Specification and Analysis of Real-Time Systems with PARAGON

Oleg Sokolsky, Insup Lee

Department of Computer and Information Science

University of Pennsylvania

Philadelphia, PA 19104, U.S.A.

sokolsky@saul.cis.upenn.edu

Hanêne Ben-Abdallah

Département d'Informatique

FSEG

Université de Sfax

B.P. 1088

3018 Sfax, Tunisia

Abstract

This paper describes a methodology for the specification and analysis of distributed real-time systems using the toolset called PARAGON. PARAGON is based on the *Communicating Shared Resources* paradigm, which allows a real-time system to be modeled as a set of communicating processes that compete for shared resources. PARAGON supports both visual and textual languages for describing real-time systems. It offers automatic analysis based on state space exploration as well as user-directed simulation. Our experience with using PARAGON in several case studies resulted in a methodology that includes design patterns and abstraction heuristics, as well as an overall process. This paper briefly overviews the communicating shared resource paradigm and its toolset PARAGON, including the textual and visual specification languages. The paper then describes our methodology with special emphasis on heuristics that can be used in PARAGON to reduce the state space. To illustrate the methodology, we use examples from a real-life system case study.

1 INTRODUCTION

As software systems become more complex and safety-critical, it is vitally important to ensure reliability properties of these systems. Most complex safety-critical systems are distributed and must function in real-time. *Formal methods* have been proposed to aid in development of safety-critical systems. They allow users to specify systems precisely and reason about them in mathematical terms. A variety of methods for dealing with hardware and software systems aimed at distributed and real-time systems have been developed. They include state machines, Petri nets, logics, temporal logics, process algebras and timed automata; the summary of existing approaches and directions for future research can be found in [Clarke and Wing 1996; Cleaveland and Smolka 1996]. As formal methods become more mature and their benefits for development of large systems can be clearly demonstrated, they are being increasingly accepted by the industry.

Most industrial designs yield specifications with very large state spaces. Therefore, tools for mechanical analysis of large specifications are essential for successful application of formal methods in industry. A number of tools based on formal methods have been put forward in the last several years in an effort to increase the usability of formal methods especially within the industrial community. Among the tools that are most widely available are the Concurrency Workbench [Cleaveland *et al.* 1993], Spin [Holzmann 1991], SMV [McMillan 1993]. Analysis of real-time systems is supported by COSPAN [Hardin *et al.* 1996], Kronos [Daws *et al.* 1995], and Uppaal [Bengtsson *et al.* 1995], to name just a few.

Even with tool support, most specifications of real-life systems are too large to be analyzed by brute force. Analysis of large systems is impossible without abstractions and simplifications that serve to reduce an infinite, or finite but unmanageable, state space of the system's specification. Users of each formalism and supporting tools employ a number of abstraction heuristics that help in creating manageable specifications of large-scale systems. Some of the used heuristics are specific to the formalism or the tool, while others are applicable to several related methods. Often when case studies are described, such heuristics are left out or mentioned only briefly. We think it is worth while to make these heuristics explicit for the benefit of future users of formal method tools.

This paper describes a methodology for the specification and analysis of distributed real-time systems using a toolset PARAGON. We describe the process of constructing a formal specification from an informal description of the system, and some of the specification patterns often observed in this process. In addition, we summarize heuristics commonly employed by PARAGON that are aimed at reducing the state space of specifications.

PARAGON is based on the process algebra ACSR [Lee *et al.* 1994] and related formalisms. Process algebras, such as CCS [Milner 1989], CSP [Hoare 1985] and ACP [Bergstra and Klop 1985], have been devel-

oped to describe and analyze communicating, concurrently executing systems. A process algebra consists of a concise language, a precisely defined operational semantics, and a notion of equivalence. The language is based on a small set of operators and a few syntactic rules for constructing a complex process from simpler components. The operational semantics describes the possible execution steps that a process can take, i.e., a process specification can be executed, and serves as the basis for various analysis algorithms.

The notion of equivalence captures when two processes behave identically, i.e., they have the same execution steps. To verify a system using a process algebra, one writes a requirements specification as an *abstract* process and a design specification as a *detailed* process. The correctness can then be established by showing that the two processes are equivalent. The most salient aspect of process algebras is that they support the *modular* specification and verification of a system. This is due to the algebraic laws that form a compositional proof system, and thus it is possible to verify the whole system by reasoning about its parts. Process algebras without the notion of time are now used widely in specifying and verifying concurrent systems.

To expand the usefulness to real-time systems, several real-time process algebras have been developed by adding the notion of time and including a set of timing operators to process algebras. In particular, these real-time process algebras provide constructs to express delays and timeouts, which are two essential concepts to specify temporal constraints in real-time systems.

Algebra of Communicating Shared Resource (ACSR) introduced by [Lee *et al.* 1994], is a timed process algebra which can be regarded as an extension of CCS. It enriches the set of operators, introducing constructs to capture common real-time design notions such as resource sharing, exception and interrupt handling. ACSR supports the notions of resources, priorities, interrupt, timeout, and process structure. The notion of real time in ACSR is quantitative and discrete, and is accommodated using the concept of timed actions. The execution of a timed action takes one time unit and consumes a set of resources defined in the timed action during that one time unit period. The execution of a timed action is subject to the availability of resources it uses. The contention for resources is arbitrated according to the priorities of competing actions. To ensure the uniform progression of time, processes execute timed actions synchronously.

ACSR is an extension of another real-time process algebra CCSR [Gerber and Lee 1994], which shares many aspects of ACSR. In particular, CCSR was the first process algebra to support the notions of both resources and priorities. CCSR, however, lacks instantaneous synchronization since all actions take exactly one time unit. ACSR extends CCSR with the notion of instantaneous events and synchronization, and includes a set of laws complete for finite state processes [Brémond-Grégoire *et al.* 1997]. To promote the use of ACSR in the specification and analysis of real-time systems, we have implemented a tool VERSA [Clarke *et al.* 1995]. PARAGON is a toolset that extends the capability of VERSA by a providing graphical user

interface, graphical specification language and visual simulation.

The paper is organized as follows. Section 2 presents the paradigm of Communicating Shared Resources, which forms the basis for analysis in PARAGON. Section 3 gives an overview of the PARAGON toolset. Section 4 describes the specification and analysis methodology of PARAGON using a real-life example. It also details heuristics that enabled us to successfully analyze this example. Section 5 gives an overview of related work. We conclude in Section 6 with a summary of our results and directions for future research.

2 OVERVIEW OF THE FORMALISM

The specification paradigm of Communicating Shared Resources (CSR) [Gerber 1991] is the basis for several process-algebraic formalisms. Among these formalisms are the real-time process algebra ACSR (the **A**lgebra of **C**ommunicating **S**hared **R**esources) [Lee *et al.* 1994; Brémond-Grégoire *et al.* 1997] and the visual specification language GCSR (the **G**raphical CSR) [Ben-Abdallah 1996; Ben-Abdallah *et al.* 1995]. The two languages have compatible semantics and can be intermixed in a large specification.

The CSR paradigm is based on the view that a real-time system consists of a set of communicating components called *processes*. Processes compete for access to a finite set of serially shared resources and synchronize with one another through communication channels. Further, parameterized specifications allow users to represent data manipulation and value passing between processes.

The use of shared resources by processes is represented by timed *actions*, and synchronization is supported via instantaneous *events*. The execution of an action is assumed to utilize a set of resources during a nonzero amount of time, measured by an implicit global clock. The execution of an action is subject to availability of resources it uses, and contention for resources is arbitrated according to priorities of competing actions. In addition, to ensure uniform progress of time, processes execute actions synchronously. Time can be either dense or discrete; in this paper, we consider only discrete time semantics, which is implemented in PARAGON. With discrete time, duration of an action is one tick of the global clock. Each action is represented by a set of resources needed for the action, each with an access priority. Example of an action is $\{(cpu, 2), (sensor, 1)\}$, which is executed only if resources *cpu* and *sensor* are not in use by a higher-priority process. Note that if some resource required by an action is unavailable, the process trying to execute the action will deadlock, unless the specification provides for an alternative behavior.

Unlike an action, the execution of an event is instantaneous and does not require any resources. Processes execute events asynchronously except when two processes synchronize through matching event names, i.e., channels. Two events match if one is an input and the other is an output on the same-name

channel. Matching pairs of events are denoted a and $\bar{a}$ in ACSR, respectively, and $a?$ and $a!$ in GCSR. Contention for channels is also resolved according to priorities of events. An input event on a channel a with priority 1 is denoted as $(a, 1)$. As is common with CCS-like process algebras, a special event τ denotes an internal activity of the process and cannot synchronize with any other event.

