PARAGON: A Paradigm for the Specification, Verification and Testing of Real-Time Systems

Hanêne Ben-Abdallah
Dept. of Electrical & Computer Engineering
University of Waterloo
Waterloo, ON N2L 3G1

Duncan Clarke
Dept. of Computer Science
University of Kentucky
Lexington, KY 40506

Insup Lee
Dept. of Computer & Information Science
University of Pennsylvania
Philadelphia, PA 19104

Oleg Sokolsky
Computer Command and Control Company
Philadelphia, PA 19103

Abstract— **The PARAGON toolset provides an environment for the modular and hierarchical design of resource-bound, real-time systems. It offers well-integrated graphical and textual specification languages with formal semantics. Both languages are based on the Algebra of Communicating Shared Resources (ACSR), a process algebra with explicit notions of time, resources and priority. The integration of the three notions widens the applicability of the PARAGON formalisms to embedded systems, control systems, and fault-tolerant systems where run-time resource requirements must be considered during the design phase. To facilitate the design of complex systems, PARAGON allows a designer to describe a system incrementally through refinement steps that preserve system properties. To increase dependentability of system models, PARAGON offers three types of analysis: automated verification of system requirements, interactive simulation, and testing. In this paper, we demonstrate the design methodology that PARAGON offers through examples.**

This research was supported in part by NSF CCR-9415346, AFOSR F49620-95-1-0508, ARO DAAH04-95-1-0092, NSF-STC-SBR-8920230 and a grant from ETRI. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Office of Scientific Research or the U.S. Government.

I. Introduction

There has been significant progress in the development of formal methods for the design of real-time systems in an effort towards increasing reliability in these systems. Formal approaches to the specification and analysis of real-time systems take many forms, including state machines, logics, and process algebras. We focus here on the algebraic paradigm, which involves the description of processes using an algebraic language and the derivation of their correctness proofs through equivalence checking, testing, and state space exploration. The algebra we use is the *Algebra of Communicating Shared Resources* (ACSR) [5].

ACSR is a timed process algebra that facilitates description of concurrent real-time systems with finite supplies of serially reusable resources. Most current real-time process algebras adequately capture delays due to process synchronization, e.g., timed extensions of the classic untimed process algebras CSP and CCS [1], [11], [16], [23], [24], [29].

These process algebras, however, abstract out resource-specific details. In contrast, the computation model of ACSR is based on the view that a real-time system consists of a set of communicating processes that compete for shared resources. The use of shared resources is modeled by timed actions whose executions are subject to the availability of resources. Contention for resources is arbitrated according to the priorities of competing actions. To ensure the uniform progression of time, processes execute timed actions synchronously. In addition to timed actions, ACSR supports instantaneous actions, called events, that do not consume any resource. Processes execute events asynchronously except when two processes synchronize through matching events. When multiple events are possible their relative importance can be prioritized.

The novelty of ACSR relative to existing real-time formalisms is its representation of resources and priority. Without an explicit notion of resources, the specification of resource-bound systems requires that some artificial means be used to model resource requirements, such as defining processes to represent resources. Models that lack explicit priorities require that a process be created for the sole purpose of arbitrating priorities and implementing preemption. Providing explicit notions of resources and priority within ACSR results in specifications that are closer analogues of the systems they model and that are easier to modify to reflect different resource allocations and scheduling disciplines.

The algebraic nature of the ACSR formalism makes it attractive in several ways. One is that the various operators allow it to support common software design methodologies such as modular and hierarchical description of a large-scale system. A second attraction is that both design (i.e. detailed) and requirements (i.e. abstract) specifications are described in the same language. Another attractive aspect is that the behavioral equivalence relation of ACSR is a congruence. This facilitates modular and hierarchical analysis of large-scale systems because it is possible to verify the whole system by reasoning about its parts. Yet another attractive aspect is that automated and semi-automated formal analysis techniques such as equivalence checking, testing, and simulation can be applied within the ACSR paradigm.

While the virtues of formal approaches to specification and analysis of software systems are well-known, it is equally well-known that 1) their textual, mathematical notation often produces obtuse descriptions, and this has impeded their application within industrial settings, and 2) their manual application to realistic problems is time consuming and error prone. As a result, many formalisms have adopted graphical notation [15], [17], [27] and are supported by automated tools [9], [10], [12], [21], [25], [26] that perform tasks such as syntax checking, semantic equivalences (e.g., CWB[9] for CCS) or pre-orders checking (e.g., FDR [12] for CSP), model checking (e.g., MT [10] for Modecharts and the TTM/RTTL verifier[25]), and interactive execution.

To avoid the mathematical notation and facilitate the use of ACSR by practitioners that are not necessarily experts in process algebras, we have developed the Graphical Communicating Shared Resources (GCSR) [4] language. The graphical syntax of GCSR

has been carefully defined so that it produces modular, hierarchical and thus scalable specifications. In addition, the semantics of GCSR and ACSR are tightly connected, which allows a designer to combine graphical and textual notations, for example to describe the high level structure of a system graphically and fill in the details textually. Furthermore, to facilitate the design within the ACSR paradigm, we have developed a toolset, called PARAGON, that assists in the graphical and textual description of real-time system models and that supports verification based on automated equivalence checking, syntactic transformations that preserve behavioral equivalence, testing and interactive execution.

The remainder of this paper is organized as follows. Section II describes the ACSR and GCSR languages. Section III describes the tools we have implemented to facilitate the use of ACSR. Section IV describes a design methodology for real-time applications in our paradigm and illustrates it through a landing gear system [28]. Finally, Section V summarizes the paper and reports on our current research within the ACSR paradigm.

II. Formalism

A. ACSR

ACSR (Algebra of Communicating Shared Resources) is a timed process algebra based on the synchronization model of CCS that includes features for representing synchronization, time, temporal scopes [20], resource requirements, and priorities. Although the time domain of ACSR can be either discrete [19] or dense [5], this paper and the GCSR/XVERSA toolset use discrete time exclusively.

The ACSR paradigm is based on the view that a real-time system consists of a set of communicating *processes* that execute on a finite set of serially reusable resources and synchronize with one another through communication channels. The use of shared resources is represented by time and resource consuming *actions*, and synchronization is supported by instantaneous *events*.

An action consists of a possibly empty set of pairs (r, p) where r is the resource name and p is its priority, with the restriction that each resource is represented in the set at most once. The action $\emptyset$ represents the idling action since no resources are used. 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 that 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 uniform progress of time, processes execute actions synchronously.

An event in ACSR consists of a pair (e, p) where e is a label and p is its priority. The special event label τ represents synchronization of two events with *complementary* event labels e and $\overline{e}$. 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 event labels.

