



University of Pennsylvania
ScholarlyCommons

Technical Reports (CIS)

Department of Computer & Information Science

March 1995

GCSR: A Graphical Language With Algebraic Semantics for the Specification of Real-Time Systems

Hanene Ben-Abdallah
University of Pennsylvania

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

Jin Young Choi
University of Pennsylvania

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

Recommended Citation

Hanene Ben-Abdallah, Insup Lee, and Jin Young Choi, "GCSR: A Graphical Language With Algebraic Semantics for the Specification of Real-Time Systems", . March 1995.

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-95-09.

This paper is posted at ScholarlyCommons. https://repository.upenn.edu/cis_reports/206
For more information, please contact repository@pobox.upenn.edu.

GCSR: A Graphical Language With Algebraic Semantics for the Specification of Real-Time Systems

Abstract

Graphical Communicating Shared Resources, GCSR, is a formal language for specifying real-time systems including their functional and resource requirements. A GCSR specification consists of a set of nodes that are connected with directed, labeled edges, which describe possible execution flows. Nodes represent instantaneous selection among execution flows, or time and resource consuming system activities. In addition, a node can represent a system subcomponent, which allows modular, hierarchical, thus scalable system specifications. Edges are labeled with instantaneous communication actions or time to describe the duration of activities in the source node. GCSR supports the explicit representation of resources and priorities to resolve resource contention. The semantics of GCSR is the Algebra of Communicating Shared Resources, a timed process algebra with operational semantics that makes GCSR specifications executable. Furthermore, the process algebra provides behavioral equivalence relations between GCSR specifications. These equivalence relations can be used to replace a GCSR specification with an equivalent specification inside another, and to minimize a GCSR specification in terms of the number of nodes and edges. The paper defines the GCSR language, describes GCSR specification reductions that preserve the specification behaviors, and illustrates GCSR with example design specifications.

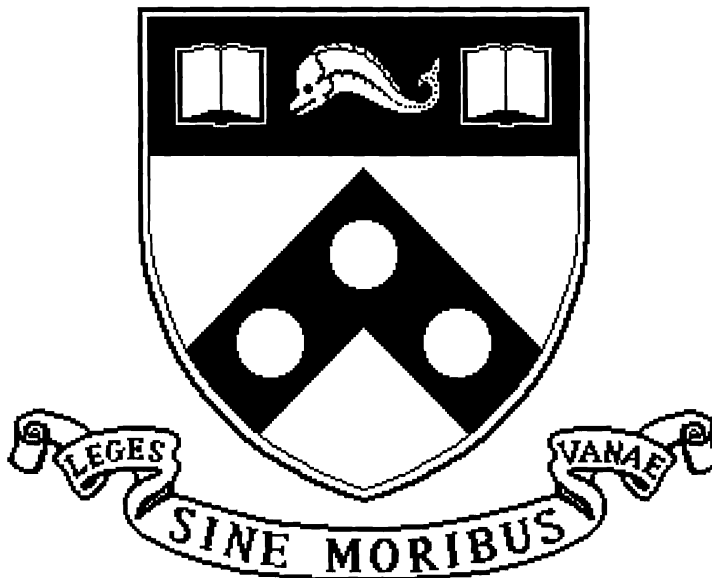
Comments

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-95-09.

GCSR: A Graphical Language with Algebraic Semantics for the Specification of Real-Time Systems

MS-CIS-95-09

Hanene Ben-Abdallah
Insup Lee
Jin-Young Choi



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

March 1995

GCSR: A Graphical Language with Algebraic Semantics for the Specification of Real-Time Systems *

Hanêne Ben-Abdallah, Insup Lee, and Jin-Young Choi
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104

March 1995

Abstract

Graphical Communicating Shared Resources, GCSR, is a formal language for specifying real-time systems including their functional and resource requirements. A GCSR specification consists of a set of nodes that are connected with directed, labeled edges, which describe possible execution flows. Nodes represent instantaneous selection among execution flows, or time and resource consuming system activities. In addition, a node can represent a system subcomponent, which allows modular, hierarchical, thus, scalable system specifications. Edges are labeled with instantaneous communication actions or time to describe the duration of activities in the source node. GCSR supports the explicit representation of resources and priorities to resolve resource contention. The semantics of GCSR is the Algebra of Communicating Shared Resources, a timed process algebra with operational semantics that makes GCSR specifications executable. Furthermore, the process algebra provides behavioral equivalence relations between GCSR specifications. These equivalence relations can be used to replace a GCSR specification with an equivalent specification inside another, and to minimize a GCSR specification in terms of the number of nodes and edges. The paper defines the GCSR language, describes GCSR specification reductions that preserve the specification behaviors, and illustrates GCSR with example design specifications.

*This research was supported in part by NSF CCR93-11622 and ARO DAAH04-95-1-0092.

1 Introduction

The Communicating Shared Resources (CSR) paradigm is an ongoing project at the University of Pennsylvania to build a framework for the development of real-time systems. This project has been motivated by a demand for a rigorous framework in which various design alternatives for a real-time system can be formally specified and rigorously analyzed and tested before implementation. This is an effort to reduce potential high cost associated with incorrect operation of real-time systems which are often embedded in safety-critical applications.

The CSR paradigm is based on the premise that the timed behavior of a real-time system is affected not only by the time its components take to execute and synchronize, but also by delays introduced due to the scheduling of actions that compete for shared resources. One of the objectives of the CSR paradigm is therefore to provide a formalism where the run-time resource requirements of a real-time system can be specified together with its functional requirements. The integration allows designers to consider resource-induced constraints during the design stage of the development cycle and to eliminate unimplementable design alternatives without expensive prototyping.

Within the CSR paradigm, the Algebra of Communicating Shared Resources, ACSR [11], has been developed as a formal specification language. ACSR is distinguished from other real-time formalisms (e.g., temporal logics, net-based models, and automata theory) by its notions of resources and priorities which allow the specification of the resource requirements of a real-time system. Two additional advantages of ACSR are its precise operational semantics and its equivalence relations. The operational semantics makes it possible to execute an ACSR specification to detect unintended behaviors of the specification before attempting to prove its correctness. The equivalence relations make it possible to compare specifications (e.g., requirements and design) and to replace one specification for another inside a larger system. This in turn makes modular specification and verification possible within the ACSR formalism. Recently, ACSR has been implemented in VERSA [2], a toolkit for the specification, syntactic manipulation, execution, and analysis of ACSR specifications.

Similar to most other formal specification languages, however, ACSR is based on a textual notation that often produces cumbersome, difficult to understand specifications. In an effort to make ACSR more accessible to average software engineers who are not necessarily experts in process algebra, we have developed a graphical language for real-time systems called Graphical Communicating Shared Resources (GCSR). Figure 1 shows the overall structure of the GCSR development environment. GCSR borrows intuitive graphical concepts of nodes and edges from control flow graphs which have been widely used in software engineering through methodologies such as Structured Analysis [4]. GCSR has several advantages: it allows scalable specification of complex systems in a modular and hierarchical fashion; it allows the integration of a system functional requirements with its resource requirements in a natural way that produces easy to understand and modify specifications; and it has a precise operational, i.e., executable, semantics that corresponds to ACSR. The GCSR-ACSR correspondence makes it possible to combine both types of specifications and to use the analysis tools provided through VERSA.

