Operational Semantics for Visual Simulation in PARAGON

Hanêne Ben-Abdallah
University of Waterloo
hanene@swen.uwaterloo.ca

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

Oleg Sokolsky
Computer Command and Control Company
sokolsky@cccc.com

Abstract

In the case of safety-critical systems simulation is more credible when it is derived through a mathematically sound interpretation of a specification. In this paper, we describe the foundation and tool to support visual simulation of specifications within PARAGON, a toolset for the formal specification, analysis and testing of real-time, resource-bound systems. Visual simulation executes a real-time system specification that is described in the Graphical Communicating Shared Resources language (GCSR). It is based on the operational semantics of GCSR which describes all possible execution steps of a GCSR specification in terms of the GCSR graphical entities. The rules of the operational semantics for GCSR have been implemented as a simulator within PARAGON. The GCSR simulator provides a visual animation of a GCSR model, which makes it easier to understand directly the behavior of a model as oppose to examining its execution traces.

1 Introduction

During the development of a real-time system, simulation plays an important role in testing and better understanding a system specification. It allows both designers and users to explore sample executions of the system and to revise the requirements when necessary. Currently, simulation can be provided either through prototyping languages (e.g., [10]), or direct execution of a specification. Prototyping however can increase the development cost and may create a gap between the prototype and design models. Direct execution of a specification, on the other hand, is more reliable especially when simulation is derived from a formal interpretation of the specification. In the case of safety-critical systems, e.g., aerospace applications, a mathematically sound interpretation of a specification is vital as it increases the credibility of simulation and reduces potential errors by avoiding gaps in the development process.

In this paper, we describe the foundation and tool to support visual simulation of specifications within PARAGON, a toolset for the formal specification, analysis and testing of real-time, resource-bound systems. Simulation in PARAGON complements other formal verification techniques based on state-space exploration, equivalence checking and testing. Visual simulation consists of executing a real-time system specification that is described in the Graphical Communicating Shared Resources language (GCSR), a formalism supported within PARAGON. The GCSR language allows a designer to model the functional, temporal and resource requirements of a real-time system. Its graphical syntax provides for the modular and hierarchical description of systems, and facilitates the visual structuring of a large system in terms of its components and their dependencies.

To provide for credible simulation of GCSR models, we have defined the operational semantics of GCSR in terms of the GCSR graphical entities. The operational semantics describes in a systematic way all possible execution steps of a GCSR specification. Currently, operational semantics have been successively applied to process algebraic languages. In the case of graphical and particularly hierarchical languages, defining the rules of the operational semantics presents several challenges, e.g., how to determine the execution steps at a given system state.

The rules of the operational semantics for GCSR have been implemented as a simulator within PARAGON. The GCSR simulator provides a visual animation of a GCSR model, which makes it easier to understand directly the behavior of a model as oppose to examining model's execution traces. It supports two operation modes: (1) a manual mode where the user interactively steps through a specification by selecting the next execution step from a list of steps possible at a system state; and (2) an automatic mode where the simulator repeatedly selects the next execution step either randomly, or driven by a given execution trace. In the automatic mode, the GCSR simulator also allows the user to set breakpoints where automatic simulation is interrupted to examine a particular execution of the system.

Paper Organization. The next section reviews relevant work in supporting visual simulation of formal languages for real-time systems. Section 3 introduces the GCSR language. Section 4 presents the rules of the operational semantics of GCSR. Section 5 first briefly describes the PARAGON tool; secondly, it discusses the issues pertinent to implementing the GCSR semantic rules and describes the functionalities of the GCSR simulator. The last section summarizes the paper and outlines future research topics.

2 Related Work

Most software engineering tools offer simulation. We next review a few that have specification languages close to GCSR and those that adopt an approach comparable to our simulation technique.

Modechart is a graphical language for the specification and analysis of real-time systems. Its syntax is based on concurrent, hierarchical finite-state machines similar to Statecharts [6]. Its fundamental four constructs are: 1) *modes* which are used to divide a system specification; 2) *actions* each of which is associated with a mode and which is executed when control enters the mode; 3) *events* which mark a time moment when something happens, such as a mode transition, a system variable change, and the start and stop of an action; and 4) lower and upper time bounds which label mode transitions and which set timing constraints between the entry and exit of a mode. The MT toolset [5, 13] implements a subset of Modechart with formal semantics defined in Real-Time Logic (RTL), a first-order predicate logic. The simulator in MT allows a user to examine sample behavior of a Modechart specification that is displayed as a bar graph: a graph of horizontal time lines one for each mode in the specification. A thick time line segment indicates that corresponding mode is active during that time interval. The MT simulator can also display the temporal behavior of actions, events and boolean variables [5]. The MT simulator allows the user to specify how to resolve non-determinism in a Modechart specification, schedule the random occurrence of an event, set breakpoints (made of events or modes), and fix the duration of an action and a simulation. Because of the different notations, the MT simulation results can be difficult to trace back to a simulated Modechart specification. Recently, an operational semantics for Modechart has been proposed in [12] which should allow the implementation of a visual simulator for Modechart specifications.

The ObjecTime tool [11] uses a variant of Statecharts [6], called RoomCharts, to describe an object-oriented model of a real-time system. The ObjecTime tool contains an interpreter that allows a user to simulate a RoomChart model. The semantics is essentially in the tool, as there is no formal definition of the interpreter. A simulation trace of a RoomChart is derived based on manual message injection, and tracing via the Angio tracing method [7]: a trace profiling method for concurrent executions that records the connection between recorded execution events and their stimuli, executing process, and system state among other information. A trace is represented in ObjecTime as a Message Sequence Chart (MSC) [8]: a graphical representation of message exchanges and state conditions of concurrent actors (i.e., processes) in the simulated RoomChart. Currently, ObjecTime does not support any visual connection between an MSC and a simulated RoomChart model; name matches is the main connection. While the MSC representation of a trace is visually more appealing than a textual format, one finds it hard to connect an MSC to a RoomChart model.

The visual simulation tool in the Concurrency Factory [4] uses a closer approach to the GCSR simulator. In this tool, a system is modeled in Graphical CCS (GCCS) as a hierarchy of finite-state machines communicating through named channels. GCCS specifications are thus very low-level. A high-level programming language VPL can also be used as input, but VPL specifications do not have a visual representation and thus cannot be simulated. Simulation of GCCS specifications is not directly based on the operational semantics. In addition, since GCSR is a higher-level specification language than the GCCS, simulation results are easier to interpret in PARAGON than in the Concurrency Factory.

3 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 execute on a finite set of serial resources and synchronize with one another through communication channels. The use of shared resources is represented by timed *actions*, and synchronization is supported by instantaneous *events*. The execution of an action takes nonzero time units with respect to a global clock, and consumes 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; priorities are static, i.e., fixed and are drawn from the set of natural numbers. To ensure the uniform progress of time, processes execute actions synchronously. Time can be either dense or discrete in our framework; in this paper, we consider discrete time only to simplify the description of the GCSR semantics and to reflect the current implementation.

Unlike an action, the execution of an event is instantaneous and never consumes any resources. Processes execute events asynchronously except when two processes synchronize through matching event names, i.e., channels.