Let P range over the domain of terms, A range over the domain of actions, e range over

the domain of event labels or τ, b range over the domain of event labels, F range over the set of event labels, I range over the set of resource names, and let C range over the domain of process names. The syntax of ACSR is defined by the following grammar:

$$\begin{array}{rcl} P & ::= & \text{NIL} \mid A^t : P \mid (e,p).P \mid P_1 + P_2 \\ & & P_1 \| P_2 \;\; P \,\triangle_t^b\, (P_1, P_2, P_3) \mid [P]_I \\ & & P \backslash\backslash I \mid P \backslash F \mid C \end{array}$$

The semantics of an ACSR process is defined in terms of a labeled transition system together with a notion of prioritized transition represented by a preemption relation. We informally describe the ACSR semantics next; for a detailed description, please refer to [19].

NIL is a process that executes no action, i.e., it is initially deadlocked. There are two prefix operators that correspond to the two types of actions. The first, $A^t : P$, executes a timed, resource-consuming action A for $t \in \mathbb{N}^+$ time units (i.e. t is a positive integer) and proceeds to the process P. The second prefix operator, $(e,p).P$, executes the instantaneous event e at priority level p, and proceeds to P. The Choice operator $P_1 + P_2$ represents possibilities – either of the processes may be chosen to execute, subject to the event offerings and resource limitations of the environment. The operator $P_1 \| P_2$ is the concurrent execution of P_1 and P_2. Note that in ACSR, execution of events can be interleaved while actions are executed synchronously by the processes in the parallel operator.

The Scope construct $P \,\triangle_t^b (P_1, P_2, P_3)$ binds the process P by a temporal scope [20], and incorporates both the features of timeouts and interrupts. P executes for a maximum of $t \in \mathbb{N}^+ \cup \{\infty\}$ time units. The scope may be exited in three ways. First, if P successfully terminates within time t by executing an event labeled with $\overline{b}$, then control proceeds to the "exception-handler" P_1 (here, b may be any label other than τ). Second, if P fails to terminate within time t, then control proceeds to the "timeout-handler" P_2. Lastly, at any time while P is executing it may be interrupted by P_3's execution of an action or event, and the scope is then departed.

The Close operator, $[P]_I$, produces a process P that uses the resources in the set I exclusively. The resource hiding operator, $P \backslash\backslash I$, eliminates all resources in I from the actions of P. The Restriction operator, $P \backslash F$, limits the behavior of P: events with labels in F are permitted to execute only if they synchronize and become the internal event τ. Each process constant C is associated with a *process definition* of the form $C \stackrel{\text{def}}{=} P$. This provides ACSR's mechanism for defining recursive processes.

When several execution steps (i.e. transitions in the labelled transition system) are simultaneously possible they are prioritized. 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: 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 [19].)

ACSR offers two basic notions of behavioral equivalence that are defined over the prioritized labeled transition system. The first equivalence relation is based on strong bisimulation [22], $\sim_\pi$, which ensures that equivalent processes match each other's labeled transitions; it is a congruence relation [13]. The second is based on weak bisimulation, $\approx_\pi$, which ensures that equivalent processes match each other's transitions that are labeled with actions and non-τ events but allows one process to make transitions on τ that an equivalent process does not match.

B. GCSR

One motivation for the development of Graphical Communicating Shared Resources (GCSR) is that, like other textual formalisms, the syntax of ACSR produces obtuse specifications that are hard to understand even in the case of simple examples. In addition, the development of GCSR addressed two concerns: support the modular, hierarchical, and thus scalable, specification of real-time systems; and benefit from the analysis techniques [6], [7] and tool set [8] developed for ACSR.

Scalability is an essential feature in any specification language (and in particular graphical languages) for large-scale applications. GCSR supports scalability through its well defined notions of modularity and hierarchy. In GCSR, the visibility scope of communication events, which reflect potential dependencies between system components, can be limited. Furthermore, GCSR's notion of hierarchy is *structured* in the sense that no edge can cross node boundaries and there is a graphical distinction between control transfer due to an interrupt versus an exception, i.e., involuntary versus voluntary release of control. These two syntactic features, in addition to the explicit representation of resources and priorities, distinguish GCSR from other graphical languages for real-time systems, e.g., Statecharts [15], Modechart [17] and the Communicating Real-time State Machines [27].

The semantics of GCSR can be defined either directly as a labeled transition system or through a translation to ACSR. As shown in [4], the second way of defining the semantics of GCSR allows a designer to interchangeably use the graphical and textual notations; for example, to specify the high-level view of a system graphically and fill the details of components textually. In addition, it allows a designer to use the notions of equivalence of ACSR to verify the correctness of a GCSR specification, as we will see shortly. Equivalence checking also distinguishes GCSR from other graphical languages for real-time systems. We next briefly describe the syntax of GCSR and its semantics through a correspondence to ACSR.

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

Figure 1 shows the graphical symbols for the five 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 non-shared resource usage. The *Name* attribute of a reference node refers to

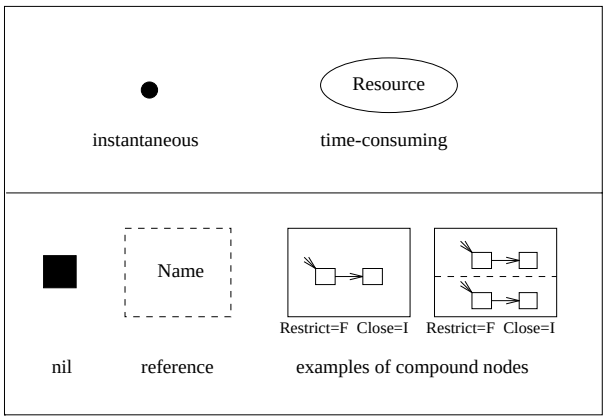

Fig. 1. GCSR nodes

the name of a GCSR process. The *Restrict* and *Close* attributes of a compound node are sets of event names and resource names, respectively.

The motivation for various node symbols in GCSR is a succinct and scalable representation of the different system activities and components. The instantaneous node requires that no delay is allowed before the next activity. In contrast, the time-consuming node describes a time and resource consuming activity. Note that the explicit specification of the required resources by a system activity makes it easy to modify any resource requirement to reflect different resource allocation and scheduling disciplines.

The *nil* node describes a halting process, i.e., end of system execution. The reference node allows the decomposition of a large specification into subspecifications which eases the visual structuring of such a specification. On the other hand, the compound node visually distinguishes a system action from a system component. It is essential in supporting scalable and modular specifications since it allows a designer to 1) group GCSR processes into a higher level entity, 2) connect several GCSR processes that are executed sequentially, and 3) reflect the fact that system components execute in parallel. In addition to the structural modularity, compound nodes also provide for 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, even if their actions do not explicitly request them.