The remainder of the paper is organized as follows. Section 2 presents the GCSR compu-

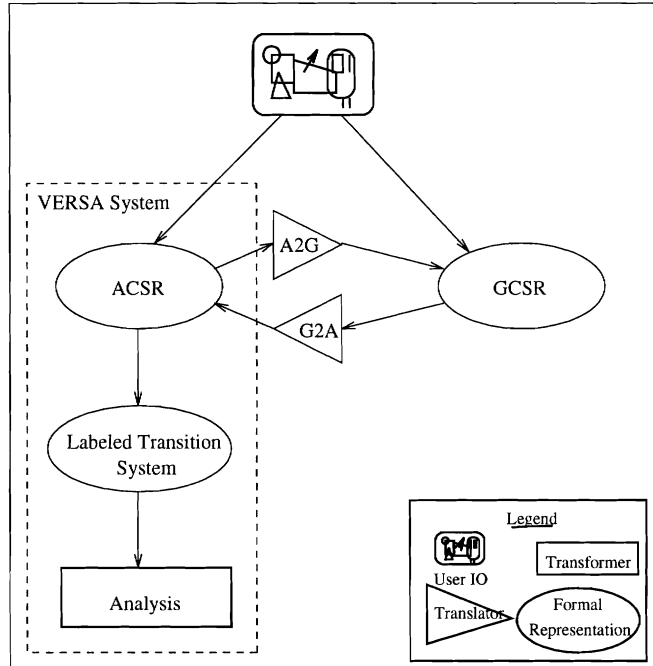


Figure 1: GCSR Specification and Analysis Environment

tation model and syntax. Section 3 reviews ACSR and describes the correspondence between GCSR and ACSR. Section 4 models and analyzes a railroad crossing system in the GCSR formalism. Section 5 reviews related work in the area of graphical, formal specification languages for real-time systems and outlines future work based on GCSR.

2 The GCSR Language

The GCSR paradigm is based on the view that a real-time system consists of a set of communicating components, called *processes*, that use a finite set of serial, shared resources for execution and that synchronize with one another. The use of shared resources is represented by timed actions, called *actions*, and synchronization is supported by instantaneous actions, called *events*. The execution of an action is assumed to take nonzero time units with respect to a global clock, and to consume a set of resources during that time. The execution of an action is subject to the availability of the resources it uses. Contention for resources is arbitrated according to the priorities of competing actions. To ensure uniform progress of time, processes execute actions synchronously.

Unlike an action, the execution of an event is instantaneous and never consumes any resource. Processes execute events asynchronously except when two processes synchronize through matching events. Priorities are used to arbitrate the choice when several events are possible. Event priorities model priorities inherited from the functional requirements and “break a tie” between competing services.

Graphically, a GCSR process is represented by a finite set of *nodes* that are connected with directed *edges*.

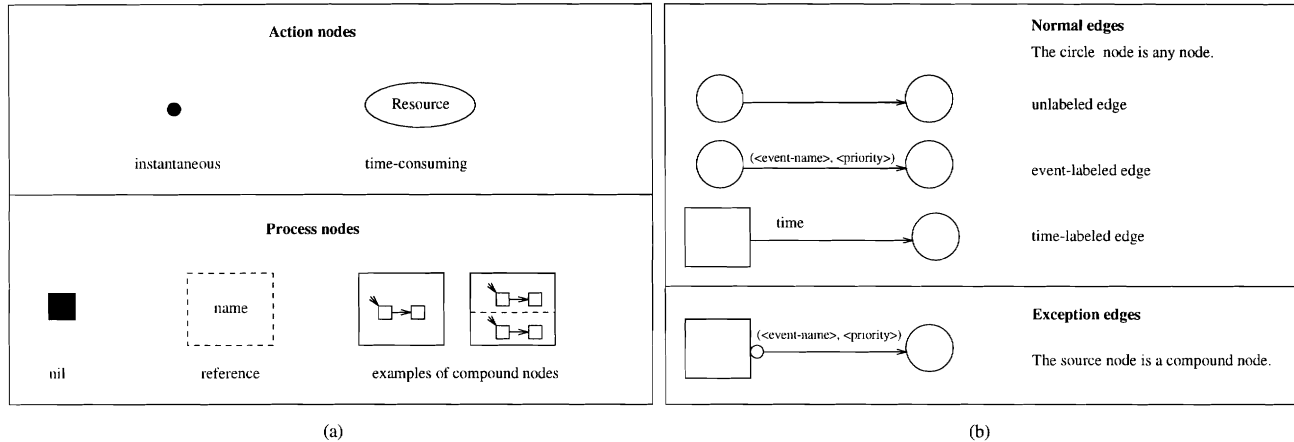


Figure 2: (a) GCSR Nodes; (b) GCSR Edges

2.1 Nodes

GCSR offers several types of nodes for a succinct and graphically clear representation of various system activities. GCSR classifies nodes into two main categories: *action* nodes and *process* nodes. Figure 2 (a) shows the graphical symbols of GCSR nodes.

An action node describes a system execution state where a basic activity, i.e., decision or timed action, is executed. Action nodes are divided into two types: 1) the *instantaneous* node describes a system state where an undelayed communication or control transfer is executed; and 2) the *time-consuming* node describes a system action, e.g., computation that takes time and consumes resources. The *Resource* attribute of a time-consuming node identifies the set of resources required for the execution of the action.

A process node describes a system execution state in which multiple activities are executed. Process nodes are divided into three types: 1) the *nil* node represents a halting process, i.e., end of system execution; 2) the *reference* node refers to a GCSR process by its name. Reference nodes help visually to structure a large specification into pieces; and 3) the *compound* node describes the details of one GCSR process or multiple GCSR processes executing in parallel.

The compound node is essential in supporting scalable specifications. It allows grouping of GCSR processes into a higher level entity. Furthermore, it can also be used to connect several GCSR processes that are executed sequentially. This makes it possible to describe a system in a modular fashion. In addition to this structural modularity, a compound node encapsulates dependencies through two attributes, *Restrict* and *Close*. The *Restrict* attribute identifies a set of events which are visible only among the GCSR processes nested inside the node. The *Close* attribute identifies a set of resources which can be used only by the nested GCSR processes and thus are unavailable outside the compound node.

2.2 Edges

GCSR divides edges into two categories shown in Figure 2 (b): *normal* edges to represent transitions between nodes at the same level, and *exception* edges to emulate transitions between nodes at different levels of nesting. The two types of edges are motivated by the desire to support a structured, hierarchical specification in which edges do not cross node boundaries. Also, we want to graphically distinguish two types of control flow: one that is externally controlled by an interacting process and one that is triggered internally through voluntary release of control by raising an exception.

A normal edge is either unlabeled or labeled with an event or time. An event is denoted by a pair $(\langle event-name \rangle, \langle priority \rangle)$, where the priority is an integer that represents the urgency of the event. We use $e?$ to denote receiving the event e and $e!$ to denote sending the event e . Time is denoted by a positive integer constant.

An exception edge has a source node that is either a compound or reference node. Control moves to the destination node of the exception edge when the process inside the source node executes an exception event that labels the exception edge. The transfer of control through an exception edge allows synchronization between a process inside a compound node with an outside node.

There can be multiple edges originating from a common source node; this allows the choice among several activities. GCSR uses ACSR notion of prioritized transitions, which is described in Subsection 3.1.2, to arbitrate among simultaneously possible transitions.

2.3 Gate Example

Figure 3 shows GCSR specification of the gate component of a railroad crossing system [7]. The GCSR specification of the gate is divided into four modules: an initial process, *Gate*, two processes responsible of lowering the gate, *GD* and *GD'*, and a process, *GU*, responsible of raising the gate.

