January 1996

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

Duncan Clarke
University of Pennsylvania

Insup Lee
University of Pennsylvania, lee@cis.upenn.edu

Follow this and additional works at: http://repository.upenn.edu/cis_reports

Recommended Citation


This paper is posted at ScholarlyCommons. http://repository.upenn.edu/cis_reports/182
For more information, please contact libraryrepository@pobox.upenn.edu.
A Hybrid Approach to Formal Verification Applied to an ATM Switching System

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.

Comments

This technical report is available at ScholarlyCommons: http://repository.upenn.edu/cis_reports/182
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

January 29, 1996

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 NSF CCR-9415346, AFOSR F49620-95-1-0508, and ARO DAAH04-95-1-0092.
## Contents

1 Introduction  

2 Algebra of Communicating Shared Resources with Value Passing and its Toolkit  
   2.1 ACSR-VP  
   2.2 VERSA  

3 The Sunshine ATM Switch Architecture  

4 Modeling a Sunshine Switch with ACSR-VP  
   4.1 Sorting Networks  
   4.2 The Trap Stage  
   4.3 The Selector  
   4.4 Banyan Output Network  
   4.5 The Delay Module  
   4.6 Final Assembly  

5 Analysis of the Sunshine Switch Model  
   5.1 Methodology  
   5.2 Empirical Results for the Sunshine Switch Model  

6 Summary  

A 2x2 Crossbar Switching Element (ACSR)  

B 2x2 Crossbar Switching Element (ACSR-VP)  

C 4x4 Merge Network  

D 8x8 Merge Network  

E 8x8 Batcher Sorting Network  

F Trap  

G Selector  

H Banyan Network  

I Delay Line  

J Sunshine Packet Switch  

K Two-Port Sequential Delay Line
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, CLX95a]. 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 \((l, p)!(3, 4, 5)\). If an event receives values a “?” operator and variable vector will be appended to it, as in \((l, p)?(a, b, c)\).
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 $\mathbf{P}(\mathcal{R} \times \mathbb{N})$, 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., $\pi_r(\{(r_1,p_1),(r_2,p_2)\}) = p_1$. 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 $x, ve$ and $be$, 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 $be_i P$ 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, $P \setminus F$, limits the behavior of $P$ by disallowing any externally observable events with labels in $F$. The process hiding operator, $P \setminus I$, 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 $\text{rec } X.P$ 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 $\text{rec } X$ operator. An agent is a syntactically valid process expression with no free process variables.

In [CLX95a] 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) “→” The second step uses a preemption relation defined for events and actions to incorporate notions of preemption and priority, producing the prioritized transition system “→π”.

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

\[
\text{ChoiceL} \quad \frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'} \quad \text{ChoiceR} \quad \frac{Q \xrightarrow{\alpha} Q'}{P + Q \xrightarrow{\alpha} Q'}
\]

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 “→”.

**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 “≺”, called the preemption relation, be a transitive, irreflexive, binary relation on actions. Then for two actions α and β, if α ≺ β, 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 (α ≺ β), if one of the following cases holds:

1. Both α and β are timed actions in Dr, where

   \( (\rho(\beta) \leq \rho(\alpha)) \land (\forall r \in \rho(\alpha).\pi_r(\alpha) \leq \pi_r(\beta)) \land (\exists r \in \rho(\beta).\pi_r(\alpha) < \pi_r(\beta)) \)

2. Both α and β are events in De, where π(α) < π(β) ∧ l(α) = l(β)

3. α ∈ Dr and β ∈ De, with l(β) = τ and π(β) > 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 πr(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 “≺.” 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, “≺.”

