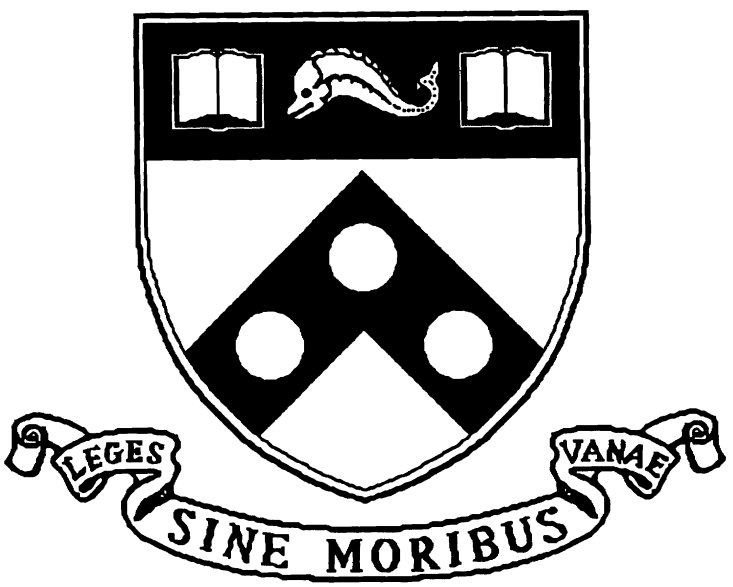
VERSA: A Tool for the Specification and Analysis of Resource-Bound Real-Time Systems

MS-CIS-93-77
GRASP LAB 359
DISTRIBUTED SYSTEMS LAB 34

Duncan Clarke
Insup Lee
Hong-liang Xie



University of Pennsylvania
School of Engineering and Applied Science
Computer and Information Science Department
Philadelphia, PA 19104-6389

September 1993

VERSA: A Tool for the Specification and Analysis of Resource-Bound Real-Time Systems *

Duncan Clarke, Insup Lee and Hong-liang Xie
University of Pennsylvania

Abstract

VERSA is a tool that assists in the algebraic analysis of real-time systems. It is based on ACSR, a timed process algebra designed to express resource-bound real-time distributed systems. VERSA supports the analysis of real-time processes through algebraic rewriting, interactive execution, and equivalence testing. This paper begins by presenting a brief overview of the process algebra ACSR, its syntax, operational semantics, and equivalence relations. VERSA's process and command syntax, its algebraic rewrite system, and its state-based analysis features are described fully. The presentation includes examples that illustrate the salient features of ACSR, and output from sample VERSA sessions that demonstrate the application of the tool to real-time systems analysis.

1 Introduction

Reliability in real-time systems can be improved through the use of formal methods for the specification and analysis of real-time systems. Formal methods treat system components as mathematical objects and provide mathematical models to describe and predict the observable properties and behaviors of these objects. There are several advantages to using formal methods for the specification and analysis of real-time systems. They include (1) the early discovery of ambiguities, inconsistencies and incompleteness in informal requirements; (2) the automatic or machine-assisted analysis of the correctness of specifications with respect to requirements; and (3) the evaluation of design alternatives without expensive prototyping.

Despite all the virtues of using formal methods, their manual application tends to be time consuming and error prone. To alleviate this strain, we have built VERSA (Verification, Execution, and Rewrite System for ACSR), an integrated toolkit whose goal is to simplify the specification and analysis of resource-bound real-time systems.

*This research was supported in part by ONR N00014-89-J-1131, ONR N00014-89-J-1131S1, and DARPA/NSF CCR90-14621. Correspondence and requests for reprints should be sent to Insup Lee, Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA, 19103-6389

VERSA is based on ACSR[1], a process algebra for describing concurrent real-time systems with explicit resource requirements. To specify and analyze real-time systems, many process algebras have been augmented to include the notion of time and a set of timed operators [2, 3, 4, 5, 6, 7, 8]. The domain of time is a partially ordered set that is either discrete or dense. A time domain is discrete if there is a least element in the domain and each element in the domain has a finite number of predecessors and a unique immediate successor. A time domain is dense if there is an element between any two elements in the domain; that is, an event can happen at an arbitrary moment in time. Dense time is a more general model of real-world phenomena, but the discrete time domain simplifies modeling and analysis in many cases, and is an accurate model of a common problem domain, that of digital systems operating from a single global clock. Although a version of ACSR with dense time is being developed, the presentation of ACSR in this paper is restricted to the discrete time domain.

The timing behavior of a real-time system depends not only on delays due to process synchronization, but also on the availability of shared resources. Most current real-time process algebras, including those cited above, adequately capture delays due to process synchronization; however, they abstract out resource-specific details. In contrast, the computation model of ACSR is based on the view that a real-time system consists of a set of communicating processes that compete for shared resources. The use of shared resources is modelled by timed actions whose executions are subject to the availability of resources. Contention for resources is arbitrated according to the priorities of competing actions. To ensure the uniform progression of time, processes execute timed actions synchronously. In addition to timed actions, ACSR supports instantaneous actions, called events, that do not consume any resource. Processes execute events asynchronously except when two processes synchronize through matching events. Priorities are also used to arbitrate the choice among multiple events that are possible at the same time.

The novelty of ACSR relative to existing real-time formalisms is its representation of resources and priority. Without an explicit notion of resources, the specification of resource-bound systems requires that some artificial means be used to model resource requirements, such as defining processes to represent resources. Models that lack explicit priorities require that a process be created for the express purpose of arbitrating priorities and implementing preemption. Providing explicit notions of resources and priority within ACSR has simplified the task of modeling common situations that arise in real-time system design, and results in specifications that are closer analogues of the systems they model.

Although mathematical formalisms for expressing processes are important in themselves, their manual application to realistic systems is time consuming and error prone. As a result, many formalisms are supported by automated tools that perform tasks such as syntax checking, state space analysis, and interactive execution. Space does not permit us to survey the vast range of formal approaches and supporting tools here, however we briefly overview a few representative examples of language-based systems for the purpose of comparison. The classic formalisms CSP[9] and CCS[10, 11] are supported by FDR[12] and CWB[13], respectively. These tools offer textual interfaces for process description, comparison of processes,

and model checking. A different emphasis is present in the Process Algebra Manipulator (PAM)[14, 15], which is in some respects more general than FDR or CWB, as it allows the user to define the language that is to be used for analysis, and supports very general algebraic analysis techniques. Tools are fundamental to Modechart[16, 17, 18] and the TTM/RTTL verifier[19], which use a graphical formalism to express the structure of concurrent systems, and provide an interpreter and a logic-based language for analysis.

Based upon the algebra ACSR, VERSA is a tool that supports an algebraic approach to systems analysis and design. Systems are described as algebraic expressions, and they can be analyzed by (1) application of rewriting rules to deduce system properties; (2) construction of a state machine, and subsequent exploration and analysis of the state space of that machine to verify safety properties such as freedom from deadlocks and equivalence of alternative process formulations; or (3) interactive execution of the process specification to explore specific system properties and sample the execution traces of the system.

VERSA differs from all of the tools mentioned previously in its support for a formalism that allows explicit modeling of resources and priorities. However, VERSA does lack some features provided by classic tools like FDR and CWB, such as model checking. PAM is similar to VERSA in that it supports algebraic analysis, but the focus of PAM is the creation and analysis of new algebras, so it is necessarily more complete in its support of algebraic manipulations and theorem proving techniques. However, PAM lacks some of VERSA's more practical features, such as state-based analysis. The graphical interface of Modechart and TTM/RTTL offers distinct advantages over VERSA's textual notation, but neither of these systems offers a formal algebraic framework to facilitate the construction of proofs based on their underlying language.

The remainder of this paper is organized as follows. Section 2 gives a brief introduction to the ACSR paradigm. Section 3 presents a detailed description of the VERSA tool including details of its architecture, implementation, process description conventions, and features that support analysis, illustrated with examples. Section 4 concludes the paper with a summary discussion and presents some ways in which we plan to extend VERSA.

2 Modeling Processes

VERSA's language for describing processes and the underlying semantics is based on ACSR (Algebra of Communicating Shared Resources), a process algebra that incorporates the notions of communication, concurrency, resources, and priorities into a single formalism. This section provides the details of ACSR necessary to understand VERSA's facilities for operating on ACSR process descriptions.

2.1 Actions

When modeling a process with algebraic expressions, the progress of the process through its interactions with external agents is modeled by the execution of discrete "actions." ACSR

uses two distinct action types to model computation: time and resource consuming actions, and instantaneous events.