The main unit of specification in GCSR is a process. Processes may be organized in a hierarchy; thus a process may have subprocesses. A GCSR process is a triple $(\mathcal{N}, \mathcal{I}, \mathcal{E})$ where $\mathcal{N}$ is a finite set of nodes; $\mathcal{I} \subset \mathcal{N}$ is a set of *initial* nodes with a distinctive initial node $n_0 \in \mathcal{I}$ which is the unique initial node at the highest level, i.e., it is not inside any node; and $\mathcal{E}$ is a set of edges. Each node has a type and based on its type can be associated with several attributes, such as a process name and a set of event names. Each edge also has a type and based on its type can have a label such as an event and a time constant. Node and edge types and attributes are discussed below.

3.1 GCSR Nodes and Edges

Figure 1 shows the graphical symbols for the various types of GCSR nodes. The *Resource* attribute of a time-consuming node is a set of (resource, priority) pairs, with the restriction that each resource is listed once; this enforces the notion of serial resource usage. The *Name* attribute of a reference node refers to the name of a GCSR process. The *Restrict* and *Close* attributes of a compound node are sets of event names and resource names, respectively. Note that there are three distinct kinds of names used in GCSR: event names, resource names, and process names. We let $\mathcal{L}$ denote the domain of event names, $\mathcal{R}$ denote the domain of resource names, and $\boldsymbol{N}$ denote the set of natural numbers.

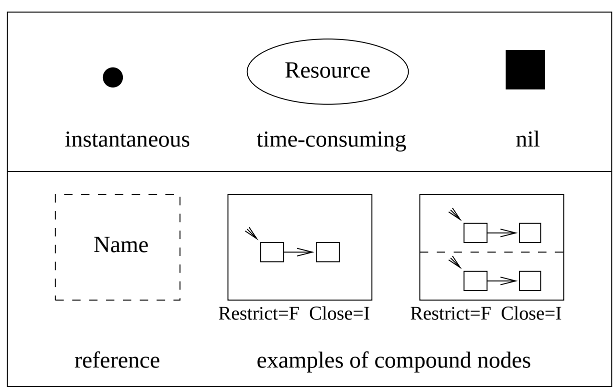


Figure 1: GCSR nodes

Intuitively, an instantaneous node requires that no delay be allowed before the next activity. In contrast, a time-consuming node describes a time consuming activity. The Resource attribute of a time-consuming node specifies the required resources for the system activity.

A nil node describes a halting process, i.e., end of system execution. A reference node allows decomposition of a large specification into subspecifications, which eases visual structuring of such a specification. Compound nodes are essential in supporting scalable and modular specifications since they allow a designer to group GCSR processes into a higher level entity by composing them sequentially and in parallel. In addition to structural modularity, compound nodes also enhance semantic modularity by encapsulating dependencies through their Restrict and Close attributes: The Restrict attribute identifies a set of events that are visible only among the GCSR processes inside the node; the Close attribute identifies a set of resources that are reserved for the nested GCSR processes.

GCSR nodes can be connected with edges to describe sequential execution. GCSR offers four types of edges shown in Figure 2. An edge can be labeled by a prioritized event or an integer representing a time value. An event is constructed by adding either a *send* ("?") or *receive* symbol ("!") to an event name. A distinguished event name τ, which cannot have either send or receive designation, is reserved for internal synchronization of a process. We call unlabeled, event-labeled, and time-labeled edges as *normal* edges. The distinct symbols for a normal edge and an exception edge are motivated by the desire to support a structured, hierarchical specification in which edges do not cross node boundaries and 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. The first type of control flow is described by a normal edge and the second by an exception edge. Control moves to the destination node of an exception edge when the process of 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 and thus emulates a transition between nodes at different levels of nesting.

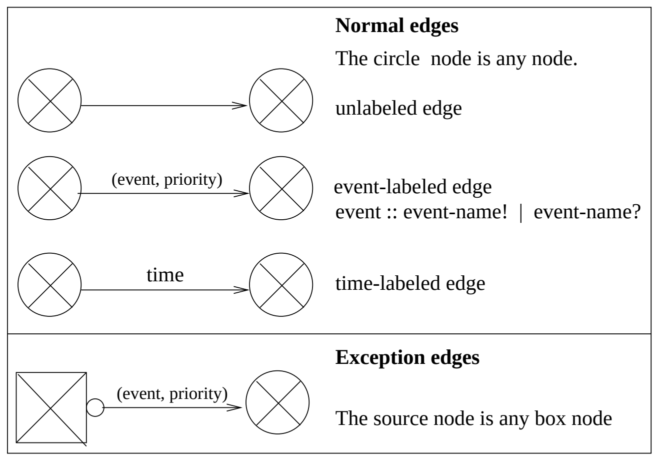


Figure 2: GCSR edges

To give a meaningful semantics to a GCSR process, we require it to be *well-formed*. That is, its sets of nodes and edges must satisfy the following rules:

1. An instantaneous node must have at least one out-edge and all of its out-edges are only unlabeled or event labeled.

2. A time-consuming node must have one outgoing time-labeled edge and no other out-edges.

3. The nil node does not have any out-edges.
4. An edge connects nodes that belong to the same GCSR process and at the same nesting level.
5. A reference and a compound node can have at most one time-labeled edge.

3.2 Informal Semantics

We use a simple example of a client-server system to illustrate the semantics of GCSR processes. In GCSR, as Figure 3 shows, the system is described at the top-level as two concurrent GCSR processes, `Client` and `Server`.

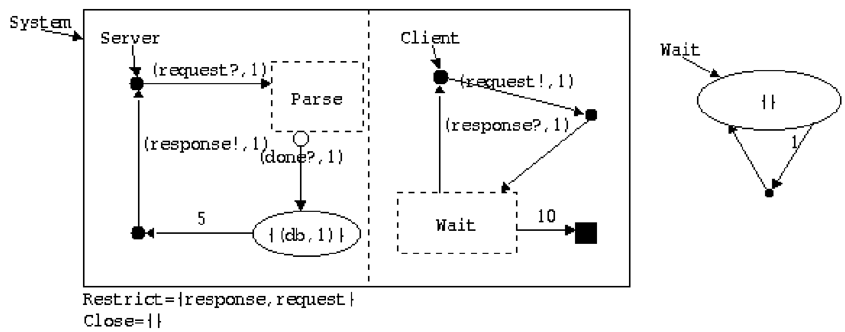


Figure 3: Client-Server Specification in GCSR

Let us concentrate on the `Server` process first. The process `Server` starts its execution with control at the instantaneous node marked with its name. At this stage, the server must instantenously *receive* the event named *request* at priority level 1, in which case control immediately transfers to the reference node and the server starts behaving as discribed by the process `Parse` responsible for parsing requests (not shown here). When parsing is completed, process `Parse` can voluntarily relinquish control by *sending* the event *done*. This event is caught by the exception edge of the reference node, and control is transferred to the time-consuming node, where the `Server` process is accessing the database (represented as the *db* resource) at priority 1. Note that there may be other servers accessing the database at the same time, possibly at higher priorities, in which case execution of this server would be preempted. The `Server` process accesses the database for 5 time units, after which it sends the *response* event to the client, and returns to the initial state.

