {dp,0,DelayPathCount-1}] /* packet will be delayed */
           + {}:SelPath[p](pty)) /* packet dropped */
      + (key >= KeyRange) -> (pathid,p)!<p>.{}:
          ((pathid,pty+18+p)?<z>.
             Choice[(z == zVal) -> (SelBan[p-zVal],pty+25)!<key-KeyRange,pay>.
                                     {}:SelPath[p](pty)
                    {zVal,0,PathCount-1}] /* packet will be output */
  + {}:{}:SelPath[p](pty)
{p,0,PathCount-1};




// Continued...
```

```
// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
// 2. Identify the lowest numbered input path destined for output so that
//    inputs can be shifted in a contiguous block down to port 0.  Otherwise
//    Banyan inputs may start at path > 0, and generate incorrect output.
// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

MinBanIn(minval) =
    (pathid,1)?<newval> (newval < minval)  ->  MinBanIn(newval)
                                          else MinBanIn(minval)
  + {}:MinBanOut(minval);
MinBanOut(minval) =
  (minval == PathCount)  ->  {}:MinBanIn(PathCount)
                        else ((pathid,1)!<minval>.MinBanOut(minval)
                              + {}:MinBanIn[PathCount]);
MinBan = MinBanIn(PathCount);

// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
// 3. Selector.
// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

SelectorPaths(pty) = Parallel[SelPath[p](pty) {p,0,PathCount-1}];
Selector(pty) = (SelectorPaths(pty) || MinBan) \ {pathid};
```

H Banyan Network

```
// Eight-input Banyan (Butterfly) network built out of 2x2 crossbars.
// Accepts input every other cycle.  Latency is three cycles.
//
// Input on B8In[0...7] <key,payload> at times 0,2,4,6,...
// Output on B8Out[0...7] <key,payload> at times 3,5,7,9,...
//
// Process:
//   Banyan8(pty) -- [pty,pty+11] is priority of events for crossbars
// Constants:
//   PayRange -- Range of payload values [0...PayRange-1]

// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
// 1. Create process to supply low priority dummy values for crossbars that
// receive no input, and absorb them so they are never output.  Attach
// appropriate dummy input generator to each crossbar.
// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

#define DummyMax 6  // largest dummy value that is needed

DummyIO(dval) =
    {}:DummyIO(dval)
  + Choice[(XIn[port],port*12)!<dval,PayRange>.DummyIO[dval] {port,0,1}]
  + Choice[(rawXOut[port],port*12)?<dval,payload>.
             (payload < PayRange)  -> (XOut[port],port*12)!<dval,payload>.
                                          DummyIO[dval]
                                    else DummyIO[dval]
           {port,0,1}];

DXBar[j](pty) =
    (   (XBar(pty+j) % [{rawXOut[0]/XOut[0],rawXOut[1]/XOut[1]},{}]
     || DummyIO(3-(-1)**(j/2)*2*((j+4)/8)-(-1)**j*(j/8)))
  \ Set[rawXOut[port] {port,0,1}]
{j,0,11};
```

```
// Continued...
```

```
// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
// 2. Crossbar interconnections.  Outputs of DXBar's 6, 7, 9, 10, 11 have
// to be tweaked to insure that dummy values don't force real values to the
// high-order output.
// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