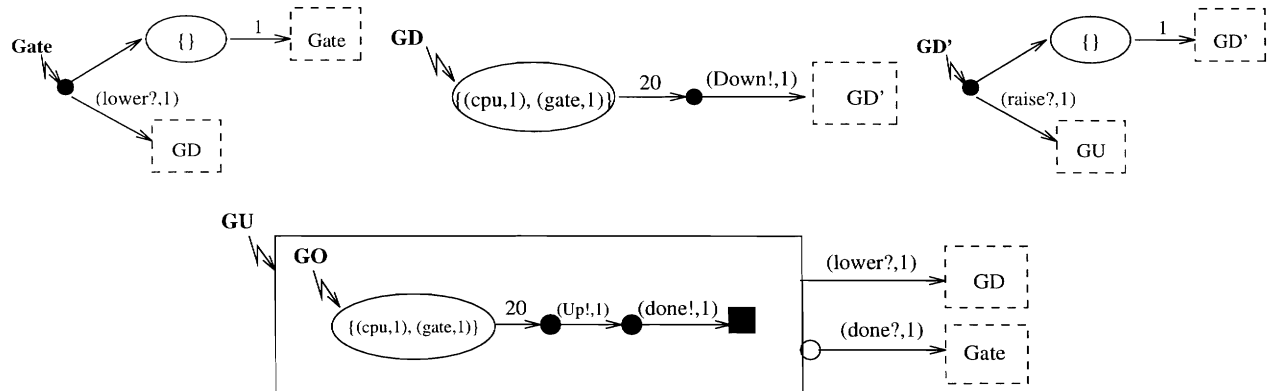


Figure 3: Gate Example

The behavior of the system is described by a control flow that starts at the initial node of *Gate* and moves across the nodes until it reaches a nil node, at which time the system ends its

execution. When control enters the initial node of **Gate** which is an instantaneous node, there are two possibilities corresponding to the outgoing edges. One possibility is to idle by moving control to the time consuming node with the empty set of resources. The other possibility is to receive the event (*lower?*, 1), in which case control moves to the reference node named **GD** and the system starts behaving like the **GD** process. In the complete specification of the railroad crossing system, where the **Gate** is running in parallel with other components, e.g, train, the semantics of GCSR ensures that the transition on receiving the *lower* event has a higher priority than the idle transition. Once control enters the initial node of **GD**, it stays in this node for 20 time units while continuously consuming the cpu resource at priority 1 and the gate resource at priority 1. Afterwards, control moves to the target instantaneous node, and so on so forth. The initial node of the process **GU** is a compound node. The process nested inside this node, **GO**, represents the gate opening activities. This process can be interrupted by the reception of the event *lower*; in this case, control moves to the process responsible of closing the gate, **GD**. If there is no *lower* request, the gate is opened in 20 time units; here, it signals *done* and control moves to the initial process **Gate**.

2.4 GCSR Processes

A *GCSR process* is a tuple $\langle N, I, E, \mathcal{L}, \mathcal{R} \rangle$ where \mathcal{L} is a set of event names, \mathcal{R} is a set of resource names, (N, E) is a directed graph with initial nodes $I \subseteq N$. The set of labeled edges $E \subseteq N \times L \times N$ is defined over the set of labels $L \subseteq \{\epsilon\} \cup (\mathcal{L} \times \mathbf{N}) \cup (\mathbf{N} \cup \{\infty\})$ where ϵ denotes an empty label.

Since GCSR is hierarchical, the set of nodes N is defined together with a *hierarchy function* ρ which defines for each node the set of GCSR processes it contains inside. That is, $\rho(n) = \{G_1, \dots, G_k\}$ if the GCSR processes G_1, \dots, G_k are contained immediately inside the node n . Given the hierarchy function ρ , a node n is either an instantaneous, time-consuming, nil, or reference node, if $\rho(n) = \emptyset$; and a node n is a compound node, if $\rho(n) = \{G_1, \dots, G_k\}$.

To assign semantics to GCSR specifications, we next describe rules to restrict how GCSR nodes and edges can be combined to form a *valid* GCSR specification.

Definition 2.1 A *valid* GCSR process has edges and nodes that satisfy the following restrictions:

- Edges:
 1. Each edge connects nodes only at the same level.
 2. For every edge (s, l, d) , and for every compound node n , $\rho(n) = \{G_1, \dots, G_k\}$, s is a node of G_i if and only if d is a node of G_i , for $i = 1, \dots, k$.
- Action nodes:
 1. Each instantaneous node can have only unlabeled or event-labeled, normal out-edges, and it can not be a sink node.
 2. Each time-consuming node must have one time-labeled out-edge and nothing else.
- Process nodes:

1. Each nil node is a sink node, i.e., has no out-edge.
2. Any reference and compound node can have multiple event-labeled normal out-edges, at most one time-labeled out-edge, and at most one exception out-edge.

Figure 4 shows examples of illegal compound node and edges. In the rest of this paper,

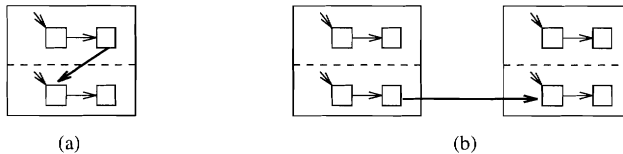


Figure 4: Illegal Compound Node (a) and Edges (a) and (b)

GCSR process means valid GCSR process.

3 The Semantics of GCSR

The underlying semantics of GCSR is the Algebra of Communicating Shared Resources (ACSR). We define the semantics of GCSR in two steps. The first step is to identify *pure* GCSR, a subset of GCSR with an obvious correspondence with ACSR. The second step is to define a set of transformations from pure GCSR to GCSR and vice versa. Thus, every valid GCSR process can be translated into an ACSR process by first converting it to a pure GCSR process and then mapping the pure GCSR process to a corresponding ACSR process.

3.1 The ACSR Language

ACSR augments CCS [13] with time and resource-consuming prioritized actions. Although ACSR has been developed for both dense time [1] and discrete time [11], the analysis toolkit VERSA supports only discrete time.

The computation model of ACSR is based on the view that a real-time system is a set of communicating processes that compete for shared resources. The use of shared resources is represented by *actions* and communication is supported by *events*. An action represents the progress of one time unit with the consumption of resources. The execution of an action, which is a set of (resource-name, priority) pairs, is subject to the availability of the resources it uses. The contention for resources is arbitrated according to the priorities of competing actions. As an example, the action $\{(cpu, 1)\}$ denotes the use of the *cpu* resource at the priority level 1, whereas the action $\{(cpu, 1), (memory, 2)\}$ denotes the use of the *cpu* resource at the priority level 1 as well as the *memory* resource at the priority level 2. The action \emptyset represents the passage of one time unit without consuming any resources, that is, idling for one time unit.

The events provide a basic mechanism for synchronization and communication between processes, and for outside observation. The execution of an event never consumes any resource. Processes execute events asynchronously except when two processes synchronize through matching events. Priorities are used to arbitrate the choice when several events are

possible at the same time. An event is denoted by a pair (a, p) , where a is the *label* of the event, and p is its *priority*. Labels are drawn from the set $\mathcal{L} \cup \bar{\mathcal{L}} \cup \{\tau\}$, where if a is a given label, we say that \bar{a} is its *inverse* label; i.e., $\bar{\bar{a}} = a$. As in CCS, the special identity label, τ , arises when two events with inverse labels are executed in parallel.

3.1.1 The Syntax of ACSR

We use A, B and C for actions and e, f and g for events. When we do not want to distinguish whether it is an action or an event, we use α and β .

The following grammar describes the syntax of ACSR processes:

$$P ::= \text{NIL} \mid A : P \mid (a, n).P \mid P + P \mid P \parallel P \mid \\ P \Delta_t^a(Q, R, S) \mid [P]_I \mid P \setminus F \mid \text{rec } X.P \mid X$$

NIL is a process that executes no action (i.e., it is initially deadlocked). There are two prefix operators, corresponding to actions and events, respectively. The first, $A : P$, executes a resource-consuming action A at the first time unit, and proceeds to the process P . On the other hand, $(a, n).P$, executes the event (a, n) , and proceeds to P . The difference here is that we consider no time to pass during the event occurrence. The Choice operator $P + Q$ represents possible executions – either of the enabled processes may be chosen to execute. The operator $P \parallel Q$ is the parallel composition of P and Q .

The Scope construct $P \Delta_t^a(Q, R, S)$ binds the process P by a temporal scope [12], and incorporates both the features of timeouts and interrupts. We call t the *time bound*, where $t \in \mathbf{N}^+ \cup \{\infty\}$ (i.e., t is either a non-negative integer or infinity). The scope may be exited in a number of ways. First, if P successfully terminates before t time units by executing an event labelled with \bar{a} , then control proceeds to the “success-handler” Q (here, a may be any label other than τ .) On the other hand, R is a timeout exception-handler; that is, if P fails to terminate before t time units, then control proceeds to R . Lastly, at any time while P is executing it may be interrupted by S , and the scope is then departed.