On the client side, the `Client` process first sends the *request* to the server and starts idling as described by process `Wait` shown next to the main process. The `Wait` process repeatedly consumes one time unit without consuming any resources. In the `Client` process, idling the execution of the `Wait` process can be aborted in two ways: one way is when the *response* event is received from the server, at which time control returns to the initial node of `Client` via the edge labeled by the receive event. The second way of aborting th execution of the `Wait` process is after 10 time units of idling, at which time control moves to the Nil node and the `Client` process deadlocks.

Note that within the `System` process, time can progress only when both processes can consume it simultaneously; this ensures the synchroneous passage of time. In particular, if process `Client` enters the Nil node, the whole system will deadlock. In addition, the two processes `Server` and `Client` must synchronize their activities locally through the event named *request* and *response*; that is, these events are only visible inside the compound node of the `System` process and cannot be used to communicate with any other process that may execute in parallel.

4 The Operational Semantics of GCSR

A GCSR process describes the operational view of a system in terms of its resource requirements and communication events under a set of temporal constraints. The overall behavior of a GCSR process can be formally described as a *labeled transition system* where each transition represents an execution step with the label being a communication event or a time and resource consuming action. As seen in the above examples, there might be several execution steps that are simultaneously possible. The selection among such execution steps is done via a notion of priority defined as a *preemption* relation [9, 3].

The execution steps in a GCSR process can be derived either directly in terms of GCSR entities or indirectly via a translation to the Algebra of Communicating Shared Resources (ACSR) [9]. The direct derivation of the execution steps guides the visual simulation of a GCSR specification, which provides intuitive understanding on the flow of execution through the sequencing of active states and transitions between them. On the other hand, the indirect derivation of the execution steps via ACSR augments GCSR with process algebraic analysis techniques, e.g., equivalence checking which current graphical languages lack. As we show in [1], both techniques of deriving the execution steps are consistent in the sense that the corresponding ACSR process has an equivalent labeled transition system to the one generated directly from the GCSR process. Due to space limitation, in this paper we focus on the direct interpretation of a GCSR process in terms of nodes and edges.

Definition 4.1 A *labeled transition system* is a four-tuple $(S, \Sigma, s_0, \rightarrow)$, where S is the set of states, Σ is the alphabet, $s_0 \in S$ is the start state, and $\rightarrow \subseteq S \times \Sigma \times S$ is the transition relation. We write $s \xrightarrow{\alpha} s'$ for a transition $(s, \alpha, s') \in \rightarrow$. □

The main difficulties in defining the states and transitions of the labeled transition system of a GCSR process stem from the time-labeled and the unlabeled edges. Time-labeled edges require a notion of "local clock" that keeps track of how long control can stay inside the source node of the edge. The local clock must be decreased with each "tick" of the system global clock and control must leave the source node when the local clock reaches zero. In addition, to ensure uniform passage of time in the system, all local clocks must be decreased at the same time. Since nodes do not have the information about how long control is allowed to remain in them, we had to consider GCSR edges that are extended with local clocks as part of the *state* definition. That is, a state in the labeled transition system is defined as a pair of a set of nodes together with a set of *relevant* edges. The nodes of a state can simultaneously have control. Each relevant edge has its source node in the state nodes and has a local clock associated with it to determine how long control is allowed to stay in the source node.

To illustrate the difficulties presented by unlabeled edges, consider the GCSR processes of Figure 4. The process $G1$ differs from $G2$ by the additional unlabeled edges. Intuitively both processes must have equivalent behavior, i.e., the same execution steps. Assume we treat unlabeled edges as "silent" transitions and allow them to be taken arbitrarily. That is, for process $G1$ control can proceed along the unlabeled edges independently in the two nested processes. For example, control can first move along the unlabeled edge from $s1$ to $s2$ while it remains in $n1$ for the other nested process; in a second step, control moves along the unlabeled edge from $n1$ to $n4$. Since both events a and b are restricted, i.e., the send and receive for either event must be executed at the same time, the process $G1$ does not have any execution step from the reached configuration; that is, $G1$ deadlocks. Process $G2$, on the other hand, does not deadlock. Thus to retain the intended semantics we cannot allow unlabeled edges to be executed independently.

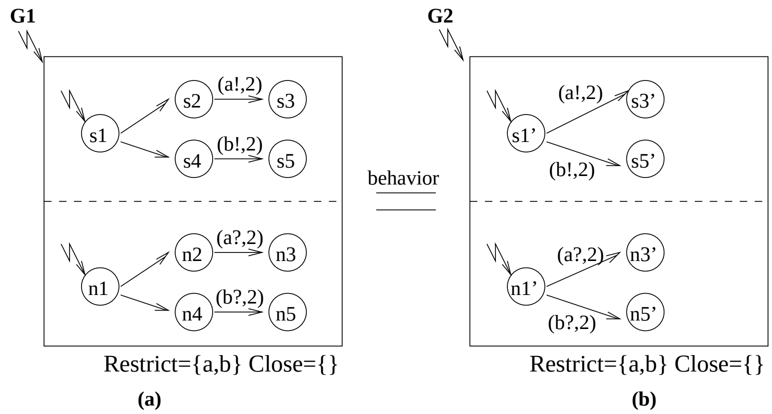


Figure 4: G1 and G2 should have the same behavior

To handle GCSR unlabeled edges, we define the transitions of the labeled transition system for a GCSR process in two steps: First, we define the semantics of GCSR without notions of parallel composition nor synchronization. At this level, we basically "prune out" the unlabeled edges and determine the sequential behavior of a process. We then define the (actual) GCSR semantics on top of the first labeled transition system, taking into account parallel execution and synchronization. The two labeled transition systems differ in their transition relation only.

Notation. In the following sections, we define the labeled transition system for a given GCSR process $G = (\mathcal{N}, \mathcal{I}, \mathcal{E})$. For an edge $e \in \mathcal{E}$, we use $\psi_{\mathcal{E}}(e)$ to retrieve its type. For a node $n \in \mathcal{N}$, we use $\psi_{\mathcal{N}}(n)$ to retrieve its type, $children(n)$ to get the set of nodes immediately contained inside n[1], $action(n)$ returns the Resource attribute of a time-consuming n, $resrc(n)$ returns the set of resource names in $action(n)$, $restrict(n)$ returns the set of event names in the Restrict attribute of a compound n, and $close(n)$ returns the set of resource names in the Close attribute of a compound n. Further, we denote the set of natural numbers as $\boldsymbol{N}$.

[1]We denote the transitive closure of the function $children$ by $children^*$.

4.1 States

Informally, a state in the labeled transition system contains all GCSR nodes that simultaneously have control, together with next possible activities that can be produced at some or all of these nodes. There are three possible types of activities: (1) take an unlabeled edge to move to another node, (2) take an event-labeled edge and produce an event, or (3) consume time and resources for one time unit. We next introduce auxiliary definitions that we will use to define a state.

Parallel nodes: Two nodes n_1 and n_2 are *parallel*, $n_1 \| n_2$, if they belong to two distinct components inside a common ancestor compound node.

Consistent nodes: A set of nodes is *consistent* if any two nodes in the set are either related through $children$, or are parallel.

Configuration: For a given consistent set of nodes $N \subseteq \mathcal{N}$, the *configuration* of N, $conf(N) = q$, is a set of nodes constructed according to the following three rules: 1) $N \subseteq q$; 2) if a node has a child in q, then it is also in q; and 3) if a compound node n is in q and any of its components does not have a node in N, then the designated initial node of the component is added to q.