- \( \{(r_1, 2), (r_2, 5)\} < \{(r_1, 7), (r_2, 5)\} \)
- \( \{(r_1, 2), (r_2, 5)\} \neq \{(r_1, 7), (r_2, 3)\} \)
- \( \{(r_1, 2), (r_2, 0)\} \not< \{(r_1, 7)\} \)
- \( (\tau, 1) < (\tau, 2) \)
- \( (a, 1) \not< (b, 2) \)
- \( (a, 2) < (a, 5) \)
- \( ((r_1, 2), (r_2, 5)) < (\tau, 2) \)

We define the prioritized transition system “→π,” which refines “→” to account for preemption.
\[ XBSort_{0}(key, pay) = (\text{swap, 1})(\text{key, pay}).(\text{swap, 1})(\text{key2, pay2}). \]
\[ \quad + \quad (\text{key} < \text{key2}) \rightarrow XBOut_{0}(\text{key, pay}) \]
\[ \quad + \quad (\text{key2} < \text{key}) \rightarrow XBOut_{0}(\text{key2, pay2}) \]
\[ XBSort_{1}(key, pay) = (\text{swap, 1})(\text{key2, pay2}).(\text{swap, 1})(\text{key, pay}). \]
\[ \quad + \quad (\text{key} > \text{key2}) \rightarrow XBOut_{1}(\text{key, pay}) \]
\[ \quad + \quad (\text{key2} > \text{key}) \rightarrow XBOut_{1}(\text{key2, pay2}) \]
\[ XBOut_{port}(key, pay) = (\text{key} < \text{KeyRange}) \rightarrow (XOut_{port, 1})(\text{key, pay}, 1).\emptyset : XBin_{port} \]
\[ \quad + \quad (\text{key} = \text{KeyRange}) \rightarrow \emptyset : XBin_{port} \]
\[ XBar = (XBin_{0} || XBin_{1}) \{\text{Path0, Path1, Idler0, Idler1}\}\{\text{swap}\} \]

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

**Definition 2.2** The labeled transition system \(\xrightarrow{\alpha}\) is defined as follows: \(P \xrightarrow{\alpha} P'\) 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 \(\alpha < \beta\).

**Example: A 2 x 2 Crossbar Switching Element.** We demonstrate the main features of the ACSR-VP paradigm by presenting two approaches to the specification of a 2 x 2 crossbar switching element. A 2 x 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 x 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.
When an input value is received on port \( \text{port} \) the \( X BI_{\text{port}} \) process consumes the \( \text{Path}_{\text{port}} \) and \( \text{Idler}_{1-\text{port}} \) resources for one time unit and continues as \( X BS_{\text{sort}_{\text{port}}} \). The \( X BS_{\text{sort}} \) processes share the key values between port processes and proceed to the output processes \( X BO_{\text{out}_{\text{port}}} \). 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 \( X BO_{\text{out}} \) processes output the key and payload values received from the sort step, consume one unit of time idling and restart the \( X BI_{\text{port}} \) processes.

The second and third lines of the the \( X BI_{\text{port}} \) 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 \( \{(\text{Path}_0, 3), (\text{Idler}_1, 3)\} \) action. Port 0's use of the \( \text{Idler}_1 \) resource preempts port 1's attempt to use it (in line three of the definition of \( X BI_{\text{port}} \)) so port 1 must execute the other timed action step \( \{(\text{Path}_1, 1), (\text{Idler}_1, 1)\} \) that passes the out-of-range key value \( \text{KeyRange} \) to \( X BS_{\text{sort}_1} \). This value is later filtered by the \( X BO_{\text{out}_1} \) 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 \( \text{Path}_{\text{port}} \) has a priority of 2, preempting the lower priority possibility. This \( \text{Path} \) consumes two units of time and restarts the \( X BI_{\text{port}} \) processes without performing comparisons.

The \( X Bar \) process hides its use of the \( \text{Path} \) and \( \text{Idler} \) resources to avoid conflict between concurrent \( X Bar \)'s. Restriction of the \text{swap} event labels insures that exchanges within \( X Bar \) use local values.