The semantic definition of ACSR and GCSR includes a preemption relation between events and actions based on their priorities. After the transitions of a process have been computed according to the semantic rules, the preemption relation is applied to remove transitions with lower priorities.

Formal semantics for ACSR and GCSR in the form of structured operational semantics (SOS) rules may be found in [Lee *et al.* 1994] and [Ben-Abdallah *et al.* 1997], respectively. In this paper, we present an informal overview of the languages and illustrate their use with an example. This allows us to introduce the most commonly used constructs of the languages and compare their utility in crafting specifications. The example represents a fixed-priority scheduler and is an extended version of the one presented in [Brémond-Grégoire *et al.* 1997]. The system consists of a scheduler and three periodic tasks with two parameters: the computation time c_i and the period p_i. The deadline for each task is assumed to be the same as its period. The example can easily be extended to a larger number of tasks and more complex scheduling policies (see [Ben-Abdallah *et al.* 1996]).

Both ACSR and GCSR provide for parameterization of a process specification by an index set, to support efficient representation families of similarly defined processes. There are two kinds of variables in specifications, *process* variables and *index variables*. Process variables represent terms of the algebra, and index variables range over elements of some index set and are used in parameterized specifications. Process variables, as well as event and resource names have fixed *arities* associated with them. We will always assume that the specifications are type correct. That is, whenever an n-tuple is used in parameterization of a name, the name has arity n.

2.1 ACSR

ACSR provides a set of operators that are similar to the common set of operators found in other process algebras: *prefix* for sequencing of actions and events; *choice* for choosing between alternatives; *parallel* for composing two processes to run in parallel; *restriction* and *hiding* for abstracting communication details or resource names; and *recursion* for describing infinite processes. As a real-time formalism, ACSR supports a variety of operators that deal with time. They allow one to *delay* execution for t time units, to *timeout* while waiting for some actions to occur, and to *bound* the time it takes to execute a sequence of actions. In addition, ACSR provides two operators, *interrupt* and *exception*, that are extremely useful in modeling

$$
\begin{array}{rcll}
System & = & [\ (\ Dispatch \parallel \Pi_{i\in\{1..3\}}\ Task[i]\ )\ \setminus\ \{start[i],\ i \in \{1..3\}\}\ ]_{\{cpu\}} & \\
Dispatch & = & \Pi_{i\in\{1..3\}}\ Start[i] & \\
Start[i] & = & (\overline{start}[i], i).D[i,0], & i \in \{1..3\} \\
D[i,j] & = & \texttt{if } j < p_i \texttt{ then } \{\} : D[i,j+1] & \\
 & & \qquad \texttt{else } Start[i], & i \in \{1..3\}, j \in \{1..p_i\} \\
Task[i] & = & (start[i],1).T[i,0] + \{\} : Task[i], & i \in \{1..3\} \\
T[i,j] & = & \{\} : T[i,j] + \texttt{if } j < c_i \texttt{ then } \{(cpu,i)\} : T[i,j+1] & \\
 & & \qquad \texttt{else } Task[i], & i \in \{1..3\}, j \in \{1..c_i\}
\end{array}
$$

Figure 1: A fixed-priority scheduler

real-time systems but are not present in other real-time process algebras. The interrupt operator makes it easy to specify reaction to asynchronous actions or events. The exception operator allows an exception to be raised any place inside a process and handled by an exception handling process.

The ACSR specification of the scheduling system is presented in Figure 1. The top-level specification, *System*, consists of the scheduler process *Dispatch* and a set of three periodic tasks. The operator $\parallel$ represents the parallel composition of *Dispatch* and the set of three tasks. The generalized parallel composition operator $\Pi_{\overline{v}}$ allows one to express parallel composition of all processes generated by index variables $\overline{v}$. In this example, it is used to represent the set of concurrent tasks. The tasks and the scheduler communicate by means of events $start[i]$, used by the scheduler to initiate task i. These events are restricted by means of the hiding operator $\setminus$, making the $start[i]$ events local to *System*. All tasks are run on the same processor, represented by the resource *cpu*. No other processes can use the same processor, which is captured by the resource closure operator $[\cdot]_{\{cpu\}}$.

The scheduler is capable of independently dispatching each task, hence the process *Dispatch* is a parallel composition of a set of processes $Start[i]$. Each process $Start[i]$ begins by sending an event $start[i]$ to task i at priority level i and then proceeds as $D[i,0]$. By sending events $start[i]$ to $Task[i]$ with different priorities, the scheduler effectively serializes the events according to their priorities. The process $D[i,j]$ idles until it is time to start $Task[i]$ again. Idling is represented as an empty action $\{\}$, which takes one unit of time without consuming any resources. Parameter j captures the amount of time elapsed since the last invocation of task i. At the end of the period, $D[i,p_i]$ proceeds as $Start[i]$.

The process $Task[i]$ captures the behavior of task i. Higher-numbered tasks have higher priorities. Each $Task[i]$ begins by receiving the $start[i]$ event from the scheduler. After a task is started, it behaves like process $T[i,j]$, where parameter j captures the amount of time spent in execution on the processor.

When the processor is busy with a higher-priority task, $T[i,j]$ idles. When the processor is available, $T[i,j]$ executes on the processor for another time unit. Such alternative behaviors are represented by the "+" operator. When c_i time units of processing are accumulated, task i becomes ready to be started again. Note that, if the deadline of the task arrives before it completes its computation, it will not be able to synchronize, via an event $start[i]$, with the scheduler and the system will deadlock. This approach to specification of periodic processes is the basis of the ACSR-based schedulability analysis [Ben-Abdallah *et al.* 1996].

2.2 GCSR

A GCSR specification represents a system as a hierarchical collection of nodes. Edges connect the nodes that belong to the same process. Intuitively, the execution of a GCSR process proceeds from node to node along edges. A *compound* node may contain subprocesses within itself. Subprocesses execute concurrently and are visually delineated by means of *separators*. Edges cannot cross boundaries of compound nodes or separators, which enhances modularity of specifications. When execution enters a compound node, processes within the node are activated and execution proceeds in each process separately, subject to synchronization rules between processes. Compound nodes also provide for restriction of synchronization events and resource closure by means of attributes *Restrict* and *Close*, respectively.

All transitions between nodes are instantaneous and therefore can be labeled only with events or with conditions on index variables. Time can pass only within nodes of certain types. A *time-consuming* node, represented as an oval labeled with an action, denotes resource consumption by a sequential process. Such node can have only one outgoing edge labeled by the duration of the action. Time can also pass in a compound node, which requires all of its subprocesses to pass time as well. There is no special construct in GCSR to express alternative behaviors (similar to the choice operator of ACSR), but whenever a node has several outgoing edges, execution can proceed through any of them, thus achieving the same effect. The choice between a set of parameterized processes can be expressed by means of a special *initial marker*.

One node in each process has to be marked as initial. This is the node that execution first enters when the process is activated. An initial markers is also used to assign a name to a process and give the ranges of index variables. Different shapes of initial markers distinguish between sequential processes, generalized composition and generalized choice.

We illustrate GCSR constructs by showing the GCSR specification of the scheduling system described earlier. The specification is presented in Figure 2. The structure of the graphical specification closely follows that of the ACSR specification in Section 2.1. The top-level description is given by the process $GSystem$, which is a parallel composition of the scheduler process $GDispatch$ and three tasks $GTask[i]$. The dashed

boxes are *reference* nodes, allowing us to refer to a process by its name. As before, events $start[i]$ are local to the scheduling system, and the resource *cpu* is used exclusively.