Timed Actions—We consider a system to be composed of a finite set of serially reusable resources, denoted by $\mathcal{R}$. An action that consumes one "tick" of time is drawn from the domain $\mathbb{P}(\mathcal{R} \times \boldsymbol{N})$ (the power set of $\mathcal{R} \times \boldsymbol{N}$), 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 priority level p. Priority values range over $\boldsymbol{N}$, with 0 being the lowest (least pressing) priority, and priority increasing with increasing p. 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, 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 action A in the resource r, 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 $\pi_r(A) = 0$.

Instantaneous Events—We call instantaneous actions *events*, which provide the basic synchronization in our process algebra. An event is denoted by a pair (a,p), where a is the *label* of the event, and p is its *priority*. Again, priority values range over $\boldsymbol{N}$ with 0 being the least pressing priority. Labels are drawn from the set $\mathcal{L} \cup \overline{\mathcal{L}} \cup \{\tau\}$, where if a is a given label, we say that $\bar{a}$ is its *inverse* label, i.e. $\bar{\bar{a}} = a$. A label and its inverse can be thought of as naming complementary ends of a communication channel. As in CCS[10], the special identity label, τ, arises when two events with inverse labels are executed in parallel.

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.

Finally, the entire domain of actions is $\mathcal{D} = \mathcal{D}_R \cup \mathcal{D}_E$, and we let α and β range over $\mathcal{D}$.

2.2 ACSR Operator Syntax and Semantics

Let P, P_1, P_2, and P_3 range over the domain of terms, and let X range over the domain of term variables. Additionally, we assume an infinite set of free term variables, FV. ACSR's syntax is given by the grammar of Figure 1.

$$\begin{array}{rcl} P & ::= & \text{NIL} \mid A:P \mid e.P \mid P_1 + P_2 \mid P_1 \| P_2 \mid \\ & & P \,\Delta_t^a\, (P_1, P_2, P_3) \mid [P]_I \mid P\backslash F \mid rec\ X.P \mid X \end{array}$$

Figure 1: Syntax of ACSR Process Expressions

NIL is a process that executes no action (i.e., it is initially deadlocked). There are two prefix operators, corresponding to the two types of actions. The first, $A : P$, executes a timed, resource-consuming action A, consumes one time unit, and proceeds to the process P. The second prefix operator, $e.P$, executes the instantaneous event e, and proceeds to P. 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 Scope construct $P \triangle_t^a (P_1, P_2, P_3)$ binds the process P by a temporal scope[20], and incorporates both the features of timeouts and interrupts. We call t the *time bound*, where $t \in \mathbb{N}^+ \cup \{\infty\}$ (i.e., t is either a non-negative integer or infinity). P executes for a maximum of t time units. The scope may be exited in a number of ways. First, if P successfully terminates within time t by executing an event labeled with $\bar{a}$, then control proceeds to the "success-handler" P_1 (here, a may be any label other than τ). Process P_2 is a timeout exception-handler; that is, if P fails to terminate within time t, then control proceeds to P_2. Lastly, at any time while P is executing it may be interrupted by P_3's execution of a timed action or instantaneous event, and the scope is then departed.

The Close operator, $[P]_I$, produces a process P that monopolizes the resources in $I \subseteq \mathcal{R}$. The Restriction operator, $P \backslash F$, limits the behavior of P. Here, no events with labels in F are permitted to execute. The process $rec\ X.P$ denotes standard recursion, allowing the specification of infinite behaviors. The term X, without a "rec" binding, is a free variable that belongs to the infinite set FV.

ACSR's formal semantics is defined in two steps. First, we develop the *unconstrained* transition system, where a transition is denoted as $P \xrightarrow{\alpha} P'$ (for P and P' processes and α an action). Within "$\rightarrow$" no priority arbitration is made between actions. Rather, we subsequently refine "$\rightarrow$" to define our prioritized transition system, "$\rightarrow_\pi$."

The two rules for the prefix operators are *axioms*, i.e. they have premises of *true*. There is one rule for time-consuming actions, and one for events.

$$\textbf{ActT}\ \frac{-}{A : P \xrightarrow{A} P} \qquad \textbf{ActI}\ \frac{-}{e.P \xrightarrow{e} P}$$

For example, the process $\{(r_1, p_1), (r_2, p_2)\} : P$ simultaneously uses resources r_1 and r_2 for one time unit, and then executes P. Alternatively, the process $(a, p).P$ executes the event "(a, p)," and proceeds to P.

The rules for Choice are identical for both timed actions and instantaneous events (and hence we use "α" as the label).

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

As an example, $(a, 7).P + \{(r_1, 3), (r_2, 7)\} : Q$ may choose between executing the event $(a, 7)$ or the time-consuming action $\{(r_1, 3), (r_2, 7)\}$. The former behavior is deduced from rule **ActI**, while the latter is deduced from **ActT**.

The Parallel operator provides the basic constructor for concurrency and communication. The first rule, **ParT**, is for two time-consuming transitions.

$$\textbf{ParT}\ \frac{P \xrightarrow{A_1} P', Q \xrightarrow{A_2} Q'}{P \| Q \xrightarrow{A_1 \cup A_2} P' \| Q'} \quad (\rho(A_1) \cap \rho(A_2) = \emptyset)$$

Note that timed transitions are truly synchronous, in that the resulting process advances only if both of the constituents take a step. Thus, care must be taken to insure that every

step of a timed computation offers one or more timed alternatives, lest the lack of a timed step should "stop the clock." The condition $\rho(A_1) \cap \rho(A_2) = \emptyset$ mandates that each resource is truly sequential, and that only one process may use a given resource during any time step.

The next three laws are for event transitions. As opposed to timed actions, events may occur asynchronously (as in CCS and related interleaving models).

$$\textbf{ParIL} \quad \frac{P \xrightarrow{e} P'}{P \| Q \xrightarrow{e} P' \| Q} \qquad \textbf{ParIR} \quad \frac{Q \xrightarrow{e} Q'}{P \| Q \xrightarrow{e} P \| Q'}$$

$$\textbf{ParCom} \quad \frac{P \xrightarrow{(a,n)} P', Q \xrightarrow{(\bar{a},m)} Q'}{P \| Q \xrightarrow{(\tau,n+m)} P' \| Q'}$$

The first two rules show that events may be arbitrarily interleaved. The last rule is for two synchronizing processes, i.e. P executes an event with the label a, while Q executes an event with the inverse label $\bar{a}$. Note that when two events synchronize, their resulting priority is the sum of their constituent priorities.

The Scope operator possesses a total of five transition rules describing the various behaviors induced by a temporal scope. The first two rules show that as long as $t > 0$ and P does not execute an event labeled with b, the executions of P continue.

$$\textbf{ScopeCT} \quad \frac{P \xrightarrow{A} P'}{P \triangle_t^b (Q, R, S) \xrightarrow{A} P' \triangle_{t-1}^b (Q, R, S)} \quad (t > 0)$$

$$\textbf{ScopeCI} \quad \frac{P \xrightarrow{e} P'}{P \triangle_t^b (Q, R, S) \xrightarrow{e} P' \triangle_t^b (Q, R, S)} \quad (l(e) \neq \bar{b}, t > 0)$$

The **ScopeE** (for "end") shows that P can depart the temporal scope by executing an event labeled with $\bar{b}$. Upon exit, the label $\bar{b}$ is converted to the identity label τ; however, the same priority is retained.

$$\textbf{ScopeE} \quad \frac{P \xrightarrow{(\bar{b},n)} P'}{P \triangle_t^b (Q, R, S) \xrightarrow{(\tau,n)} Q} \quad (t > 0)$$

The next rule, **ScopeT** (for "timeout"), is applied whenever the scope times out, i.e. when $t = 0$. At this point, control proceeds to the exception handler R.

$$\textbf{ScopeT} \quad \frac{R \xrightarrow{\alpha} R'}{P \triangle_t^b (Q, R, S) \xrightarrow{\alpha} R'} \quad (t = 0)$$

Finally, **ScopeI** shows that the process S may interrupt (and kill) P while the scope is still active. Note that the interrupt step may be one of several options offered simultaneously, and whether or not the interrupting action is executed and the scope is terminated will

depend upon the priorities of the competing events and the preemption relation that will be defined shortly. Also note that scopes may be nested arbitrarily deeply. When nested scopes use the same action to trigger an interrupt, the interrupting action will propagate upward after each interruption. This follows from rule ScopeI, which specifies that the interrupting action is preserved.

