Specification and Analysis of Resource-Bound Real-Time Systems

MS-CIS-91-96
LOGIC & COMPUTATION 43
DISTRIBUTED SYSTEMS LAB 9

Richard Gerber
Insup Lee

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

November 1991

Specification and Analysis of Resource-Bound Real-Time Systems*

Richard Gerber
Dept. of Computer Science
University of Maryland
College Park, MD 20742
rich@cs.umd.edu

Insup Lee
Dept. of Computer and Info. Science
University of Pennsylvania
Philadelphia, PA 19104
lee@cis.upenn.edu

Abstract. We describe a layered approach to the specification and verification of real-time systems. Application processes are specified in the CSR application language, which includes high-level language constructs such as timeouts, deadlines, periodic processes, interrupts and exception-handling. Then, a configuration schema is used to map the processes to system resources, and to specify the physical communication links between them. To analyze and execute the entire system, we automatically translate the result of the mapping into the CCSR process algebra. CCSR characterizes CSR's resource-based computation model by a priority-sensitive, operational semantics, which yields a set of equivalence-preserving proof rules. Using this proof system, we perform the algebraic verification of our original real-time system.

Keywords: Real-time, specification, configuration, verification, proof systems, process algebras, programming languages.

Contents

*This research was supported in part by ONR N00014-89-J-1131 and NSF CCR-9014621.

1 Introduction

Once strictly the province of assembly-language programmers, real-time computing has developed into an important area of research. This is a welcome sign, since the practice of building real-time systems is still dominated by the use of arcane and *ad hoc* techniques. As a step toward redressing this problem, there has recently been a spate of progress in the development of real-time formal methods. Much of this work has fallen into the traditional categories of untimed systems – for example, temporal logics (as in [14, 1]), assertional methods (as in [16, 7]), net-based paradigms (as in [10, 3]) and process algebras (as in [15, 17]).

In this paper we address the problem of shared resources in real-time systems. As in [9], we model a real-time system not only by its functionality and timing constraints, but also as a collection of one or more shared resources. Each resource is inherently sequential in nature; that is, a resource only has the capacity to execute a solitary event at any time. This constraint quite naturally leads to an interleaving notion of concurrency at the resource level of the system, where we assume that a priority ordering is used to arbitrate between simultaneous resource requests. At the system level, true parallelism occurs when a group of resources are executed simultaneously. In such an environment, various factors can influence the real-time behavior of a system: the number of resources, their timing characteristics, their ordering of priorities, the connectivity between them, and the processes hosted on them.

To study the subtle interplay between these factors, we have developed a two-tiered framework called Communicating Shared Resources (or CSR). At the top layer is the CSR specification language which is used to describe the functional aspects of a real-time system as well as its resource requirements. The specification language consists of an application language and a configuration language. The application language possesses high-level constructs to specify communication primitives, timeouts, delays, interrupts, deadlines, periodic processes and exceptions. The configuration language is used to define the *structure* of the complete system. A configuration schema maps application processes to system resources, specifies the topology of interconnection network, replicates system components and declares priorities. The bottom layer is the Calculus for CSR (or CCSR), which is a process algebra based on a computation model that captures the notions of priority and resource. An automatic translator accepts the application and configuration components of a CSR specification, and then translated them into a CCSR term. The correctness of the CSR specification is verified using the proof system of CCSR.

This paper is organized as follows: In Section 2.1, we present an overview of the CSR Specification Language, with both its syntax and its informal semantics. We proceed to show how a resource-constrained, real-time system can be assembled using a configuration schema. Section 3 is devoted to the CCSR process algebra – its execution model, semantics and the proof system that it yields. In Section 4 we show how a CSR specification and a configuration schema is translated into CCSR, and in Section 5 we show how the CCSR proof rules can then be used to verify that the original CSR program is correct.

2 The CSR Specification Language

A major objective of the CSR paradigm is to facilitate the specification of real-time processes, and then support a static evaluation of various design alternatives. To this end, the CSR paradigm supports the separation of a real-time system's functional specification from its configuration. The functional spcification, which is written in the CSR application language, describes the high-level timing characteristics of a system. The configuration schema, written in the CSR Configuration Language, characterizes the interralationship between these processes; i.e., their resource mappings, their communication links, etc. This separation of concerns more easily lends itself to an evaluation of various system designs. At the CSR application language level, one need only concern oneself with the functionality of each process. The configuration language then gives us the ability to "assemble" the component programs in as intelligent a manner as possible. By experimenting with different process mappings and communication channel connections, one can change the topology of a system without rewriting the actual CSR application specification.

2.1 The Application Language

A CSR application program consists of a collection of processes, each of which contains a declaration section and a CSR statement. The declaration section reserves the ports, local execution events, and free timing parameters that the process will use, while the statement specifies the process' basic control structure. CSR statements include send and receive commands, sequential composition, timeouts, periodic loops, interrupts and exception-handling.

Table 1 shows the syntax of the language; CSR keywords are in boldface, $\langle\ \rangle$ indicates non-terminal symbols, "id" represents an identifier, and "num" denotes a positive integer. We also use $|$ to indicate alternatives and [ ] to indicate optional syntax.

Declarations. Declarations fall into two categories – event declarations and timing declarations. An event declaration reserves identifiers associated either with communication ports or local execution actions. These CSR events are generically called **atoms** as they are used in "atomic" CSR statements: **input** atoms are used in **recv** statements, **output** atoms are used in **send** statements, and **local** atoms are used in **exec** statements.

A timing declaration reserves one or more free *time variables* that may be used in a CSR time-constrained statement such as **wait**, **every** and **scope**. When a time variable appears in such a statement, it becomes the configuration schema's responsibility to bind it to an integer constant. As is shown in Table 1, constants may also be used in these statements.

```
         ⟨program⟩  ::=  ⟨proc⟩ ⟨program⟩ | ⟨proc⟩
            ⟨proc⟩  ::=  process id ⟨decls⟩ ⟨stmt⟩
           ⟨decls⟩  ::=  ⟨decl⟩ [ ⟨decls⟩ ]
            ⟨decl⟩  ::=  input ⟨ids⟩ | output ⟨ids⟩ | local ⟨ids⟩ | timevar ⟨ids⟩
             ⟨ids⟩  ::=  id [ , ⟨ids⟩ ]
            ⟨stmt⟩  ::=  ⟨atomic⟩ | skip | wait ⟨time⟩ | idle | ndet(⟨atomic⟩,num,num)
                     |   ⟨stmt⟩ ; ⟨stmt⟩ | ⟨loop-stat⟩ | ⟨every-stat⟩
                     |   ⟨scope-stat⟩ | ⟨interleave-stat⟩
          ⟨atomic⟩  ::=  exec(id) | recv(id) | send(id)
       ⟨loop-stat⟩  ::=  loop do ⟨stmt⟩ od
      ⟨every-stat⟩  ::=  every ⟨time⟩ do ⟨stmt⟩ od
      ⟨scope-stat⟩  ::=  scope do ⟨stmt⟩ [ ⟨triggers⟩ ] od
        ⟨triggers⟩  ::=  interrupt ⟨atomic⟩ → ⟨stmt⟩ ⟨triggers⟩
                     |   timeout ⟨time⟩ → ⟨stmt⟩
 ⟨interleave-stat⟩  ::=  interleave do ⟨stmt⟩ & ⟨stmt⟩ od
            ⟨time⟩  ::=  num | id
```

Table 1: The CSR Application Language

Statements. An atomic statement – i.e., **send**, **recv** or **exec** – requires one time unit to execute, at which time it terminates. And while it cannot be interrupted *during* execution, it does have the capacity to idle indefinitely *before* execution. There are two reasons for this. First, **recv** and **send** require synchronization with other processes. Second, all three statements must also wait to be scheduled since many processes may be interleaved on the same resource.

The **wait** t statement specifies pure delay for t time units, after which it terminates. (We mandate that t be greater than 0.) The **skip** statement terminates at the first time unit; that is, **skip** is equivalent to **wait** 1. At the other extreme is the **idle** statement, which endlessly idles and never terminates.

The sequential composition of S and T, "S;T", is standard: first S executes until termination, immediately after which T is initiated. For an atomic statement S, **ndet**(S,m,n) executes S between m and n times, where $1 \leq m \leq n$. If m is not equal to n, the exact number of executions is resolved nondeterministically. For example, the statement **ndet**(**send**(ch),3,5) means that **send**(ch) is executed between 3 and 5 times.

There are two kinds of repetitive constructs: **loop** and **every**. The **loop** statement is used for simple, nonterminating loops. For example, **loop do** S **od** endlessly cycles, executing the statement S. On the other hand, the **every** construct denotes a statement with cyclic behavior of a positive periodicity. For example **every** 6 **do** S **od** has a period of 6 time units; that is, every 6 time units S is reinitiated. There are three possible outcomes of each iteration: (1) S may terminate *exactly* at 6 time units, and then be restarted; (2) S may terminate early, in which case the loop idles for the remainder of the period, and then gets restarted; and (3) S may not terminate within the period, in which case it is aborted, and then restarted.