The Close operator, $[P]_I$, produces a process P that monopolizes the resources in the set I . The Restriction operator, $P \setminus F$, limits the behavior of P . Here, no events with labels in F are permitted to execute. The process $\text{rec } X.P$ denotes standard recursion, allowing the specification of infinite behaviors.

To simplify the description of large systems, we augment the ACSR syntax with a definition operator as well as indexed processes and events. We use $X \stackrel{\text{def}}{=} P$ to refer to the process expression P by the process name X . We use subscripts to define indexed processes and events, e.g., P_1 and (a_2, p) .

3.1.2 The Semantics of ACSR

The semantics of ACSR is defined in two steps. First, we develop the *unconstrained* transition system, where a transition is denoted as $P \xrightarrow{\alpha} P'$. Within “ \rightarrow ” no priority information is used to prune impossible executions; we subsequently refine “ \rightarrow ” to define our prioritized transition system, “ \rightarrow_π .”

The two rules for the prefix operators are *axioms*; i.e., they have premises of *true*. There is one rule for an action, and one for an event.

$$\mathbf{ActT} \frac{-}{A : P \xrightarrow{A} P} \qquad \mathbf{ActI} \frac{-}{(a, n).P \xrightarrow{(a, n)} P}$$

For example, the process $\{(cpu, 1)\} : C_{1,1}$ can use the *cpu* resource at priority level 1 for one time unit and proceeds to the process $C_{1,1}$. Alternatively, the task process $T_1 \stackrel{\text{def}}{=} (s_1, 1).C_{1,0}$ can execute the event $(s_1, 1)$ and proceeds to the process $C_{1,0}$.

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

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

As an example, the process $C_{1,c_1} \stackrel{\text{def}}{=} \emptyset : C_{1,c_1} + (s_1, 1).C_{1,0}$ may choose between idling for one time unit or executing the event $(s_1, 1)$. The former behavior is deduced from rule **ChoiceL**, while the latter is deduced from **ChoiceR**.

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

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

where $\rho(A)$ is the set of resources used by the action A ; e.g., $\rho(\{(cpu, 1), (buffer, 2)\}) = \{cpu, buffer\}$. Note that timed transitions are synchronous, in that the resulting process advances only if both of the constituents take a step. 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 actions, events may occur asynchronously.

$$\mathbf{ParIL} \frac{P \xrightarrow{(a, n)} P'}{P \parallel Q \xrightarrow{(a, n)} P' \parallel Q} \qquad \mathbf{ParIR} \frac{Q \xrightarrow{(a, n)} Q'}{P \parallel Q \xrightarrow{(a, n)} P \parallel Q'}$$

$$\mathbf{ParCom} \frac{P \xrightarrow{(a, n)} P', Q \xrightarrow{(\bar{a}, m)} Q'}{P \parallel Q \xrightarrow{(\tau, n+m)} P' \parallel Q'}$$

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

Example 3.1 Consider the following two processes:

$$\begin{aligned} P &\stackrel{\text{def}}{=} (s, 3).P_1 + \{(cpu_1, 8)\} : P_2 \\ Q &\stackrel{\text{def}}{=} (\bar{s}, 5).Q_1 + \{(cpu_2, 7)\} : Q_2 \end{aligned}$$

The compound process $P\|Q$ admits the following four transitions:

$$\begin{aligned} P\|Q &\xrightarrow{(s,3)} P_1\|Q && \text{[by ParIL]} \\ P\|Q &\xrightarrow{(\bar{s},5)} P\|Q_1 && \text{[by ParIR]} \\ P\|Q &\xrightarrow{(\tau,8)} P_1\|Q_1 && \text{[by ParCom]} \\ P\|Q &\xrightarrow{\{(cpu_1,8),(cpu_2,7)\}} P_2\|Q_2 && \text{[by ParT]} \end{aligned}$$

Note that an event transition always executes before the next “tick” of the global clock. \square

The construction of **ParCom** ensures that the *relative* priority ordering among events with the same labels remains consistent even after communication takes places. The following example shows how the ordering is preserved.

Example 3.2 Consider the following static priority scheduler where the dispatcher D has the choice between instantiating two tasks T_1 and T_2 :

$$\begin{aligned} D &\stackrel{\text{def}}{=} (\bar{s}, 5).D_1 + (\bar{s}, 3).D_2 \\ T &\stackrel{\text{def}}{=} (s, 2).T_1 + (s, 3).T_2 \end{aligned}$$

Thus, in T the second choice is preferred, while in D the first choice is preferred. There are eight possible transitions for $D\|T$:

$$\begin{array}{ll} D\|T \xrightarrow{(\bar{s},5)} D_1\|T & D\|T \xrightarrow{(\bar{s},3)} D_2\|T \\ D\|T \xrightarrow{(s,2)} D\|T_1 & D\|T \xrightarrow{(s,3)} D\|T_2 \\ D\|T \xrightarrow{(\tau,7)} D_1\|T_1 & D\|T \xrightarrow{(\tau,5)} D_2\|T_1 \\ D\|T \xrightarrow{(\tau,8)} D_1\|T_2 & D\|T \xrightarrow{(\tau,6)} D_2\|T_2 \end{array}$$

While there are now four possible transitions labelled with τ , the addition of priorities in **ParCom** ensures that the original relative orderings are maintained. Note that the τ -transition with the highest priority is that associated with the derivative $D_1\|T_2$. These transitions had the highest priorities in their original constituent processes. \square

The Scope operator possesses a total of five transition rules, which describe 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 labelled with \bar{b} , the executions of P continue.

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

$$\text{ScopeCI} \frac{P \xrightarrow{(a,n)} P'}{P \Delta_t^b(Q, R, S) \xrightarrow{(a,n)} P' \Delta_t^b(Q, R, S)} \quad (\bar{a} \neq b, t > 0)$$

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

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

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

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

Finally, **ScopeI** shows that the process S may interrupt (and kill) P while the scope is still active.

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

Example 3.3 Consider the following specification: “Execute P for a maximum of 100 time units. If P executes an event labelled with \bar{b} in that time, then stop the system. However, if P fails to finish within 100 time units, then start executing R . At any time during the execution of P , allow interruption by an event $(c, 3)$, which will halt P , and initiate the interrupt-handler S .” This system may be realized by the following process:

$$P \Delta_{100}^b(\text{NIL}, R, (c, 3).S).$$

□

The Restriction operator defines a subset of 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. Actions, on the other hand, remain unaffected.

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

Example 3.4 Restriction is particularly useful in “forcing” the synchronization between concurrent processes. In Example 3.1, synchronization on s and \bar{s} is not forced, since $P \parallel Q$ has transitions labelled with s and \bar{s} . On the other hand, $(P \parallel Q) \setminus \{s\}$ has only two transitions:

$$(P \parallel Q) \setminus \{s\} \xrightarrow{(\tau, 8)} (P_1 \parallel Q_1) \setminus \{s\} \quad \text{and} \quad (P \parallel Q) \setminus \{s\} \xrightarrow{\{(cpu_1, 8), (cpu_2, 7)\}} (P_2 \parallel Q_2) \setminus \{s\}$$

In effect, the restriction declares that s and \bar{s} define a “local channel” between P and Q . □

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 an action A . If A utilizes less than the full resource set I , the action is augmented with $(r, 0)$ pairs for each unused resource $r \in I - \rho(A)$.

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

$$\mathbf{CloseI} \quad \frac{P \xrightarrow{(a,n)} P'}{[P]_I \xrightarrow{(a,n)} [P']_I}$$

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

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

where $P[rec X.P/X]$ is the standard notation for substitution of $rec X.P$ for each free occurrence of X in P .