It can be seen that the configuration is the maximal consistent set containing N. It can therefore be used to describe the set of nodes that simultaneously have control. To represent a state adequately, we have to extend the configuration with information about how long control can stay in any particular node. We therefore add to a configuration the set of edges whose source nodes are in a configuration, and extend each of these edges with a local clock.

Extended edges: An *extended* edge is a tuple (e, c), where $e \in \mathcal{E}$ and c is an integer not exceeding the label of the edge when e is time-labeled and 0 otherwise. In an *initially extended* edge, c is equal to the value of the label. We use $\mathcal{E}^+$ to denote the set of extended edges generated by the edges in $\mathcal{E}$ and $extend(\mathcal{E})$ to denote the set of initially extended edges generated by the edges in $\mathcal{E}$.

Relevant edges: An extended edge is *relevant* to a set of nodes q if its source node is in q. The set $relevant(q)$ denotes all extended edges relevant to q.

We now have the constituents of a state.

State: The set of *states* of the labeled transition system for G is

$$S = \{(q, E) \in (2^{\mathcal{N}} \times 2^{\mathcal{E}^+}) |\, conf(q) = q \;\wedge\; E \subseteq relevant(q)\}$$

Start state: The *start state* of the labeled transition system of G is

$$s_0 = (conf(\{n_0\}),\ relevant(conf(\{n_0\})) \cap extend(\mathcal{E}))$$

where n_0 is the distinguished initial node of G. We take the intersection of the relevant edges and the initially extended edges to ensure that the initial state s_0 is a state where control has just entered the set of nodes and no time has elapsed yet.

4.2 Transitions

The computation model of GCSR is such that a GCSR process, G, can execute either an event instantaneously or an action which consumes time and resources. A transition in the labeled transition system of G therefore represents either event-labeled edges that are taken in G, or resource consumption inside time-consuming nodes. Transitions are labeled with elements from the set

$$\begin{aligned}\Sigma \subseteq\ & \{(a?, p) \mid a \in \mathcal{L} \wedge p \in \boldsymbol{N}\} \\ & \cup \{(a!, p) \mid a \in \mathcal{L} \wedge p \in \boldsymbol{N}\} \\ & \cup \{(\tau, p) \mid p \in \boldsymbol{N}\} \\ & \cup\ 2^{\{(r,p) \mid r \in \mathcal{R} \wedge p \in \mathbf{N}\}}\end{aligned}$$

The transition relations of the labeled transition systems at both levels will make use of two auxiliary functions: *next* and *new*. Each function takes a state (q, E) and a relevant extended edge e. The function *next* returns the next state either when the edge e is taken, or after one time unit elapses in the state (q, E). The function *new* returns the set of nodes which control enters either when the edge e is taken, or after one time unit elapses in the state (q, E). The formal definitions of these functions can be found in [1].

We note that when the extended edge e is time-labeled, the function *next* decreases by one time unit the local clocks of the edge e and those edges whose source nodes are ancestors of e. (Any time-labeled edge out of a parallel node will not be decremented at this level; it will be decremented at the second level.) Any edge whose local clock is 1 is taken and, thus, is removed from the set of relevant edges. The source node and its children are removed from the configuration. The added nodes correspond to the configuration of the target node of a taken edge that is at the highest nesting level. The set of nodes *new* contains all added nodes as well as those nodes which remained in q but whose local clocks are decremented.

Notation. To distinguish between the labeled transition systems at the two levels, we denote the transition relation at the first level as $\stackrel{\alpha}{\longmapsto}$ and at the second level as $\xrightarrow{\alpha}$.

4.2.1 Transitions at the First Level

Transitions at the first level prune out unlabeled edges to produce sequential behavior of the GCSR process. More specifically, a transition at the first level is the result of taking unlabeled edges that will lead to a node where a visible (event or resource consumption) activity can be produced. Transitions at this level can be either instantaneous or time-consuming. They are defined by the following two rules.

Instantaneous transition. An instantaneous transition is the result of taking unlabeled edges that lead to a node where an event-labeled edge can be taken. It is defined by the rule **PreActI**.

$$\textbf{PreActI} \quad \frac{\begin{array}{c}((s_0,\epsilon,s_1),0)\in E_0,\ ((s'_1,\epsilon,s_2),0)\in E_1,\cdots,\\ ((s'_{k-1},\epsilon,s_k),0)\in E_{k-1},\ ((s'_k,(a*,p),s_{k+1}),0)\in E_k\end{array}}{(q_0,E_0)\overset{(a*,p)}{\longmapsto}(q',E')} \quad cond(PreActI)$$

where $a*$ denotes τ, $a!$, or $a?$ for $a \in \mathcal{L}$ and

$$\begin{array}{rl} cond(PreActI) = & \bigwedge_{i=1,k} s'_i \in children^*(s_i) \\ & \wedge\ \exists q_1,\cdots,q_k \in \mathcal{N}.\ \bigwedge_{i=1,k}(q_i,E_i) = next((q_{i-1},E_{i-1}),((s'_i,\epsilon,s_{i+1}),0)) \\ & \wedge\ (q',E') = next((q_k,E_k),((s'_k,(a*,p),s_{k+1}),0)) \end{array}$$

The condition $cond(PreActI)$ ensures the following: 1) the event-labeled edge is produced at a node that is in the same component as the taken unlabeled edges. That is, the taken unlabeled edges sequentially lead to the node where the event is executed; and 2) the set of extended edges in the predicate are the result of taking the unlabeled edges according to the *next* function. As an example of a transition that violates the first part of condition $cond(PreActI)$, consider the GCSR process $G1$ in Figure 4 (a): First take the unlabeled edge from $s1$ to $s2$, second the unlabeled edge from $n1$ to $n4$, and then the $(a!,2)$-edge to conclude that

$$\begin{array}{l}(\{n,s1,n1\},\{((s1,\epsilon,s2),0),((s1,\epsilon,s4),0),((n1,\epsilon,n2),0),((n1,\epsilon,n4),0)\}) \\ \overset{(a!,2)}{\longmapsto}(\{n,s3,n4\},\{((s4,(b?,2),n5),0)\}).\end{array}$$

In this case, after taking the second unlabeled edge, the event $(a!,2)$ does not come from a sequentially connected node but rather comes from a parallel component. A legal transition requires taking the edge labeled with $(a!,2)$ after taking the first unlabeled edge; that is, the following transition is legal:

$$\begin{array}{l}(\{n,s1,n1\},\{((s1,\epsilon,s2),0),((s1,\epsilon,s4),0),((n1,\epsilon,n2),0),((n1,\epsilon,n4),0)\}) \\ \overset{(a!,2)}{\longmapsto}(\{n,s3,n1\},\{((n1,\epsilon,n4),0),\ ((n1,\epsilon,n2),0)\}).\end{array}$$

To motivate the need for this restriction, consider the GCSR processes of Figure 4. In the GCSR process $G1$ of Figure 4 (a), the events a and b are restricted and thus both components inside the compound node must synchronize on them as we will see in the next section. Allowing the first transition, therefore, creates a deadlock in $G1$. This makes it behave differently from the GCSR process $G2$ of Figure 4 (b) which does not have the unlabeled edges and does not deadlock.