The **scope** statement allows the specification of interrupts and a deadline associated with a statement. A typical scope statement is as follows: **scope do** S triggers **od**. The body, S, of the scope statement executes, and if and when a triggering condition is raised the execution flow is transferred to its associated exception handler. There are four kinds of trigger guards: send, recv, exec and timeout. We use the following fragment to explain the construct's behavior:

```
scope do
   S
interrupt recv(ch1) → S1
interrupt send(ch2) → S2
interrupt recv(ch1) → S3
interrupt exec(a) → S4
timeout 100 → S5
od
```

Upon the initiation of the scope, the statement S starts executing and the timeout counter – initially set to 100 – starts decrementing upon entering the scope. During the execution of S, the atomic expressions (or "interrupts") are enabled. S is executed until one of the following occurs: (1) S terminates, in which case the timeout and interrupts are disabled, and the entire statement terminates; (2) an interrupt is executed, in which case S is aborted, the timeout and other interrupts are disabled, and the associated handler is then executed; or (3) the timeout occurs, in which case S is aborted, the interrupts are disabled, and the timeout exception handler S5 is initiated.

If there is contention between two interrupts at the same time, the selection is made by the two action's synchronization constraints, by their priority, and as a last resort, nondeterministically. Contention between an interrupt, and an atomic action in S is similarly resolved.

We note that if no **timeout** condition were present, S would not have to terminate within 100 time units; similarly, if there were no interrupts, the only two possibilities would be for S to terminate, or for a timeout exception to be raised.

The CSR paradigm provides two kinds of concurrency: one for processes executing on different resources and another for processes executing on the same resource. As we have stated, it is the responsibility of the configuration schema to define the resource location of each process, and thus, the characteristics of inter-process concurrency. However, we additionally provide the facility for intra-process concurrency, which is defined by the **interleave** statement. For example, the statement, **interleave do** S & T **od**, interleaves the actions of S and T, scheduling them according to their synchronization and priority constraints. The entire statement terminates when both S and T have reached terminating states. We note that one can use the interleave statement to describe a hardware interrupt handling, in which the interrupted statement is resumed after handling the interrupt.

The semantics of the interleave is such that pure idle time is never interleaved with other actions; instead, it is "overlapped." Thus, the fragment

interleave do wait 2 & **recv**(ch1);**recv**(ch2) **od**

is equivalent to "**recv**(ch1);**recv**(ch2)", as the two receives are overlapped with the idles. Thus, **skip** functions as an identity in the **interleave** statement.

```
process S
  local sense, error
  output ch
  timevar t
  every 6 do
    exec(sense);
    scope do idle
    interrupt send(ch) → skip
    timeout t → exec(error)  od
  od

process M
  local compute
  input ch
  loop do
    recv(ch); exec(compute); exec(compute)
  od
```

Figure 1: Specification of a Sensor-Monitor System

2.2 A Sensor-Monitor System

Figure 1 displays the two process "S" and "M," where S specifies a sensor process and M specifies the monitor process with which it communicates. We assume that our time granularity is 1 millisecond.

The sensor has a 6 millisecond period, within which it (1) takes a sample reading from the environment, and then (2) attempts to communicate the result to M along channel ch. However, because the environment is subject to very rapid changes, the communication must be made within t milliseconds; otherwise the data becomes worthless. If the communication is successfully accomplished within t milliseconds, S sleeps for the remainder of the period. Otherwise, it records an error.

As for the monitor, its control structure consists solely of an endless loop. During every iteration, the process first waits for S to communicate its data. When the data is received, statistics are computed for 2 milliseconds within a critical section, after which M again waits for communication with S.

2.3 The Configuration Language

The CSR Application Language is designed for the functional description of real-time processes. These individual CSR processes are without "concurrent context"; that is, their relationship to the overall system is left unspecified. This relationship is defined in a *configuration schema* written the CSR configuration language. A system configuration contains the following information: (1) processes are mapped to the system resources; (2) priorities are assigned to the various atoms declared in each process; (3) time variables are bound to integer constants; (4) channels are created by making connections between

⟨sys⟩	::=	⟨config⟩ ⟨sys⟩ | ⟨mainsys⟩
⟨config⟩	::=	**configurator** id ([⟨formals⟩]) ⟨declarations⟩ **end**
⟨formals⟩	::=	⟨formal⟩ [; ⟨formals⟩]
⟨formal⟩	::=	**resource** ⟨ids⟩ | **priority** ⟨ids⟩ | **timeval** ⟨ids⟩
⟨mainsys⟩	::=	**main** ⟨declarations⟩ **end**
⟨declarations⟩	::=	⟨decl⟩ [⟨declarations⟩]
⟨decl⟩	::=	**resource** ⟨ids⟩
	|	**system** id = id ([⟨actuals⟩])
	|	**process** id ⟨attrspec⟩
	|	**assign** ⟨anyids⟩ **on** id
	|	**close** ⟨ids⟩
	|	**connect** ⟨fullids⟩
⟨actuals⟩	::=	⟨value⟩ [, ⟨actuals⟩]
⟨value⟩	::=	num | id
⟨attrspec⟩	::=	⟨attrtype⟩ ⟨attrlist⟩ ⟨attrspec⟩
⟨attrtype⟩	::=	**inport** | **outport** | **local** | **timevar**
⟨attrlist⟩	::=	id (⟨value⟩) [, ⟨attrlist⟩]
⟨ids⟩	::=	id [, ⟨ids⟩]
⟨fullids⟩	::=	fullid [, ⟨fullids⟩]
⟨anyids⟩	::=	anyid [, ⟨anyids⟩]

Table 2: The CSR Configuration Language

the processes' ports; and (5) if we so desire, designated resources are *closed*, meaning that no further processes will be allocated to them.

The configuration language is hierarchical in that a configured system may be an amalgamation of subsystems, which may themselves have subsystems. We also provide for *parameterized configurations*, so that the parent systems may pass both resources and priority assignments to their child subsystems.

Table 2 presents the complete grammar for the Configuration Language. We use the same notational conventions as in the CSR grammar, with the following addition: a "fullid" is an alphanumeric string interspersed with dots, such as "sys.prog.a1", and "anyid" represents either a fullid or an ordinary identifier.

A configuration schema consists of a **main** system (or the root), as well as a (possibly empty) set of configurators for subsystems. Within each component is a set of declarations that define the system characteristics.

We can illustrate the Configuration Language by using it to build a *dual* sensor-monitor system, derived from the sensor and monitor processes in Figure 1. In our new scenario we have two sensors, S1.S and S2.S, each of which with the functionality of our original sensor S. We now bind the S1.S's time variable t to 2 milliseconds, and S2.S's time variable t to 4 milliseconds. Thus S2.S's sampled data has greater temporal persistence than that of S1.S. We also have two monitors M1.M and M2.M, each of which with the functionality of our original monitor M. We use the configuration shown in Figure 3 to "draw" the system layout depicted in Figure 2.

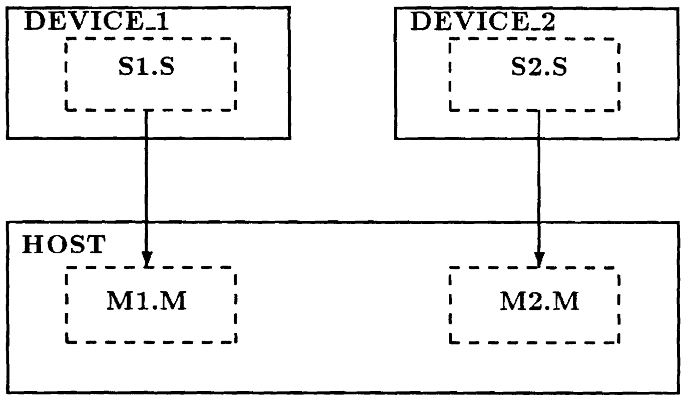

Figure 2: Process to Resource Mapping

Before delving into the functionality of configurators and subsystems, we first consider the basic system-building declarations: **resource**, **process**, **assign**, **close** and **connect**. As we have stated, the resource is the defining aspect of our real-time model, in that a single resource may only execute one atom at a time. Thus the number of resources in a system may significantly alter its runtime behavior. Within a configuration, resources are denoted by the **resource** declaration; e.g., in Figure 3 identifies the three resources, Device1, Device2 and Host.

Each process used in the system is identified using a **process** declaration, which has the syntax that is quite similar to the header section in a CSR process. That is, it defines the process name, the atoms used within the process and any free time variables that must be bound.

The **assign** declaration is used to map processes onto resources. For example, in Figure 3, processes S1.S and S2.S are mapped to resources Device1 and Device2, respectively, and processes M1.M and M2.M are mapped to resource Host.