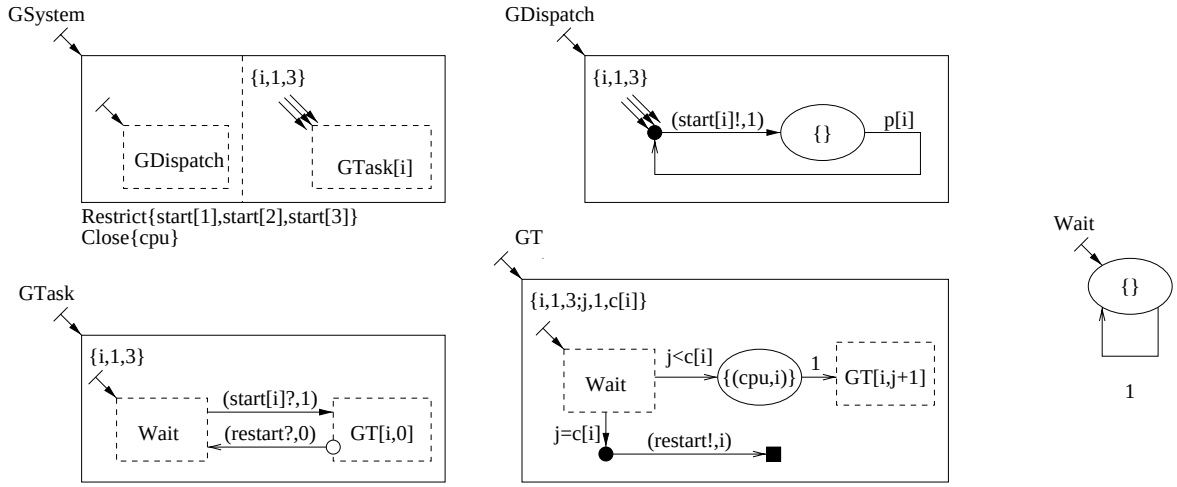

Figure 2: Graphical specification of the scheduling system

The process $GDispatch$ is, in turn, a parallel composition of three parameterized processes, each of which first sends a $start[i]$ event and then idles for p_i time units before repeating itself. The process $GTask$, parameterized by index i, represents the set of concurrent tasks. Each task initially waits (i.e., repeatedly idles) to be started by the scheduler, ready to accept the *start* event. After it receives the event, the task behaves as process GT until GT relinquishes control by sending the event *restart*. At this stage, $GTask$ returns to its initial state. It is important to notice that $GTask$ uses the event *restart* to synchronize not with a concurrent process, but with a subprocess of itself. We use a special type of edge to denote this situation. The typical use of this construct is to catch exceptions thrown by a subprocess, and therefore we call it an *exception* edge.

The process GT, parameterized with i, the task number and j, the amount of computation performed by the task, idles until it either reaches the completion of computation ($j = c_i$) or can continue its computation for one more time unit ($j < c_i$ and the resource *cpu* is not used by a higher-priority process). In the latter case, $GT[i, j]$ becomes $GT[i, j + 1]$. The process $Wait$, used by the processes $GTask$ and GT, represents idling and is equivalent to the ACSR process $X = \{\} : X$.

Note that although the ACSR and GCSR specifications are very close, one is not a translation of the other. In particular, the graphical specification uses an exception (*restart*), while the textual one does not. We demonstrate in Section 3 that the ACSR and the GCSR specifications are equivalent.

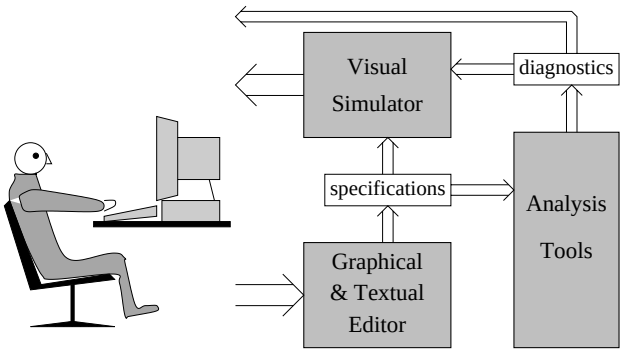

Figure 3: Structure of PARAGON toolset.

2.3 Relationship Between ACSR and GCSR

The operational semantics for ACSR and GCSR give rise to the same model for both languages. This allows one to mix ACSR and GCSR expressions together in the same specification. The semantic rules form the basis for translations from ACSR to GCSR and vice versa. Although it is not difficult to see the essence of these translations from the above example, we refer the reader to [Ben-Abdallah 1996; Ben-Abdallah *et al.* 1995] for the details of translations. It is shown there that an unparameterized ACSR specification, translated into GCSR and back, is strongly equivalent to the original one. The translation can be extended to parameterized specifications in a straightforward way.

3 THE PARAGON TOOLSET

To facilitate specification and verification via the ACSR and GCSR formalisms, we have developed the PARAGON toolset. Figure 3 shows the overall structure of the toolset, which consists of three main tools: the graphical/textual editor, the visual simulator, and the back-end analysis tool.

Specifications are constructed via the editor, and may be verified using the analysis tool or simulated using the simulator. The back-end analyzer performs analysis and presents the user with the outcome. If verification fails, the verification tool produces diagnostic information that is used to find the problem. The diagnostic information takes the form of a trace that can be used to drive the simulator, thus helping to locate the problem.

PARAGON allows the user to intermix graphical GCSR expressions with textual ACSR ones within the same specification. Since both specification languages have compatible formal semantics, one can use the two languages in a large specification as appropriate. It is known that a high-level description of a system can easily be represented graphically, which allows the user to see the overall interconnections between

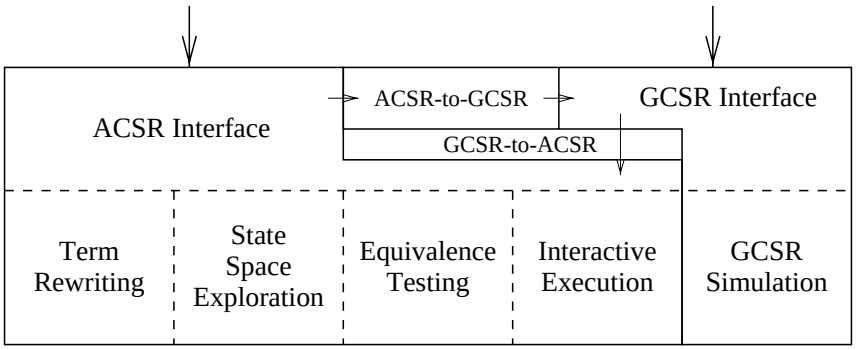

Figure 4: The functionality of PARAGON

components at a single glance. On the other hand, lower-level specifications can be very cumbersome in a graphical form. In that case, a textual specification may be preferable, both in terms of readability and the time it takes to construct the specification. Moreover, different users may have different personal preferences between graphical and textual specifications.

Figure 5 shows the GCSR editor displaying a part of the scheduler specification. The user interface of the editor consists of a drawing area and a number of buttons that allow the user to draw graphical objects and perform the standard editing operations. Menus at the top of the editor window provide capabilities to load and save specifications, open new editor windows, etc. In particular, the `Tools` menu, shown at right, offers to check consistency of the specification, translate the graphical specification into ACSR, and provides the connection to the back-end analysis tool. Other items in this menu allow the user to perform refinement of GCSR specifications. Details of GCSR refinement are given in [Ben-Abdallah 1996] and are beyond the scope of this presentation.

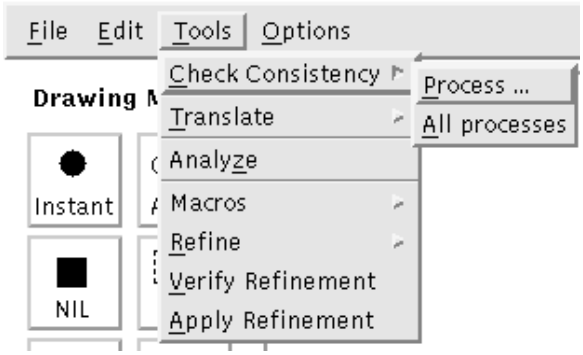

Figure 6: `Tools` menu

PARAGON uses XVERSA [D.Clarke *et al.* 1996] as the back-end analysis tool. XVERSA, which is an X windows interface to VERSA [Clarke *et al.* 1995], has been designed for analysis of ACSR specifications. GCSR specifications are exported into XVERSA by translating them into ACSR. Analysis functionality added by XVERSA is illustrated in Figure 4. State-space exploration routines allow users to perform reachability analysis and deadlock detection of PARAGON specifications. Equivalence checking performs comparison of two specifications using a variety of behavioral equivalences, including prioritized strong and weak bisimulations [Lee *et al.* 1994]. The term rewriting module is a simple theorem-proving environment