RDXBar[ i](pty) = DXBar[ i](pty) {i,0,5};
RDXBar[ 6](pty) =
    (    (DXBar[ 6](pty) % [{Internal/XOut[1]},{}])
     || (rec Stable .   (Internal,1)?<key,pay>.
                          (key == 5)  ->  (XOut[0],pty+6)!<key,pay>.Stable
                                     else (XOut[1],pty+6)!<key,pay>.Stable
                        + {}:Stable))
  \ {Internal};
RDXBar[ 7](pty) =
    (    (DXBar[ 7](pty) % [{Internal/XOut[1]},{}])
     || (rec Stable .   (Internal,1)?<key,pay>.
                          (key == 5)  ->  (XOut[0],pty+7)!<key,pay>.Stable
                                     else (XOut[1],pty+7)!<key,pay>.Stable
                        + {}:Stable))
  \ {Internal};
RDXBar[ 8](pty) = DXBar[ 8](pty);
RDXBar[ 9](pty) =
    (    (DXBar[ 9](pty) % [{Internal/XOut[1]},{}])
     || (rec Stable .   (Internal,1)?<key,pay>.
                          (key == 2)  ->  (XOut[0],pty+9)!<key,pay>.Stable
                                     else (XOut[1],pty+9)!<key,pay>.Stable
                        + {}:Stable))
  \ {Internal};
RDXBar[10](pty) =
    (    (DXBar[10](pty) % [{Internal/XOut[1]},{}])
     || (rec Stable .   (Internal,1)?<key,pay>.
                          (key == 4)  ->  (XOut[0],pty+10)!<key,pay>.Stable
                                     else (XOut[1],pty+10)!<key,pay>.Stable
                        + {}:Stable))
  \ {Internal};
RDXBar[11](pty) =
    (    (DXBar[11](pty) % [{Internal/XOut[1]},{}])
     || (rec Stable .   (Internal,1)?<key,pay>.
                          (key == 6)  ->  (XOut[0],pty+11)!<key,pay>.Stable
                                     else (XOut[1],pty+11)!<key,pay>.Stable
                        + {}:Stable))
  \ {Internal};




// Continued...
```

```
// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
// 3. Stages are constructed from interconnected X-bar's
// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

B8[0][j](pty) =
    RDXBar[j](pty)
  % [Set[B8In[i*4+j]/XIn[i],Internal[i+j*2]/XOut[i] {i,0,1}],{}]
{j,0,3};
B8[1][j](pty) =
    RDXBar[j+4](pty)
  % [Set[Internal[2*j-3*(j/2)+4*i]/XIn[i],
         Internal[8+i+2*j]/XOut[i] {i,0,1}],{}]
{j,0,3};
B8[2][j](pty) =
    RDXBar[j+8](pty)
  % [Set[Internal[8+2*i+j%2+4*(j/2)]/XIn[i],B8Out[i+2*j]/XOut[i] {i,0,1}],{}]
{j,0,3};

// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
// 4. Banyan8 is constructed from parallel composition of stages
// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Banyan8(pty) =
  (   (  (      Parallel[B8[0][j](pty) {j,0,3}]
         || {}:Parallel[B8[1][j](pty) {j,0,3}]) \ Set[Internal[l] {l,0,7}])
   ||     {}:{}:Parallel[B8[2][j](pty) {j,0,3}]) \ Set[Internal[l] {l,8,15}];
```

I Delay Line

```
// Delay.  Latches packets and redelivers them when the input ports
// are ready.  Accepts inputs at any time.  Latency is zero.
//
// Input on DelayIn[port] <key,payload> at any time.
// Output on DelayOut[port] <key,payload> whenever there is someone to
// synchronize with.
//
// Process:
//   Delay
// Constants:
//   DelayPathCount -- Range of port indices [0...DelayPathCount-1]

// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
// 1. Paths.  All are defined identically, with the exception of the port
//    number.
// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

DelayPath[p] =
  (DelayIn[p]?<key,pay>,1).DelayLatch[p](key,pay) + {}:DelayPath[p]
{p,0,DelayPathCount-1};

// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
// 2. Latches. Hold packets until someone is ready to synchronize.
// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

DelayLatch[p](key,pay) =
  {}:DelayLatch[p](key,pay) + (DelayOut[p],1)!<key,pay>.DelayPath[p]
{p,0,DelayPathCount-1};

// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
// 3. Delay module is constructed from parallel composition of paths
// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Delay = Parallel[DelayPath[p] {p,0,DelayPathCount-1}];
```

J Sunshine Packet Switch

```
// Sunshine ATM Switch.  Accepts input every other cycle. Latency is
// 6 (sort) +2 (trap) +6 (sort) +1 (select) +3 (banyan) = 18 cycles.
//
// Input on in[0...6] <key,payload> at 0,2,4,6,...
// Output on out[0...6] <key,payload> at 18,20,22,24,...
//
// Process:
//   Sunshine
// Constants:
//   None

// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
// 1. Initializing Parameters and load crossbar
// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

#define KeyRange 6
#define PayRange 2
#define PathCount 8
#define DelayPathCount 2
#include <atm/xbar.acsr>

// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
// 2. Load first sorter, relabeling high order inputs as ``in'' and connecting
// outputs to trap stage
// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