In determining the timing behavior of processes on a particular resource, it is useful to know whether the resource is going to host additional processes. The **close** declaration defines that no more processes may be assigned to the named resources. The **close** declaration in Figure 3 denotes that S1.S and S2.S are the *sole* occupants of Device1 and Device2, respectively.

We use **connect** to define connections between atoms; that is, they must execute synchronously when their resources are combined in a system. In Figure 3, the first **connect** declaration specifies that the S1.S and M1.M processes share a point-to-point link. When they execute in parallel, the atoms S1.S.ch and M1.M.ch must always execute simultaneously. We note that the meaning of the **connect** declaration is *transitive*. That is, the following declarations imply that P.a, Q.b, R.c and S.d are all mutually connected:

connect P.a, Q.b
connect R.c, S.d
connect P.a, S.d

Hierarchical, Parameterized Configurations. The Configuration Language supports hierarchical schemas, which can be constructed using the following features:

```
configurator MakeSense(timevar u)
  process S
    local sense(1), error(1)
    outport ch(0)
    timevar t(u)
end

configurator MakeMon(priority chpri)
  process M
    local compute(3)
    inport ch(chpri)
end

main
  resource Device1, Device2, Host
  system S1 = MakeSense(2)
  system S2 = MakeSense(4)
  system M1 = MakeMon(2)
  system M2 = MakeMon(1)
  assign S1.S on Device1
  assign S2.S on Device2
  assign M1.M, M2.M on Host
  close Device1, Device2, Host
  connect S1.S.ch, M1.M.ch
  connect S2.S.ch, M2.M.ch
end
```

Figure 3: Hierarchical Configuration of Sensor-Monitor System

- The **configurator** declaration, which is used to denote configuration "generators"; that is, entities that must be "instantiated" to create a particular configuration.
- The **system** declaration, which "instantiates" a configurator, and thus creates a new subsystem within its scope.

For example, Figure 3 includes a configurator MakeMon. By invoking the configurator twice, the **main** schema creates two "copies" of the system defined in Conf, and in this way we derive the processes M1.M and M2.M. Moreover, M1.M.ch has a priority of 2, and M2.Mon.ch has priority 1.

A configurator may itself make reference to other configurators, which results in a subsystem that owns other subsystems within its scope. The only restriction we make is that no forward references are allowed; this rules out any possibility of "recursive" configurator invocation. Those readers familiar with the ML programming language will note that the relationship between configuration schemas and configurators is similar to that between structures and functors [6].

3 The Calculus for Communicating Shared Resources

In this section we describe the CCSR language and its computation model, developing such notions as resources, priority, synchronization and preemption. We proceed to show that CCSR's semantics gives rise to a set of substitutive proof rules.

3.1 The Computation Model

The basic unit of computation is the *event*, which is used to model both local resource execution as well as inter-resource synchronization. When executed by a resource, each event consumes exactly one time unit. We let Σ represent the universal set of events.

Since a system potentially consists of many resources, multiple events may be observed at any time throughout the course of its execution. We call such observances *actions*, and they are represented by sets in $\mathbb{P}(\Sigma)$. In general, we let the letters a, b and c range over the event set Σ, and the letters A, B and C range over the action set $\mathbb{P}(\Sigma)$.

Termination. The termination event, or "$\surd$", has the unique property that it is not "owned" by any particular resource. Rather, if $\surd \in A$ for some action A, this means that the system executing A is capable of terminating.

Resources and Actions. We consider individual resources to be inherently sequential in nature. That is, at each time unit a resource is capable of executing *at most* a single event. Actions that consist of multiple events must be formed by the synchronous execution of multiple resources. We denote $\mathcal{R}$ to represent the set of resources available to a system, and let i, j, and k range over $\mathcal{R}$. For all i in $\mathcal{R}$ we denote Σ_i as the collection of events exclusively "owned" by resource i:

$$\forall i \in \mathcal{R}, \forall j \in \mathcal{R} \,.\, i \neq j,\ \Sigma_i \cap \Sigma_j = \emptyset$$

This type of alphabet partitioning is similar to that found in the I/O Automata model [12], where it is used to define a notion of fairness. However, here it is used to help mandate our resource-induced mutual exclusion condition. The domain of actions executable by any CCSR term, "$\mathcal{D}$", is defined as follows:

$$\mathcal{D} \;=\; \{A \in p(\Sigma) \,|\, \forall i \in \mathcal{R},\ |A \cap \Sigma_i| \leq 1\}$$

where "$p(\Sigma)$" denotes the set of finite subsets of Σ, and "$|S|$" denotes the cardinality of a finite set "S". For a given action A, we use the notation $\rho(A)$ to represent the resource set that executes events in A: $\rho(A) = \{i \in \mathcal{R} \,|\, \Sigma_i \cap A \neq \emptyset\}$. Note that since for all i, $\surd \notin \Sigma_i$, $\rho(A) = \rho(A - \{\surd\})$.

Priority. At any point in time many events may be competing for the ability to execute on a single resource. We help arbitrate such competition through the use of a priority ordering, $\pi \in \Sigma \to \mathbb{N}$. Using π, we define the preorder "$\leq_p$" that reflects the notion of priority over the domain $\mathcal{D}$. For all $A, B \in \mathcal{D}$, $A \leq_p B$ if and only if for all i in $\rho(A) \cup \rho(B)$,

$$\begin{array}{l} A \cap \Sigma_i = \emptyset \vee \\ (\exists a.\ A \cap \Sigma_i = \{a\} \wedge \pi(a) = 0) \vee \\ (\exists a \exists b.\ A \cap \Sigma_i = \{a\} \wedge B \cap \Sigma_j = \{b\} \wedge \pi(a) \leq \pi(b)) \end{array}$$

Based on this definition, we use the notation "$A <_p B$" to represent that A has lower priority than B; i.e., $A \leq_p B$ and $B \not\leq_p A$.

Synchronization. In CCSR, the lowest form of communication is accomplished through the simultaneous execution of synchronizing events. The model treats such synchronizing events as being statically "bound" together by the various connections between system resources. To capture this property we make use of what we call *connection sets.* A connection set is a set of events that exhibits the "all or none" property of event synchronization: At time t, if any of the events in a given connection set wish to execute, they all must execute. More formally, a connection set is an equivalence class formed by the equivalence relation "$\bullet\!\!-\!\!\bullet$".

Definition 3.1 $\bullet\!\!-\!\!\bullet \subseteq \Sigma \times \Sigma$ *is an equivalence relation, where* $a \bullet\!\!-\!\!\bullet\, b$ *denotes that* a *is connected to* b. *We use the notation* $connections(a)$ *to represent the equivalence class (or connection set) of* a. □

We say that an action is synchronized with respect to a resource set $I \subseteq \mathcal{R}$ if $sync_{(I)}(A)$ holds, where

$$sync_{(I)}(A) \quad \text{iff} \quad A = (\textstyle\bigcup_{a \in A} connections(a)) \cap (\bigcup_{i \in I}(\Sigma_i \cup \{\surd\}))$$

That is, if $sync_{(I)}(A)$ holds, A cannot make any additional connections with the resources in I. Also, it is often convenient to be able to decompose an action A into two parts: that which is fully synchronized (or resolved), and that which is not (or still unresolved). To do this, we make use of the following two definitions:

$$\begin{array}{rcl} res(A) & = & \bigcup\{B \subseteq A \,|\, sync_{(\mathcal{R})}(B)\} \\ unres(A) & = & A - res(A) \end{array}$$

Idle Events. For every $i \in \mathcal{R}$, there is an idle event τ_i^0 in Σ_i such that $\pi(\tau_i^0) = 0$. The way to interpret these events is as follows. A process may idle in two ways – it may either release its processor during the idle time (represented by the execution of no event) or it may hold its processor (represented by the execution of an idle event). We add that such idle events are local with respect to their own resources; that is, they belong to their own connection sets.

3.2 The CCSR Language and its Semantics

The syntax of CCSR resembles, in some respects, that of SCCS [13]. Let $\mathcal{E}$ represent the domain of terms, and let E, F, G and H range over $\mathcal{E}$. Additionally we assume an infinite set of free term variables, FV, with X ranging over FV and $free(E)$ representing the set of free variables in the term E. Let $\mathcal{P}$ represent the domain of closed terms, which we call *agents* or alternatively, *processes*, and let P, Q, R and S range over $\mathcal{P}$. The following grammar defines the terms of CCSR:

$$E \;:=\; NIL \mid A : E \mid E + E \mid E\,{}_I\|_J\, E \mid E \,\Delta_t^B\, (E, E, E) \mid [E]_I \mid fix(X.E) \mid X$$

The semantics of $\mathcal{E}$ is defined by a labeled transition system $\langle \mathcal{E}, \rightarrow, \mathcal{D} \rangle$, which is a relation $\rightarrow \subseteq \mathcal{E} \times \mathcal{D} \times \mathcal{E}$. We denote each member (E, A, F) of "$\rightarrow$" as "$E \xrightarrow{A} F$". We call this transition system *unconstrained*, in that no priority arbitration is made between actions. We then use "$\longrightarrow$" to define a prioritized transition system $\langle \mathcal{E}, \longrightarrow_\pi, \mathcal{D}\rangle$, which is sensitive to preemption. This two-phased approach greatly simplifies the definition of "$\longrightarrow_\pi$"; similar tactics have been used by [2] in their treatment of CCS priority, and by [8] in their semantics for maximum parallelism.