Time-consuming transition. A time-consuming transition at the first level is the result of taking unlabeled edges that lead to a time-consuming node where time and resources can be consumed.

$$\textbf{PreActT}\ \ \frac{((s_0,\epsilon,s_1),0)\in E_0,\ ((s_1',\epsilon,s_2),0)\in E_1,\cdots,\\ ((s_{k-1}',\epsilon,s_k),0)\in E_{k-1},\ ((s_k',t,s_{k+1}),c)\in E_{k-1}}{(q_0,E_0)\xmapsto{A}(q',E')}\quad cond(PreActT)$$

where

$$\begin{aligned} cond(PreActT) = {} & \textstyle\bigwedge_{i=1,k} s_i' \in children^*(s_i) \\ & \wedge\ \psi_{\mathcal{N}}(s_k') = \text{time-consuming}\ \wedge\ A = action(s_k') \\ & \wedge\ \exists q_1,\cdots,q_{k-1}\in\mathcal{N}.\ (\ \textstyle\bigwedge_{i=1,k-1}(q_i,E_i) = next((q_{i-1},E_{i-1}),((s_i',\epsilon,s_{i+1}),0))) \\ & \wedge\ (q',E') = next((q_{k-1},E_{k-1}),((s_k',t,s_{k+1}),c)) \end{aligned}$$

The condition $cond(PreActT)$ is similar to $cond(PreActI)$, in that it ensures that the taken unlabeled edges sequentially lead to the time-consuming node where the resources are consumed for one time unit.

Definition 4.2 For the transition rule **PreActI**, we say that the edge $((s_k',(a*,p),s_{k+1}),0)$ *initiates* the transition $(q_0,E_0)\xmapsto{(a*,p)}(q',E')$. Similarly, for the transition rule **PreActT**, we say that the edge $((s_k',t,s_{k+1}),c)$ *initiates* $(q_0,E_0)\xmapsto{A}(q',E')$.
To be brief, when the states (q_0,E_0) and (q',E') are clear from their context, we simply write $((s_k',(a*,p),s_{k+1}),0)$ initiates $\xmapsto{(a*,p)}$, and $((s_k',t,s_{k+1}),c)$ initiates $\xmapsto{A}$. □

4.2.2 Transitions at the Second Level

Given the transitions at the first-level, which filter out the unlabeled edges, the labeled transitions at the second level define the GCSR semantics, including parallel composition and synchronization.

In the sequel, we adopt the following notation: $\overline{a*}$ denotes the inverse of event label a, e.g., $\overline{a!} = a?$. We also overload the syntax of the function new and use $new(q)$ for $new((q,E),((s,\alpha,d),c))$ when (q,E) and $((s,\alpha,d),c)$ are clear from their context.

Instantaneous transition. An instantaneous transition is labeled with an event. It is the result of three possible ways of taking relevant edges: 1) take an event-labeled edge that is labeled with an unrestricted event; 2) take two event-labeled edges with inverse event labels; or 3) take an event-labeled edge and an exception edge with inverse event labels. These transitions are described by the following three rules.

$$\textbf{ActI}\ \ \frac{(q,E)\xmapsto{(a*,p)}(q',E')}{(q,E)\xrightarrow{(a*,p)}(q',E')}\quad cond(ActI)$$

where

$$
\begin{aligned}
cond(ActI) = (s,(a*,p),d),0) \text{ initiates } & \overset{(a*,p)}{\longmapsto} \\
\Longrightarrow \quad & \\
\psi_{\mathcal{E}}((s,(a*,p),d)) = & \text{ event-labeled} \wedge \\
\forall n \in \mathcal{N}.(\quad & s \in children^*(n) \Longrightarrow \\
& a \notin restrict(n) \wedge \\
& \forall (n,(\overline{a*},p'),n'),0) \in \mathcal{E}.\psi_{\mathcal{E}}((n,(\overline{a*},p'),n')) \neq \text{ exception edge })
\end{aligned}
$$

Condition $cond(ActI)$ states that the event-labeled edge which initiates the first-level transition has a *visible* event a; that is, the event a is not restricted in any ancestor node of s and does not synchronize with an exception edge from any ancestor node of s. The second case is captured by rule **Exception** defined shortly.

The second type of instantaneous transitions is when events are synchronized. This is possible when there are two first-level, instantaneous transitions which can be executed in parallel and which are labeled with inverse events, i.e., $a!$ and $a?$. There are two possible ways to synchronize events: simultaneously taking two event-labeled edges, and simultaneously taking an event-labeled edge and an exception edge with an inverse event label. The two cases are described by the **ParCom** and **Except** rules, respectively.

$$
\textbf{ParCom} \quad \frac{(q,E) \overset{(a!,p)}{\longmapsto} (q_1,E_1), (q,E) \overset{(a?,p')}{\longmapsto} (q_2,E_2)}{(q,E) \xrightarrow{(\tau,p+p')} (q',E')} \quad cond(ParCom)
$$

where

$$
\begin{aligned}
q' &= conf(\,(q_1 \cap q_2) \cup (new(q_1) \cup new(q_2))\,) \\
E' &= \{((n,l,n'),c) \in (E_1 \cup E_2) |\, n \in q'\} \\
cond(ParCom) &= (\,((s_1,(a!,p),d_1),0) \text{ initiates } \overset{(a!,p)}{\longmapsto} \\
&\quad\ \wedge ((s_2,(a?,p'),d_2),0) \text{ initiates } \overset{(a?,p')}{\longmapsto}\ ) \\
&\quad \Longrightarrow \\
&\quad (\psi_{\mathcal{E}}((s_1,(a!,p),d_1)) = \psi_{\mathcal{E}}((s_2,(a?,p'),d_2)) = \text{event-labeled} \\
&\quad\ \wedge\ s_1 \| s_2 \\
&\quad\ \wedge\ \forall n.(\ \ s_1 \in children^*(n) \wedge s_2 \notin children^*(n) \\
&\qquad\quad \Longrightarrow \\
&\qquad\quad a \notin restrict(n)\ \wedge \\
&\qquad\quad \forall ((n,(a!,p''),n'),0) \in \mathcal{E}.\psi_{\mathcal{E}}((n,(a!,p''),n')) \neq \text{exception}) \\
&\quad\ \wedge\ \forall n.(\ \ s_1 \notin children^*(n) \wedge s_2 \in children^*(n) \\
&\qquad\quad \Longrightarrow \\
&\qquad\quad a \notin restrict(n)\ \wedge \\
&\qquad\quad \forall ((n,(a?,p''),n'),0) \in \mathcal{E}.\psi_{\mathcal{E}}((n,(a?,p''),n')) \neq \text{exception}))
\end{aligned}
$$

To understand the new configuration q', we note the following: q_1 and q_2 share compound nodes that contain 1) either s_1 or s_2 or both, and 2) nodes that are parallel to s_1 and s_2

but do not participate in the transitions. Furthermore, q_1 may contain nodes with unlabeled out-edges that will lead to s_2; a similar comment applies to q_2 and node s_1. The latter nodes will not be in $q_1 \cap q_2$ and should not be in the final configuration. The new nodes in q_1 and q_2 however must be in the new configuration since control just moved there. Finally, to construct q', we take the configuration of the above sets of nodes to ensure that we include those nodes which are parallel to s_1 and s_2 but did not participate in the transitions at the first level.