$$\textbf{ScopeI} \quad \frac{S \xrightarrow{\alpha} S'}{P \triangle_t^b (Q, R, S) \xrightarrow{\alpha} S'} \qquad (t > 0)$$

The Restriction operator defines a subset of instantaneous events that are excluded from the behavior of the system. This is done by establishing a set of labels, F ($\tau \notin F$), and deriving only those behaviors that do not involve events with those labels. Note that while $P\backslash F$ restricts P from communicating with other processes using labels in F, concurrent subprocesses within P are free to interact with one another using these labels. Thus, restriction can be viewed as the assignment of dedicated channels for communication within a process. Note also that time-consuming actions are unaffected by restriction.

$$\textbf{ResT} \quad \frac{P \xrightarrow{A} P'}{P\backslash F \xrightarrow{A} P'\backslash F} \qquad\qquad \textbf{ResI} \quad \frac{P \xrightarrow{(a,n)} P'}{P\backslash F \xrightarrow{(a,n)} P'\backslash F} \qquad (a, \bar{a} \notin F)$$

While Restriction assigns dedicated channels to processes, the Close operator assigns dedicated resources. When a process P is embedded in a closed context such as $[P]_I$, we ensure that there is no further sharing of the resources in I. Assume that P executes a time-consuming action A. If A utilizes less than the full resource set I, the action is augmented with $(r, 0)$ pairs for each unused resource $r \in I - \rho(A)$. The way to interpret Close is as follows. A process may idle in one of two ways: it may either release its resources during the idle time (represented by $\emptyset$), or it may hold them. Close ensures that the resources are held. (Instantaneous events are not affected.)

$$\textbf{CloseT} \quad \frac{P \xrightarrow{A_1} P'}{[P]_I \xrightarrow{A_1 \cup A_2} [P']_I} \qquad (A_2 = \{(r, 0) \mid r \in I - \rho(A_1)\})$$

$$\textbf{CloseI} \quad \frac{P \xrightarrow{e} P'}{[P]_I \xrightarrow{e} [P']_I}$$

The operator $rec\ X.P$ denotes guarded recursion, allowing the specification of infinite behaviors.

$$\textbf{Rec} \quad \frac{P[^{rec\ X.P}/_X] \xrightarrow{\alpha} P'}{rec\ X.P \xrightarrow{\alpha} P'}$$

$P[^{rec\ X.P}/_X]$ is the standard notation for substitution of $rec\ X.P$ for each free occurrence of X in P. By guarded recursion, we mean that all occurrences of X in $rec\ X.P$ are preceded by some action α. For example, $rec\ X.(A : X)$ is guarded, while $rec\ X.(X + A : X)$ is not.

As an example, consider $rec\ X.(A : X)$, which executes the resource-consuming action "A" forever. By **ActT**, $A : (rec\ X.(A : X)) \xrightarrow{A} rec\ X.(A : X)$, so by **Rec**, $rec\ X.(A : X) \xrightarrow{A} rec\ X.(A : X)$.

2.3 Priority and Preemption

The prioritized transition system is based on *preemption*, which incorporates our 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.

Example 2.1 The following examples show some comparisons made by the preemption relation, "$\prec$."

a. $\{(r_1, 2), (r_2, 5)\} \prec \{(r_1, 7), (r_2, 5)\}$
b. $\{(r_1, 2), (r_2, 5)\} \not\prec \{(r_1, 7), (r_2, 3)\}$
c. $\{(r_1, 2), (r_2, 0)\} \prec \{(r_1, 7)\}$
d. $\{(r_1, 2), (r_2, 1)\} \not\prec \{(r_1, 7)\}$
e. $(\tau, 1) \prec (\tau, 2)$
f. $(a, 1) \not\prec (b, 2)$
g. $(a, 2) \prec (a, 5)$
h. $\{(r_1, 2), (r_2, 5)\} \prec (\tau, 2)$ □

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

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

a) $P \xrightarrow{\alpha} P'$ *is an unprioritized transition, and*

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

2.4 Example: Mutual Exclusion Problem

In this section, we demonstrate the main features of the ACSR paradigm by presenting two approaches to the specification of a mutual exclusion problem.

Example 2.2 Consider tasks A and B which run on cpu_1 and cpu_2, respectively, and share a common data structure called *data* with different priorities. The tasks send their requests for *data* via the channels $chan_1$ and $chan_2$, respectively. To ensure mutual exclusion, *data* is protected by critical sections in each of the tasks. The running of the tasks inside their critical sections is captured by a fixed number of actions prc_1 and prc_2, respectively, and outside the critical sections by an arbitrary number of actions run_1 and run_2, respectively. The computation steps in which the tasks commit to moving into their critical section are req_1 and req_2, respectively.

The actions are defined as follows:

$$\begin{array}{llll} prc_1 = \{(data,1),(cpu_1,1)\}, & run_1 = \{(cpu_1,1)\}, & req_1 = \{(cpu_1,1),(chan_1,1)\} \\ prc_2 = \{(data,2),(cpu_2,1)\}, & run_2 = \{(cpu_2,1)\}, & req_2 = \{(cpu_2,1),(chan_2,1)\} \end{array}$$

We give two different solutions to the mutual exclusion problem in ACSR. The first one adopts the traditional approach of using semaphores to protect the critical sections. The resulting specification is the process S:

$$\begin{aligned} S &= ([P_1 \parallel P_2 \parallel PV]_{\{data\}}) \setminus \{p, v\} \\ P_1 &= (rec\ X.\ run_1 : X) \triangle_\infty (\text{NIL}, \text{NIL}, req_1 : P\_cs_1) \\ P\_cs_1 &= run_1 : P\_cs_1 + (p,1)\,.\,prc_1 : prc_1 : prc_1 : (v,1)\,.\,P_1 \\ P_2 &= (rec\ X.\ run_2 : X) \triangle_\infty (\text{NIL}, \text{NIL}, req_2 : P\_cs_2) \\ P\_cs_2 &= run_2 : P\_cs_2 + (p,2)\,.\,prc_2 : prc_2 : prc_2 : prc_2 : (v,1)\,.\,P_2 \\ PV &= \emptyset : PV + (\overline{p},1)\,.\,(rec\ X.\ (\emptyset : X + (\overline{v},1)\,.\,PV)) \end{aligned}$$

Process S specifies that the system is made up of three concurrent tasks P_1, P_2, and PV that monopolize the resource *data*, and do not share their semaphore's p and v operations with any other processes. P_1 and P_2 represent the competing tasks that share the *data* resource. P_1 and P_2 loop until they receive a signal (via resource $chan_1$ or $chan_2$, respectively)

indicating that there are data to process in the buffer. Each then proceeds to P_cs_1 and P_cs_2, respectively, where they idle if the semaphore is unavailable, or obtain the semaphore, proceed through their critical section, relinquish the semaphore, and return to their initial state. Process PV models a binary semaphore, offering communication on $\overline{p}$, and then waiting until a synchronization on $\overline{v}$ occurs before offering $\overline{p}$ again.

The second solution to the same problem exploits the sequential nature of ACSR's resources and leads to a simpler specification of the mutual exclusion problem, represented by the process T:

$$\begin{aligned} T &= [Q_1 \parallel Q_2]_{\{data\}} \\ Q_1 &= (rec\ X.\ run_1 : X) \triangle_\infty (\text{NIL}, \text{NIL}, req_1 : Q\_cs_1) \\ Q\_cs_1 &= run_1 : Q\_cs_1 + prc_1 : prc_1 : prc_1 : Q_1 \\ Q_2 &= (rec\ X.\ run_2 : X) \triangle_\infty (\text{NIL}, \text{NIL}, req_2 : Q\_cs_1) \\ Q\_cs_2 &= run_2 : Q\_cs_2 + prc_2 : prc_2 : prc_2 : prc_2 : Q_2 \end{aligned}$$

The resource *data* in the process T is closed to guarantee that if *data* is available while requested by Q_1 or Q_2, it will be taken immediately.

The different priorities of the two subtasks are enforced by $(p,1) \prec (p,2)$ in process S and by $prc_1 \prec prc_2$ in process T. As we will see later, the correctness of S follows from that of T after certain equivalence relations between the two processes are established. □

2.5 Equivalence of Processes

Our 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 processes is based on the concept of *bisimulation*[21], 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 $(P',Q') \in r$.* □

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 [22], 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's operators[23]. Similarly, "$\sim_\pi$" is the largest strong bisimulation over "$\rightarrow_\pi$," and we call it a *prioritized strong equivalence.*

When comparing specifications and implementations we often find that because different objectives were pursued in formulating the two process expressions (for example, simplicity for the specification, and efficiency for the implementation), the internal synchronization actions of the two processes are not identical. Consequently, even though the two processes may display identical "external" behavior (i.e. non-τ event labels and timed action steps), there may be τ actions in one process that do not correspond directly with τ actions in the other. (Recall that synchronization replaces the complementary event labels with a single τ event.) For those situations where matching of external behaviors is sufficient, a weaker form of equivalence, *weak bisimulation*[10] is used.