The Unconstrained Transition System, "$\longrightarrow$". Table 3 presents the unconstrained transition system, "$\longrightarrow$". Throughout, we use the following notation. For a given set of resources $I \subseteq \mathcal{R}$, we let Σ_I represent the set $\bigcup_{i \in I} \Sigma_i$. Also, $A * B = (A - \{\surd\}) \cup (B - \{\surd\}) \cup (A \cap B)$; that is, the termination event "$\surd$" is an element of $A * B$ if and only if it is in both A and B.
Inaction. The term NIL executes no action whatsoever, and thus it possesses no transition.
Action. The Action operator, "$A : E$", has the following behavior. At the first time unit, the action A is executed, proceeded by the term E.
Choice. The Choice operator represents selection – either of the terms can be chosen to execute, subject to the constraints of the environment. For example, the term $(A : E) + (B : F)$ may execute A and proceed to E, or it may execute B and proceed to F.
Parallel. The Parallel operator defines the resources that can be used by the two terms, and also forces synchronization between them. Here, $I, J \subseteq \mathcal{R}$ are the resources allotted to E and F, respectively. In the case where $I \cap J \neq \emptyset$, E and F may be able to share certain resources. But as we have stated, such resource-sharing must be interleaved.

The first two conditions define the resources on which the terms E_1 and E_2 may execute, while the third condition stipulates that single resources may not execute more than one event at a time. The final condition defines our notion of inter-resource synchronization; that is, A_1 and A_2 may execute simultaneously if and only if they are connected in the following sense: If some event $a \in A_1$ is connected to an event $b \in \Sigma_J$, then b *must* appear in A_2, and *vice versa.*
Scope. The Scope construct $E \, \Delta_t^B \, (F, G, H)$ binds the term E by a temporal scope [11], and it incorporates both the features of timeouts and interrupts. We call t the *time bound* and B the *termination control*, where $t \in \mathbf{N}^+ \cup \{\infty\}$ (i.e., t is either a positive integer or infinity), and $B = \{\surd\}$ or $B = \emptyset$.

The four rules for the Scope operator, correspond to the four actions that may be taken while a term E is bound by a temporal scope: 1) continuing (ScopeC), 2) successfully terminating (ScopeE), 3) timing out with an exception-handler (ScopeT) and 4) being interrupted (ScopeI).
Close. The Close operator assigns terms to occupy *exactly* the resource set denoted by the index I. First, the action A may not utilize *more* than the resources in I; otherwise it is not admitted by the transition system. If the events in A utilize less than the set I, the action is augmented with the "idle" events from each of the unused resources. For example, assume E executes an action A, and that there is some $i \in I$ such that $i \notin \rho(A)$. In $[E]_I$, this gap is filled by including τ_0^i in A. Here we use the notation $\mathcal{T}_J^0$ to represent *all* of the 0-priority events from the resource set J: $\mathcal{T}_J^0 = \{\tau_j^0 \mid j \in J\}$.
Recursion. The term $fix(X.P)$ denotes recursion, allowing the specification of infinite behaviors. As an example, consider the term that indefinitely executes the action "A": $fix(X.(A : X))$. By the Action rule, $A : fix(X.(A : X)) \xrightarrow{A} fix(X.(A : X))$, so by the

Action : $A : E \xrightarrow{A} E$

ChoiceL : $\dfrac{E \xrightarrow{A} E'}{E + F \xrightarrow{A} E'}$ **ChoiceR :** $\dfrac{F \xrightarrow{A} F'}{E + F \xrightarrow{A} F'}$

Parallel :

$$\frac{E_1 \xrightarrow{A_1} E_1',\ E_2 \xrightarrow{A_2} E_2'}{E_1\, {}_I\|_J\, E_2 \xrightarrow{A_1 * A_2} E_1'\, {}_I\|_J\, E_2'} \quad \left(\begin{array}{l} \rho(A_1) \subseteq I,\ \rho(A_2) \subseteq J, \\ \rho(A_1) \cap \rho(A_2) = \emptyset,\ sync_{(I \cup J)}(A_1 * A_2) \end{array}\right)$$

ScopeC : $\dfrac{E \xrightarrow{A} E'}{E \,\Delta_t^B\, (F, G, H) \xrightarrow{A} E' \,\Delta_{t-1}^B\, (E, F, G)} \quad (t > 1,\ \surd \notin A)$

ScopeE : $\dfrac{E \xrightarrow{A} E'}{E \,\Delta_t^B\, (F, G, H) \xrightarrow{A * B} F} \quad (t \geq 1,\ \surd \in A)$

ScopeT : $\dfrac{E \xrightarrow{A} E'}{E \,\Delta_t^B\, (F, G, H) \xrightarrow{A} G} \quad (t = 1,\ \surd \notin A)$

ScopeI : $\dfrac{H \xrightarrow{A} H'}{E \,\Delta_t^B\, (F, G, H) \xrightarrow{A} H'} \quad (t \geq 1)$

Close : $\dfrac{E \xrightarrow{A} E'}{[E]_I \xrightarrow{A \cup (\mathcal{T}_I^0 - \mathcal{T}_{\rho(A)}^0)} [E']_I} \quad (\rho(A) \subseteq I)$

Recursion : $\dfrac{E[fix(X.E)/X] \xrightarrow{A} E'}{fix(X.E) \xrightarrow{A} E'}$

Con : $\dfrac{E \xrightarrow{A} E'}{X \xrightarrow{A} E'} \quad (X \stackrel{\text{def}}{=} E)$

Table 3: Unconstrained Transition System

Recursion rule, $fix(X.(A:X)) \xrightarrow{A} fix(X.(A:X))$.
Con. The Constant rule just mandates that if a name "X" is defined to represent the term "E", then X can make the same transitions that E can.

Preemption and the Prioritized Transition System. The prioritized transition system is based on the notion of *preemption*, which unifies CCSR's treatment of synchronization, resource-sharing, and priority. Let "$\prec$", called the *preemption order*, be a transitive, irreflexive, binary relation on actions. Then for two actions A and B, if $A \prec B$, we can say that "A is preempted by B". This means that in all real-time contexts, if a system can choose between executing either A or B, it will execute B.

Definition 3.2 *For all $A \in \mathcal{D}$, $B \in \mathcal{D}$, $A \preceq B$ if and only if*

$$\rho(A) = \rho(B) \wedge unres(A) = unres(B) \wedge res(A) \leq_p res(B)$$

The relation "$\preceq$" defines a preorder over $\mathcal{D}$, and we say $A \prec B$ if $A \preceq B$ and $B \not\preceq A$, i.e., $\rho(A) = \rho(B) \wedge unres(A) = unres(B) \wedge res(A) <_p res(B)$. □

Now we are ready to define the transition system $\langle \mathcal{E}, \rightarrow_\pi, \mathcal{D} \rangle$, grounded in our notion of preemption.

Definition 3.3 *The labeled transition system $\langle \mathcal{E}, \rightarrow_\pi, \mathcal{D} \rangle$ is a relation $\rightarrow_\pi \subseteq \mathcal{E} \times \mathcal{D} \times \mathcal{E}$ and is defined as follows: $(E, A, E') \in \rightarrow_\pi$ (or $E \xrightarrow{A}_\pi E'$) if:*

1. *$E \xrightarrow{A} E'$, and*

2. *For all $A' \in \mathcal{D}$, $E'' \in \mathcal{E}$ such that $E \xrightarrow{A'} E''$, $A \not\prec A'$.* □

Equivalence between processes is based on the concept of *strong bisimulation*, which is defined in the usual sense (see [13]). We denote "$\sim_\pi$" as the largest strong bisimulation over the transition system $\langle \mathcal{E}, \rightarrow_\pi, \mathcal{D} \rangle$, and we call it *prioritized strong equivalence.* The following two theorems state characterize some fundamental properties of CCSR. Theorem 3.1 states that "$\sim_\pi$" forms a congruence over the CCSR operators; that is, whenever two terms are equivalent, all CCSR "contexts" will preserve their equivalence. Theorem 3.2 states that fixpoints exist and are unique. For detailed proofs, refer to [4].