Condition $cond(ParCom)$ ensures that 1) both edges are event-labeled edges, 2) they are parallel edges, and 3) their events can propagate from nested nodes up to a common ancestor node where they can synchronize; that is, the events are not restricted in an ancestor node that contains one edge but not the second, and they do not synchronize with an exception edge at such an ancestor node. Synchronization with an exception edge is handled by the following rule.

$$\textbf{Except} \quad \frac{(q,E) \xmapsto{(a*,p)} (q_1,E_1), (q,E) \xmapsto{(\overline{a*},p')} (q',E')}{(q,E) \xrightarrow{(\tau,p+p')} (q',E')} \quad cond(Except)$$

where

$$\begin{aligned}
Cond(Except) = {} & (((s,(a*,p),d),0) \text{ initiates } \xmapsto{(a*,p)} \wedge ((s',(a*,p'),d'),0) \text{ initiates } \xmapsto{(\overline{a*},p')}) \\
& \Longrightarrow \\
& (\psi_{\mathcal{E}}((s,(a*,p),d)) = \text{event-labeled} \wedge \psi_{\mathcal{E}}((s',(\overline{a*},p'),d')) = \text{exception} \\
& \wedge\, s \in children^*(s') \\
& \wedge\, \forall((n,(\overline{a*},p'),n'),0) \in \mathcal{E}.(s \in children^*(n) \wedge n \in children^*(s') \Longrightarrow n = s') \\
& \wedge \forall n \in \mathcal{N}.(s \in children^*(n) \wedge n \in children^*(s') \Longrightarrow a \notin restrict(n))\ )
\end{aligned}$$

Condition $Cond(Except)$ ensures the following: 1) only one of the initiating edges is an exception edge; 2) the source of the exception edge, s', contains the event-labeled edge; 3) s' is the first ancestor with the exception edge; and 4) the event is not restricted before reaching the node s'.

Time-consuming transition. The last type of labeled transitions is labeled with an action that represents the consumption of resources for one time unit.

$$\textbf{ParT} \quad \frac{(q,E) \xmapsto{A_1} (q_1,E_1), \cdots, (q,E) \xmapsto{A_k} (q_k,E_k)}{(q,E) \xrightarrow{A} (q',E')} \quad cond(ParT)$$

where

$$
\begin{array}{rcl}
q' & = & conf(\bigcap_{i=1,k} q_i \cup \bigcup_{i=1,k} new(q_i)) \\
E' & = & \bigcup_{i=1,k}\{((n,l,n'),c) \in E_i \mid n \in new(q_i)\} \\
A & = & \bigcup_{i=1,k} A_i \\
cond(ParT) & = & \forall i,j = 1,\cdots k.(i \neq j \Longrightarrow resrc(A_i) \cap resrc(A_j) = \emptyset) \\
 & & \wedge\ \forall (q,E) \overset{A'}{\longmapsto} (q'',E'').\ (\ ((s,t,d),c) \text{ initiates } \overset{A'}{\longmapsto} \\
 & & \quad \Longrightarrow \exists((s_i,t_i,d_i),c_i) \text{ initiates } \overset{A_i}{\longmapsto}.\,(s = s_i \wedge A' = A_i)\ \vee\ s\ \not\| s_i\ ) \\
 & & \wedge\ (N = \bigcup_{i=1,k}\{n \mid ((s_i,t_i,d_i),c_i) \text{ initiates } \overset{A_i}{\longmapsto} \wedge s_i \in children^*(n)\},\ \wedge \\
 & & \quad conf(N) = N\ )
\end{array}
$$

The new configuration q' is constructed in a similar way to the parallel, instantaneous transition rule **ParCom**. To construct the new set of relevant edges, E', we can not take the union of the relevant edges E_i for $i = 1, \cdots, k$ because the *next* function does not reduce the local clocks of nodes that are parallel to the source node of the time-labeled edge being considered. Thus, the sets of relevant edges E_i contain time-labeled edges of parallel nodes whose local clocks were not decreased. To make sure E' reflects synchronous time passage, it must select those edges in E_i which reflect time passage. These edges have their source nodes selected by the function *new* and must be in the set E'.

Condition $Cond(ParT)$ ensures that a time consuming transition is allowed under three restrictions: 1) the participating time-consuming nodes have disjoint sets of resources. This restriction stems from the assumption that all resources are serial, and thus at any time at most one action can execute on a particular resource; 2) the set of transitions at the first level is maximal, i.e., any other time-consuming transition at the first level is from a node that is not parallel to another participating node; and 3) the configuration from which the time consuming step is executed contains only the time-consuming nodes which initiate the transitions at the first level and their ancestor compound nodes. This ensures that the time-consuming nodes are parallel and that the time-consuming step occurs only if all parallel components can participate– and thus time progresses synchronously.

To deal with the Close attribute of compound nodes in a GCSR process, the condition of rule **ParT** must be changed to construct the set A incrementally from the lowest level of nesting up; refer to [1] for details. To deal with the reference nodes, whenever a reference node n is in a configuration of a state, the set of relevant edges of the corresponding state is computed by syntactically substituting the reference node with a new compound node n' that has one nested process, G', that corresponds to n.

4.3 Prioritized Operational Semantics

The prioritized operational semantics of GCSR is defined through the *prioritized labeled transition system*, "$\rightarrow_\pi$", which uses the preemption relation "$\prec$" of ACSR [3, 9] to refine the unconstrained labeled transition system "$\rightarrow$".

Informally, the selection among two transitions that are simultaneously possible is either nondeterministic, or is arbitrated according to the following three rules for selecting a transition labeled with α over a transition labeled with β ($\beta \prec \alpha$): 1) α and β are events with the same label and α has a higher priority; 2) α and β are actions and α uses a subset of the resources used in β at priorities no lower than in β and with at least one resource in α at a higher priority; or 3) α is a τ event (i.e., synchronization of two events) with a non-zero priority and β is a time-consuming action. (The technical reasons behind these rules are rather complicated and outside the scope of this paper [9].)

Definition 4.3 The labeled transition system "$\rightarrow_\pi$" is defined as follows: $q \xrightarrow{\alpha}_\pi q'$ if
a) $q \xrightarrow{\alpha} q'$ is an unprioritized transition, and
b) There is no unprioritized transition $q \xrightarrow{\beta} q''$ such that $\alpha \prec \beta$. □

5 Simulation of GCSR Specifications

The PARAGON[2] toolset [2] facilitates the specification and analysis of real-time systems modeled in the GCSR language. We next briefly overview PARAGON, and then focus on how PARAGON supports simulation of GCSR specifications as one of its analysis tools.

5.1 The PARAGON Toolset

The PARAGON toolset is jointly developed by the University of Pennsylvania and Computer Command and Control Company. It is a collection of tools for the specification and analysis of real-time systems modeled in GCSR and the Algebra of Communicating Shared Resources (ACSR) [9]. Figure 5 shows the overall architecture of PARAGON.