Definition 2.4 *If $t \in \mathcal{D}^*$, then $\hat{t} \in (\mathcal{D} - \{\tau\})^*$ is the sequence derived by deleting all occurrences of τ from t.* □

Definition 2.5 *If $t = \alpha_1 \cdots \alpha_n \in \mathcal{D}^*$, then $E \stackrel{t}{\Rightarrow} E'$ if*

$$E(\stackrel{(\tau,p)}{\longrightarrow})^* \stackrel{\alpha_1}{\longrightarrow} (\stackrel{(\tau,p)}{\longrightarrow})^* \cdots (\stackrel{(\tau,p)}{\longrightarrow})^* \stackrel{\alpha_n}{\longrightarrow} (\stackrel{(\tau,p)}{\longrightarrow})^* E',$$

where p represents an arbitrary priority value that can vary from event to event. □

Definition 2.6 *For a given transition system "$\longrightarrow$", any binary relation r is a* weak bisimulation *if, for $(P,Q) \in r$ and $\alpha \in \mathcal{D}$,*

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

In other words, if P (or Q) can take a step on $\alpha \in \mathcal{D}$, then Q (or P) must also be able to take a step (or steps) on $(\tau,p)^*\alpha(\tau,p)^*$. And if P (or Q) can take a step on (τ,p), then Q (or P) may or may not take a step (or steps) on $(\tau,p)^+$. It is possible to prove the existence of a largest weak bisimulation over "$\rightarrow_\pi$" in a manner analogous to the case for prioritized strong equivalence, and we call this relation a *prioritized weak equivalence.* Prioritized weak equivalence is not a congruence, meaning that equivalent processes may not behave identically when substituted into larger contexts, but it is a useful comparison for evaluating processes meant to operate independently.

3 The VERSA Environment

The VERSA environment has been designed to allow integrated use of algebraic manipulation of specifications and state-space exploration based analysis. The design has been implemented using object-oriented techniques and the C++ language to facilitate maintenance, enhancement, and portability. The sections that follow describe the VERSA system. Section 3.1 provides an overview of the system architecture. Section 3.2 describes VERSA's input/output conventions, process syntax, command syntax, and features.

3.1 Structure

Figure 2 shows the overall architecture of the VERSA system. The architecture is divided into three major areas: front-end, intermediate representation, and analysis and rewriting.

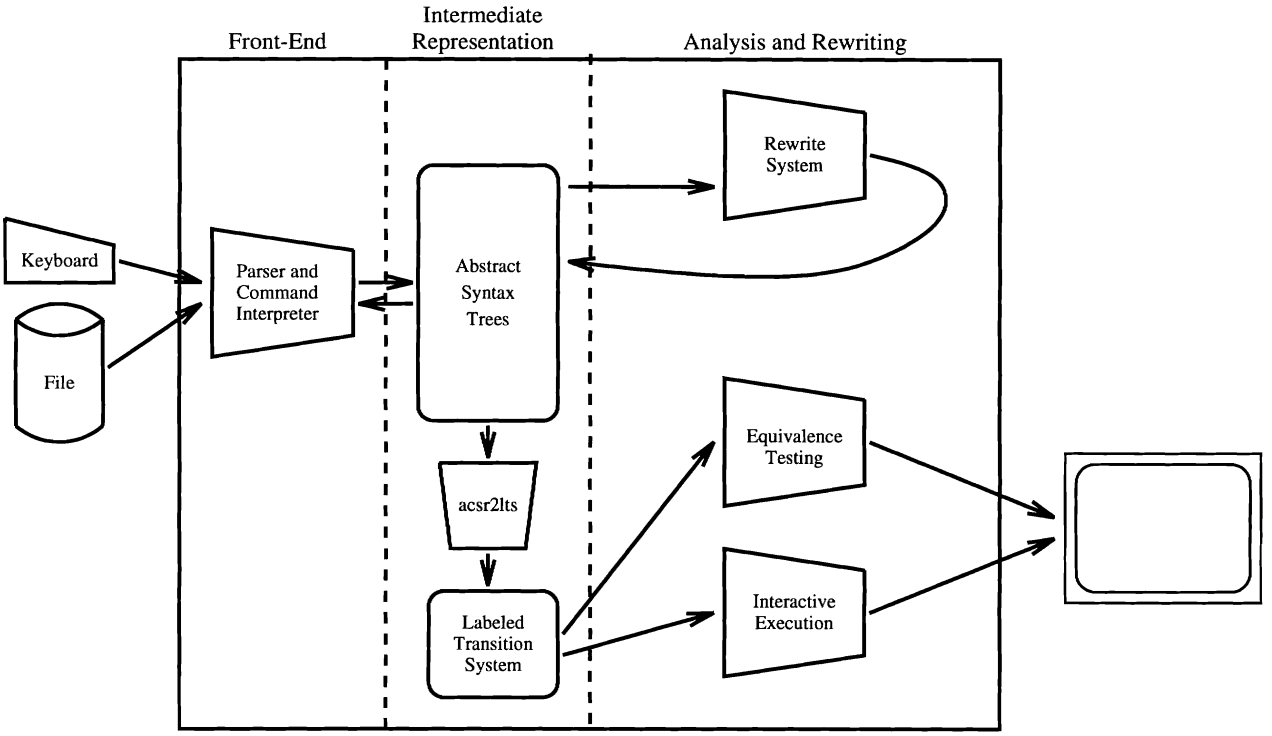


Figure 2: VERSA System Structure

The front-end is responsible for management of input sources, parsing of input commands and process definitions, and presentation of a consistent interface to the user. The current user interface is keyboard- and file-based, allowing commands and process descriptions to be entered directly from the keyboard, input from a file, or a combination of keyboard and file input. The lexical analysis and parsing of the input is handled by a Lex/Yacc based parser.

Analysis and rewriting provides support for automatic and user-directed analysis of system specifications. At present there are three major functions implemented: rewriting, equivalence testing, and interactive execution.

The rewrite system facilitates the rewriting of ACSR process expressions according to sound algebraic laws that preserve prioritized strong equivalence. At the direction of the user, the rewrite system applies pre-defined algebraic laws to one or more processes, producing a new process that may be bound to a new, or pre-existing process variable. In this way, algebraic proofs of the equivalence of process expressions may be developed.

Equivalence testing and interactive execution operate on a labeled transition system (LTS). The LTS for one or more processes is produced by an algorithm that expands the process to produce a labeled transition system representing all possible executions. The

LTS construction algorithm also prunes edges made unreachable by the semantics of the prioritized transition system, in most cases reducing the size of the resulting state machine.

Processes can be tested using a number of different notions of equivalence including syntactic equivalence, a weaker syntactic equivalence which allows renaming of process variables and simple changes in structure, prioritized strong equivalence, and prioritized weak equivalence. In the order listed, these notions of equivalence increase in computational complexity and decrease in "strength" (i.e., equate more terms).

The interactive execution feature allows user-directed execution of process specifications. The user may interactively step through the LTS one action at a time, produce traces from random executions of the LTS, save process configurations to a stack for later analysis while an alternate path is explored, and analyze the size and deadlock characteristics of the LTS resulting from their process.

3.2 Representation and Operations

While the architecture of the VERSA system's design is important for assuring high performance, ease of use, maintainability, and extensibility, what matters most to the user is the ease with which processes can be described, manipulated, and analyzed. In this section we present an overview of VERSA's input syntax, command language, rewrite system, and interactive execution environment.

3.2.1 Input conventions and facilities

One of the primary goals of the VERSA system is to make the language in which processes are represented as readable as possible. Toward that end, the syntax has been kept as close to the original ACSR syntax as possible, and syntactic conventions from mathematics have been adopted to simplify the description of large systems of process expressions.

VERSA's syntax for describing processes is necessarily different from pure ACSR, because of ACSR's extensive use of subscripting, superscripting, and characters not available on standard ASCII keyboards. The differences between the ACSR syntax (Figure 1) and VERSA syntax are shown in Table 1. The syntax of operators not appearing in this table is identical to the original ACSR syntax.