Theorem 3.1 *Prioritized strong equivalence is a congruence with respect to the CCSR operators. That is, for $E, F, G \in \mathcal{E}$, if $E \sim_\pi F$ and $free(G) = \{X\}$, then $G[E/X] \sim_\pi G[F/X]$.*

Theorem 3.2 *For any term E, $fix(X.E)$ is the unique solution to the recursive equation $X \sim_\pi E$.*

Choice(1)	$E + NIL = E$
Choice(2)	$E + E = E$
Choice(3)	$E + F = F + E$
Choice(4)	$(E + F) + G = E + (F + G)$
Choice(5)	$(A : E) + (B : F) = B : F \quad \text{if } A \prec B$
Par(1)	$E\,{}_I\Vert_J\, NIL = NIL$
Par(2)	$E\,{}_I\Vert_J\, F = F\,{}_J\Vert_I\, E$
Par(3)	$(E\,{}_I\Vert_J\, F)\,{}_{(I\cup J)}\Vert_K\, G = E\,{}_I\Vert_{(J\cup K)}\,(F\,{}_J\Vert_K\, G) \quad \text{if } I\cap J = \emptyset,\ J \cap K = \emptyset,\ I \cap K = \emptyset$
Par(4)	$E\,{}_I\Vert_J\,(F + G) = (E\,{}_I\Vert_J\, F) + (E\,{}_I\Vert_J\, G)$
Par(5)	$(A : X)\,{}_I\Vert_J\,(B : Y) = \begin{cases} (A * B) : (X\,{}_I\Vert_J\, Y) & \text{if } \rho(A) \subseteq I,\ \rho(A)\cap\rho(B) = \emptyset, \\ & \quad \rho(B) \subseteq J,\ sync_{(I\cup J)}(A * B) \\ NIL & \text{otherwise} \end{cases}$
Scope(1)	$NIL \triangle_t^B (F, G, H) = H$
Scope(2)	$(E_1 + E_2) \triangle_t^B (F, G, H) = (E_1 \triangle_t^B (F, G, H)) \ + \ (E_2 \triangle_t^B (F, G, H))$
Scope(3)	$(A : E) \triangle_t^B (F, G, H) = \begin{cases} (A * B : F) + H & \text{if } \surd \in A \\ (A : (E \triangle_{t-1}^B (F, G, H))) + H & \text{if } \surd \notin A \text{ and } t > 1 \\ (A : G) + H & \text{otherwise} \end{cases}$
Close(1)	$[NIL]_I = NIL$
Close(2)	$[E + F]_I = [E]_I + [F]_I$
Close(3)	$[A : E]_I = \begin{cases} (A \cup (\mathcal{T}_I^0 - \mathcal{T}_{\rho(A)}^0)) : [E]_I & \text{if } \rho(A) \subseteq I \\ NIL & \text{otherwise} \end{cases}$
Close(4)	$[[E]_I]_J = \begin{cases} [E]_J & \text{if } I \subseteq J \\ NIL & \text{otherwise} \end{cases}$

Table 4: The Axiom System, $\mathcal{A}$

3.3 An Axiomatization of CCSR

The axioms in the CCSR proof system, $\mathcal{A}$, are enumerated in Table 4. In [4] we show that $\mathcal{A}$, (augmented with standard laws for substitution), is sound with respect to prioritized equivalence; further, $\mathcal{A}$ is complete for finite fragments of CCSR. Using the Choice and Parallel laws in $\mathcal{A}$, we can derive an analogue to the Expansion Law from CCS.

Theorem 3.3 (Expansion Law) *Let K and L be finite index sets such that for all $k \in K$, $l \in L$, $A_k : E_k \in \mathcal{E}$ and $B_l : F_l \in \mathcal{E}$. Then*

$$\mathcal{A} \vdash (\sum_{k\in K} A_k : E_k)_I\|_J(\sum_{l\in L} B_l : F_l) \;=\; \sum_{\substack{k\in K,\ l\in L,\\ \rho(A)\subseteq I,\ \rho(A)\cap\rho(B)=\emptyset,\\ \rho(B)\subseteq J,\ sync_{(I\cup J)}(A*B)}} (A_k * B_l) : (E_k\,{}_I\|_J F_l)$$

4 Translation of CSR Processes Into CCSR Terms

Using CSR to specify a real-time system is certainly more natural and intuitive affair than using the CCSR process algebra. However, CCSR provides a solid foundation to formally analyze the properties of real-time systems; e.g., it enjoys a well-behaved operational semantics, a rich equational structure and a congruence relation over terms. Our objective in this section is to assign formal meaning to CSR processes by translating them into CCSR terms.

In addition to the standard CCSR notation, we make prolific use of the following derived terms:

$$\boxed{\begin{array}{rcl} IDLE & \stackrel{\text{def}}{=} & \emptyset : IDLE \\ TERM & \stackrel{\text{def}}{=} & \{\surd\} : TERM \\ \delta_\infty(a) & \stackrel{\text{def}}{=} & (\{a,\surd\} : TERM) + (\emptyset : \delta_\infty(a)) \\ E \rhd F & \stackrel{\text{def}}{=} & E \,\triangle^{\emptyset}_{\infty}\, (F, NIL, NIL) \end{array}}$$

That is, the $IDLE$ process sleeps indefinitely, while $TERM$ is the process that indefinitely signals termination. For some event $a \in \Sigma$, the process "$\delta_\infty(a)$" is an "asynchronizer"; it may indefinitely idle before executing a (the exact time of execution depends on the system's characteristics, such as the priority of a, it's synchronization requirements, etc.). The term $E \rhd F$ (pronounced "E pipe F") executes E until it signals termination, at which time control passes to F. Using the ScopeC and ScopeE rules from Table 3, we can derive the two transition rules for "$\rhd$":

$$\frac{E \xrightarrow{A} E'}{E \rhd F \xrightarrow{A} E' \rhd F}\ (\surd \notin A) \qquad \frac{E \xrightarrow{A} E'}{E \rhd F \xrightarrow{A-\{\surd\}} F}\ (\surd \in A)$$

4.1 The Translation Approach

Within this section we let S and T range over CSR statements, we let at range over atoms, and act range over atomic actions (e.g., **send**(at) is an atomic action). Now consider the following generic process definition, where pid is the process name, ⟨decls⟩ is the declaration section and S is the process body:

$$\begin{array}{l}\textbf{process}\ pid\\ \quad \langle \text{decls}\rangle \\ \quad S\end{array}$$

Throughout we assume that the atoms in the process are consistent, in that no single atom is used for multiple functions (e.g., for both **send** and **exec** atomic actions). That being the case, the function "Tevent" accepts a CSR atomic action and produces an uninterpreted CCSR event:

$$\begin{array}{rcl}\text{Tevent}(\textbf{send}(at)) &=& pid.at\\ \text{Tevent}(\textbf{recv}(at)) &=& pid.at\\ \text{Tevent}(\textbf{exec}(at)) &=& pid.at\end{array}$$

Thus all atomic actions are generically mapped to an event that identifies both the process pid and the atom name, at. Next we define translation function, Tstat, which accepts CSR *statements* within the process pid, and produces a CCSR term as its result.

Atomic Statements. Since all atomic statments have the capacity to idle until being scheduled, the "δ_∞" suits our purposes here.

$$\begin{array}{rcl}\text{Tstat}(\textbf{send}(at)) &=& \delta_\infty(\text{Tevent}(\textbf{send}(at)))\\ \text{Tstat}(\textbf{recv}(at)) &=& \delta_\infty(\text{Tevent}(\textbf{recv}(at)))\\ \text{Tstat}(\textbf{exec}(at)) &=& \delta_\infty(\text{Tevent}(\textbf{exec}(at)))\end{array}$$

In other words, the translated process may idle before both executing $pid.at$ and signaling termination. This is easily illustrated by the transition rules, which yield:

$$\begin{array}{l}\text{Tstat}(\textbf{recv}(at)) \xrightarrow{\emptyset} \text{Tstat}(\textbf{recv}(at))\\ \text{Tstat}(\textbf{recv}(at)) \xrightarrow{\{pid.at,\surd\}} TERM\end{array}$$

Skip and Wait. The CSR statement **wait** t "vamps" for its first $t-1$ time units, and then goes into a terminating state at its tth time unit (**wait** t is defined only for $t \geq 1$).

$$\text{Tstat}(\textbf{wait}\ t) = \begin{cases} TERM & \text{if } t = 1\\ IDLE\ \Delta^{\emptyset}_{t-1}\,(NIL, TERM, NIL) & \text{otherwise}\end{cases}$$

Recall that **skip** is simply syntactic sugar for **wait** 1.