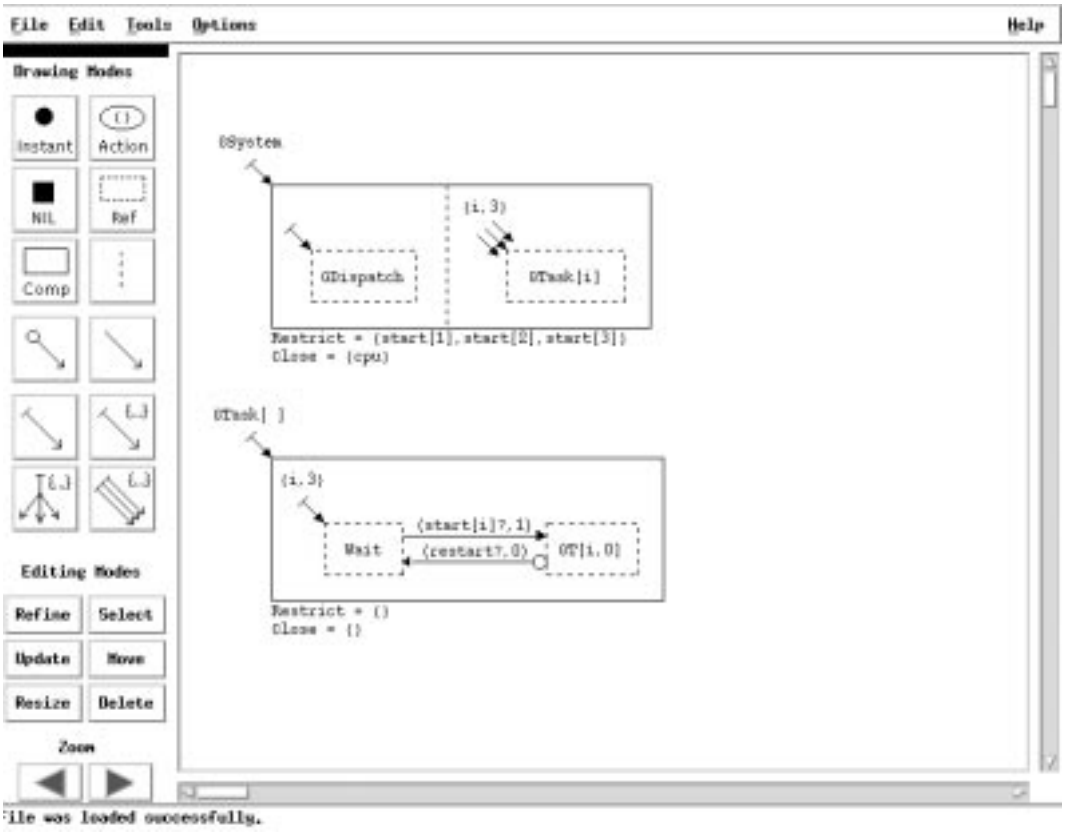

Figure 5: The GCSR Editor

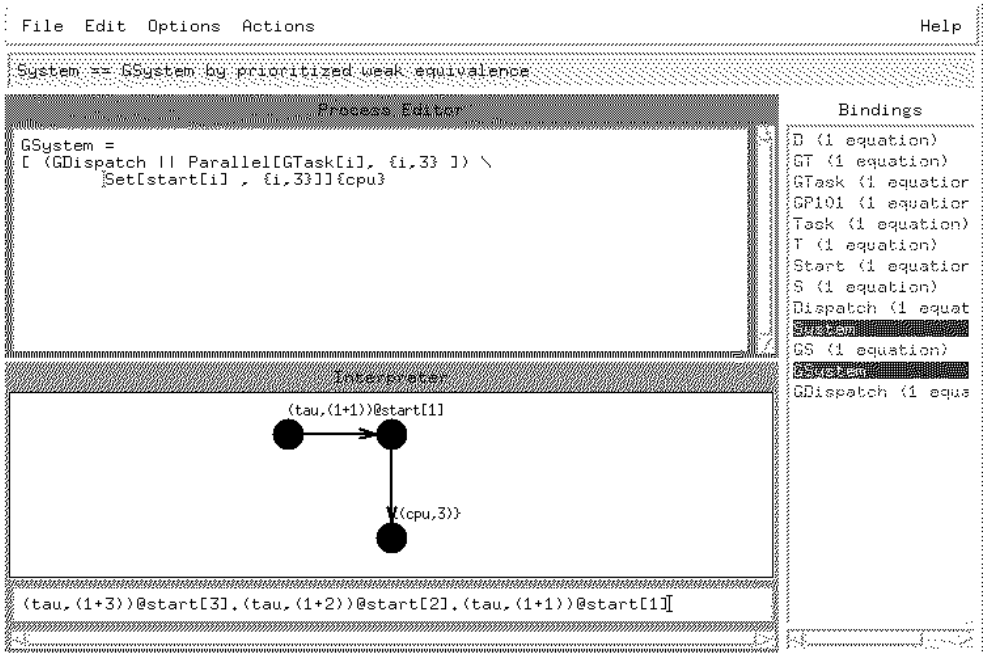

Figure 7: Verification tool XVERSA

for carrying out algebraic proofs about the specifications. Visual simulation of GCSR specifications and interactive execution of ACSR specifications provide for step-by-step manual exploration of the state space of a specification. In addition, the GCSR simulator provides a number of advanced simulation features such as the automatic simulation mode that allows users to set breakpoints.

Continuing the scheduler example, we demonstrate equivalence of the specification of Figure 1 with that of Figure 2. To do the comparison, the GCSR specification has been translated into ACSR. Figure 7 shows the user interface of XVERSA with both specifications loaded. The process window shows the translation of the GCSR process *GSystem*. In the list of process names (the Bindings window on the right), processes *System* and *GSystem* are selected for comparison. The message window at the top of the screen displays the result of analysis: the ACSR and GCSR specifications are equivalent.

4 ANALYSIS METHODOLOGY OF PARAGON

To make the best use of the capabilities of PARAGON, we developed a methodology for specification and analysis of large-scale real-time systems. The methodology summarizes our experience with specification of real-time systems in PARAGON, gained in several case studies. PARAGON was used for specification and analysis of the Production Cell benchmark [Lewerenz and T. Lindner 1995] in [Ben-Abdallah 1996], Sunshine ATM switch [Clarke and Lee 1996], redundancy management system of a reusable launch spacecraft [Sokolsky

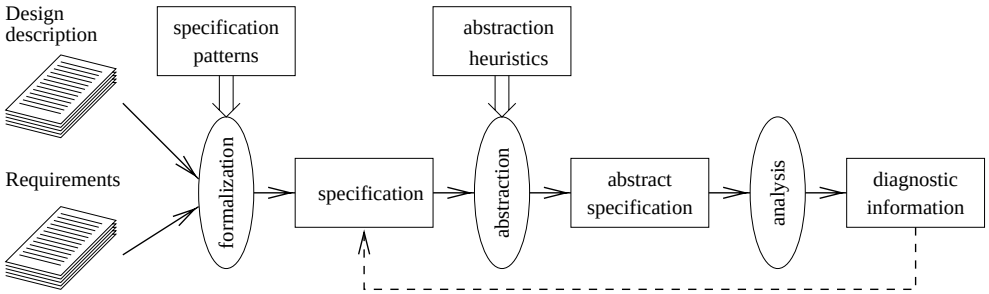

Figure 8: PARAGON methodology

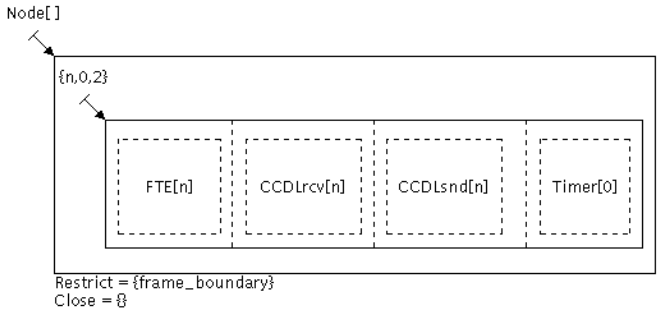

Figure 9: Structure of an RMS node

et al. 1998], schedulability analysis of the submarine sonar radar [Ben-Abdallah *et al.* 1996].

The main part of the methodology is a process for 1) constructing a formal specification of a system and its requirements from their informal descriptions, and 2) performing their analysis using a number of common approaches. In addition, the methodology includes a set of patterns that commonly occur in specifications, together with a set of abstraction heuristics. The overall view of the PARAGON methodology is presented in Figure 8, and is described in detail in Section 4.1.