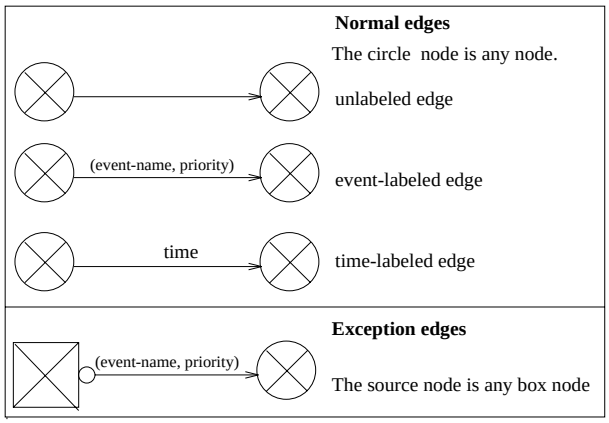

Fig. 2. GCSR edges

GCSR nodes can be connected with edges to describe sequential execution. GCSR offers four types of edges shown in Figure 2. We call unlabeled, event-labeled, and time-labeled edges *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.

In [4] we define a set of simple syntactic rules for a *well-formed* GCSR process. These rules basically disallow 1) an edge to cross a node boundary, 2) an edge to connect nodes in parallel components inside a compound node, 3) a node to have multiple time-labeled outgoing edges, and 4) an instantaneous node to have a time-labeled outgoing edge.

GCSR-ACSR Correspondence. Figure 3 shows the main steps of the translation of a GCSR process to an ACSR process, where the translation function is denoted as T.

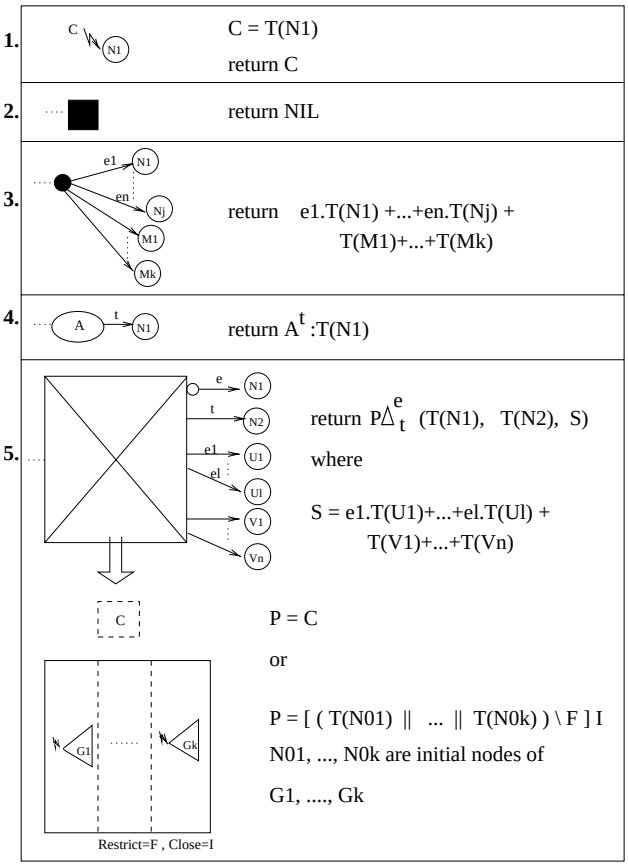

Fig. 3. GCSR to ACSR Translation, T.

The translation starts from the initial node of a GCSR process and recursively traverses all reachable nodes. Step 1 binds the translation of an *initial* node to the ACSR process variable name C and returns the ACSR process variable C; this step is done for each initial GCSR node. In step 2, the *nil* node is translated to the NIL ACSR process. In step 3, an instantaneous node is translated to an ACSR Choice process; each ACSR subprocess in the Choice process corresponds either to the translation of the target node of an unlabeled edge out of the instantaneous node, or to an event-prefix ACSR process where the event is the label of an edge out of the instantaneous node and the next process in the event-prefix process is the translation of the target node of the edge. Note that in this translation the event syntax must be converted from GCSR to ACSR, i.e., $(a!, p)$ and $(a?, p)$ are converted to $(\overline{a}, p)$ and $(a?, p)$, respectively. In step 4, a time-consuming node is translated to an action-prefix ACSR process where the duration of the action is the label of the time-labeled edge out of the time-consuming node. In step 5, the translation of the *box* node produces a Scope ACSR process $P \ \Delta_t^e \ (P_1, P_2, P_3)$ where the main process P is the translation of the box node without its outgoing edges.

Note that due to space limitations, Figure 3 abstracts out several details such as how

loops are handled, and equivalence preserving optimizations applied when some edges are missing in a GCSR specification. The reverse translation, from ACSR to GCSR, is also possible.

C. Example: Aircraft Landing Gear

The landing gear system is a case study of the Swedish JAS 39 Gripen defense and fighter aircraft [28]. In [28], the authors used an automaton-based language to model the system and prove that their model ensures one safety property: the door and landing gear do not collide while in motion. In this section, we illustrate how to model the aircraft landing gear system using the GCSR and ACSR formalisms. In subsequent sections, we will analyze our model for the above safety requirement and additional timeliness requirements.

The physical system, i.e. plant consists of a door and a landing gear. The goal of the case study is to model a software controller that receives commands from a pilot to lower or raise the landing gear. To function autonomously, the plant is equipped with sensors that report the status of the plant to the controller, and actuators, for the controller to operate the plant.

When it receives the command to lower the landing gear, the software controller must first inquire the status of the door and landing gear. If the door is closed and the landing gear is raised, the controller signals the door to open, waits until the door is completely open, signals the landing gear to move down, waits for it to be completely lowered, and then signals the door to close again. When it receives the command to raise the landing gear, the controller also inquires the status of the door and landing gear. If the door is closed and the landing gear is lowered, the controller first signals the door to open, signals the landing gear to move up after the door is open, waits until the landing gear is raised, and then signals the door to close.

The software controller operates the door and landing gear within a set of timing and safety requirements. The timing assumptions about the system are summarized in Table I. The controller must report to the pilot the result of executing a command within a fixed deadline. One safety requirement in the original case study [28] is that the controller must ensure that the door and gear never collide while in motion. In addition, our design will check for and report failures in the door and gear.

TABLE I
TIMING ASSUMPTIONS

Door	
$Tdoor = 10$	time to open/close door
Landing gear	
$Tgear = 10$	time to lower/raise gear
Controller	
$Tdoor + 1$	deadline for door's response
$Tgear + 1$	deadline for gear's response

Our design. Our design models a sample of both unshared and shared resources. The unshared resources *door* and *gear* represent physical components in the plant each of which is only manipulated by its corresponding software/hardware component. In addition, our design models the shared and critical resource *space* which describes the space in which both the gear and door move during their operation. This resource can be used by

one component at a time. An attempt by the door and gear components to use it simultaneously indicates a collision between the door and gear, which is a safety violation.