#include <atm/batcher8.acsr>
SunSorter1 =
      B8(1)
    % [Set[in[p-DelayPathCount]/BatchIn[p] {p,DelayPathCount,PathCount-1}]]
  % [Set[TrapIn[p]/BatchOut[p] {p,0,PathCount-1}]];

// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
// 3. Load trap stage and connect outputs to second sorter
// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

#include <atm/trap.acsr>
SunTrap =
  {}:{}:{}:{}:{}:{}:(  Trap
                     % [Set[BatchIn[p]/TrapOut[p] {p,0,PathCount-1}]]);




// Continued...
```

```
// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
// 4. Load second sorter and connect outputs to selector
// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

#define KeyRange 12
#include <atm/batcher8.acsr>
SunSorter2 =
  {}:{}:{}:{}:{}:{}:{}:{}:(  B8(25)
                           % [Set[SelIn[p]/BatchOut[p] p,0,PathCount-1}]]);
#define KeyRange 6

// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
// 5. Load selector and connect outputs to delay and banyan network
// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

#include <atm/selector.acsr>
SunSelector =
  {}:{}:{}:{}:{}:{}:{}:{}:{}:{}:{}:{}:{}:{}:
        Selector(25) \ Set[SelDel[p] {p,DelayPathCount,PathCount-1}]
      % [Set[DelayIn[p]/SelDel[p] {p,0,DelayPathCount-1}]]
    % [Set[B8In[p]/SelBan[p] {p,0,PathCount-1}]];

// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
// 6. Load banyan network and relabel outputs as ``out''
// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

#include <atm/banyan.acsr>
SunBanyan =
  {}:{}:{}:{}:{}:{}:{}:{}:{}:{}:{}:{}:{}:{}:{}:
    Banyan8(49) % [Set[out[p]/B8Out[p] {p,0,PathCount-1}]];

// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
// 7. Load delay and connect outputs to low order inputs of first sorter
// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

#include <atm/delay.acsr>
SunDelay = Delay % [Set[BatchIn[p]/DelayOut[p] {p,0,DelayPathCount-1}]];

// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
// 8. Sunshine is parallel composition of six components
// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

TS = (SunTrap || SunSorter2) \ Set[BatchIn[p] {p,0,PathCount-1}];
STS = (SunSorter1 || TS) \ Set[TrapIn[p] {p,0,PathCount-1}];
STSS = (STS || SunSelector) \ Set[SelIn[p] {p,0,PathCount-1}];
STSSB = (STSS || SunBanyan) \ Set[B8In[p] {p,0,PathCount-1}];
STSSBD = (STSSB || SunDelay) \ Set[DelayIn[p],BatchIn[p] {p,0,DelayPathCount-1}];

Sunshine = STSSBD;
```

K Two-Port Sequential Delay Line

```
// Two port sequential delay.  Latches packets and redelivers them when
// the input ports are ready.  Accepts inputs at any time.  Latency is zero.
//
// Input on DelayIn[port] <key,payload> at any time.
// Output on DelayOut[port] <key,payload> whenever there is someone to
// synchronize with.
//
// Process:
//   DelaySeq
// Constants:
//   none

// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
// 1. Initial state: neither buffer full.
// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

DelaySeq =
    {}:DelaySeq
  + (DelayIn[0],2)?<key0,pay0>.DelaySeq0(key0,pay0)
  + (DelayIn[1],1)?<key1,pay1>.DelaySeq1(key1,pay1);

// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
// 2. One buffer full.
// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

DelaySeq0(key0,pay0) =
    {}:DelaySeq0(key0,pay0)
  + (DelayOut[0],1)!<key0,pay0>.DelaySeq
  + (DelayIn[1],1)?<key1,pay1>.DelaySeq01(key0,pay0,key1,pay1);
DelaySeq1(key1,pay1) =
    {}:DelaySeq1(key1,pay1)
  + (DelayOut[1],1)!<key1,pay1>.DelaySeq
  + (DelayIn[0],2)?<key0,pay0>.DelaySeq01(key0,pay0,key1,pay1);

// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
// 3. Both buffers full.
// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

DelaySeq01(key0,pay0,key1,pay1) =
    {}:DelaySeq01(key0,pay0,key1,pay1)
  + (DelayOut[0],1)!<key0,pay0>.DelaySeq1(key1,pay1)
  + (DelayOut[1],1)!<key1,pay1>.DelaySeq0(key0,pay0);
```