In this section, we illustrate the methodology using a recent case study. The case study analyzes a part of the redundancy management system (RMS) of a spacecraft. The results of the case study are described in [Sokolsky *et al.* 1998]; here we only use fragments of the RMS specification as examples. The RMS was designed and implemented by AlliedSignal, Inc., and has been extensively tested. The purpose of the RMS is to ensure tolerance of any single fault. Our specification is based on the original list of informal requirements for the RMS and the pseudo-code that was used in the RMS design. The RMS consists of three identical *nodes*. Each RMS node contains a Fault-Tolerant Executive (FTE) and a Cross Channel Data Link (CCDL) sender and receiver used to broadcast the information to other RMS nodes. The structure of an RMS node is shown in Figure 9. The FTE is a cyclic executive that repeats its execution every *frame*, according to a timer local to the node.

4.1 The Process

Specification and analysis is performed based on the documentation provided by the customer. The documentation usually includes a set of correctness requirements and a description of the system design. The system description can be high-level or detailed, and may take the form of design diagrams or pseudo-code of the system. Requirements are usually represented informally, using English language. They describe the set of properties that the system has to have in order to be considered correct. We have found that formalization of requirements is the most time-consuming part of the specification process, which involves extensive communication with developers of the system until requirements are precise enough to be transformed into a formal representation. The graphical notation of GCSR facilitates the communication since it shields the customer from the mathematical foundation; additionally, the simulator of PARAGON allows one to animate a specification for better understanding. A useful side effect of the requirement formalization process is that requirements become more complete by bringing out implicit assumptions made by the developers.

The methodology of PARAGON presumes separate roles for the customer, an engineer who designs the system, and the expert who is proficient in formal methods but not necessarily in the specific problem domain. It is desirable to bring these two roles closer. Undoubtedly, this will happen in the future. Much research is being conducted to make formal methods more intuitive to engineers [Lee and Sokolsky 1997; Nelken and Francez 1996]. However, given today's state of technology, an expert in formal methods is necessary in most realistic cases as a mediator between the engineer and the tool.

The specification of a system is constructed by the expert and the customer working in close cooperation. The structure of the specification is driven by the structure of the design, but the translation is often far from mechanical. The major role of the expert is to devise suitable abstractions to be applied to the specification. Abstractions should, on the one hand, be effective enough to reduce the size of the specification so that it can be analyzed by the PARAGON tools. On the other hand, abstractions have to preserve the properties expressed by the system requirements. Abstractions are covered in more detail in Section 4.3 using examples from the RMS case study.

The main difficulty of the specification process is deciding which abstractions to use. The customer may not be familiar with the details of abstraction, but she has the intimate knowledge of the system behavior that would allow the team to estimate which abstractions would be effective and safe to use. The specification process includes extensive interaction between the expert and the customer, which helps to transfer this knowledge to the expert.

The two most common methods of analysis in PARAGON are state-space exploration and equivalence checking. State-space exploration usually takes the form of deadlock detection, which is invaluable

in detecting synchronization and scheduling errors in the system design. More importantly, verification of safety properties can be reduced to deadlock detection by employing tester processes. Specification of such a tester is one of the patterns described below. A tester process is composed in parallel with the system. The effect is as if the tester process is observing the behavior of the system and issues a special failure event or forces a deadlock when a violation of the property is detected. The composite specification is given as input to the analysis tool, which looks for reachability of the failure event, or a deadlock induced by the tester. In case of a failure, the analysis tool produces an execution trace leading to the undesirable event or a deadlocked state. This trace can be run through the simulator, which visualizes the execution and helps the user to discover the source of the problem.

This testing approach is useful if the requirements of the system are given in the form of a collection of properties that the system has to satisfy, or constraints on the system behavior. On the other hand, if the requirements are in the form of high-level system behaviors, the requirements and the system specification can be directly compared to each other. In this case, equivalence checking is more appropriate. PARAGON supports verification of a number of well-known process equivalences, extended to take priorities into consideration. See [Lee *et al.* 1994] for details about these equivalences.

4.2 Specification Patterns

Specification patterns serve the same purpose in crafting a formal description of a system as design patterns do in the implementation process. They include fragments of specifications that one can reuse. One such pattern has already been introduced in Section 2 when we presented the scheduling example. The ACSR process $Task$ and GCSR process $GTask$ represent a periodic task that commonly occurs in real-time systems. Variations of this pattern may include other parameters such as release jitter or additional resource requirements, but the overall structure remains the same.

Other patterns discussed in this section address the representation of data storage and manipulation, and the use of preemption in specifications. The last pattern is used during analysis of specifications and presents a common way to express safety properties of a specification.

Representation of data. Most systems use variables to store data, and exchange data values between subsystems. Data variables and value-passing are modeled in PARAGON specifications using parameterization. There are two ways to represent data. The first includes the current value of a variable as a parameter to the process definition. An example of such parameterized specification is given by the simplified version

of the hardware timer model contained in each RMS node:

$$\begin{aligned} Timer[tick] &= \{\} : Timer[(tick+1)\%tmax] \\ &+ (\overline{local\_time}[tick], 0).Timer[tick] \\ &+ \texttt{if } tick = 0 \texttt{ then } (\overline{frame\_b}, 0).Timer[tick] \quad \{tick \in 0..tmax-1\} \end{aligned}$$

Here, % denotes the modulo integer operator, and $tmax$ represents the number of timer ticks in a frame. Index variable $tick$ ranges from 0 to $tmax - 1$. The timer reports the value of local time to the node, and signals a frame boundary with the *frame_b* signal at the beginning of the frame.

The other approach, which is commonly used in process algebras without value passing capabilities, introduces a separate process for each variable in the system. This process holds the current value of the variable and communicates it to other processes that need the value. These other processes may also request to change the current value. For example, the following process is used in the RMS specification to hold the timestamp of an incoming CCDL message from node c:

$$\begin{aligned} Tstamp[c, v] &= \{\} : Tstamp[c, v] \\ &+ (\overline{getStamp}[c, v], 0).Tstamp[c, v] \\ &+ \Sigma_n \ (setStamp[c, n], 0).Tstamp[c, n], \end{aligned}$$

where v (the current value) and n (the new value) are drawn from the range of admissible timestamp values. Operator Σ_n is the generalized choice operator, allowing the process to accept any of $\overline{setStamp}[c, v]$ events and thus to assign the new value v to the variable. Priorities of events in such a process are normally set to 0 so that concurrent accesses to a variable by several processes can be governed based solely on priorities of the processes.

The first of the two approaches represents a variable local to the process $Timer$, which can be only read by other processes. In the second case, $Tstamp$ is shared between several processes.

The use of preemption. The notion of preemption between events and actions is primarily used to arbitrate resource access between processes. However, preemption can also be used to express other aspects of the system, which are usually difficult to express with other process-algebraic approaches.

For example, priorities allow us to test for the presence of an event and provide an alternate behavior for the case of its absence. In most other process algebras, one cannot do such test without explicitly introducing an additional event to denote the absence of the first one. With ACSR, on the other hand, we can test for an event directly. Consider the specification fragment in Figure 10. It shows a process that will interact with P via the event *ready* when P can do so, or else resort to some default activity instead. The event *ready* that is used to signal the availability of P is restricted. This means that if P does not offer the matching event, the only possibility for progress is the internal τ event that leads to process *Default*. If P

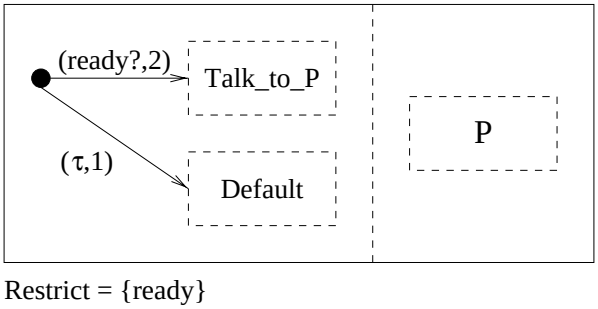

Figure 10: Test for an event

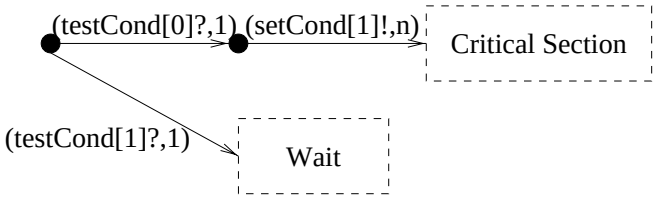

Figure 11: Atomic execution

does offer the *ready* event, then synchronization occurs, resulting in a τ event with a higher priority that will preempt the default event.