Each sensor and actuator is represented by a communication event. Since our formalisms lack continuous data, sensory information only reflects the critical status of the plant, e.g. door completely open or completely closed. Intermediate sensory information is ignored. The synchronization events for the set of sensors and actuators in the system are summarized in Table II.

Figure 4 shows the high-level structure of our design which contains three components: the processes *Door* and *Gear* which model the two components in the plant, and the process *Controller* which is the software controller. The three components execute concurrently and privately coordinate with one another through the synchronization events shown in Table II. In addition, the three components reserve the resources *door*, *gear* and *space*.

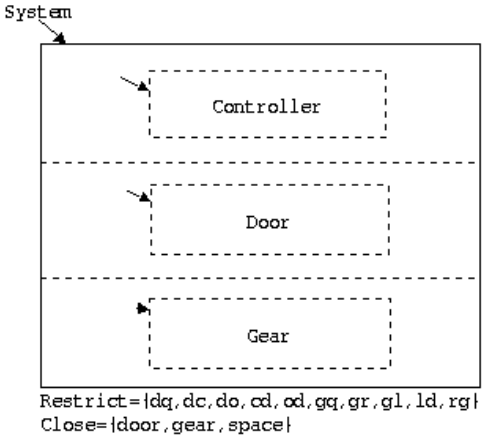

Fig. 4. High Level GCSR Specification of the Landing Gear System

Figure 5 shows the ACSR specification of the door component. Initially, the door is assumed to be closed and behaves as described by the process *Closed*. This latter has four possible behaviors: instantaneously send the event $(\overline{dc}, 1)$ to indicate its closed status and remain closed; instantaneously receive the event *cd* at priority level 1 after which it remains closed; instantaneously receive the event *od* at priority level 2 after which it starts behaving as described by the process *Opening*; or idles by consuming time and no resources. As shown in Figure 5, within the *System* process the *Door* process synchronizes its behavior with the *Controller* process through the events *dc*, *cd* and *od*. Thus, when synchronization occurs, these events will become the special, internal event τ. The prioritized semantics of ACSR and GCSR is such that when any synchronization is possible, the *Door* process favors it over idling. Also, when all synchronizations are possible, the *Door* process favors the event *od* over the other possible behaviors; this is reflected by the high priority of the event *od*. However, the selection between the events *dc* and *cd* is nondeterministic since they have equal priorities.

The process *Opening* describes the activity of opening the door: the process starts behaving like the process *DoorBusyO* for *Tdoor* time units. The process *DoorBusyO* can consume the resource *door* at priority level 1 together with the resource *space* at priority level 1 for one time unit. The process *DoorBusyO* uses the resources in a non-preemptive way. In addition, the process *DoorBusyO* can receive the event (*od*,1) which instructs it to open the door. Since it is already opening the door, the process receives the event and goes on carrying out the door-opening activ-

TABLE II
SYNCHRONIZATION EVENTS

Controller	→	Door	Controller	→	Landing gear	}	Actuators
od	:	open door	lg	:	lower gear		
cd	:	close door	rg	:	raise gear		
Door	→	Controller	Landing gear	→	Controller	}	Sensors
do	:	door is open	gl	:	gear is low		
dc	:	door is closed	gr	:	gear is raised		

$$
\begin{array}{lcl}
\text{Door} & \stackrel{\text{def}}{=} & \text{Closed} \\
\text{Closed} & \stackrel{\text{def}}{=} & (\overline{dc},1).\text{Closed} + (cd,1).\text{Closed} + (od,2).\text{Opening} + \emptyset : Closed; \\
\text{Opening} & \stackrel{\text{def}}{=} & \text{DoorBusyO} \triangle_{\text{Tdoor}} (\text{NIL}, \text{Open}, (cd,2).\text{Closing}) \\
\text{Closing} & \stackrel{\text{def}}{=} & \text{DoorBusyC} \triangle_{\text{Tdoor}} (\text{NIL}, \text{Closed}, (od,2).\text{Opening}) \\
\text{DoorBusyO} & \stackrel{\text{def}}{=} & (od,1).\text{DoorBusyO} + \{(\text{door},1),(\text{space},1)\} : \text{DoorBusyO} \\
\text{DoorBusyC} & \stackrel{\text{def}}{=} & (cd,1).\text{DoorBusyC} + \{(\text{door},1),(\text{space},1)\} : \text{DoorBusyC} \\
\text{Open} & \stackrel{\text{def}}{=} & (\overline{\text{do}},1).\text{Open} + (od,1).\text{Open} + (cd,2).\text{Closing} + \emptyset : \text{Open}
\end{array}
$$

Fig. 5. ACSR Specification of the Door Component

ity. The execution of the process *DoorBusyO* can be aborted in two ways within the process *Opening*: One is through a timeout after *Tdoor* time units from its instantiation and after which the process *Open* is started; a second way is through receiving the event $(\overline{cd},2)$ from the controller, in which case the process *Closing* is started to close the door. Synchronization with the controller has a higher priority than consuming time and the resource *door* as long as the timer *Tdoor* has not expired.

The specification of the *Gear* process is similar to the process *Door*. When it is being lowered or raised, the *Gear* process simultaneously uses the *gear* and *space* resources in a non-preemptive mode. Figure 6 shows the GCSR specification of the *Controller* process. Execution starts at the reference node marked with *Controller*. At this time, the controller starts idling as described by the *Wait* process. The idling can be interrupted by the reception of the event *cm_dn* at priority level 3, at which time execution moves inside the top compound node. Inside this node, two processes execute in parallel: one process, *LowerGear*, describes the controller's instructions to open the door, lower the gear, and then close the door again; the second describes the fact that the controller is always ready to receive the command to lower the gear—event (*cm_dn*?,3) which is accepted without the controller initiating additional attempts to lower the gear. Execution of these two processes can be interrupted at any time by the reception of a command to raise the gear—event (*cm_up*?,3). When this command is received, the controller in-