As an example, consider $rec X.(\{(cpu, 1)\} : X)$, which indefinitely executes the action “ $\{(cpu, 1)\}$.” By **ActT**,

$$\{(cpu, 1)\} : (rec X.(\{(cpu, 1)\} : X)) \xrightarrow{\{(cpu, 1)\}} rec X.(\{(cpu, 1)\} : X),$$

so by **Rec**,

$$rec X.(\{(cpu, 1)\} : X) \xrightarrow{\{(cpu, 1)\}} rec X.(\{(cpu, 1)\} : X).$$

We define the unprioritized labelled transition system “ \rightarrow ” as follows: $P \xrightarrow{\alpha} P'$ if the transition on α is derivable by one of the rules described in this section.

The Prioritized Transition System. The prioritized transition system is based on *pre-emption*, which incorporates our treatment of synchronization, resource-sharing and priority. We use “ \prec ” to denote the preemption relation. For two actions or events, α and β , if $\alpha \prec \beta$, then we 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 . Informally, there are three cases for $\alpha \prec \beta$: 1) α and β are events with the same label and β has a higher priority; 2) α and β are actions and β uses a subset of resources with at least one at a higher priority than in α ; or 3) β is a τ event with a non-zero priority while α is an action.

Example 3.5 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. $(a, 1) \not\prec (b, 2)$ if $a \neq b$
- g. $(a, 2) \prec (a, 5)$
- h. $\{(r_1, 2), (r_2, 5)\} \prec (\tau, 2)$ □

We next define the prioritized transition system “ \rightarrow_π ,” which simply refines “ \rightarrow ” to account for preemption.

Definition 3.1 *The labelled transition system “ \rightarrow_π ” is defined as follows: $P \xrightarrow{\alpha} P'$ if*

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

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

Example 3.6 In Example 3.2, prioritized transition eliminates five of the eight possible transitions, leaving the following:

$$\begin{array}{ccc}
 D\|T \xrightarrow{(s,3)}_\pi D\|T_2 & & D\|T \xrightarrow{(\bar{s},5)}_\pi D_1\|T \\
 D\|T \xrightarrow{(\tau,8)}_\pi D_1\|T_2 & &
 \end{array}$$

since $(s, 2) \prec (s, 3)$, and $(\bar{s}, 3) \prec (\bar{s}, 5)$, and $(\tau, p) \prec (\tau, 8)$ for $p = 5, 6, 7$. □

3.1.3 Analysis in ACSR

The precise operational semantics of ACSR, as described earlier, makes it possible to execute an ACSR process to examine unintended behaviors. Furthermore, one of the advantages of a process algebraic formalism is its support of notions of equivalence that can be used to verify the correctness of a process, e.g., design specification, with respect to another process, e.g., requirements specification. ACSR offers notions of process equivalence that are based on the concept of *bisimulation* [14]. ACSR uses the two common notions of bisimulation, *strong bisimulation* and *weak bisimulation*. Furthermore, ACSR augments the two notions of bisimulation with priorities to define the largest prioritized strong and weak bisimulations, which are called (*prioritized*) *strong equivalence* (\sim_π) and (*prioritized*) *weak equivalence* (\approx_π), respectively. The strong equivalence, \sim_π , is a congruence relation with respect to all ACSR operators [5]. This allows modular analysis in ACSR which is essential in the verification of large-scale systems.

Using the notions of equivalence, ACSR offers *syntax-based analysis* and *semantics-based analysis* techniques. In the syntax-based analysis, ACSR defines a set of laws that can be used to syntactically transform one process to an equivalent process. In the case of discrete time, ACSR has a sound and complete set of laws that characterize strong equivalence for finite-state processes. The reader is referred to [1] for more details.

Semantics-based analysis in ACSR uses an effective algorithmic approach to verify the equivalence of any finite-state processes. The basic idea of this analysis technique is to compute the largest bisimulation of two processes. The algorithm [3, 10] minimizes the combined labelled transition systems of two processes to be compared with respect to bisimulation.

Recently, we augmented ACSR with a Hiding operator $P \setminus H$ which conceals the identify of the resources in H from the behaviors (i.e., prioritized labelled transition system) of the process P . The Hiding operator augments semantics-based analysis capabilities of ACSR. It allows the comparison of ACSR processes irrespectively of their resource usage which might be considered as detail information. This is very useful when one is interested only in the timing behavior of a process while resource usage is irrelevant. The following example illustrates how verification is facilitated through the Hide operator.

Example 3.7 Consider the following processes:

$$\begin{aligned} P &\stackrel{def}{=} \{(cpu1, 1)\} : P + \{(cpu2, 1)\} : P \\ Q &\stackrel{def}{=} \{(cpu2, 1)\} : Q + \{(cpu3, 1)\} : Q \\ R &\stackrel{def}{=} P \parallel Q \end{aligned}$$

Suppose we want to prove that the process R does not reach a deadlock state. One way is to construct such an ACSR requirements specification R' and to show that R and R' are bisimilar. Since any two of the three resources $cpu1, cpu2, cpu3$ can be used during each time unit, it is cumbersome to specify R' so that all possible resource use patterns are explicitly enumerated. But, we are not interested in the patterns of resource usage. Thus, the requirements specification can be rephrased as to show that the process $R \setminus \{cpu1, cpu2, cpu3\}$ never deadlocks. Since the resource information of R is hidden, we can use a ACSR requirements specification, $rec X. \emptyset : X$. The correctness of R can then be proved by showing that

$$R \setminus \{cpu1, cpu2, cpu3\} \approx_{\pi} rec X. \emptyset : X.$$

□

3.2 The Semantics of Pure GCSR

In this section, we define the semantics of a subset of valid GCSR, called *pure GCSR*. Pure GCSR corresponds to ACSR; that is, there is a translation between pure GCSR to ACSR (\mathcal{T}_{GA}) and vice versa (\mathcal{T}_{AG}). The main idea behind the two translations is the same, we therefore describe only \mathcal{T}_{GA} in detail.

Definition 3.2 A *pure GCSR* process is a valid GCSR such that:

1. any instant node has either one normal, event-labeled out-edge, or two normal, unlabeled out-edges;
2. any reference node is a sink node;

3. any compound node, n , has either one or two nested components, i.e., $\rho(n) = \langle G \rangle$ or $\rho(n) = \langle G_1, G_2 \rangle$; and
4. any compound node with out-edges has one nested component, one exception out-edge, one time-labeled out-edge, and one normal unlabeled edge.

Process **GD** of Figure 3 is a pure GCSR process, however, processes **Gate**, **GD'**, and **GU** are not. Figure 5 shows their pure GCSR representation.

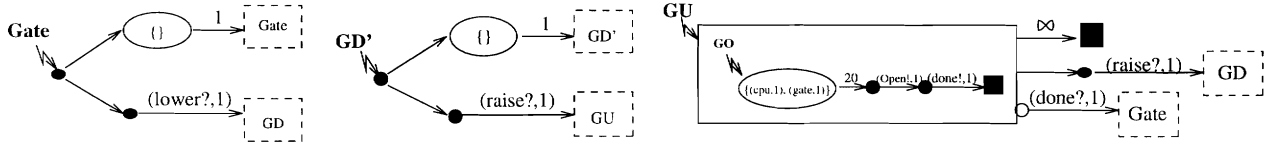


Figure 5: Pure GCSR processes

Each pure GCSR process corresponds to an ACSR process. The translation of a GCSR process consists of translating the nodes and the edges. The translation starts from the initial node and is recursively applied to processes reachable from the initial nodes. The results of the translations are then combined using an ACSR operator. Figure 6 describes the nine steps of translating a GCSR process to ACSR. The translation function is called \mathcal{T}_{GA} .

Step 1 binds the translation of the GCSR process to the process variable name P . In step 2, an event prefix process is created from the event that labels the normal edge out of the instantaneous node and the translation of the GCSR process that starts at the target node. In step 3, an action prefix process is created from the time-consuming node and its time-labeled out-edge. In step 4, each process that starts at the target node of the unlabeled edge is translated and the result is combined through the Choice operator. In step 5, the nil node is translated to the NIL process. In step 6, the reference node is translated to a process variable with the referenced name. In step 7, the GCSR process inside the compound node is translated, and then the Restrict and Close attributes of the compound node are used in the Restrict and Close operators; the attributes are ignored if they are the empty sets. In step 8, the two GCSR processes inside the compound node are translated and combined through the Parallel operator. The Restrict and Close attributes of the compound node are used in the Restrict and Close operators. In step 9, the GCSR process inside the compound node is translated and used as the main process in the Scope operator. The translation of target process of the exception edge produces the success-handler process. The translation of the time-labeled edge produces the timeout process. Finally, the translation of the GCSR process that starts at the target node of the unlabeled edge produces the interrupt process.