Priorities can also be used to ensure atomic executions of a sequence of events. For example, consider the implementation of the *test-and-set* operation in Figure 11. The process tests the value of a boolean condition variable by means of events $testCond[0]$ and $testCond[1]$. If the variable is already set, the process is suspended (enters a $Wait$ state). Otherwise, it sets the value of the condition to 1 and proceeds into the critical section. It is important that test and set operations are performed atomically. In order to express atomicity in the specification, we set the priority of event $setCond[1]$ to n, where n is higher than any intervening event. This way, once the process performs event $testCond[0]$, $setCond[1]$ will preempt all other events and the atomicity will be preserved. Without priorities, one has to explicitly include locking mechanisms into the specification in order to ensure atomicity of execution. These locking mechanisms are usually specified as separate processes, which obscure the specification.

Testers for safety requirements. The purpose of formal analysis is to verify the specification with respect to its requirements. Most requirements in real-time systems are safety properties. Moreover, requirements are often expressed in terms of some required sequential behavior. In PARAGON, such requirements can be verified via a testing approach. Requirements are specified as tester processes that are composed in parallel with the system specification. A tester process for a requirement is a sequential process that watches system behaviors for potential violations of the requirement, and signals a failure if it detects such a

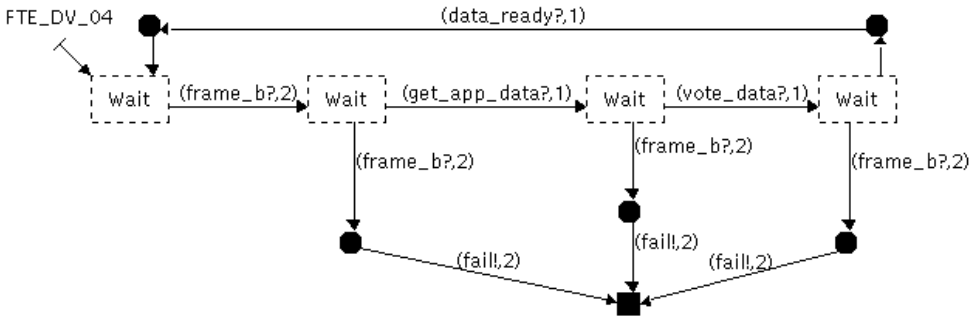

Figure 12: Tester for a safety requirement

violation. As an example, consider the requirement that an RMS node must collect application data at each frame boundary, vote on the data, and signal the availability of the voted data before the end of the current frame. The tester process for this requirement is shown in Figure 12. A *fail* event is issued by the tester if a frame boundary event *frame_b* occurs before the prescribed sequence of events is completed. A general automata-based approach to construct testers for safety properties is discussed in [Alpern and Schneider 1989]. Another approach that generates a tester automatically from a temporal logic formula is presented in [Aceto *et al.* 1998].

4.3 Abstraction Heuristics

This section presents various abstractions that are commonly used in PARAGON and illustrates them with examples from the RMS specification. The RMS is a complex system, whose behavior heavily depends on the data exchanged between the nodes. Analysis of the RMS in PARAGON would be impossible without abstractions because of the large state space.

A number of abstractions can be applied during the specification of a system. Whether or not the abstraction is safe to be used often depends on a specific property to be checked. For example, if an abstraction does not preserve the temporal information, it is unsafe to use if we need to verify a time-sensitive property of the system. However, property-dependent abstractions can greatly reduce the size of the state space of a specification. The drawback of this property-dependent approach is that each property is verified against a different abstract specification and it is difficult to maintain consistency between several versions of the specification, which inevitably change during the analysis process. To reduce the impact of this drawback, we employ parameterization in our specifications, so that we could switch between different abstract specifications by changing parameters.

Some abstractions are precise with respect to a property in the sense that the property is true in

the detailed specification if and only if it is true in the abstract specification. Others are conservative approximations. That is, when the property is true in the abstract specification, it is also true in the detailed one. The converse, however, may not hold. In these cases, when verification fails, it is the job of the expert to tell whether the failure is due to a design fault, or is induced by the abstraction.

Most abstractions are aimed at reducing the number of states and transitions in the model of a specification. The remaining abstractions are used to reduce the size of the internal representation of each state in the labeled transition system. Below, we give a summary of the most commonly used heuristics. The presented abstractions are not specific to ACSR/GCSR specifications and can be employed by most other analysis tools.

Limiting the ranges of data variables. This abstraction is precise if the maximum value of a variable can be decided statically. Otherwise, this abstraction may not be safe to use. Fortunately, the analysis tool can help us make the abstraction precise. We can use reachability analysis to see if events representing assignment of values that exceed the range of the variable ever occur in the specification. If no such events are reachable, then the abstraction is safe to use. An example of this reduction is the variable $Tstamp$ for recording the time stamps of incoming CCDL messages, described in the previous section. The RMS synchronization algorithm considers only messages arriving within a certain time interval called the *error window*. Therefore, we do not need to record all possible timestamps. Instead, we can limit ourselves to values within the window plus one additional value to represent all timestamp values outside the window.

An extreme case of this abstraction is when, instead of the value of a variable, we record the value of an expression on this variable. For example, let the process input some value v from a channel and then test, possibly much later, whether v is less then 5. In this case, instead of storing the value in a variable that would require the full range of possible values, we can store the value of $v < 5$.

Minimize the number of variables. This abstraction is similar to register minimization performed by most compilers. When we are specifying a system that has two variables, which are always used disjointly, both variables can be combined in a single variable. The range of the new variable is the larger of the two ranges. Consider the voting procedure conducted by each of the RMS nodes. A simplified version of it for node n can be represented as follows:

$$Voting[n] = (\Pi_m Data[m] \| DoVote \| Result) \backslash \{data[k, v], result[v]\}$$

where n, m, k range over the set of nodes ($\{0,1,2\}$) and v is drawn from the set of admissible values for the voted data item. Each $Data[i]$ process represents the value received from node i (including $i = n$). Process $DoVote$ uses events $data$ to collect the values of each variable, and then event outputs $result$ to record the

outcome of voting. One can notice that the values held by $Data[i]$ processes are not used again after voting, and the new values are not received until the next frame, when the result of voting is no longer important. This allows us to use $Data[n]$ to hold the result of voting and dispose of $Result$ process:

$$Voting2[n] = (\Pi_m Data[m] \| DoVote) \backslash \{data[m, v]\}$$

The effect of this abstraction is twofold. On the one hand, it reduces the number of states generated by the algorithm. Indeed, consider the state of the specification after the value of $Data[n]$ is updated but before voting is carried out. At that moment, $Result$ contains a value from the previous frame, and each combination of values of the two variables will give rise to a distinct state, which are equivalent to each other with respect to any property that we may expect to check. $Voting2$ collapses all such states together. On the other hand, the reduced specification contains one parallel process fewer. Therefore, the abstraction reduces the size of the internal representation of each state, which depends on the number of parallel processes in the respective ACSR term. When the number of states is large, this reduction will amount to substantial memory savings. The abstraction is always precise.

Partition the specification according to the property. Often a requirement concerns only a part of the whole system. Many of the RMS requirements describe the properties of a single node. For example, the requirement shown in Figure 12 specifies behavior of one node regardless of the behavior of the other nodes. Therefore, we can consider description of one node in isolation, which considerably reduces the state space. We refer to this kind of partitioning as the *vertical* partitioning of the system.

Sometimes, a requirement assumes that all components of the system are in a specific functional mode. In that case, a reduced specification may be constructed to reflect only the required mode, which we call the *horizontal* partitioning. Some of the RMS requirements that deal with fault detection and synchronization between nodes assume that the nodes are in "steady state" mode. Therefore, the specification that we used to analyze these properties did not contain any of the "startup" modes.

Horizontal partitioning not only reduces the state space of the specification, but also makes the tester processes for the properties simpler. Figure 13 shows a fragment of the tester process for monitoring the skew between two nodes, used in analysis of synchronization requirements. The tester collects the values of local time from the two nodes every 20 time units and checks whether the limit on clock skew has been exceeded. Event $fail$ is sent when the limit is exceeded. In general, this process has to check that each node is in the steady state. Here, the reduced version omits the check since all nodes are in the steady state by construction.