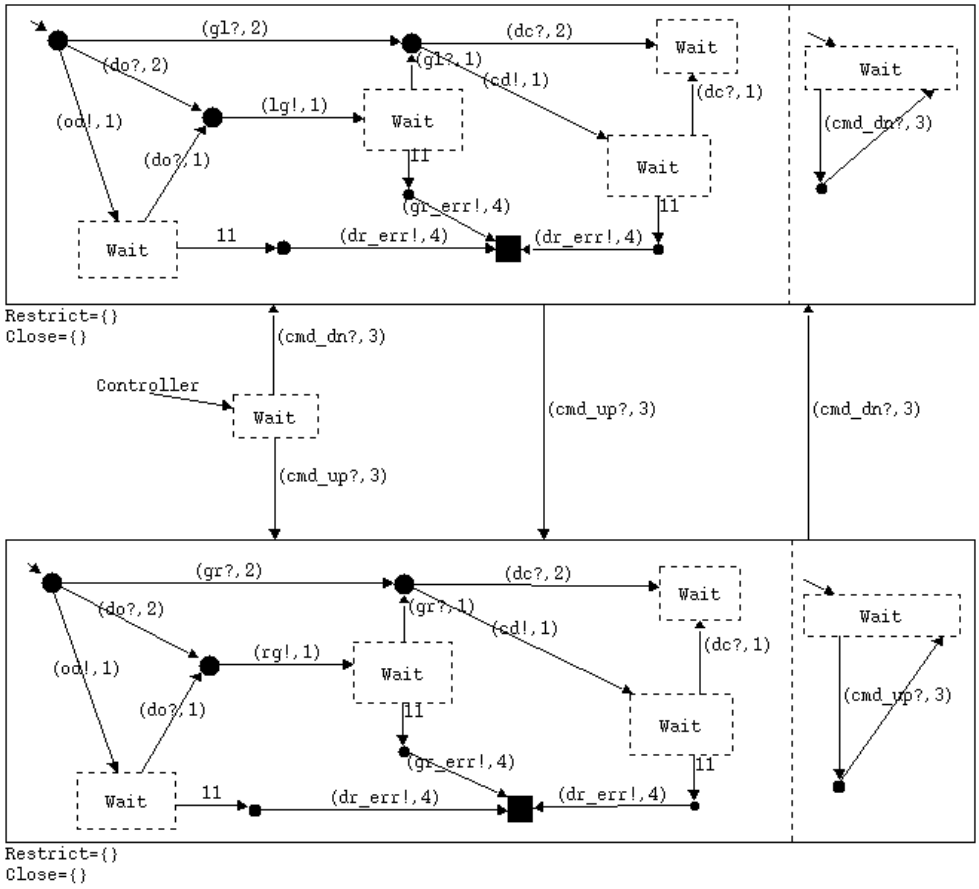

Fig. 6. GCSR Specification of the Controller Component

terrupts its instructions to open the door and lower the gear and starts instructing the door to open, the gear to move up, and then the door to close again.

Our design accounts for potential failures of the door and gear. For example, in the process *LowerGear* after signaling the door to open, the controller waits for at most one time unit more than the time needed to open the door. If by this time the controller does not receive the event *do* which indicates that the door is open, it assumes a failure has occurred in the door and signals an error, event (*dr_err*!, 4), then immediately enters a deadlocked state that is represented by a *nil* node. Our design also includes similar error detection for the gear. The semantics of GCSR and ACSR ensures that if the controller (or any of the other parallel processes) deadlocks, so does the process *System*. We will use these deadlocks and special error-marking events to test for potential malfunctioning of the door and gear.

III. The PARAGON Toolset

We have implemented a toolset, called PARAGON, to facilitate the use of the ACSR paradigm for real-time systems modeling and analysis. Figure 7 shows the overall structure of PARAGON. The user's view of the toolset is provided by the GCSR, XVERSA

and VERSA user interfaces. The analysis of ACSR and GCSR specifications is carried out by the VERSA system that is accessed through these interfaces.

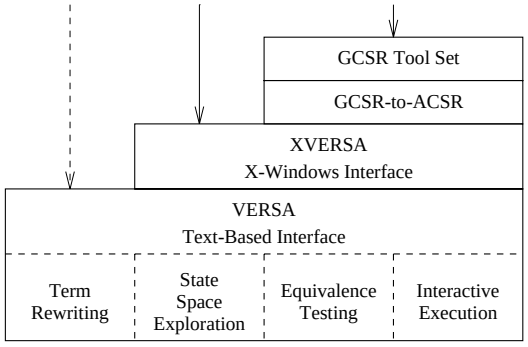

Fig. 7. Architecture of the PARAGON toolset

The user interfaces are responsible for management of input/output streams. They allow processes to be entered as graphical GCSR process descriptions or as ACSR processes using a text-based notation. Graphical input of GCSR specifications is managed with drawing support functions, syntax-checking functions, and automated translation of GCSR to ACSR. The text-based notation accepted by XVERSA enhances the ACSR process algebra with indexing, which can be used to emulate value-passing.

Within VERSA there are four major functional areas for analyzing processes: term rewriting, state space exploration, equivalence testing, and interactive execution.

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

State space exploration, equivalence testing and interactive execution operate on a labeled transition system (LTS) representation of the system being analyzed. The LTS for one or more processes is produced by an algorithm that expands the process to produce a labeled transition system representing all possible executions. The LTS construction algorithm also prunes edges made unreachable by the semantics of the prioritized transition system, in most cases reducing the size of the resulting LTS.

State space exploration analysis can be used to determine key properties of a system's LTS. These include (1) number of states and transitions; (2) presence of deadlocked states; (3) states capable of *Zeno* behaviors (*i.e.* infinite sequences of instantaneous events); (4) states that require synchronization to take place before time can progress; and (5) reachability of specific externally observable events.

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

The interactive execution feature allows user-directed execution of process specifications. The user may interactively step through the LTS one action at a time, pro-

duce traces from random executions of the LTS, save process configurations to a stack for later analysis while an alternate path is explored, and analyze the size and deadlock characteristics of the LTS resulting from their process.

IV. Design Support

A. Specification

The various operators in ACSR, such as the parallel and scope operators, allow the specification and verification of a system in a modular and hierarchical fashion, which makes the development of a large scale real-time system more manageable. The precise, compositional semantics of the operators allows one to replace any component of a requirements specification by an equivalent component, and retain any correctness proved using the same equivalence. Thus, one can develop a design specification by a series of component-wise design refinements, starting with a requirements specification and gradually replacing its components with detailed components until a desired design specification is realized.

Another support of modular and hierarchical design within ACSR and GCSR is through a set of *refinement* operations that allow a designer to introduce implementation details gradually, e.g. add communication events, rename communication events, show the structure of a process for a time-consuming action, and tighten an estimate resource requirements [3]. These notions of refinement are supported through a set of rewrite rules for ACSR and graphical transformations for GCSR, that syntactically manipulate an abstract specification to add details. In addition, refinement in ACSR and GCSR has a precise semantics that insures that each trace of the refined specification mimics a trace in the abstract specification in such a way that each timed occurrence of an abstract event is preserved in the refined trace.

Another feature of the ACSR paradigm that facilitates real-time system specification is the precise correspondence between ACSR and GCSR, which allows a designer to combine textual and graphical descriptions. For example, a designer can describe the high level structure of a system graphically and fill in the detailed description of components textually. This helps a designer to visualize the structure of the system and the dependencies between its component expressed as communication events.