The syntax used for binding process expressions to names is a semicolon terminated assignment statement. For example, Figure 3 shows two alternative VERSA descriptions of a two-place semaphore $PV2$ composed of two copies of the process PV defined in Example 2.2. As the example demonstrates, an alternative way to write recursive processes is by using mutually recursive references to process names. It is actually the preferred way in many cases because it is more revealing. On the right-hand sides of the assignment statements, the types of `p` and `v` (labels for events) are inferred from their usage. Implicit typing is used throughout VERSA, so there is no need to explicitly declare the types of identifiers used as process variables, resource names, or event labels.

Binding of process variables within process expressions is dynamic, meaning that subsequent changes to process variable bindings can change the meaning of processes that reference

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`
$\emptyset$	{} or `idle`

Table 1: Pure ACSR vs. VERSA Syntax

```
PV2  =  PV || PV;
PV   =  {}:PV + ('p,1).(rec X.({}:X+('v,1).PV));

PV2  =  PV || PV;
PV   =  {}:PV + ('p,1).PV';
PV'  =  {}:PV' + ('v,1).PV;
```

Figure 3: Alternative VERSA Descriptions of 3-place Semaphore

them. For example, if process P is defined in terms of process Q, and Q is later re-bound to a new process expression, then the behavior of P may change as well.

Frequently, realistic process descriptions involve bounded counting which is implemented as progression through a finite sequence of similar states. When writing a process in a mathematical context, it is common to use subscripts to describe such a set of nearly identical items with a single concise expression. VERSA also allows such subscripting, as the 5-place semaphore shown in Figure 4 demonstrates. The syntax for index variable definition is borrowed from Mathematica[24], and the semantics are similar to the traditional mathematical conventions for subscripting. That is, event labels, resource names, and process variables may be of any arity, provided the arity is used consistently; index variables can only be defined in terms of expressions involving integer constants and previously defined index variables; and all references to indices must be bound within the process definition of which they are a part.

The full syntax of an index definition is

$$\{Var,\ (Finish)|(Start, Finish,\ [Step, [Conditional]])\},$$

where Var is the name of the variable being defined as an integer index; $Start$ is the initial value of Var (1 if omitted); $Finish$ is the largest value Var may attain; $Step$ is a positive

Pure ACSR	$PV5_0 = \{\} : PV5_0 + (\overline{p}, 1) \,.\, PV5_1$ $PV5_i = \{\} : PV5_i + (\overline{p}, 1) \,.\, PV5_{i+1} + (\overline{v}, 1) \,.\, PV5_{i-1}, 1 \leq i \leq 4$ $PV5_5 = \{\} : PV5_5 + (\overline{v}, 1) \,.\, PV5_4$

VERSA Syntax	`PV5[0] = {}:PV5[0] + ('p,1).PV5[1];`
	`PV5[i] = {}:PV5[i] + ('p,1).PV5[i+1] + ('v,1).PV5[i-1] {i,1,4};`
	`PV5[5] = {}:PV5[5] + ('v,1).PV5[4];`

Figure 4: Five-Place Semaphores Using Subscripted Processes

value used to increment Var; and *Conditional* is a predicate on the value of Var, constants, and values of indices from outer scopes, which determines whether or not each iteration of the index will be used to create a process. Nested indices are denoted by a comma-separated list of index definitions, evaluated from left to right. Thus,

```
P[i] = (a,1).NIL {i,1,10};
```

defines ten processes: `P[1]`, `P[2]`, $\cdots$, `P[10]` while

```
Q[i,j] = (b,1).NIL {i,1,4},{j,1,i};
```

defines a lower-triangular matrix of ten processes: `Q[1,1]`, `Q[2,1]`, `Q[2,2]`, `Q[3,1]`, $\cdots$, `Q[4,3]`, `Q[4,4]`, and

```
R[i] = (c,1).NIL {i,1,10,2};
S[i] = (d,1).NIL {i,1,10,1,((i%2)==1)};
```

each define five processes with odd indices. (The symbol `%` is the modulus operator.)

Along with the power to bind large numbers of processes in a single concise expression comes the need to operate on arbitrarily large collections of process variables and labels in a general way. VERSA addresses this problem by adding generalized versions of ACSR's binary operators. For example, Figure 5 describes an alternative implementation of a five-place semaphore through parallel composition of five copies of the process PV defined previously.

To further generalize process specifications and improve their readability, a preprocessing facility that performs macro substitution and file inclusion is also provided. The preprocessor has a syntax identical to the preprocessor used in the "C" programming language. The `#include<...>` directive includes files from a standard directory, and the `#define...` directive defines symbolic constants for substitution wherever they appear in subsequent input. Generalized macro definitions with parameter substitution and the `#undef...` directive for removing definitions are also implemented. Some uses of the preprocessing facility can be seen in later examples.

Pure ACSR	$PV5 = PV \parallel PV \parallel PV \parallel PV \parallel PV$
VERSA Syntax	`PV5 = Parallel[PV, {i,1,5}];`

Figure 5: Five-Place Semaphores Using Generalized Operator

3.2.2 The algebraic rewrite system

The algebraic approach to formal analysis of systems is a three-step process. First, the system is described as an easily understood, easily verified set of process expressions known as a specification. Then, another set of process expressions is constructed to reflect a realistic formulation of the specification that uses all of the operators of the algebra necessary for a concise and efficient implementation of the specification. Last, the user formally proves that the implementation respects the specification by algebraic manipulations according to a pre-defined set of algebraic laws that have been shown to preserve the notion of equivalence that is of interest. For VERSA, there are 32 such laws, shown in Section A of the Appendix, which preserve prioritized strong equivalence (defined in Section 2). These laws are adapted from [25], which presents a set of laws that are sound and complete for finite state ACSR agents.

Rewriting is carried out using a function-style syntax, whereby the name of the law is applied to the process being rewritten. For example, the process expression that results from applying law Choice(1) to the process $(a,1).Q + NIL$ is specified in VERSA as `Choice1((a,1).Q + NIL)`. Frequently the process that is being manipulated is not in the exact form required by the law, and so operators for abstracting and expanding substructure are provided. The *fold* operator has the syntax `fold(`*process*`,`*process_variable*`)`, and it yields the process that results from replacing every occurrence of the process bound to *process_variable* in *process* by the variable itself. The *unfold* operator has the syntax `unfold(`*process*`,`*process_variable*`)`, and it yields the process that results from replacing every occurrence of *process_variable* in *process* by the process expression bound to *process_variable*.

Example 3.1 In this example, we demonstrate how to prove by rewriting in VERSA that the process definition

$$PV = (\overline{p},1)\,.\,rec\ X.((\overline{v},1)\,.\,PV + \{\}:X) + \{\}:PV$$

is equivalent to

$$PV = \{\}:PV + (\overline{p},1)\,.\,rec\ X.(\{\}:X + (\overline{v},1)\,.\,PV)$$

The VERSA script is as follows where law Choice(3) is $P + Q = Q + P$:

```
Ready:  PV = ('p,1).rec X.(('v,1).PV + {}:X) + {}:PV;                    (1)
Ready:  Q = ('v,1).PV + ({}:X);                                          (2)
Ready:  PV = fold(PV, Q);                                                (3)
Ready:  PV?                                                              (4)
('p,1).(rec X.(Q)) + ({}:PV)
Ready:  Q = choice3(Q);                                                  (5)
Ready:  Q?                                                               (6)
({}:X) + ('v,1).PV
Ready:  PV = unfold(PV,Q);                                               (7)
Ready:  PV?                                                              (8)
('p,1).(rec X.(({}:X) + ('v,1).PV)) + ({}:PV)
Ready:  PV = choice3(PV);                                                (9)
Ready:  PV?                                                              (10)
({}:PV) + ('p,1).(rec X.(({}:X) + ('v,1).PV))
```

Step (1) defines the process PV to be manipulated. Step (2) binds the subprocess to be isolated by a fold operation to the variable Q. Step (3) hides the substructure of PV that corresponds to the process bound to Q. Step (4) confirms this operation by displaying the new binding for PV. With the process bound to Q now in the correct form for law Choice(3), Step (5) applies the law to Q to swap the two choices in Q, as confirmed in Step (6). Step (7) unfolds occurrences of process variable Q in PV according to the binding registered previously, as confirmed in Step (8). Finally, Step (9) swaps the two choices of PV to conclude the proof.