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

Initially \( X BarSeq \) accepts any input event. If an input event is received on port \( p \) then \( X BarSeq \) proceeds to a state that allows any input key and payload to be accepted on the complementary
\[XBarSeq = (XIn_0, 1)(key_0, pay_0)\]
\[+ (XIn_1, 1)(key_1, pay_1).XBarSeq'(key_0, pay_0, key_1, pay_1)\]
\[+ \emptyset : (\tau, 2).(\tau, 2).(XOut_0, 1)!{(key_0, pay_0)} : XBarSeq\]

\[XBarSeq'(key_0, pay_0, key_1, pay_1) = \emptyset : (\tau, 2).(\tau, 2)\]
\[ (key_0 \leq key_1) \rightarrow (XOut_1, 1)!{(key_1, pay_1)} : XBarSeq\]
\[+ (XOut_1, 1)!{(key_1, pay_1)} : XBarSeq\]
\[+ (key_0 > key_1) \rightarrow (XOut_0, 1)!{(key_0, pay_0)} : XBarSeq\]
\[+ (XOut_1, 1)!{(key_0, pay_0)} : XBarSeq\]

Figure 4: Sequential ACSR-VP model of a \(2 \times 2\) Switching Element

port \((1-p)\). Once both input values have been received computation advances through an idle step and two \(\tau\) 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 “\(\rightarrow\)”, 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 \((P’, Q’) \in r\). 

9
To achieve similar behavior using ACSR we would create the following process:

\[
\text{Therm} = \sum_{i=0}^{67} (\text{temp}_i, 1).\text{Heat} + \sum_{i=68}^{78} (\text{temp}_i, 1).\text{Idle} + \sum_{i=79}^{129} (\text{temp}_i, 1).\text{Cool}
\]

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 [CLX95b].

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).
Table 1: Pure ACSR vs. VERSA Syntax

<table>
<thead>
<tr>
<th>Pure ACSR</th>
<th>VERSA Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a, a', a, a; \tau$</td>
<td>$a, 'a, a', a[i]$</td>
</tr>
<tr>
<td>$NIL$</td>
<td>Any capitalization of NIL</td>
</tr>
<tr>
<td>$</td>
<td></td>
</tr>
<tr>
<td>$P\Delta^a(Q,R,S)$</td>
<td>scope($P,a,t,Q,R,S$)</td>
</tr>
<tr>
<td>$\infty$</td>
<td>inf or infinite or infinity</td>
</tr>
<tr>
<td>$[P]_I$</td>
<td>$[P]_I$</td>
</tr>
<tr>
<td>$P [R_e, R_a]$</td>
<td>$P[R_e,R_a]$</td>
</tr>
<tr>
<td>$R_e$ and $R_a$</td>
<td>${11'/11,12'/12,...,ln'/ln}$</td>
</tr>
<tr>
<td>$\emptyset$</td>
<td>{} or idle</td>
</tr>
<tr>
<td>Definition of index $i$</td>
<td>{i, begin, end, step, predicate}</td>
</tr>
<tr>
<td>$\sum_{i=0}^N P$</td>
<td>Choice($P {1,1,N}$)</td>
</tr>
<tr>
<td>$\prod_{i=0}^N P$</td>
<td>Parallel($P {1,1,N}$)</td>
</tr>
<tr>
<td>${a_i</td>
<td>i = 1,\ldots,N \land pred(i)}$</td>
</tr>
</tbody>
</table>

- **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 [CLX95b].

**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 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 */ * block delimiters or the C++ line delimiter, //.

**Example: 2x2 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  \hfill (1)
ready> #define PtyMax 1  \hfill (2)
ready> #define PtyStep 1  \hfill (3)
ready> #define KeyRange 6  \hfill (4)
ready> #define PayRange 2  \hfill (5)
ready> #include "XBar.acsr"  \hfill (6)
ready> #include "XBarSeq.acsr"  \hfill (7)
ready> XBar[1] == XBarSeq?  \hfill (8)
   \text{true (by prioritized strong equivalence)}
ready> XBar[1]!  \hfill (9)
XBar[1] <1> --(XIn[0,0,0],1)-->  \hfill \text{(10)}
  <2> --(XIn[0,0,1],1)-->  
  \ldots
  <25> --{}--> no-name
[XBar[1]] ready> show stats
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 $2 \times 2$ 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.