To illustrate one of the notions of refinement in GCSR, let us revisit the landing gear example and refine our controller design shown in Figure 6. We can refine this design by adding a report to the pilot after each command has been successfully carried out by the door and gear. For this, we rely on the modularity of refinement in GCSR and focus on refining the processes in charge of carrying out the commands, e.g., LowerGear and which are part of the *Controller* component of the *System* process. One new event is ($gdn!$, 4) and it is inserted between each occurrence of the event dc and the next *Wait* process. A similar event ($gup!$,4) is added to report the fact that the gear has been raised. The refined *Controller* process is shown in Figure 8.

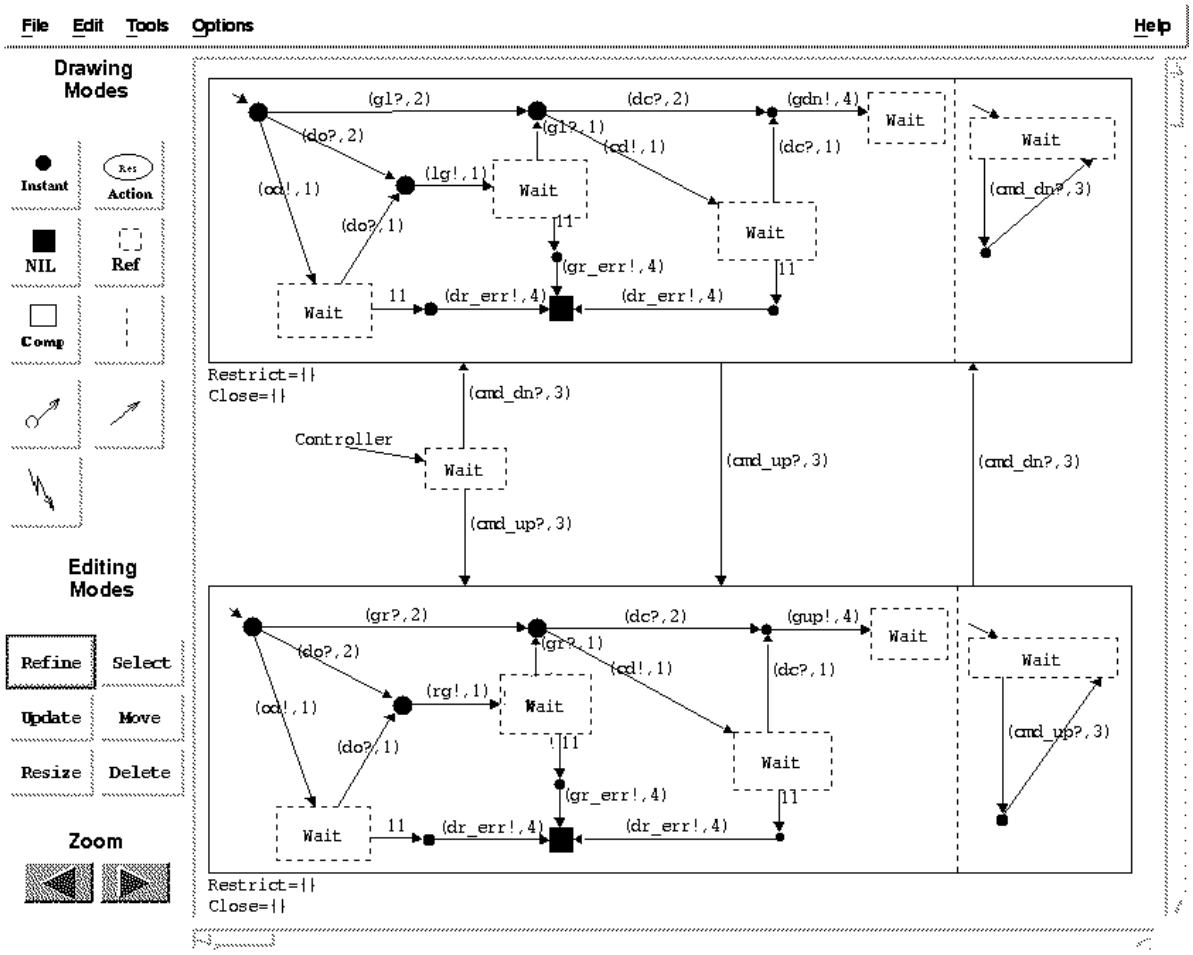

Fig. 8. Refined Controller

B. Verification

The primary methods for verifying the correctness of GCSR and ACSR processes are (1) using bisimulation checking algorithms to verify the prioritized strong equivalence of a design model and a high-level correctness specification; and (2) using state space exploration to detect the presence of deadlocked states.

Equivalence Testing. When a real-time system model is based on the structure of a proposed design, the model is likely to contain non-sequential operators such as parallel composition and temporal scope. Although these operators are critical for creating a model that accurately and succinctly describes the intended design, they make it difficult to determine the model's operational behavior by inspection. However, if the system's correct behavior (ignoring constraints on the design such as a need for redundancy or distributed control) has a succinct sequential specification that is correct by inspection, the correctness of the design model can be verified by proving the two descriptions are bisimilar.

For example, consider the model of *System* presented in Figure 4 of Section II-C. The model contains nested processes with syn-

chronization events, temporal scopes, and resource requirements to facilitate a concise description and accurately reflect the hardware implementation. The system also has a sequential correctness specification SystemSeq

$$\begin{aligned} \text{SystemSeq} \stackrel{\text{def}}{=} \; & \emptyset : \text{SystemSeq} \\ & + (\text{cmd\_dn}, 3).\text{SystemSeq} \\ & + (\text{cm\_up}, 3).\text{SystemSeq} \end{aligned}$$

which describes in simple terms a recursive behavior that essentially accepts commands and consumes time. The correctness of this formulation of the system requirements can be verified by inspection: the process SystemSeq never deadlocks, it is always ready to accept commands, and it never produces any failure marking event.

Deadlock Detection. An approach to verification that can be applied to non-terminating systems is to use resource contention deadlocks to indicate that an unsafe state is reachable. Recall that the notion of resource defined in ACSR's semantic model does not allow two or more timed actions to execute simultaneously if they require the same resource. If a system model is constructed in such a way that a resource must be held while executing in a non-sharable critical region, then an attempt by two or more processes to execute simultaneously in that critical region will introduce a deadlock.