Figure 7 shows the translation of the pure GCSR processes **Gate**, **GD'**, and **GU** of Figure 5 and **GD** of Figure 3. Note that $(lower?, 1)$ in GCSR is represented as $(lower, 1)$ in ACSR and $(Down!, 1)$ as $(\overline{Down}, 1)$.

We can use structural induction to prove the following lemma.

Lemma 3.1 For every pure GCSR process G , $\mathcal{T}_{GA}(G)$ is an ACSR process and for every ACSR process P , $\mathcal{T}_{AG}(P)$ is a pure GCSR process.


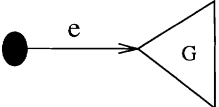
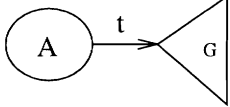
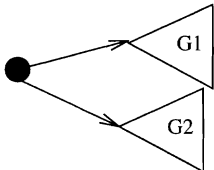


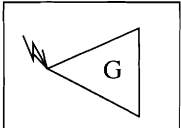
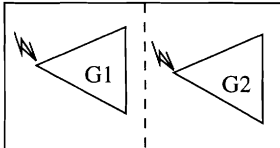
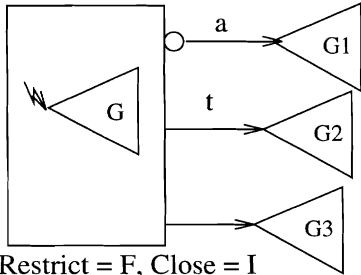
1.		$P = T(G)$
2.		$e.T(G)$
3.		$\underbrace{A:A:\dots:A:T(G)}_{t \text{ times}}$
4.		$T(G1) + T(G2)$
5.		NIL
6.		P
7.	 Restrict = F, Close = I	$[T(G) \setminus F]I$
8.	 Restrict = F, Close = I	$[(T(G1) \parallel T(G2)) \setminus F]I$
9.	 Restrict = F, Close = I	$([T(G) \setminus F]I) \Delta_t^a (T(G1), T(G2), T(G3))$

Figure 6: Pure GCSR to ACSR Translation

Gate	$\stackrel{\text{def}}{=} \emptyset : \text{Gate} + (\text{lower}, 1).\text{GD}$
GD	$\stackrel{\text{def}}{=} \{(cpu, 1), (gate, 1)\}^{20} : (\overline{Down}, 1).\text{GD}'$
GD'	$\stackrel{\text{def}}{=} \emptyset : \text{GD}' + (\text{raise}, 1).\text{GU}$
GU	$\stackrel{\text{def}}{=} \text{GO} \Delta_{\infty}^{\text{done}} (\text{Gate}, \text{NIL}, (\text{lower}, 1).\text{GD})$
GO	$\stackrel{\text{def}}{=} \{(cpu, 1), (gate, 1)\}^{20} : (\overline{Up}, 1).(\overline{done}, 1).\text{NIL}$

Figure 7: Translation of the Gate Example

Furthermore, we can establish the uniqueness of the correspondence between pure GCSR and ACSR as stated in the next Theorem.

Theorem 3.1 For every pure GCSR process G , we have $\mathcal{T}_{AG}(\mathcal{T}_{GA}(G)) \equiv G$, and for every ACSR process P , we have $\mathcal{T}_{GA}(\mathcal{T}_{AG}(P)) = P$, where \equiv denotes labeled graph isomorphism and $=$ denotes syntactic equality between processes modulo commutativity for the parallel and choice operators.

3.3 Transformation of GCSR Specifications

In addition to the verification of the correctness of a GCSR specification, another advantage of ACSR equivalence relations is their use as the basis to minimize or expand GCSR specifications. Figure 8 shows transformations (minimization or expansion) that produce equivalent specifications according to the prioritized strong equivalence of ACSR, \sim_{π} .

Eliminate process nodes. There are two transformations that allow the elimination of process nodes: one to replace a reference to a process with a cyclic edge (Figure 8, 1) and the other to eliminate an unreachable nil node (Figure 8, 2).

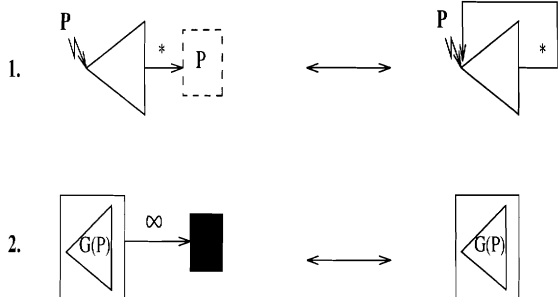
Eliminate edges. There are two transformations to eliminate edges in a GCSR process: one to remove unnecessary unlabeled edges (3) and the second to merge consecutive “identical” nodes (4).

Eliminate boxes. Extra nested compound nodes, which can be due to reference node binding, can be eliminated (5 and 6).

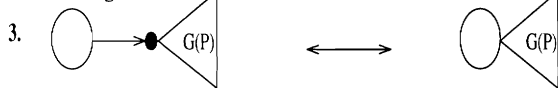
Eliminate duplicates. The last GCSR transformation merges “identical” portions in a GCSR process.

The GCSR processes **Gate** and **GD'** of Figure 3 can be obtained from the GCSR processes in Figure 5 by eliminating unlabeled edges (transformation 3). The GCSR process **GU** in Figure 3 can be obtained from the pure GCSR process in Figure 5 by eliminating the ∞ time labeled edge and its target node (transformation 2). Figure 9 (a) shows the result of eliminating the reference node **GD'** from the process **GD'** (transformation 1), and (b)

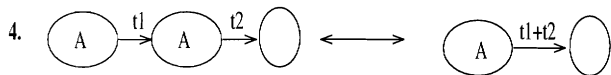
Eliminate process nodes



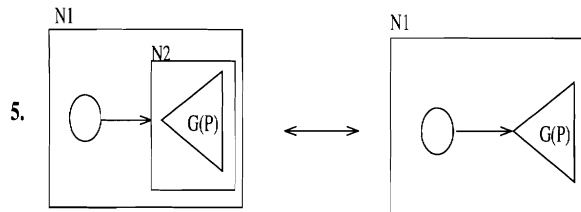
Eliminate edges



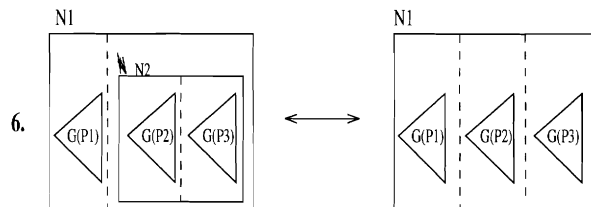
If instantaneous node has one unlabeled incoming edge



Eliminate boxes



Conditions: $N2.Restrict = N1.Restrict$
 $N2.Close = N1.Close$



Conditions: $N2.Restrict = N1.Restrict$
 $N2.Close = N1.Close = \{ \}$

Eliminate duplicates

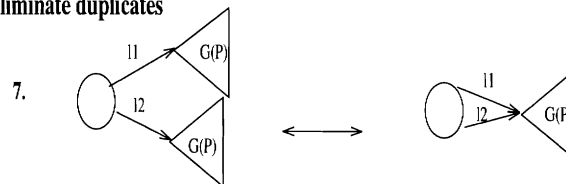


Figure 8: GCSR Transformations and Short Hand Notation

shows the result of binding the reference node labeled GD' in the process GD of Figure 3, finally (c) shows the resulting of eliminating the extra compound node which is created in (b) (transformation 5).