Coarse-grain time steps. PARAGON specifications are based on a discrete time model. Each time-consuming transition represents time being incremented by one tick of an implicit global clock. When we

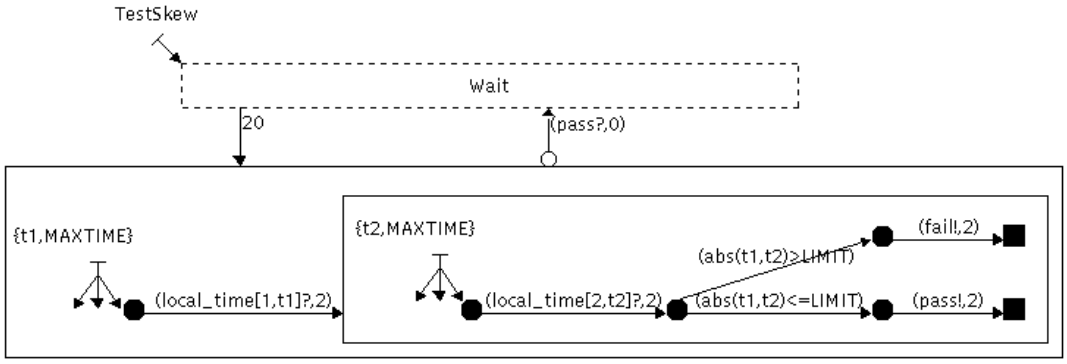

Figure 13: Tester process for testing clock skew

have to model absolute time values as was the case with the RMS specification, we have to assume the value of this unit of time in the specification. In the RMS specification, this value determines $tmax$, the number of ticks of the implicit clocks in one frame, and a number of other parameters, such as the limit on clock skew between nodes used in the tester process *TestSkew* of Figure 13. Clearly, these parameters significantly affect the size of the state space of the specification.

In general, increasing the tick value is not safe. In the case of the RMS, however, behavior of the system consists of a number of operations that can be treated as instantaneous for the tick values we considered. These operations are separated by delays that are significantly larger than one tick. Therefore, in this case the abstractions were precise for all variations of the tick value. As part of our on-going research, we are currently investigating general conditions that make this heuristic safe to use.

Reducing complex computations to non-determinism. When a complex computation produces a small set of values, substantial savings may be gained by abstracting away the details of the computation and producing the same values non-deterministically. This approach has been used in the RMS specification when checking the requirement FTE_DV_04 shown in Figure 12. Indeed, with respect to the property, the result of voting is irrelevant as long as some data is offered to the application. This abstraction is a conservative approximation of the actual behavior of the system, since it loses the information about which input produces which output. As a result, the abstract specification may exhibit some spurious behaviors. For example, in the RMS specification, voting may produce an error for perfectly good data. In our experience, this approach works best when analyzing rather "static" properties, such as FTE_DV_04.

5 RELATED WORK

Real-time specification. Specification of real-time systems can be carried out using a variety of formalisms. Several real-time process algebras exist, with both discrete-time [Moller and Tofts 1990; Hennessy and Regan 1991] and dense-time semantics [Reed and Roscoe 1988; Nicollin *et al.* 1991; Yi 1991]. The automaton model has been extended for real time in [Alur and Dill 1994]. A number of real-time logics are used to specify systems, for example [Jahanian and Mok 1986; Ramakrishna *et al.* 1996].

Tools. There is a wide variety of tools for formal specification and analysis of systems. Tools range from fully automatic verifiers for finite-state systems to theorem-provers based on higher-order formalisms. Among the most widely used automatic tools are the Concurrency Workbench [Cleaveland *et al.* 1993], Spin [Holzmann 1991], SMV [McMillan 1993]. Theorem provers such as PVS [Owre *et al.* 1992] often can handle larger classes of systems, but usually require more involvement from formal methods experts for successful application.

Many of the tools support visual specification languages, which make specifications easier to read and more intuitive for engineers. The design of GCSR was, to some extent, influenced by Statecharts [Harel 1987]. Specification of systems using Statecharts is supported by StateMate [Harel *et al.* 1990] environment. ModeChart [Mok *et al.* 1996] extends Statecharts for the specification of timing constraints and is supported by the toolset called MT. Concurrency Factory [Cleaveland *et al.* 1994] employs a visual specification language with process-algebraic semantics.

Reduction techniques. The importance of abstraction has been long understood by the formal methods researchers. In [Daws and Tripakis 1998], a set of abstractions specific to the timed-automata framework is described. Two abstractions for SCR requirement specifications are presented in [Bharadwaj and Heitmeyer 1997]. The advantage of these abstractions is that they can be computed automatically.

The use of non-determinism to avoid detailed specification of complex computations is an important part of COSPAN analysis methodology [Kurshan 1994]. In this methodology, non-determinism in the specification is gradually resolved, making specification more and more precise until the desired property is proved or a counterexample is found (or the specification becomes too large to analyze). A related technique for analysis of real-time systems [Alur *et al.* 1993] allows one to start with a simplified specification that ignores timing information and gradually add more and more of the timing constraints.

Partial-order reductions [Godefroid and Wolper 1993; Peled 1996] is another successful approach for managing the state-space explosion. The knowledge of the structure of a distributed system can be used to

rule out some interleavings of events during execution. This approach is different from abstraction in that reduction is applied during analysis, rather than during specification.

Run-time reductions have also been considered in the context of model checking for timed automata [Alur *et al.* 1992; Sokolsky and Smolka 1995]. The algorithms combine property-dependent minimization with model checking.

6 CONCLUSIONS

We have presented an overview of the CSR specification paradigm and its supporting toolset PARAGON. The paradigm provides for modular easy-to-maintain specifications of distributed real-time systems in both visual and textual formats. PARAGON features a collection of tools that operate on these specifications and allow users to analyze and refine them. This paper summarizes our experience with constructing and analyzing specifications using PARAGON. This experience supplements the description of the toolset and helps the user use it effectively. We have discussed various abstraction heuristics that are commonly used during analysis to be able to handle larger systems. The specification and analysis paradigm of PARAGON has been tested in a variety of case studies and proved itself scalable enough to handle real-life designs.

We are working to improve PARAGON further and make it scale to handle even larger systems. One of the drawbacks of the current implementation is the way parameterization is handled. Parameterized specifications are often used to model exchange of data between processes. Parameterization is a very expensive way to achieve this effect, since all analysis is performed by PARAGON at the ground level. The solution to this inefficiency is to introduce value-passing capabilities directly into the CSR specification paradigm, as we have in a value-passing process algebra ACSR-VP [Kwak *et al.* 1998]. An implementation of symbolic analysis algorithms for ACSR-VP specifications is currently under way.

Acknowledgments. This research was supported in part by NSF CCR-9415346, NSF CCR-9619910, AFOSR F49620-95-1-0508, AFOSR F49620-96-1-0204, ONR N00014-97-1-0505 (MURI), ARO DAAG55-98-1-0393, ARO DAAG55-98-1-0466. We would like to thank anonymous referees for their insightful comments.

REFERENCES

Aceto, L., A. Burgueño, and K. G. Larsen (1998), "Model Checking via Reachability Testing for Timed Automata," In *Proceedings of TACAS '98*, LNCS 1384, pp. 263–280.

Alpern, B. and F. B. Schneider (1989), "Verifying Temporal Properties without Temporal Logic," *ACM Transactions on Programming Languages and Systems 11*, 1, 147–167.

Alur, R., C. Courcoubetis, N. Halbwachs, D. Dill, and H. Wong-Toi (1992), "Minimization of Timed Transition Systems," In *Proceedings of CONCUR'92*, LNCS 630.

Alur, R. and D. L. Dill (1994), "The Theory of Timed Automata," *Theoretical Computer Science 126*, 2.

Alur, R., A. Itai, R. Kurshan, and M. Yannakakis (1993), "Timing Verification by Successive Approximation," In *Proceedings of CAV '92*, LNCS 663, Springer-Verlag.

Ben-Abdallah, H. (1996), "Graphical Communicating Shared Resources: A Language for The Specification, Refinement and Analysis of Real-Time Systems," Ph.D. thesis, Department of Computer and Information Science, University of Pennsylvania.

Ben-Abdallah, H., Y.-S. Kim, and I. Lee (1996), "Schedulability and Safety Analysis in GCSR," In *Proceedings of WORDS '96: IEEE 2nd International Workshop on Object-oriented Real-time Dependable Systems*.