Ndet. The CSR statement **ndet**(act, m, n) nondeterministically executes the atomic action *act* between m and n times, where $1 \leq m \leq n$. Exactly *when* each iteration will execute is, of course, sensitive to the characteristics of the fully configured system. For now we have to retain *all* possible times of execution, which is exactly the purpose of the "δ_∞" operator.

$$\text{Tstat}(\mathbf{ndet}(act, m, n)) =$$
$$\begin{cases} \text{Tstat}(act) & \text{if } n = m = 1 \\ \text{Tstat}(act) + (\text{Tstat}(act) \rhd \text{Tstat}(\mathbf{ndet}(act, 1, n-1))) & \text{if } m = 1 \text{ and } n \neq 1 \\ \text{Tstat}(act) \rhd \text{Tstat}(\mathbf{ndet}(act, m-1, n-1)) & \text{otherwise} \end{cases}$$

Sequential Composition. Our pipe operator was tailor-made to implement CSR's sequential composition. The term Tstat$(S;T)$ executes Tstat(S) until it signals termination; at the very next time unit, Tstat(T) is executed.

$$\text{Tstat}(S;T) = \text{Tstat}(S) \rhd \text{Tstat}(T)$$

Scope. In translating the **scope** statement, we require the use of an auxiliary translation function, Tint, for and any interrupts specified within the scope.

$$\text{Tint}(\mathbf{interrupt}\ act_1 \rightarrow S_1 \ldots \mathbf{interrupt}\ act_n \rightarrow S_n) = \sum_{i=1}^{n} \{\text{Tevent}(act_i)\} : \text{Tstat}(S_i)$$

There are four alternatives for **scope**, corresponding to each of the possible varieties of triggers that bound the scope: (1) no triggers, (2) a timeout trigger, (3) interrupt triggers and (4) both timeout and interrupt triggers.

$$\text{Tstat}(\mathbf{scope\ do}\ S\ \mathbf{od}) = \text{Tstat}(S)$$
$$\text{Tstat}(\mathbf{scope\ do}\ S\ \mathbf{timeout}\ t \rightarrow T\ \mathbf{od}) =$$
$$\text{Tstat}(S)\ \Delta_t^{\{\surd\}}\ (TERM, \text{Tstat}(T), NIL)$$
$$\text{Tstat}(\mathbf{scope\ do}\ S\ interrupts\ \mathbf{od}) =$$
$$\text{Tstat}(S)\ \Delta_\infty^{\{\surd\}}\ (TERM, NIL, \text{Tint}(interrupts))$$
$$\text{Tstat}(\mathbf{scope\ do}\ S\ interrupts\ \mathbf{timeout}\ t \rightarrow T\ \mathbf{od}) =$$
$$\text{Tstat}(S)\ \Delta_t^{\{\surd\}}\ (TERM, \text{Tstat}(T), \text{Tint}(interrupts))$$

Infinite Behaviors – Loop and Every. The CSR **loop** construct endlessly executes its loop body; thus we can use our pipe combinator in conjunction with fixpoint:

$$\text{Tstat}(\mathbf{loop\ do}\ S\ \mathbf{od}) = fix(X.\ (\text{Tstat}(S) \rhd X))$$

Recall that the statement **every** t **do** S **od** has a guaranteed period of t time units (where we require that $t \geq 1$). Again the CCSR scope operator is quite helpful in helping us realize the semantics for **every**. First, in creating a scope body of Tstat$(S) \rhd IDLE$, we produce a term that executes Tstat(S), and subsequently idles indefinitely. Thus the body performs all of the work of S, but never signals termination. This allows us to use the timeout argument of the scope operator to repeat the period, as in the following translation:

$$\text{Tstat}(\mathbf{every}\ t\ \mathbf{do}\ S\ \mathbf{od}) = fix(X.\ ((\text{Tstat}(S) \rhd IDLE)\ \Delta_t^{\emptyset}\ (NIL, X, NIL)))$$

Interleave. In presenting the formulation for the **interleave** we assume our process is configured on some resource $R.pid$. That is, let $events(S)$ be the set of events appearing in the term Tstat(S). Then to translate **interleave do** S & T **od**, we postulate that $\rho(events(\mathrm{Tstat}(S)) \cup events(\mathrm{Tstat}(T))) \subseteq \{R.pid\}$. Thus, the translation of the **interleave** statement is:

$$\mathrm{Tstat}(\textbf{interleave do } S \ \& \ T \textbf{ od}) = \mathrm{Tstat}(S) \ {}_{\{R.pid\}}\|_{\{R.pid\}} \ \mathrm{Tstat}(T)$$

From the definition of $sync$, we have that for any event a such that $\rho(\{a\}) = \{R.pid\}$, $sync_{(\{R.pid\})}(\{a\})$ holds, as does $sync_{(\{R.pid\})}(\{a, \surd\})$. And since S and T execute on the same resource, we can derive the following rule for **interleave**:

$$\frac{\mathrm{Tstat}(S) \xrightarrow{A_1} P_1, \ \mathrm{Tstat}(T) \xrightarrow{A_2} P_2}{\mathrm{Tstat}(\textbf{interleave do } S \ \& \ T \textbf{ od}) \xrightarrow{A_1 * A_2} P_1 \, {}_{\{R.pid\}}\|_{\{R.pid\}} \, P_2} \quad (|A_1 * A_2| \leq 1)$$

4.2 Translating the Configuration Schema

The configuration schema is used for two purposes – to determine the characteristics of the action domain, and to fix the final structure of the CCSR system. Each invocation of the **resource** declaration builds up the resource domain, $\mathcal{R}$. The **process** declaration determines the event alphabet, Σ, as well as the priority function π. The **assign** declaration generates the concurrent structure of the final CCSR term, and also determines the resource mapping, ρ. When **connect** is used – as in "**connect** P.a,P.b" – it states that "P.a •—• P.b".

4.3 CCSR Translation of Sensor-Monitor Process Specification

We revisit the sensor-monitor example, and apply our Tstat translation function to the processes in Figure 1 and the configuration in Figure 3. The result is displayed in Table 5; for readability, we have taken the liberty to rewrite fixpoint expressions using equivalent Constant definitions. With the action domain's construction from Table 6, as well as the system's syntax, we may now perform analysis using CCSR's transition system and its proof rules. Indeed, this is precisely what we shall do in Section 5.

5 Using the CCSR Proof Rules

In this section we present a sketch of a correctness proof of the Sensor-Monitor System, as portrayed in Tables 6 and 5. Recall that our safety constraint is as follows: neither Sensor process ever reaches a state in which its *error* event is enabled. We demonstrate this by showing that there is some $System'$ such that $System \sim_\pi System'$, and further,

$$
\begin{aligned}
System &\overset{\text{def}}{=} (R\_Device1 \;{}_{\{Device1\}}\|_{\{Device2\}}\; R\_Device2) \;{}_{\{Device1,Device2\}}\|_{\{Host\}}\; R\_Host \\
R\_Device1 &\overset{\text{def}}{=} [S1.S]_{\{Device1\}} \\
R\_Device2 &\overset{\text{def}}{=} [S2.S]_{\{Device2\}} \\
R\_Host &\overset{\text{def}}{=} [M1.M \;{}_{\{Host\}}\|_{\{Host\}}\; M2.M]_{\{Host\}} \\
S1.S &\overset{\text{def}}{=} (\delta_\infty(S1.S.sense) \rhd \\
&\qquad (IDLE\, \Delta_2^{\emptyset}\, (NIL, \delta_\infty(S1.S.error), \{S1.S.ch\} : TERM)) \rhd \\
&\quad IDLE)\, \Delta_6^{\emptyset}\, (NIL, S1.S, NIL) \\
S2.S &\overset{\text{def}}{=} (\delta_\infty(S2.S.sense) \rhd \\
&\qquad (IDLE\, \Delta_4^{\emptyset}\, (NIL, \delta_\infty(S2.S.error), \{S2.S.ch\} : TERM)) \rhd \\
&\quad IDLE)\, \Delta_6^{\emptyset}\, (NIL, S2.S, NIL) \\
M1.M &\overset{\text{def}}{=} \delta_\infty(M1.M.ch) \rhd (\delta_\infty(M1.M.compute) \rhd (\delta_\infty(M1.M.compute) \rhd M1.M)) \\
M2.M &\overset{\text{def}}{=} \delta_\infty(M2.M.ch) \rhd (\delta_\infty(M2.M.compute) \rhd (\delta_\infty(M2.M.compute) \rhd M2.M))
\end{aligned}
$$

Table 5: CCSR Sensor-Monitor Process Specification

that neither $S1.S.error$ nor $S2.S.error$ appears in the syntax of $System'$. Specifically,

$$
\begin{aligned}
System' &\overset{\text{def}}{=} [\{S1.S.sense, S2.S.sense\} : T]_{\{Device1,Device2,Host\}} \\
T &\overset{\text{def}}{=} \{S1.S.ch, M1.M.ch\} : \{M1.M.compute\} : \{M1.M.compute\} : \\
&\quad \{S2.S.ch, M2.M.ch\} : \{M2.M.compute\} : \\
&\quad \{S1.S.sense, S2.S.sense, M2.M.compute\} : T
\end{aligned}
$$

To simplify the proof, we make use of the definitions in Table 7. We refer quite frequently to these terms, and defining them here alleviates the need to repetitively copy them.

Recall that the "$\rhd$" operator is derived directly from Scope, so there is no formal need to provide separate rewrite rules for it. However, here we present some pertinent equations as a lemma; once we have done so, we no longer have to appeal to its definition in Section 4.