It is also possible to imbed the application of the law within the process definition, since application of a law to a process expression is recognized as a process expression by the parser. Thus, the multi-step derivation shown above can be carried by an alternate path of reasoning according to

```
PV = Choice3(('p,1).rec X. Choice3(('v,1).PV + {}:X) + {}:PV);
```
□

Our experience with the current command-based interface for rewriting is that a burdensome number of law applications, folding, and unfolding steps is required to prove even trivial properties. We expect that the graphical user interface we are developing will make the rewrite system much more comfortable to use with its point-and-click style of applying laws.

3.2.3 Queries and testing of preemption and process equivalence

As seen in Example 3.1, the VERSA command language includes a query facility. Supported query types include displaying the binding of process variables, displaying the types of other (non-process variable) identifiers, comparing actions using ACSR's preemption relation, and comparing processes for equality with respect to several different definitions of equivalence.

Comparing a pair of actions to test whether one is preempted by the other is handled by the query

$$action_1 \textit{ rel-op } action_2?$$

where *rel-op* is either `<` or `>`, and $action_1$ and $action_2$ are timed actions or events. For example,

```
(p,1) < (p,2)?
```

reports "`true`" since untimed events with the same label are comparable, and `(p,2)` has a higher priority than `(p,1)`. The query

```
{(data,2),(cpu1,1)} < (p,1)
```

reports "`not comparable`" since the event τ is the only event comparable with a timed action. On the other hand,

```
{(data,2),(cpu1,1)} < (tau,1)
```

reports "`true`" since the actions are comparable and an internal action with non-zero priority preempts a timed action.

Processes are compared using the query

$$P_1 \texttt{ == } P_2?$$

where P_1 and P_2 are process variables. The types of equality that are currently tested are as follows, in order of increasing computation complexity and decreasing strength:

Syntactic identity—The first test performed is an attempt to determine whether the two processes have identical abstract syntax tree structure, up to and including identical names for all labels. This is such a strong notion of equivalence that it is rarely used; however, it is useful for validating the final step of an algebraic proof in which two process expressions have been manipulated to a point where they have identical syntax.

Syntactic equivalence via unique fixed-point induction[26]—If the test for identical structure and naming fails, the next test to be performed is a weaker syntactic equivalence that allows renaming of process variables between the two expressions under consideration, and trivial differences in structure. Though the test is efficient (termination is guaranteed in time proportional to the length of the process expressions), it applies only to guarded sequential processes[10, 26]. Its most common use is as a final comparison after algebraic manipulation has produced two process expressions arbitrarily different in naming of process variables, and differing structurally only in the way process variables are used to represent substructure. The implementation of unique fixed-point induction is based on the algorithm presented in [15].

The processes P and Q of Figure 6, though not syntactically identical, can be shown to be equivalent via unique fixed-point induction.

```
P    =  (p,1).P' + (v,1).(p,2).{}:P;
P'   =  (v,1).{}:NIL;
Q    =  (p,1).(v,1).{}:NIL + (v,1).Q';
Q'   =  (p,2).{}:Q;
```

Figure 6: Equivalent Processes

Prioritized strong equivalence—If the tests of syntactic equivalence fail, the next test that is performed is for prioritized strong equivalence, as described in Section 2.

The test for prioritized strong equivalence is carried out by converting the process descriptions in question to state machines (LTS's, as discussed above) and applying a state minimization algorithm that preserves strong bisimulation[27] to the two machines simultaneously, as if they represented a single graph with two disjoint components. If the resulting minimal state machine has a state that includes the start state of both of the original machines, the original machines are strongly bisimilar. As conversion from an algebraic expression to an LTS requires a number of operations exponential in the size of the expressions, testing of prioritized strong equivalence can be slow for systems that produce a large number of states. For obvious reasons, infinite state processes (some of which are representable via a single finite algebraic expression) cannot be tested by this means.

The processes P and Q of Figure 7, though not equivalent syntactically, can be shown to be equivalent via prioritized strong equivalence.

```
P  =  ((p,1).(v,1).NIL || ('p,1).('v,1).NIL)\{p,v};
Q  =  (tau,1).(tau,1).NIL;
```

Figure 7: Equivalent Processes

Prioritized weak equivalence—If all other tests of equivalence fail, the final test that is performed is for prioritized weak equivalence, as described in Section 2.

The test for prioritized weak equivalence is carried out by first eliminating internal (τ) transitions from the LTS's produced by the prioritized strong equivalence test. For every state of each machine, internal (τ) transitions out of each state are replaced by all external (non-τ) transitions reachable from that state by one or more τ steps. Once all of the τ steps have been eliminated, the resulting state machines are compared using the state minimization algorithm of [27] to determine whether they are bisimilar or not.

It is important to note that elimination of τ edges *does not* involve reapplying the preemption relation. Recall that, during the initial state machine construction, edges representing preempted events and actions that could never be executed were eliminated. This process of pruning edges is *not* reapplied as part of the elimination of τ edges, since the elimination of τ edges is meant to remove unobservable internal actions, not to change the sequences of observable external actions that can be executed.

The processes P and Q of Figure 8, though not strongly bisimilar, can be shown to be equivalent via prioritized weak equivalence.

```
P  =  (tau,3).{(cpu1,1)}:(tau,3).{(cpu2,1)}:NIL;
Q  =  {(cpu1,1)}:{(cpu2,1)}:(tau,3).NIL;
```

Figure 8: Weakly Equivalent Processes

Finally, if two process are not strongly equivalent the `whynot?` command will display a shortest pair of traces that distinguishes them. The `whynot^?` command will display a shortest sequence of actions that distinguishes two processes according to the definition of prioritized weak equivalence.

Example 3.2 We have seen two implementations of five-place semaphores in Figures 4 and 5. Now we proceed to prove that they are equivalent using VERSA. The following VERSA script inputs the process definitions:

```
Ready:  #define Sp ('p,1)
Ready:  #define Sv ('v,1)
Ready:  #define N  5
Ready:
Ready:  GS[0] = {}:GS[0] + Sp.GS[1];
Ready:  GS[i] = {}:GS[i] + Sp.GS[i+1] + Sv.GS[i-1]  {i,1,N-1};
Ready:  GS[N] = {}:GS[N] + Sp.GS[N-1];
Ready:
Ready:  PV    = {}:PV + Sp.(rec X. ({}:X + Sv.PV));
Ready:  PVN   = Parallel[PV, {i,1,N}];
```

Note that the `#define` directive is used to define the abbreviations `Sp`, `Sv` and the constant `N`. We can use the query "`GS[0] == PVN?`" to test if the two implementations of five-place semaphores are equivalent:

```
Ready:  GS[0] == PVN?
    UFI not applied--operands are not guarded.
    true (by prioritized strong equivalence)
```

VERSA concludes that the two implementations are strongly bisimilar. □

Example 3.3 We test if the two specifications of the mutual exclusion problem in Section 2.4 are equivalent. The contents of the file "`mutual-exclusion.acsr`" which contains the process definitions of the two specifications are shown in Figure 9.

The VERSA script to test the equivalence of the processes `S` and `T` is shown below:

```
Ready:  #include <mutual-exclusion.acsr>
Ready:  S == T?
    UFI not applied--operands are not guarded.
    false (by prioritized strong equivalence)
    true (by prioritized weak equivalence)
Ready:  whynot?
    prefix:  --{(cpu1,1),(cpu2,1),(chan2,1),(data,0)}-->
    unmatched left:
     --(tau,(2+1))@p-->
    unmatched right:
     --{(cpu1,1),(chan1,1),(data,2),(cpu2,1)}-->
     --{(cpu1,1),(data,2),(cpu2,1)}-->
```

The `#include` directive is used to input the process definitions. A query `S == T?` is then issued to test the equivalence of `S` and `T`. As one can see, the processes `S` and `T` are not equivalent via prioritized strong equivalence but are equivalent via prioritized weak equivalence. To find out why `S` and `T` failed the test for strong equivalence, a query `whynot?` is entered. The returned messages displays that, after both processes execute the action `{(cpu1,1),(cpu2,1),(chan2,1),(data,0)}` which represents that task A running and task B requesting to use `data`, the process `S` will execute the synchronized event `p` (the P semaphore operation), but in the process `T`, both of the next steps will have task B taking `data`. Indeed, since `S` uses explicit events to model a protocol that enforces mutual exclusion and `T` does not, they cannot be strongly equivalent. Consequently, weak equivalence is the strongest equivalence possible.