For example, to ensure that our landing gear system design is safe, we need to check that there is never a collision between the door and the landing gear. First, we note that each component can separately execute forever; this is due to the recursive definition of the processes. Second, the concurrent behavior of the components consists of synchronized events (i.e. τ's) interleaved by timed usage of the *door*, *gear* and *space* resources, possibly at priority zero. Thus, by the operational semantics of parallel execution, if all the events are synchronized and there is no resource contention, then the components will execute forever; in other words, the system will not reach a deadlocked state. Furthermore, if we ignore the synchronization events and conceal the identities of the resources, the concurrent behavior of the components consists of infinite sequences of idling actions, i.e. $\emptyset$, and reception of the command events *cmd_dn* and *cmd_up* which is described by the process SystemSeq. From this, we can conclude that our design is safe if the following bisimulation holds:

$$System \backslash\backslash \{door, gear, space\} \approx_{\pi} SystemSeq$$

where the operation $\backslash\backslash$ hides the resource names in the behavior of the process.

In our design, there are two sources of deadlocks which would make the above equivalence fail: contention on the *space* resource, which indicates collision between the door and gear, and miscommunication between the controller and physical devices. For the timing assumptions shown in Table I, we used our automated GCSR-to-ACSR translation and the VERSA toolset to prove that our design has no deadlocked states and it satisfies the above bisimulation; and thus our design is safe with respect to the collision criterion. This test is performed automatically by the VERSA system using its equivalence testing features. LTS's for System and SystemSeq are constructed, and a state-minimization algorithm[18] is applied to determine whether the two LTS's are bisimilar. If they are, then the correctness of the complex System process can be verified by in-

specting the SystemSeq process. If they are not equivalent, then there is some fault in System that causes it to behave differently than SystemSeq. In this case, VERSA will produce a state where the two processes have distinguishing behaviors.

C. Testing

Both of the verification techniques described in Section IV-B rely on computing the LTS representation of the system model being analyzed. Therefore they are only applicable in situations where the LTS representation of the process can be computed in a reasonable amount of time and space. In cases where this is not possible, an alternative approach is to use testing techniques to explore specific system behaviors without computing the entire system state space.

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

For example, the specification of the landing gear *System* requires that at most 30 time units will elapse following a *cmd_dn* before the *gdn* response is received ($Tdoor + Tgear + Tdoor = 30$). Thus, one possible test of System is as shown in Figure 9. If P is a correct implementation of the System specification for this input, then the state space of $R = (P \parallel T)\backslash\{\text{cmd\_up,cmd\_dn,gup,gdn}\}$ will contain the $\overline{success}$ event. If P accepts the cmd_dn input but produces incorrect output, or no output within 30 time units, then R will emit $\overline{failure}$ and deadlock at time 31.

$$
\begin{aligned}
T_1 &= (\overline{cmd\_dn}, 1).T_1' \\
T_1' &= (IdleDn \triangle_{31} \\
&\qquad (NIL, \\
&\qquad (\overline{failure}, 1).NIL, \\
&\qquad (gdn, 1).(\overline{success}, 1).NIL) \\
IdleDn &= \emptyset : IdleDn + (gup, 1).IdleDn
\end{aligned}
$$

Fig. 9. Sample Test of System

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

Bottom-up techniques are not applicable to our process testing technique because the computation of the LTS for $(P \parallel T)\backslash E$ will begin by computing the LTS for P. When

the computation of the entire state space of P is intractable, VERSA's top-down translation makes testing possible because a single test process will exercises a small subset of the state space of the process being tested.

Testing Techniques. We derive our tests by viewing the processes being tested as *black-boxes* with behavior that can be characterized by inputs, outputs and timing. Testing seeks to verify the acceptance of inputs at the correct time, and the generation of corresponding correct outputs at the correct time. We have successfully applied two black-box testing techniques from software engineering[2] to verify large ACSR process models: (1) exhaustive testing; and (2) partition testing.

Exhaustive testing exercises all possible behaviors of a process by applying all possible inputs and checking the process outputs. If the process being tested accepts only finitely many inputs and has only executions of finite duration, then exhaustive testing of the input domain is sufficient to verify the correctness of the process. Process models generally have finite input domains (this can be verified by inspection of the process structure) but infinite executions are common in realistic system models. Where infinite executions exist, simplifying assumptions regarding the maximum length of a non-repeating execution, or the maximum length of an execution that will be of interest have to be made.

Partition testing is based on a partitioning of inputs into classes, all elements of which exercise the same internal functions of the process. If the partitioning is correct, then exercising a single input from each class is sufficient to verify correctness of the process as a whole. The weakness of this method lies in the need to partition the functions of a process when the internal structure of the process is unknown. As reported in the literature[14], partition testing has intuitive appeal, but its quantitative merits are limited.

A more complete test of *cmd_up* and *cmd_dn* inputs of the landing gear System process could be performed by exercising both the *cmd_up* and *cmd_dn* inputs and all of their possible interactions. Because $System$ is non-terminating, there are infinitely many inputs that exercise these scenarios. We concentrate our attention here on single inputs of each command, beginning from the start state. A test suite to exercise *cmd_up* and *cmd_dn* and a single interruption of one command by the other, beginning in the start state, is shown in Figure 10.

Test T_2 exercises the *cmd_up* input for the system's start state, and verifies that *gup* is returned within the required interval. Test T_2 applys *cmd_dn* in the system's start state, and then nondeterministically applys *cmd_up* at every time instant up to and including the instant when *gdn* is returned. Then it is verified that *gup* is returned within the required interval. Thus, T_3 verifies the ability of a *cmd_dn* command to be properly interrupted by *cmd_up* at any instant up to and including the moment when the gear is fully lowered. Test T_4 performs a similar task, checking the ability of *cmd_up* to be properly interrupted by *cmd_dn* at any time.

We used VERSA to apply these tests to System and explored the state space of each of the resulting LTS's to determine that each contained $\overline{success}$ and was $\overline{failure}$ free. Therefore, System executes the tests T_1, T_2,

$$
\begin{aligned}
T_2 &= (\overline{cmd\_up}, 1).(IdleUp \triangle_{31} (NIL, (\overline{failure}, 1).NIL, (gup, 1).(\overline{success}, 1).NIL) \\
IdleUp &= \emptyset : IdleUp + (gdn, 1).IdleUp \\
\\
T_3 &= (\overline{cmd\_dn}, 1).(IdleDn \triangle_{31} (NIL, (\overline{failure}, 1).NIL, (gdn, 1).T2 + (\tau, 0).T2) \\
\\
T_4 &= (\overline{cmd\_dn}, 1).(IdleDn \triangle_{31} (NIL, (\overline{failure}, 1).NIL, (gdn, 1).T4') \\
T_4' &= (\overline{cmd\_up}, 1).(IdleUp \triangle_{31} (NIL, (\overline{failure}, 1).NIL, (gup, 1).T1 + (\tau, 0).T1)
\end{aligned}
$$

Fig. 10. Test processes for landing gear *System*

T_3 and T_4 successfully. The combined state space of the application of all four tests to *System* contained 500 states and 556 edges, as compared with 2766 states and 8432 edges for the full state space of *System*. This represents an 82% reduction in states and a 93% reduction in edges for testing compared to exploration of the full state space of *System*

V. Conclusion

We have presented tools and techniques for modeling resource bound real-time systems. The presentation was based on the ACSR process algebra and the GCSR graphical process description language. Tools and techniques for constructing and analyzing system models using GCSR and ACSR were presented.