Lemma 5.1 For any CCSR terms $E, F, G \in \mathcal{E}$,

$$
\begin{aligned}
&\text{Pipe(1)} \quad NIL \rhd E = NIL \\
&\text{Pipe(2)} \quad (E + F) \rhd G = (E \rhd G) + (F \rhd G) \\
&\text{Pipe(3)} \quad (A : E) \rhd F = \begin{cases} (A * \emptyset) : F & \text{if } \surd \in A \\ A : (E \rhd F) & \text{otherwise} \end{cases}
\end{aligned}
$$

- Resources: $\mathit{Device1},\ \mathit{Device2},\ \mathit{Host} \in \mathcal{R}$
- Resource Mapping:

$$\rho(\{S1.S.sense, S1.S.error, S1.S.ch\}) = \{Device1\}$$
$$\rho(\{S2.S.sense, S2.S.error, S2.S.ch\}) = \{Device2\}$$
$$\rho(\{M1.M.compute, M1.M, ch, M2.M.compute, M2.M.ch\}) = \{Host\}$$

- Connection Sets:

$\{S1.S.ch, M1.M.ch\},\ \{S2.S.ch, M2.M.ch\},\ \{S1.S.sense\},\ \{S2.S.sense\},$
$\{S1.S.error\},\ \{S2.S.error\},\ \{M1.M.compute\},\ \{M2.M.compute\}$

- Priorities:

$Device1:\ \pi(S1.S.sense) = 1,\ \pi(S1.S.error) = 1,\ \pi(S1.S.ch) = 0$
$Device2:\ \pi(S2.S.sense) = 1,\ \pi(S2.S.error) = 1,\ \pi(S2.S.ch) = 0$
$Host:\ \pi(M1.M.compute) = 3,\ \pi(M1.M.ch) = 2,\ \pi(M2.M.compute) = 3,\ \pi(M2.M.ch) = 1$

Table 6: Resources, Resource Mapping, Connections and Priorities

$$\begin{aligned}
S1.S' &\overset{\text{def}}{=} (IDLE\, \Delta^{\emptyset}_{2}\, (NIL, \delta_{\infty}(S1.S.error), \{S1.S.ch\} : TERM) \triangleright \\
&\quad IDLE)\, \Delta^{\emptyset}_{5}\, (NIL, S1.S, NIL) \\
S1.S'' &\overset{\text{def}}{=} (\delta_{\infty}(S1.S.sense) \triangleright \\
&\qquad (IDLE\, \Delta^{\emptyset}_{2}\, (NIL, \delta_{\infty}(S1.S.error), \{S1.S.ch\} : TERM)) \triangleright \\
&\quad IDLE)\, \Delta^{\emptyset}_{5}\, (NIL, S1.S, NIL) \\
S2.S' &\overset{\text{def}}{=} (IDLE\, \Delta^{\emptyset}_{4}\, (NIL, \delta_{\infty}(S2.S.error), \{S2.S.ch\} : TERM) \triangleright \\
&\quad IDLE)\, \Delta^{\emptyset}_{5}\, (NIL, S2.S, NIL) \\
M1.M' &\overset{\text{def}}{=} \delta_{\infty}(M1.M.compute) \triangleright (\delta_{\infty}(M1.M.compute) \triangleright M1.M) \\
M2.M' &\overset{\text{def}}{=} \delta_{\infty}(M2.M.compute) \triangleright (\delta_{\infty}(M2.M.compute) \triangleright M2.M) \\
Sys' &\overset{\text{def}}{=} ([S1.S']_{\{Device1\}}\ {}_{\{Device1\}}\|_{\{Device2\}}\ [S2.S']_{\{Device2\}})\ {}_{\{Device1,Device2\}}\|_{\{Host\}}\ R\_Host
\end{aligned}$$

Table 7: Proof Definitions

Proof: Pipe(1) follows directly from the definition of "$\rhd$", Scope(1). Pipe(2) is just a restatement of Scope(2), while Pipe(3) follows from Scope(3). □

Proof Sketch. By using the definition of "δ_∞," the laws in $\mathcal{A}$ and Lemma 5.1, we can obtain that:

(1) $S1.S \;=\; (\{S1.S.sense\} : S1.S') \;+\; (\emptyset : S1.S'')$

Now invoking Close(2) and two applications of Close(3), we arrive at:

(2) $R\_Device1 \;=\; \{S1.S.sense\} : [S1.S']_{\{Device1\}} \;+\; \{\tau^0_{Device1}\} : [S1.S'']_{\{Device1\}}$

We now have the first opportunity to exploit Choice(5); that is, to prune the proof tree. Note that $\{\tau^0_{Device1}\} \prec \{S1.S.sense\}$, and thus,

(3) $R\_Device1 \;=\; \{S1.S.sense\} : [S1.S']_{\{Device1\}}$

Similarly,

(4) $R\_Device2 \;=\; \{S2.S.sense\} : [S2.S']_{\{Device2\}}$

Now we turn our attention to the R_Host resource. We can use the definition of "δ_∞", Pipe(2), Pipe(3) and finally Theorem 3.3 to derive that:

(5)
$$\begin{aligned} & M1.M\,_{\{Host\}}\|_{\{Host\}}\,M2.M \\ = \; & \{M1.M.ch\} : (M1.M'\,_{\{Host\}}\|_{\{Host\}}\,M2.M) \\ + \; & \{M2.M.ch\} : (M1.M\,_{\{Host\}}\|_{\{Host\}}\,M2.M') \\ + \; & \emptyset : (M1.M\,_{\{Host\}}\|_{\{Host\}}\,M2.M) \end{aligned}$$

Next we apply Close(2) and Close(3) to derive a transformation of R_Host. Note that no priority elimination may take place here, since the actions $\{M1.M.ch\}$, $\{M2.M.ch\}$ and $\{\tau^0_{Host}\}$ are mutually uncomparable under "$\prec$."

(6)
$$\begin{aligned} R\_Host \;=\; & \{M1.M.ch\} : [M1.M'\,_{\{Host\}}\|_{\{Host\}}\,M2.M]_{\{Host\}} \\ + \; & \{M2.M.ch\} : [M1.M\,_{\{Host\}}\|_{\{Host\}}\,M2.M']_{\{Host\}} \\ + \; & \{\tau^0_{Host}\} : R\_Host \end{aligned}$$

We may now exploit Theorem 3.3 to arrive the following rewrite for the $System$ term.

(7) $System \;=\; \{S1.S.sense, S2.S.sense, \tau^0_{Host}\} : Sys'$

Using the Scope, Pipe, Choice and Close laws we have that

(8)
$$\begin{aligned} & [S1.S']_{\{Device1\}} \\ = \; & \{S1.S.ch\} : [IDLE\,\Delta^{\emptyset}_4\,(NIL, S1.S, NIL)]_{\{Device1\}} \\ + \; & \{\tau^0_{Device1}\} : [((IDLE\,\Delta^{\emptyset}_1\,(NIL, \delta_\infty(S1.S.error), \{S1.S.ch\} : TERM)) \\ & \qquad \rhd IDLE)\,\Delta^{\emptyset}_4\,(NIL, S1.S, NIL)]_{\{Device1\}} \end{aligned}$$

And similarly,

$$\begin{aligned}
(9)\quad & [S2.S']_{\{Device2\}} \\
= \; & \{S2.S.ch\} : [IDLE \, \Delta^{\emptyset}_4 \, (NIL, S2.S, NIL)]_{\{Device2\}} \\
+ \; & \{\tau^0_{Device2}\} : [((IDLE \, \Delta^{\emptyset}_3 \, (NIL, \delta_\infty(S2.S.error), \{S2.S.ch\} : TERM)) \\
& \qquad \rhd IDLE) \, \Delta^{\emptyset}_4 \, (NIL, S2.S, NIL)]_{\{Device2\}}
\end{aligned}$$

Now we invoke Theorem 3.3 to derive a transformation for Sys', using our results so far.