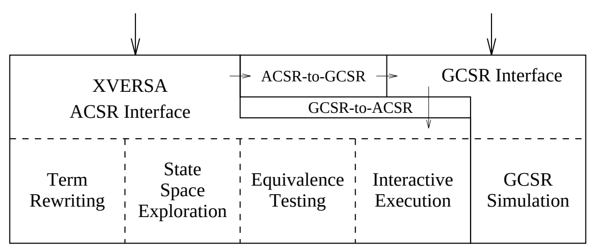


Figure 5: PARAGON Architecture

For a complete integration of the textual and graphic languages, PARAGON provides automated translation from ACSR to GCSR and vice versa. In addition, PARAGON offers two user interfaces: a GUI for GCSR specifications, and a text-based user interface for ACSR specifications. Both user interfaces provide access to analysis back-end tools that include process-algebraic analysis techniques along with the GCSR simulator.

The process-algebraic analysis techniques feature: state-space exploration which provides for deadlock detection and reachability analysis; equivalence testing for several variants of priority-based and unprioritized behavioral equivalences; textual based interactive execution of specifications; and term rewriting which allows one to construct algebraic proofs of process equivalence.

[2]PARAGON stands for Process-algebraic Analysis of Real-time Applications with Graphics-Oriented Notation.

The features of the GCSR simulator in PARAGON include: automated and interactive interpretation of system behavior; maintenance of an execution history trace; and commands for user manipulation of execution.

5.2 The Algorithm to Compute Transitions

The GCSR simulator employs an algorithm based on the operational semantics introduced in Section 4. The *current state* of a simulation is represented as a configuration. At a given current state, each of the rules of the operational semantics produces a set of transitions that are enabled. Visually, the nodes of a current state and the enabled transitions are highlighted. A direct implementation of the two-level semantics, however, may lead to highlighted GCSR entities that seem to produce a counter-intuitive behavior. Figure 6 illustrates the two main cases for the counter-intuitiveness.

The specification in Figure 6 (a) can either use the resource `gate` for three time units, or idle if the resource is not available. Thus, according to the semantics of GCSR, two timed transitions are enabled in the initial node `c1`. Visually, a direct implementation of the semantic rules would highlight the node `c1`, the unlabeled edges out of `c1`, and the two time-passing actions as part of the possible behavior at node `c1`. However, the resulting simulation state visually gives two impressions that are counter-intuitive to the GCSR semantics, yet, without contradicting it. One impression is that time is simultaneously passing in both time-consuming nodes. This impression contradicts the notion of time determinism employed by GCSR: Once the system is passing time, it is doing so in a unique way. The second counter-intuitive impression is that time is passing in the instantaneous node `c1` which as mentioned in Section 3.1 allows no time delay. Indeed, `c1` is not a state where time passes. Rather, it is a state where the choice between two possibilities to pass time is made. To clarify this situation, we introduced in the implementation a new type of instantaneous transitions at the second-level, which we call "commitment to pass time" transitions. Now, in the initial state containing the node `c1`, the simulator presents two enabled instantaneous transitions: it can commit to pass time in either of the two time-passing nodes.

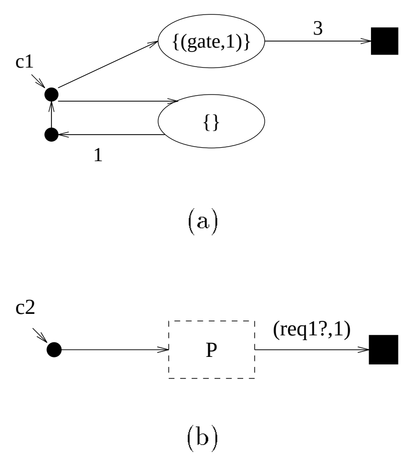


Figure 6: Making semantics intuitive.

The specification in Figure 6 (b) presents another counter-intuitive case. Since the only outgoing edge from the initial node `c2` is unlabeled, all transitions enabled in the initial node of process `P` should also be enabled at the node `c2`. In addition, the interrupt edge out of the reference node `P` should be enabled. The intuition for interrupt edges is that they are the means to leave the scope of a process, in this case process `P`. However, from the user's viewpoint, when the current simulation state is at node `c2`, simulation has not yet entered the scope of process `P`. To avoid this visual ambiguity, we added in the implementation another type of instantaneous transition at the second-level called "scope entry" transition.

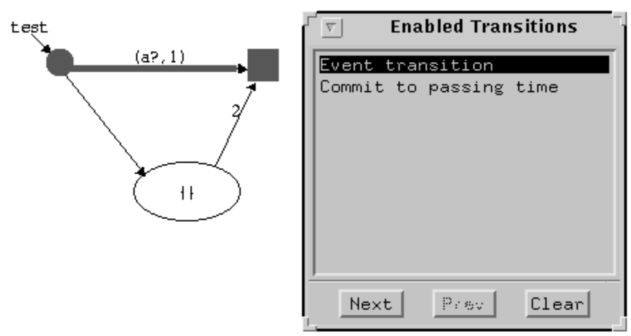
Note that the algorithm for computing enabled transitions with the additional two transitions remains faithful to the operational semantics of GCSR. Time commitment and scope entry transitions temporarily obscure some of the transitions enabled in a current simulation state. In doing so, they impose a certain structure on the set of enabled transitions, but do not modify the set itself.

5.3 Operation Modes

The GCSR simulator supports two modes of operation: *automatic* and *interactive* simulation. In either mode, the GCSR simulator receives commands from the user to guide a simulation by manipulating the simulation state and produced behavior. Commands can be entered either through buttons on the mode panel of the main simulation window or through mouse clicks as described below.

Automatic Simulation Mode. In this mode, the GCSR simulator repeatedly chooses at random the next transition to be executed from the list of enabled transitions. Automatic simulation can be started and interrupted by the user at the click of a button. Once started, an automatic simulation continues until it is interrupted by the user, a breakpoint is reached, or there are no more enabled transitions to be taken.

Interactive Simulation Mode. As an alternative to automatic execution, the user can instruct the GCSR simulator to take enabled transitions one at a time. The simulator maintains a list of enabled transitions, and displays it to the user in a special pop-up window. The user interacts with the simulator via the enabled transition window. When the user selects (via a mouse click) an item from the list of enabled transitions, the simulator highlights the corresponding edges and relevant nodes in the GCSR specification window. A second user click on the selected item deactivates the choice and unhighlights the corresponding GCSR specification entities. To instruct the simulator to execute a selected transition, the user can either click on the `Select Step` button from the enabled transition window, or double-click on the list item. Alternatively, the user may instruct the simulator to select a random transition from the list.



Execution History Manipulation. In both simulation modes, the GCSR simulator displays in a pop-up window the simulated behavior of the GCSR specification as a history trace. The history trace consists of the initial configuration for the trace followed by a list of executed transitions. The user can inspect a history trace by clicking on any of its elements. As a response, the simulator highlights the GCSR entities that correspond to the selected trace element.

Another functionality provided through the execution history window is to restart a simulation from a past state. When the user double-clicks on a transition recorded in the trace, the simulator restores the simulation state prior to the selected transition, and the transition is executed again.

State Manipulation. In addition to trace inspection and manipulation, the GCSR simulator allows the user to inquire about the current state of simulation and to enter a new simulation state. This feature allows the user to start simulation in a portion of the GCSR specification that they need to understand better.

To enter a specific simulation state, the user first selects via a mouse click the GCSR nodes that should be part of the new simulation state. The user can then activate the `Enter State` button to capture the new simulation state. The simulator completes the selected set of nodes to form a full configuration if necessary. It issues an error message if the selection is not a consistent set of nodes (see Section 4.1), e.g., when two of the selected nodes belong to the same sequential process and thus cannot be active together.