In our experience, no single verification technique will be effective in all cases. Exhaustive verification based on state space exploration is mainly applicable to small-scale systems, or mission-critical subsystems of a larger design. To analyze large systems, the user has to concentrate on specific aspects of the system's behavior. We think that our testing approach provides a formal basis for this. By combining the two techniques, the user can reach the desired level of confidence in correctness of a given design.

Our current research goals include the automatic derivation of tests that satisfy a time-based partition testing criteria. We intend to augment the existing toolset with: (1) a tool for creating a high-level timing specification diagram; and (2) tools that will use these timing specifications to automatically derive a complete partition test suite to check proper implementation of the timing constraints. We also want to explore how to use the generated tests to drive the graphical simulator of GCSR specifications, which is a part of the PARAGON toolset.

An important aspect of a specification paradigm is the right balance between textual and graphical specification. It seems that high-level specifications are better comprehended in the graphical form, while more detailed ones may be too tedious to enter graphically. Therefore, another area of interest lies in closer integration of graphical (GCSR) and textual (ACSR) specification paradigms.

References

[1] J.C.M. Baeten and J.A. Bergstra. Discrete Time Process Algebra. In *CONCUR 92*, pages 401–420. LNCS 630, Springer Verlag, 1992.

[2] B. Beizer. *Software Testing Techniques.* Van Nostrand Reinhold, second edition, 1990.

[3] 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.

[4] Hanêne Ben-Abdallah, Insup Lee, and Jin-Youg Choi. A graphical language with formal semantics for the specification and analysis of real-time systems. In *IEEE Proceedings of Real-Time Systems Symposium (RTSS' 95)*, Pisa, Italy, December 1995.

[5] P. Brémond-Grégoire, I. Lee, and R. Gerber. ACSR: An Algebra of Communicating Shared Resources with Dense Time and Priorities. In *Proc. of CONCUR '93*, 1993.

[6] Jin-Youg Choi, Insup Lee, and Hong-Liang Xie. The specification and schedulability analysis of real-time systems using acsr. In *IEEE Proceedings of Real-Time Systems Symposium (RTSS' 95)*, Pisa, Italy, December 1995.

[7] Duncan Clarke and Insup Lee. A hybrid approach to formal verification applied to an atm switching system. Technical report, Dept. of CIS, Univ. of Pennsylvania, Dec 1995.

[8] 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.

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

[10] 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 the IEEE Real-Time Systems Symposium*, pages 12–22, Raleigh-Durham, North Carolina, December 1-3 1993.

[11] J. Davies and S. Schneider. An Introduction to Timed CSP. Technical Report PRG-75, Oxford University Computing Laboratory, UK, August 1989.

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

[13] 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.

[14] R. Hamlet and R. Taylor. Partition testing does not inspire confidence. *IEEE Transactions on Software Engineering*, 16(12):1402–1411, December 1990.

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

[16] M. Hennessy and T. Regan. A Temporal Process Algebra. Technical Report 2/90, Univ. of Sussex, UK, April 1990.

[17] 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.

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

[19] 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.

[20] I. Lee and V. Gehlot. Language Constructs for Distributed Real-Time Programming. In *Proc. IEEE Real-Time Systems Symposium*, 1985.

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

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

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

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

[25] J. Ostroff. A Verifier for Real-Time Properties. *Journal of Real-Time Systems*, 4:5–35, 1992.

[26] Y. Ramakrishna, P. Melliar-Smith, L.Moser, L.Dillon, and G. Kutty. Really Visual Temporal Reasoning. In *Proc. of IEEE Real-Time Systems Symposium*, pages 262–273, December 1993.

[27] A. C. Shaw. Communicating Real-Time State Machines. *IEEE Transactions on Software Engineering*, 18(9):805–816, September 1992.

[28] M. Westhead and S. Nadjm-Tehrani. A study of decompositional verification of hybrid systems. Technical Report LiTH-IDA-95-30, Department of Computer and Information Science, University of, Linköping, Sweden, 1995.

[29] Wang Yi. CCS + Time = An Interleaving Model for Real Time Systems. In *Proc. of Int. Conf. on Automata, Languages and Programming*, July 1991.

Hanêne Ben-Abdallah received the B.S. degree in Computer Science and Mathematics in 1989 from the University of Minnesota, Minneapolis, the M.S.E. degree in 1991 and Ph.D. degree in 1996 in Computer and Information Science from the University of Pennsylvania.

She is currently a postdoctoral research fellow in the Department of Electrical and Computer Engineering at the University of Waterloo. Her research interests include distributed and real-time systems, methodologies and tools for applying formal methods in software engineering, and graphical software engineering environments.

Duncan Clarke received the B.S. degree in computer science from Michigan State University, East Lansing, in 1986, the M.S. degree in computer science from Rensselaer Polytechnic Institute, Troy, in 1987, and the Ph.D. degree in computer science from the University of Pennsylvania, Philadelphia, in 1996.

He is currently a Visiting Assistant Professor in the Department of Computer Science at the University of Kentucky. His research interests include tools and techniques for applying formal methods, software engineering, and real-time systems.

Insup Lee received the B.S. degree in mathematics from the University of North Carolina, Chapel Hill, in 1977, and the Ph.D. degree in computer science from the University of Wisconsin, Madison, in 1983.

He is currently an Associate Professor in the Department of Computer and Information Science at the University of Pennsylvania, where he has been since 1983. His research interests include distributed systems, real-time computing, programming languages, operating systems, formal methods and software engineering.

He was the Co-Chair of the Program Committee for the 1992 IEEE Real-Time Systems Symposium, and the General Co-Chair for the 1993 IEEE Real-Time Systems Symposium. He was the Co-Chair of the Program Committee for First International Workshop on Real-Time Computing Systems and Applications held in December 1994 at Seoul, Korea. He was the Co-Chair of CONCUR '95: Internation Conference on Concurrency Theory held in August 1995 at the University of Pennsylvania. He was the Co-Chair of the Program Committee for Third International Workshop on Real-Time Computing Systems and Applications held in October 1996 at Seoul, Korea. He is on the editorial board of IEEE Transactions on Computers.

Oleg Sokolsky received a B.S./M.S. degree in Computer Science from St. Petersburg Technical University (1988) and a Ph.D. degree in Computer Science from SUNY at Stony Brook (1996). He is currently employed as a Computer Scientist with Computer Command and Control Company. His research interests include specification and verification of distributed systems and graphical specification environments.