The above result establishes that the correctness of the process `S` follows from the correctness of the process `T`. □

3.2.4 Executing process specifications

The LTS used for testing equivalence is also a natural base upon which to build an interactive execution environment (i.e. an interpreter) for ACSR processes. This environment is entered by typing the name of the process to execute followed by `!` or `^!`. The first form starts the interpreter with the LTS corresponding exactly to the process to be interpreted, and the second starts the interpreter with a τ-free LTS derived using the same methods used for prioritized weak equivalence testing.

Within the interpreter, execution can be controlled in the following ways:

```
     #define prc1   {(data, 1),(cpu1,1)}
     #define prc2   {(data, 2),(cpu2,1)}
     #define run1   {(cpu1, 1)}
     #define run2   {(cpu2, 1)}
     #define req1   {(cpu1, 1),(chan1,1)}
     #define req2   {(cpu2, 1),(chan2,1)}

//-------------------------------------------------------------------------
//  Specification using P,V operations
//-------------------------------------------------------------------------
     S     = ([P1 | P2 | PV] {data}) \ {p,v};
     P1    = scope((rec X. run1:X), dummy, infty, NIL, NIL, req1:P_cs1);
     P_cs1 = run1:P_cs1 + (p,1).prc1:prc1:prc1:(v,1).P1;
     P2    = scope((rec X. run2:X), dummy, infty, NIL, NIL, req2:P_cs2);
     P_cs2 = run2:P_cs2 + (p,2).prc2:prc2:prc2:prc2:(v,1).P2;
     PV    = {}:PV  + ('p,1).rec X.({}:X+('v,1).PV);

//-------------------------------------------------------------------------
//  Specification using no P,V operations
//-------------------------------------------------------------------------
     T     = [Q1 | Q2] {data};
     Q1    = scope((rec X. run1:X), dummy, infty, NIL, NIL, req1:Q_cs1);
     Q_cs1 = run1:Q_cs1 + prc1:prc1:prc1:Q1;
     Q2    = scope((rec X. run2:X), dummy, infty, NIL, NIL, req2:Q_cs2);
     Q_cs2 = run2:Q_cs2 + prc2:prc2:prc2:prc2:Q2;
```

Figure 9: VERSA Script of Specifications for Mutual Exclusion Problem

- Advance the execution along a selected edge.
- Backtrack along the current execution trace by one or more steps.
- Execute the process until a deadlocked state or a state with more than one outgoing edge is reached.
- Execute the process until a deadlocked state is reached, making a random choice whenever confronted with two or more outgoing edges.
- Set an upper bound on the number of actions that can be accumulated in a trace. When this limit is reached, no more steps will be allowed.
- Save the current state and trace information in a stack for later re-use. This allows exploration of alternative paths without the need to restart from the initial state for each path.
- Restore a saved state and trace pair.
- Discard the current trace.
- Display the current node, nodes reachable from the current node, the current setting of the trace length limit, the contents of the stack, or simulated time elapsed thus far in the execution.
- Display the current trace of timed and untimed actions.
- Display the current trace of timed actions and events, omitting τ steps.
- Display global statistics from the state machine under test. The statistics reported include (1) a count of reachable states; (2) a count of reachable edges, by label type; (3) a count of deadlocked states; (4) a count of livelocked states (states that participate in cycles of untimed events); and (5) a count of states that are capable of "stopping the clock," i.e. states that offer only untimed event transitions with which the external environment may never synchronize, thus halting the progress of time.

Example 3.4 As an example of interactive execution of ACSR processes, we analyze the behavior of the process `T` in Example 3.3 to justify that it does satisfy the conditions of mutual exclusion.

```
Ready:  #include <mutual-exclusion.acsr>
Ready:  T!
    T <1>  --{(cpu1,1),(chan1,1),(cpu2,1),(chan2,1),(data,0)}--> T->a
      <2> --{(cpu1,1),(chan1,1),(cpu2,1),(data,0)}--> T->b
      <3> --{(cpu1,1),(cpu2,1),(chan2,1),(data,0)}--> T->c
      <4> --{(cpu1,1),(cpu2,1),(data,0)}--> T->d
```

```
Ready [T]:  step 1
    T->a <1> --{(cpu1,1),(data,2),(cpu2,1)}--> T->a->
Ready [T->a]:
```

In the above VERSA script, the file "mutual-exclusion.acsr" is input and the interactive execution of the process `T` is started by issuing command "`T!`". `T` has four possible initial steps. We enter "`step 1`" to choose the first initial step which represents the situation where both tasks request to enter the critical sections. The next step then shows that task B (with higher priority) will get into the critical section immediately, while task A keeps running outside the critical section. Thus, when both tasks request to enter their critical sections, task B will enter without delay.

Let us continue the above script by backtracking and choosing the second initial step and see what happens.

```
Ready [T->a]:  back
    T <1> --{(cpu1,1),(chan1,1),(cpu2,1),(chan2,1),(data,0)}--> T->a
      <2> --{(cpu1,1),(chan1,1),(cpu2,1),(data,0)}--> T->b
      <3> --{(cpu1,1),(cpu2,1),(chan2,1),(data,0)}--> T->c
      <4> --{(cpu1,1),(cpu2,1),(data,0)}--> T->d
Ready [T]:  step 2
    T->b <1> --{(data,1),(cpu1,1),(cpu2,1),(chan2,1)}--> T->b->a
         <2> --{(data,1),(cpu1,1),(cpu2,1)}--> T->b->b
Ready [T->b]:
```

The command "`back`" is entered to backtrack one step. The second initial step is chosen (by "`step 2`") which corresponds to task A requesting to enter its critical section. Then the process `T` will be shown to have two next steps and task A will enter its critical section in both cases. This shows that whenever `data` is available and requested by a process, it will be taken without delay.

Now we look at the situation where one task is within its critical section while the other is waiting outside its critical section. We continue the above script and proceed to the next step where task B requests to enter its critical section.

```
Ready [T->b]:   step 1
    T->b->a <1> --{(data,1),(cpu1,1),(cpu2,1)}--> T->b->a->
Ready [T->b->a]:  step
    T->b->a-> <1> --{(data,1),(cpu1,1),(cpu2,1)}--> T->b->a-[2]->
Ready [T->b->a->]:  step
    T->b->a-[2]-> <1> --{(cpu1,1),(chan1,1),(data,2),(cpu2,1)}--> T->a->
                  <2> --{(cpu1,1),(data,2),(cpu2,1)}--> T->c->b
Ready [T->b->a->[2]->]:
```

We can see that, after task A finishes its processing in the critical section, the process `T` will have two next steps and both will have task B entering its critical section. This

demonstrates that no task will have to wait indefinitely before entering its critical section. □

4 Summary and Future Directions

We have described ACSR, a process algebra that includes features for representing synchronization, time, temporal scopes, resource requirements, and priority. We have also presented VERSA, a tool for interactive specification and analysis of resource-bound real-time systems. VERSA automates tedious tasks involved in the algebraic approach to the formal analysis of such systems. We consider here a few possible enhancements to the existing system.

The equivalence tests that have been implemented thus far are only the most basic tests possible. Many more equivalences are planned, some of which are unique to the ACSR paradigm. For example, that priority values match exactly in a strong bisimulation is perhaps too strong a requirement, since all that matters after the first transition of a process is the relative priorities of the remaining transitions. Research into equivalence based on matching of relative priorities has led to definitions of unprioritized equivalence and unprioritized congruence.

The state machines into which processes are translated for testing equivalence and interpretation is the simplest labeled transition system model imaginable. We are investigating ways to augment this translation with a translation from ACSR processes to CTSM (Communicating Timed State Machine)[28]. It is hoped that translating processes to CTSM will allow them to be represented with significantly fewer states, although the impact that this more compact representation will have on the complexity of analysis has yet to be determined.

Other desirable features would be the addition of a database that would allow the VERSA environment to be saved between analysis sessions, implementation of a graphical user interface based on a standard windowing package, and implementation of a model checking facility.

5 Acknowledgements

The goals and overall design of the VERSA system are a product of numerous discussions within the Real-Time Systems Group at the University of Pennsylvania. We are grateful to the referees for their many constructive comments.

References

[1] 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.

[2] L. Aceto and D. Murphy. On the Ill-Timed but Well-Caused. In *Proc. CONCUR'93, International Conference on Concurrency Theory.* LNCS 715, Springer-Verlag, August 1993.

[3] J.C.M. Baeten and J.A. Bergstra. Real time process algebra. *Formal Aspects of Computing*, 3(2):142–188, 1991.

[4] M. Hennessy and T. Regan. A Process Algebra for Timed Systems. Technical Report 5/91, Univ. of Sussex, UK, April 1991.