Ben-Abdallah, H., I. Lee, and J.-Y. Choi (1995), "A Graphical Language with Formal Semantics for the Specification and Analysis of Real-Time Systems," In *Proceedings of IEEE Real-Time Systems Symposium*, IEEE Computer Society Press.

Ben-Abdallah, H., I. Lee, and O. Sokolsky (1997), "Operational Semantics for Visual Simulation in PARAGON," In *Proceedings of IEEE National Aerospace and Electronics Conference*.

Bengtsson, J., K. G. Larsen, F. Larsson, P. Petersson, and W. Yi (1995), "Uppaal - a Tool Suite for Automatic Verification of Real-Time Systems," In *Proceedings of the 4th DIMACS Workshop on Verification and Control of Hybrid Systems*.

Bergstra, J. and J. Klop (1985), "Algebra of Communicating Processes with Abstraction," *Journal of Theoretical Computer Science 37*, 77–121.

Bharadwaj, R. and C. Heitmeyer (1997), "Verifying SCR requirements specifications using state exploration," In *Proceedings of* 1^{st} *ACM SIGPLAN Workshop on Automatic Analysis of Software*.

Brémond-Grégoire, P., J. Choi, and I. Lee (1997), "A Complete Axiomatization of Finite-state ACSR Processes," *Information and Computation 138*, 2, 124–159.

Clarke, D. and I. Lee (1996), "Testing-Based Analysis of Real-Time System Models," In *Proceedings of International Test Conference*.

Clarke, D., I. Lee, and H.-L. Xie (1995), "VERSA: A Tool for the Specification and Analysis of Resource-Bound Real-Time Systems," *Journal of Computer and Software Engineering 3*, 2.

Clarke, E. M. and J. M. Wing (1996), "Formal Methods: State of the Art and Future Directions," *ACM Computing Surveys 28*, 4, 626–643.

Cleaveland, R., J. N. Gada, P. M. Lewis, S. A. Smolka, O. V. Sokolsky, and S. Zhang (1994), "The Concurrency Factory – Practical Tools for Specification, Simulation, Verification and Implementation of Con-

current Systems," In *Proceedings of the DIMACS Workshop on Specification Techniques for Concurrent Systems, Princeton, NJ.*.

Cleaveland, R., J. Parrow, and B. Steffen (1993), "The Concurrency Workbench: A Semantics-Based Tool for the Verification of Concurrent Systems," *ACM TOPLAS 15*, 1.

Cleaveland, R. and S. A. Smolka (1996), "Strategic Directions in Concurrency Research," *ACM Computing Surveys 28*, 4, 607–625.

Daws, C., A. Olivero, S. Tripakis, and S. Yovine (1995), "The Tool KRONOS," In *Proceedings of Workshop on Hybrid Systems and Autonomous Control*, volume 1066 of *LNCS*, Springer-Verlag, pp. 208–219.

Daws, C. and S. Tripakis (1998), "Model Checking of Real-Time Reachability Properties Using Abstractions," In *Proceedings of TACAS '98*, LNCS 1384, pp. 313–329.

D.Clarke, H. Ben-Abdallah, I. Lee, H.-L. Xie, and O. Sokolsky (1996), "XVERSA: an integrated graphical and textual toolset for the specification and analysis of resource-bound real-time systems," In *Proceedings of Computer-Aided Verification '96*, number 1102 In LNCS, Springer-Verlag, pp. 402–405.

Gerber, R. (1991), "Communicating Shared Resources: A Model for Distributed Real-Time Systems," Ph.D. thesis, Department of Computer and Information Science, University of Pennsylvania.

Gerber, R. and I. Lee (1994), "A Resource-Based Prioritized Bisimulation for Real-Time Systems," *Information and Computation 113*, 1, 102–142.

Godefroid, P. and P. Wolper (1993), "Using Partial Orders for the Efficient Verification of Deadlock Freedom and Safety Properties," *Formal Methods in System Design 2*, 2, 149–164.

Hardin, R., Z. Har'El, and R. Kurshan (1996), "COSPAN," In *Proceedings of CAV '96*, pp. 423–427.

Harel, D. (1987), "Statecharts: A Visual Formulation for Complex Systems," *Science of Computer Programming 8*, 3, 231–274.

Harel, D. *et al.* (1990), "STATEMATE: A Working Environment for the Development of Complex Reactive Systems," *IEEE Transactions on Software Engineering 16*, 4, 403–414.

Hennessy, M. and T. Regan (1991), "A Process Algebra for Timed Systems," Technical Report 5/91, CS, University of Sussex.

Hoare, C. A. R. (1985), *Communicating Sequential Processes*, Prentice Hall Intl.

Holzmann, G. (1991), *Design and Validation of Computer Protocols*, Prentice Hall.

Jahanian, F. and A. K. Mok (1986), "Safety Analysis of Timing Properties in Real-Time Systems," *IEEE Transactions on Software Engineering 12*, 9, 890–904.

Kurshan, R. P. (1994), *Computer-aided verification of coordinating processes : the automata-theoretic approach*, Princeton University Press.

Kwak, H.-H., I. Lee, A. Philippou, J.-Y. Choi, and O. Sokolsky (1998), "Symbolic Schedulability Analysis

of Real-time Systems," In *Proceedings of IEEE Real-Time Systems Symposium*, IEEE Computer Society Press.

Lee, I., P. Bremond-Gregoire, and R. Gerber (1994), "A Process Algebraic Approach to the Specification and Analysis of Resource-Bound Real-Time Systems," *Proceedings of the IEEE 82*, 1.

Lee, I. and O. Sokolsky (1997), "A Graphical Property Specification Language," In *Proceedings of 2nd IEEE Workshop on High-Assurance Systems Engineering*, IEEE Computer Society Press.

Lewerenz, C. and E. T. Lindner (1995), *Formal Development of Reactive Systems: Case Study Production Cell*, volume 891 of *LNCS*, Springer-Verlag.

McMillan, K. L. (1993), *Symbolic Model Checking: An Approach to the State Explosion Problem*, Kluwer Academic Publishers.

Milner, R. (1989), *Communication and Concurrency*, Prentice Hall Intl.

Mok, A., D. Stuart, and F. Jahanian (1996), "Specification and Analysis of Real-Time Systems: Modechart Language and Toolset," In *Formal Methods for Real-Time Computing*, C. Heitmeyer and D. Mandrioli, Eds., J. Wiley and Sons, pp. 33–54.

Moller, F. and C. Tofts (1990), "A Temporal Calculus of Communicating Systems," In *Proceedings of CONCUR'90*, LNCS 458.

Nelken, R. and N. Francez (1996), "Automatic Translation of Natural Language System Specifications into Temporal Logic," In *Proceedings of CAV '96*, LNCS 1102, Springer-Verlag, pp. 360–371.

Nicollin, X., J.-L. Richier, J. Sifakis, and J. Voiron (1991), "ATP: an Algebra for Timed Processes," In *Real-Time: Theory in Practice*, J. de Bakker *et al.*, Ed., Proceedings of REX Workshop, LNCS 600.

Owre, S., J. Rushby, and N. Shankar (1992), "PVS: A Prototype Verification System," In *Proceedings of CADE '92*, volume 607 of *Lecture Notes in Artificial Intelligence*, pp. 748–752.

Peled, D. (1996), "Combining Partial Order Reductions with On-the-fly Model Checking," *Formal Methods in System Design 8*, 39–64.

Ramakrishna, Y. S., P. M. Melliar-Smith, L. E. Moser, L. K. Dillon, and G. Kutty (1996), "Interval logics and their decision procedures. Part II: A real-time interval logic," *Theoretical Computer Science 170*, 1–2, 1–46.

Reed, G. M. and A. W. Roscoe (1988), "A Timed Model for Communicating Sequential Processes," *Theoretical Computer Science 58*.

Sokolsky, O. and S. A. Smolka (1995), "Local Model Checking for Real-Time Systems," In *Proceedings of CAV '95*, P. Wolper, Ed., LNCS 939, pp. 211–224.

Sokolsky, O., M. Younis, I. Lee, H.-H. Kwak, and J. Zhou (1998), "Verification of the Redundancy Management System for Space Launch Vehicle," In *Proceedings of 1998 Real-Time Technology and Applications*

Symposium., IEEE Computer Society Press, pp. 220–229.

Yi, W. (1991), "A Calculus of Real Time Systems," Ph.D. thesis, Chalmers University of Technology.