Breakpoint Handling. Another way to focus a simulation in the automatic simulation mode is via breakpoints, each of which is a consistent set of GCSR nodes. That is, a breakpoint is a possibly partial configuration– partial in the sense it may not cover all processes in the specification. The user sets a breakpoint by selecting GCSR nodes via mouse clicks. The user can set several breakpoints at a time, and the simulator manages the breakpoints in a special window in a way similar to the execution history list.

During an automatic simulation, the simulator checks after each execution step whether the current simulation state contains the nodes in a breakpoint. When a breakpoint is encountered, the simulator highlights the reached breakpoint in the list of breakpoints, highlights the GCSR entities corresponding the current state, stops the automatic simulation and switches to the interactive mode.

5.4 Simulation with Reference Nodes

Reference nodes are used in a specification to partition it into manageable modules. When simulation reaches a reference node, the referenced GCSR process is activated. Activation of a referenced process depends on where it is stored. When the referenced process is specified in the same database (i.e., it is displayed in the same window), the simulator adjusts the drawing area to focus it on the referenced process, and simulation proceeds as usual. Figure 7

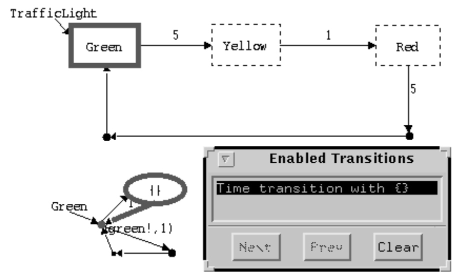


Figure 7: Simulation with a reference to a locally defined process.

illustrates an example.

When the referenced process is stored in a different database, a secondary simulator window is created where the process is loaded, and simulation proceeds concurrently in multiple windows. We note that all secondary simulator windows are slave windows in the sense they can only display and highlight GCSR objects. Simulation is driven through the main simulator window.

It is possible for several reference nodes in concurrent GCSR processes to refer to the same process. In this case, two or more of these nodes can be active at the same time. Since simulation in each of them precedes independently of others, in order to represent the state of the simulation faithfully, we need several graphical representations of the same process. When the first instance of the referenced process becomes active, it is simulated "in place", i.e., in the simulator window where it is displayed. If the second reference node becomes active while the first instance is still involved in the simulation, the process is cloned and displayed in another secondary window. To save screen space, the new simulation window is made just large enough to show the process clone. Figure 8 illustrates an example.

6 Summary and Future Outlooks

We have presented an operational semantics for the GCSR language in terms of the language's graphical entities. We then described a simulator that implements the semantic rules to animate a GCSR specification. The direct simulation of a GCSR specification allows a designer to use diagnostic traces, produced by the verification engine of PARAGON, to drive the simulation in order to debug a specification. Because simulation is currently based on the same semantics that is used for verification, one is assured of the consistent treatment of traces by the two tools in PARAGON.

An important direction of our future research and development efforts is to conduct large scale applications, and to develop efficient verification algorithms based on the operational semantics presented in this paper. Currently, verification of GCSR specifications is performed by a translation to ACSR models, and then using the verification engine. The translational semantics for GCSR has been shown to coincide with the operational semantics presented here. However, verification algorithms based directly on the operational semantics may lead

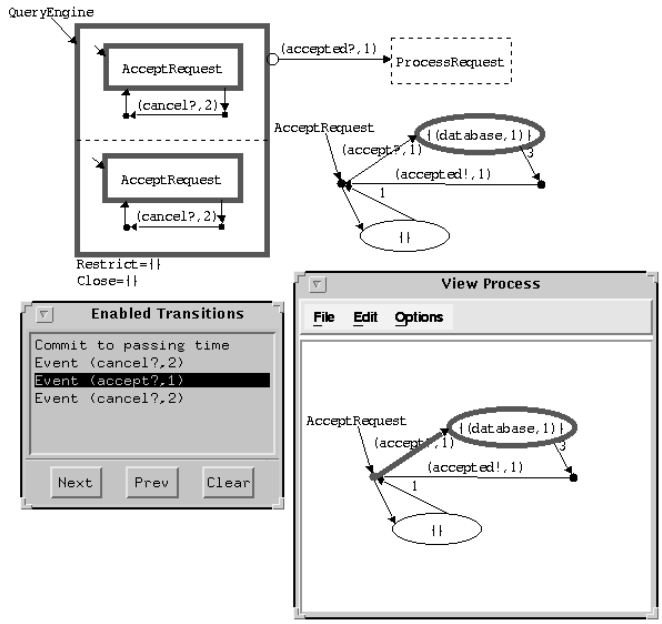


Figure 8: Simulation of concurrent references.

to significant improvement in performance.

Availability of PARAGON

The PARAGON toolset for GCSR and ACSR is implemented using the GNU project C++ compiler, the Lex/Yacc compiler, the Library of Efficient Data types and Algorithms (LEDA) class library, and the OSF X/Motif toolkit. A beta release of the toolset can be obtained by contacting Oleg Sokolsky (sokolsky@cccc.com). Documentation and related papers can be accessed from the URL page http://www.cis.upenn.edu/r̃tg/home.html.

References

[1] H. Ben-Abdallah. *GCSR: A Graphical Language for the Specification, Refinement and Analysis of Real-Time Systems.* PhD thesis, Department of Computer and Information Science, University of Pennsylvania, 1996. Tech Rep IRCS-96-18.

[2] Hanêne Ben-Abdallah, Duncan Clarke, Insup Lee, and Oleg Sokolsky. PARAGON: A Paradigm for the Specification, Verification, and Testing of Real-Time Systems. In *IEEE Aerospace Conference*, Feb 1-8 1997.

[3] Patrice Brémond-Grégoire. *Aprocess 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.

[4] R. Cleaveland, P.M. Lewis, S.A. Smolka, and O. Sokolsky. The concurrency factory: A development environment for concurrent systems. In *Proceedings of CAV'96*, LNCS 1102. Springer-Verlag, 1996.

[5] P.C. Clements, C.L. heitmeyer, B.G. Labaw, and A. T. Rose. Mt: A toolset for specifying and analyzing real-time systems. In *Proc. of IEEE Real-Time Systems Symposium*, pages 12–22, Raleigh-Durham, North Carolina, December 1-3 1993.

[6] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[7] Curtis E. Hrischuk. Formalizing Angio tracing. Technical Report SCE95-03, Real-Time and Distributed Systems Group, Carleton University, Ottawa, ON, Canada K1S 5B6, February 1995.

[8] ITU-T. Recommendation Z.120. ITU - Telecommunication Standardization Sector, Geneva, Switzerland, May 1996.

[9] 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*, pages 158–171, Jan 1994.

[10] Luqi and M. Ketabchi. A Computer-Aided Prototyping System. *IEEE Software*, March 1988.

[11] ObjecTime Ltd, 340 March Road, Kanata, ON, Canada K2K 2E4. *ObjecTime Toolset Guide*, July 1995.

[12] Carlos Puchol, Aloysius K. Mok, and Douglas A. Stuart. Compiling Modechart specifications. In *IEEE Proceedings of Real-Time Systems Symposium (RTSS' 95)*, Pisa, Italy, December 1995.

[13] 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.