GCSR to Pure GCSR. It is possible to show that every valid GCSR process can be transformed to a pure GCSR process by using the above transformations. Since every pure GCSR process corresponds to an ACSR process, we therefore establish the fact that every valid GCSR process corresponds to an ACSR process. Note also that since pure GCSR is a subset of GCSR, and since every ACSR process has a corresponding pure GCSR process, this implies that ACSR is the same as valid GCSR.

4 Example: A Busy Railroad Crossing System

The system to be developed controls a gate at a railroad crossing [7]. The railroad crossing lies in a region delimited by entry and exit sensors that detect the entry and exit of trains. The goal is to develop a system that operates the crossing gate subject to the following two

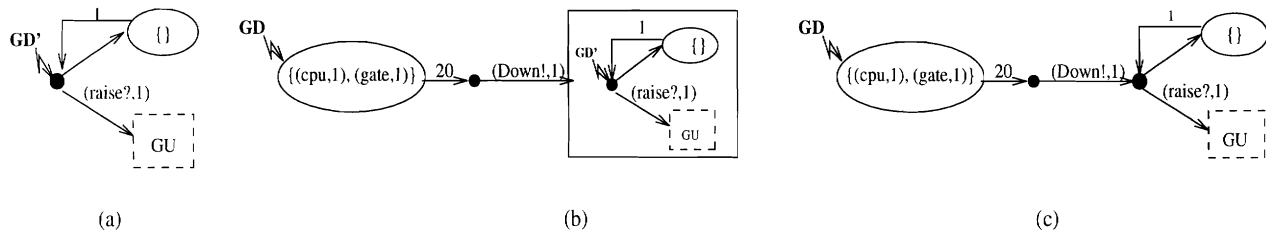


Figure 9: (a) GD' after eliminating reference node GD' ; (b) GD after resolving reference node GD' ; (c) GD after eliminating extra box.

properties:

- safety*— when a train is in the crossing, the gate is down; and
- utility*— when no train is in the crossing, the gate is up.

Figure 10 shows the GCSR specification of the railroad crossing system to which we added cars to model a *busy railroad crossing*, BRC. Our design contains three main components: **Train** and **Car** which describe the environment, and **Gate** which describes the computer system that controls the gate. The **Train** synchronizes its activities with the **Gate** by sending the sensory-event *lower* when it nears the crossing and *raise* when it leaves the crossing. In addition, the **Train** marks its critical position in the crossing region by producing the *observable* event $(Nc!, 1)$ when it is near the crossing, the event $(Ic!, 1)$ when it is in the crossing, and the event $(Pc!, 1)$ when it passes the crossing. A **Train** takes 25 time units to enter the crossing, 10 time units to pass it, and at least 15 time units to return to the crossing again.

While the **Train** synchronizes its activities with the **Gate**, the **Car** does not. This models cars that try to pass the crossing any time they can, possibly by sneaking around the gate when it is down. To detect potential collisions between a train and a car, we use the shared resource *crossing*. A collision is marked by a deadlock that is due to a train that must access the *crossing* resource at the same time as a car. To arbitrate the resource sharing, a train in the crossing has a higher priority to access the *crossing* resource than a car that arrives to the crossing.

The **Train** process states that once a train enters the crossing region, which is marked by $(Nc!, 1)$, it triggers the entry-sensor $(lower!, 1)$, and then spends 25 time units approaching the crossing while at the same time trying to capture the resource *crossing*; this is described by process **TN** which executes for 25 time units. In process **TN**, the time-consuming node with no resources can be repeatedly visited until the first time it is possible to enter the node with resources $\{(crossing, 5)\}$ at which time control loops in this node. Since the resource *crossing* is closed in **BRC**, entering the node with resources $\{(crossing, 5)\}$ is at the earliest time it is enabled. Consequently, in the case that there is no car already in the crossing (i.e., **Car** is in the idling node), the train enters this node immediately, which makes the resource *crossing* unavailable to any car that arrives later.

To verify that the detailed GCSR specification **BRC** of Figure 10 satisfies the safety and utility properties, we use the requirements specification \mathbf{Spec}_3 of Figure 11. The ACSR process \mathbf{Spec}_3 focuses on describing the desired behavior of the **BRC**. It does not show the system architecture, i.e., components and abstracts out resource usage. This specification consists mainly of sequential executions and thus is easier to inspect for the safety and

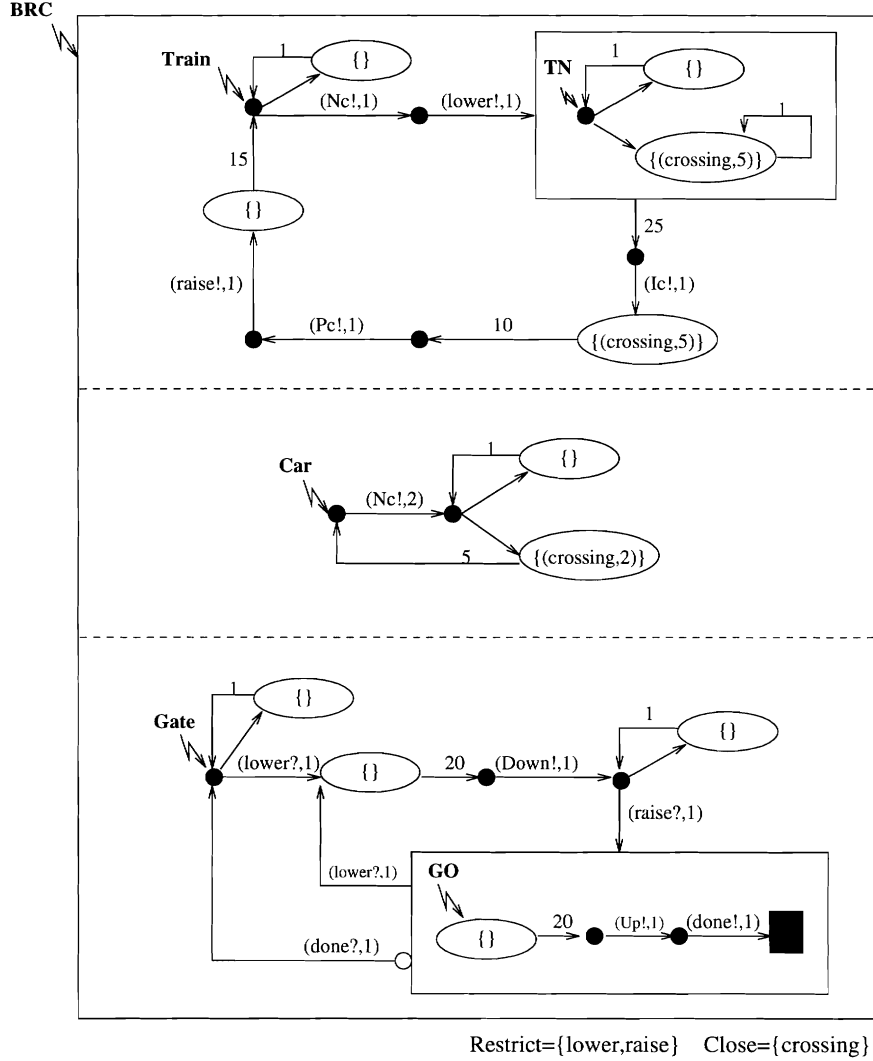


Figure 10: GCSR Specification of Busy Railroad Crossing

utility properties. We can see that in Spec_3 , the gate is down whenever there is a train in the crossing: each occurrence of $Down$ precedes an occurrence of Ic (line (4)) and each occurrence of Up is preceded by an occurrence of event Pc (from line (4) to line (7)). In addition, we can inspect in Spec_3 that the train and the car will not collide: there is no deadlock in Spec_3 due to resource contention. Furthermore, one can easily inspect that Spec_3 satisfies the utility property: if no event $(\overline{Nc}, 1)$ ever occurs, the gate remains in its initial position (line (3)), which is by assumption, up; on the other hand, if no event $(\overline{Nc}, 1)$ occurs within 15 time units of the last Pc , the event Up occurs (line (7)).