[5] F. Moller and C. Tofts. A Temporal Calculus of Communicating Systems. In *Proceedings of CONCUR '90*, pages 401–415. LNCS 458, Springer Verlag, August 1990.

[6] X. Nicollin and J. Sifakis. The Algebra of Timed Processes ATP: Theory and Application. Technical Report RT-C26, Institut National Polytechnique De Grenoble, November 1991.

[7] G.M. Reed and A.W. Roscoe. Metric Spaces as Models for Real-Time Concurrency. In *Proceedings of Mathematical Foundations of Computer Science*, pages 1–12. LNCS 298, Springer Verlag, April 1987.

[8] Wang Yi. CCS + Time = An Interleaving Model for Real Time Systems. In *Proceedings of the International Conference on Automata, Languages and Programming*, pages 217–228. LNCS 510, Springer Verlag, July 1991.

[9] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–676, August 1978.

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

[11] F. Moller and C. Tofts. A Temporal Calculus of Communicating Systems. In *Proceedings of CONCUR '90*, pages 401–415. LNCS 458, Springer Verlag, August 1990.

[12] Formal Systems (Europe) Ltd., 3 Alfred Street—Oxford OX1 4eH—UK. *Failures Divergence Refinement: User Manual and Tutorial*, April 1993.

[13] R. Cleaveland, J. Parrow, and B. Steffen. The concurrency workbench: A semantics-based tool for the verification of concurrent systems. *ACM Transactions on Programming Languages and Systems*, 15(1):36–72, January 1993.

[14] H. Lin. *PAM User Manual.* School of Cognitive and Computing Sciences, University of Sussex, 1991.

[15] H. Lin. Pam: A process algebra manipulator. In *Proceedings of the Third Workshop on Computer Aided Verification*, pages 136–146. LNCS 575, Springer Verlag, July 1991.

[16] F. Jahanian and A. Mok. Modechart: A specification language for real-time systems. *IEEE Transactions on Software Engineering*, 1994. To appear.

[17] C. Heitmeyer, B. Labaw, P. Clements, and A. Mok. Engineering CASE Tools to Support Formal Methods for Real-Time Software Development. In *Proc. Fifth International Workshop on Computer-Aided Software Engineering*, pages 110–113. IEEE Computer Society Press, July 1992.

[18] P. Clements, C. Heitmeyer, B. Labaw, and A. Rose. MT: A Toolset for Specifying and Analyzing Real-Time Systems. In *Proc. Real Time Systems Symposium*, pages 12–22. IEEE Computer Society Press, December 1993.

[19] J. Ostroff. A verifier for real-time properties. *Journal of Real-Time Systems*, 4(1):5–35, March 1992.

[20] I. Lee and V. Gehlot. Language Constructs for Distributed Real-Time Programming. In *Proc. IEEE Real-Time Systems Symposium*, pages 57–66. IEEE Computer Society Press, December 1985.

[21] D. Park. Concurrency and Automata on Infinite Sequences. In *Proc. of 5th GI Conference*, pages 167–183. LNCS 104, Springer Verlag, March 1981.

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

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

[24] S. Wolfram. *Mathematica: A System for Doing Mathematics by Computer.* Addison Wesley, second edition, 1991.

[25] P. Brémond-Grégoire, Jin-Young Choi, and Insup Lee. The soundness and completeness of acsr (algebra of communicating shared resources). Technical Report MS-CIS-93-59, Department of Computer and Information Science, University of Pennsylvania, June 1993.

[26] M. Hennessy. *Algebraic Theory of Processes.* MIT Press Series in the Foundations of Computing. MIT Press, 1988.

[27] P. C. Kanellakis and S. A. Smolka. CCS expressions, finite state processes, and three problems of equivalence. *Information and Computation*, 86(1):43–68, May 1990.

[28] I. Kang and I. Lee. A state minimization algorithm for communicating state machines with arbitrary data space. Technical Report MS-CIS-93-07, Department of Computer and Information Science, University of Pennsylvania, February 1993.

A ACSR Laws

Choice(1)	$P + \text{NIL} = P$
Choice(2)	$P + P = P$
Choice(3)	$P + Q = Q + P$
Choice(4)	$(P + Q) + R = P + (Q + R)$
Choice(5)	$A_1 : P_1 + A_2 : P_2 = A_2 : P_2$ if $A_1 \prec A_2$
Choice(6)	$(a_1, n_1).P_1 + (a_2, n_2).P_2 = (a_2, n_2).P_2$ if $(a_1, n_1) \prec (a_2, n_2)$
Choice(7)	$A : P + (\tau, n).Q = (\tau, n).Q$ if $n > 0$
Par(1)	$\text{NIL} \| \text{NIL} = \text{NIL}$
Par(2)	$(A : P) \| \text{NIL} = \text{NIL}$
Par(3)	$(a, n).P \| \text{NIL} = (a, n).(P \| \text{NIL})$
Par(4)	$P \| Q = Q \| P$
Par(5)	$(P \| Q) \| R = P \| (Q \| R)$
Par(6)	$(\sum_{i \in I} A_i : P_i \ + \ \sum_{i \in J} (a_j, n_j).Q_j) \ \| \ (\sum_{k \in K} B_k : R_k \ + \ \sum_{l \in L} (b_l, m_l).S_l)$

$$
= \left[\begin{array}{l}
\quad \sum_{\substack{i \in I, j \in J, \\ \rho(A_i) \cap \rho(B_j) = \emptyset}} (A_i \cup B_j) : (P_i \| Q_j) \\
+ \ \sum_{j \in J} (a_j, n_j).(P_j \| (\sum_{k \in K} B_k : R_k \ + \ \sum_{l \in L} (b_l, m_l).S_l)) \\
+ \ \sum_{l \in L} (b_l, m_l).((\sum_{i \in I} A_i : P_i \ + \ \sum_{j \in J} (a_j, n_j).Q_j)) \| S_l \\
+ \ \sum_{\substack{j \in J, l \in L, \\ a_j = \bar{b_l}}} (\tau, n_j + m_l).(Q_j \| S_l)
\end{array} \right.
$$

Table 2: ACSR Laws (Part 1 of 2)

Scope(1)	$A : P \stackrel{b}{\triangle}_t (Q, R, S) = A : (P \stackrel{b}{\triangle}_{t-1} (Q, R, S)) + S$ if $t > 0$
Scope(2)	$(a, n).P \stackrel{b}{\triangle}_t (Q, R, S) = (a, n).(P \stackrel{b}{\triangle}_t (Q, R, S)) + S$ if $t > 0 \wedge a \neq b$
Scope(3)	$(a, n).P \stackrel{b}{\triangle}_t (Q, R, S) = (\tau, n).Q + S$ if $t > 0 \wedge a = b$
Scope(4)	$P \stackrel{b}{\triangle}_0 (Q, R, S) = R$
Scope(5)	$(P_1 + P_2) \stackrel{b}{\triangle}_t (Q, R, S) = P_1 \stackrel{b}{\triangle}_t (Q, R, S) \ + \ P_2 \stackrel{b}{\triangle}_t (Q, R, S)$
Scope(6)	$\text{NIL} \stackrel{b}{\triangle}_t (Q, R, S) = S$
Res(1)	$\text{NIL}\backslash F = \text{NIL}$
Res(2)	$(P + Q)\backslash F = (P\backslash F) + (Q\backslash F)$
Res(3)	$(A : P)\backslash F = A : (P\backslash F)$
Res(4)	$((a, n).P)\backslash F = (a, n).(P\backslash F)$ if $a \notin F \wedge \overline{a} \notin F$
	$((a, n).P)\backslash F = NIL$ if $a \in F \vee \overline{a} \in F$
Res(5)	$(P\backslash F_1)\backslash F_2 = P\backslash(F_1 \cup F_2)$
Res(6)	$([P]_I)\backslash F = [P\backslash F]_I$
Close(1)	$[\text{NIL}]_I = \text{NIL}$
Close(2)	$[P + Q]_I = [P]_I + [Q]_I$
Close(3)	$[A_1 : P]_I = (A_1 \cup A_2) : [P]_I$ where $A_2 = \{(r, 0) \mid r \in I - \rho(A_1)\}$
Close(4)	$[(a, n).P]_I = (a, n).[P]_I$
Close(5)	$[[P]_I]_J = [P]_{I \cup J}$
Close(6)	$[P\backslash F]_I = ([P]_I)\backslash F$
Rec(1)	$recX.P = P[recX.P/X]$

Table 3: ACSR Laws (Part 2 of 2)