More specifically, we can use the VERSA analysis tools and our translation mechanism to prove that

$$\mathcal{T}_{GA}(\text{BRC}) \parallel \{\text{crossing}\} \approx_{\pi} \text{Spec}_3$$

where the operator $P \parallel H$ conceals the identities of the resources in H in the executions of process P and \approx_{π} is the prioritized weak equivalence in ACSR. Thus, we can conclude that our design specification BRC satisfies the safety and utility properties.

Spec_3	$\stackrel{\text{def}}{=} (\overline{Nc}, 2).((\overline{Nc}, 1).\emptyset^{20} : \text{GDown} + \emptyset : ((\overline{Nc}, 1).\emptyset^4 : (\overline{Nc}, 2).\emptyset^{16} : \text{GDown}$	(1)
	$+ \emptyset : ((\overline{Nc}, 1).\emptyset^3 : (\overline{Nc}, 2).\emptyset^{17} : \text{GDown} + \emptyset : ((\overline{Nc}, 1).\emptyset^2 : (\overline{Nc}, 2).\emptyset^{18} : \text{GDown}$	(2)
	$+ \emptyset : \text{Spec}_3)))))$	(3)
GDown	$\stackrel{\text{def}}{=} (\overline{Down}, 1).\emptyset^5 : (\overline{Ic}, 1).\emptyset^{10} : (\overline{Pc}, 1).\emptyset^5 : (\overline{Nc}, 2).\emptyset^5 : (\overline{Nc}, 2).\emptyset^5 : \text{Spec}'_3$	(4)
Spec'_3	$\stackrel{\text{def}}{=} (\overline{Nc}, 2).((\overline{Nc}, 1).\emptyset^{20} : \text{GDown} + \emptyset : ((\overline{Nc}, 1).\emptyset^4 : (\overline{Nc}, 2).\emptyset^{16} : \text{GDown}$	(5)
	$+ \emptyset : ((\overline{Nc}, 1).\emptyset^3 : (\overline{Nc}, 2).\emptyset^{17} : \text{GDown} + \emptyset : ((\overline{Nc}, 1).\emptyset^2 : (\overline{Nc}, 2).\emptyset^{18} : \text{GDown}$	(6)
	$+ \emptyset : ((\overline{Nc}, 1).\emptyset^1 : (\overline{Nc}, 2).\emptyset^{19} : \text{GDown} + \emptyset : (\overline{Up}, 1).\text{Spec}_3)))))$	(7)

Figure 11: Requirements Specification of the Busy Railroad Crossing

5 Conclusions

The pioneering work in CRSMs, Statecharts, and Modechart helped us understand graphical constructs essential in a specification language to describe the behavior of a real-time system in a modular and hierarchical fashion. The Communicating Real-time State Machine (CRSM) formalism [16] combines the graphical representation of state machines together with time and synchronous communication over unidirectional channels. CRSM's formal semantics is given as a history of traces that describe the timed occurrences of communication and internal computation. Statecharts [6] augmented the finite state machine formalism with hierarchy, and broadcasting mechanism. Communication is done via shared variables and events that are visible across all states and levels of a specification. Statecharts' formal semantics is based on timed sequences of system state changes. Modechart [9] enhanced the features of Statecharts with timing constructs such as alarms and deadlines, and limiting transition labels to avoid semantic anomalies. Its semantics is based on Real Time Logic, a first-order predicate logic [8]. The toolset MT [15] has been implemented to specify and model real-time systems using a subset of Modechart. MT can be used to specify real-time properties written in RTL, e.g., safety properties and verify that a given Modechart specification satisfies them.

The underlying formalisms of the above languages, however, assume idealistic run-time environments. In particular, they lack notions of resources and priorities which are essential to capture accurately the behavior of a real-time system. While it might be possible to encode the run-time resource requirements with functional requirements in these languages as boolean conditions, the resulting specification may be complex and hard to modify. Furthermore, these languages lack an equivalence relation that can be used to verify the correctness of a specification with respect to another.

In addition to supporting the specification of resources and priorities in an intuitive way that produces easy to understand and modify specifications, the GCSR language we presented in this paper is distinguished by its syntax and precise algebraic semantics. GCSR offers several types of nodes and edges for a modular and structured hierarchical specification. GCSR semantics, which is defined through the ACSR process algebra, makes it possible to

execute a GCSR specification, verify its correctness through various notions of behavioral equivalence relation, and minimize it or expand it to include more details.

There are several extensions for GCSR on which we are currently working. The first extension is to augment VERSA with a graphical environment where a real-time system can be modeled in GCSR. The second extension is to use GCSR as the front-end for an environment for the top-down design of real-time systems. In this environment, a real-time system is specified through an iterative process of implementation detail additions, called *refinement*. These refinements will be represented through graphical transformation operations. They will also have essential semantic properties such as preservation of deadlock freedom and timed occurrences of events between the original specification and the refined one.

References

- [1] Patrice Brémond-Grégoire. *A process Algebra of Communicating Shared Resources with Dense Time and Priorities*. PhD thesis, Department of Computer and Information Science, The University of Pennsylvania, Philadelphia, PA 19104, 1994. Tech. Report MS-CIS-94-24.
- [2] Duncan Clarke, Insup Lee, and Hong-Liang Xie. VERSA: A Tool for the Specification and Analysis of Resource-Bound Real-Time Systems. *Journal of Computer and Software Engineering*, 3(2), April 1995.
- [3] R. Cleaveland, J. Parrow, and B. Steffen. A Semantics-Based Verification Tool for Finite-State Systems. In *Proc. of Protocol Specification, Testing, and Verification, IX*, pages 287–302. Elsevier Science Publishers B.V., 1990.
- [4] T. DeMacro. *Structured Analysis and System Specification*. Prentice-Hall, Inc., Yourdon Press, Englewood Cliffs, NJ, 1987.
- [5] R. Gerber. *Communicating Shared Resources: A Model For Distributed Real-Time Systems*. PhD thesis, Department of Computer and Information Science, The University of Pennsylvania, Philadelphia, PA 19104, 1991.
- [6] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.
- [7] C. Heitmeyer, R. Jeffords, and B. Labaw. A benchmark for comparing different approaches for specifying and verifying real-time systems. In *Proc. 10th IEEE Workshop on Real-Time Operating Systems and Software*, May 1993.
- [8] F. Jahanian, R. Lee, and A. K. Mok. Semantics of modechart in real time logic. In *Proceedings of the 21st Hawaii International Conference on System Science*, January 1988.
- [9] F. Jahanian and A. K. Mok. Modechart: A specification language for real-time systems. *IEEE Transactions on Software Engineering (to appear)*, November 1989. IBM Tech Report RC 15140.

- [10] P. C. Kanellakis and S. A. Smolka. CCS Expressions, Finite State Processes, and Three Problems of Equivalence. *Information and Computation*, 86:43–68, 1990.
- [11] I. Lee, P. Brémont-Grégoire, and R. Gerber. A Process Algebraic Approach to the Specification and Analysis of Resource-Bound Real-Time Systems. *Proceedings of the IEEE*, pages 158–171, Jan 1994.
- [12] I. Lee and V. Gehlot. Language Constructs for Distributed Real-Time Programming. In *Proc. IEEE Real-Time Systems Symposium*, 1985.
- [13] R. Milner. *A Calculus for Communicating Systems*. LNCS 92, Springer-Verlag, 1980.
- [14] D. Park. Concurrency and Automata on Infinite Sequences. In *Proc. of 5th GI Conference*. LNCS 104, Springer Verlag, 1981.
- [15] Anne T. Rose, Manuel A. Pérez, and Paul C. Clements. *Modechart Toolset User's Manual*. Center for Computer High Assurance Systems Information Technology Division, Naval Research Laboratory. Washington, DC 20375-5320, NRL/MR/5540-94-7427 edition, February 1994.
- [16] A. C. Shaw. Communicating Real-Time State Machines. *IEEE Transactions on Software Engineering*, 18(9):805–816, September 1992.