$$\begin{aligned}
(10)\quad Sys' \; = \; & \{S1.S.ch, \tau^0_{Device2}, M1.M.ch\} : \\
& \quad (([IDLE \, \Delta^{\emptyset}_4 \, (NIL, S1.S, NIL)]_{\{Device1\}} \; {}_{\{Device1\}}\|_{\{Device2\}} \\
& \quad [((IDLE \, \Delta^{\emptyset}_3 \, (NIL, \delta_\infty(S2.S.error), \{S2.S.ch\} : TERM)) \\
& \qquad \rhd IDLE) \, \Delta^{\emptyset}_4 \, (NIL, S2.S, NIL)]_{\{Device2\}}) \; {}_{\{Device1, Device2\}}\|_{\{Host\}} \\
& \quad [M1.M' \, {}_{\{Host\}}\|_{\{Host\}} \, M2.M]_{\{Host\}}) \\
+ \; & \{\tau^0_{Device1}, S2.S.ch, M2.M.ch\} : \\
& \quad (([((IDLE \, \Delta^{\emptyset}_1 \, (NIL, \delta_\infty(S1.S.error), \{S1.S.ch\} : TERM)) \\
& \qquad \rhd IDLE) \, \Delta^{\emptyset}_4 \, (NIL, S1.S, NIL)]_{\{Device1\}}) \; {}_{\{Device1\}}\|_{\{Device2\}} \\
& \quad [IDLE \, \Delta^{\emptyset}_4 \, (NIL, S2.S, NIL)]_{\{Device2\}} \; {}_{\{Device1, Device2\}}\|_{\{Host\}} \\
& \quad [M1.M \, {}_{\{Host\}}\|_{\{Host\}} \, M2.M']_{\{Host\}}) \\
+ \; & \{\tau^0_{Device1}, \tau^0_{Device2}, \tau^0_{Host}\} : \\
& \quad (([((IDLE \, \Delta^{\emptyset}_1 \, (NIL, \delta_\infty(S1.S.error), \{S1.S.ch\} : TERM)) \\
& \qquad \rhd IDLE) \, \Delta^{\emptyset}_4 \, (NIL, S1.S, NIL)]_{\{Device1\}}) \; {}_{\{Device1\}}\|_{\{Device2\}} \\
& \quad [((IDLE \, \Delta^{\emptyset}_3 \, (NIL, \delta_\infty(S2.S.error), \{S2.S.ch\} : TERM)) \\
& \qquad \rhd IDLE) \, \Delta^{\emptyset}_4 \, (NIL, S2.S, NIL)]_{\{Device2\}}) \; {}_{\{Device1, Device2\}}\|_{\{Host\}} \\
& \quad Host
\end{aligned}$$

But we can reduce the term's complexity by using Choice(5), i.e., preemption.

$$\begin{aligned}
(11)\quad Sys' \; = \; & \{S1.S.ch, \tau^0_{Device2}, M1.M.ch\} : \\
& \quad (([IDLE \, \Delta^{\emptyset}_4 \, (NIL, S1.S, NIL)]_{\{Device1\}} \; {}_{\{Device1\}}\|_{\{Device2\}} \\
& \quad [((IDLE \, \Delta^{\emptyset}_3 \, (NIL, \delta_\infty(S2.S.error), \{S2.S.ch\} : TERM)) \\
& \qquad \rhd IDLE) \, \Delta^{\emptyset}_4 \, (NIL, S2.S, NIL)]_{\{Device2\}}) \; {}_{\{Device1, Device2\}}\|_{\{Host\}} \\
& \quad [M1.M' \, {}_{\{Host\}}\|_{\{Host\}} \, M2.M]_{\{Host\}})
\end{aligned}$$

There is a leap to the next step, in which the term Sys' is "flattened out." However, the procedure is similar to the previous one in the previous steps, and we omit the details for the sake of brevity.

$$\begin{aligned}
(12)\quad Sys' \; = \; & \{S1.S.ch, \tau^0_{Device2}, M1.M.ch\} : \{\tau^0_{Device1}, \tau^0_{Device2}, M1.M.compute\} : \\
& \{\tau^0_{Device1}, \tau^0_{Device2}, M1.M.compute\} : \{\tau^0_{Device1}, S2.S.ch, M2.M.ch\} : \\
& \{\tau^0_{Device1}, \tau^0_{Device2}, M2.M.compute\} : \\
& \{S1.S.sense, S2.S.sense, M2.M.compute\} : Sys'
\end{aligned}$$

Now, by Close(3) we have that

$$(13)\quad \mathit{System}' \;=\; \{S1.S.\mathit{sense}, S2.S.\mathit{sense}, \tau^0_{\mathit{Host}}\} : [T]_{\{\mathit{Device1},\mathit{Device2},\mathit{Host}\}}$$

and thus it remains to be shown that $\mathit{Sys}' \sim_\pi [T]_{\{\mathit{Device1},\mathit{Device2},\mathit{Host}\}}$. But by six applications of Close(3), it follows that

$$\begin{aligned}(14)\quad & [T]_{\{\mathit{Device1},\mathit{Device2},\mathit{Host}\}} \\ &= \{S1.S.\mathit{ch}, \tau^0_{\mathit{Device2}}, M1.M.\mathit{ch}\} : \{\tau^0_{\mathit{Device1}}, \tau^0_{\mathit{Device2}}, M1.M.\mathit{compute}\} : \\ &\quad \{\tau^0_{\mathit{Device1}}, \tau^0_{\mathit{Device2}}, M1.M.\mathit{compute}\} : \{\tau^0_{\mathit{Device1}}, S2.S.\mathit{ch}, M2.M.\mathit{ch}\} : \\ &\quad \{\tau^0_{\mathit{Device1}}, \tau^0_{\mathit{Device2}}, M2.M.\mathit{compute}\} : \\ &\quad \{S1.S.\mathit{sense}, S2.S.\mathit{sense}, M2.M.\mathit{compute}\} : [T]_{\{\mathit{Device1},\mathit{Device2},\mathit{Host}\}}\end{aligned}$$

Now since fixpoints are unique, it follows that $\mathit{Sys}' \sim_\pi [T]_{\{\mathit{Device1},\mathit{Device2},\mathit{Host}\}}$, as they solve the same recursive equation. □

The importance of preemption elimination (law Choice(5)) cannot be underestimated here. A simple way to illustrate this is to set $\pi(S1.S.\mathit{ch})$ to 1, and thus to give $S2.S.\mathit{ch}$ the same priority as that of $S1.S.\mathit{ch}$. In this case, the initial choice between the actions $\{S1.S.\mathit{ch}, \tau^0_{\mathit{Device2}}, M1.M.\mathit{ch}\}$ and $\{\tau^0_{\mathit{Device1}}, S2.S.\mathit{ch}, M2.M.\mathit{ch}\}$ becomes nondeterministic. And if the branch corresponding to $\{\tau^0_{\mathit{Device1}}, S2.S.\mathit{ch}, M2.M.\mathit{ch}\}$ is taken, $M2.M.\mathit{compute}$ will execute for 2 time units, during which time the execution of $M1.M.\mathit{ch}$ will be blocked. But since the deadline for $S1.S.\mathit{ch}$ will have expired, $S1.S.\mathit{error}$ would have been executed. In fact, we can prove that in this case, $\mathit{System} \sim_\pi \mathit{System}''$, where

$$\begin{aligned}&\mathit{System}'' \overset{\text{def}}{=} [\{S1.S.\mathit{sense}, S2.S.\mathit{sense}\} : T']_{\{\mathit{Device1},\mathit{Device2},\mathit{Host}\}} \\ &T' \overset{\text{def}}{=} (\{S1.S.\mathit{ch}, M1.M.\mathit{ch}\} : \{M1.M.\mathit{compute}\} : \{M1.M.\mathit{compute}\} : \\ &\qquad \{S2.S.\mathit{ch}, M2.M.\mathit{ch}\} : \{M2.M.\mathit{compute}\} : \\ &\qquad \{S1.S.\mathit{sense}, S2.S.\mathit{sense}, M2.M.\mathit{compute}\} : T') \\ &\quad + (\{S1.S.\mathit{ch}, M1.M.\mathit{ch}\} : \{M2.M.\mathit{compute}\} : \{M2.M.\mathit{compute}, S1.S.\mathit{error}\} : \\ &\qquad \emptyset : \emptyset : \{S1.S.\mathit{sense}, S2.S.\mathit{sense}, M2.M.\mathit{compute}\} : T')\end{aligned}$$

That is, $S1.S$ may starve completely, and record an error during each period.

6 Conclusion

We have described a two-tiered approach to the specification and verification of real-time systems. The CSR Specification Language is used to design the functionality of a real-time program, and an accompanying configuration schema is then used to specify its structure. To analyze the resulting system, we automatically translate it into a term in the CCSR process algebra.

As we have shown, the CCSR semantics heavily rely on our notions of resource allocation, synchronization and priority. By considering the very subtle interplay between these factors, we have defined a notion of preemption that leads to a congruence relation over the terms. In turn, this result yields a fully substitutive proof system, facilitating the algebraic verification of our original real-time system.

The CSR Specification Language has continually matured since we introduced it in [5], and its expressivity has correspondingly increased. In previous versions, for example, the **scope** construct was allowed only one trigger, which could be either a timeout *or* an interrupt. It is true that scopes could be nested, but nesting alone does not provide for the same expressivity as multiple triggers. Consider, for example, the following fragments $T1$ and $T2$:

```
T1 = scope do
        S
     interrupt recv(ch1) → S1
     timeout 50 → S2
     od
```

```
T2 = scope do
        scope do
           S
        interrupt recv(ch1) → S1
        od
     timeout 50 → S2
     od
```

Certainly $T1$ and $T2$ do not have the same meaning. In $T1$, if **recv**(ch1) is executed, then the timeout is disabled; in $T2$ the timeout remains active even while the exception-handler S1 is being executed. Since many real-time systems have multiple interrupts, it seems that the new **scope** statement is a vast improvement.

Another improvement over the older versions of CSR is the decoupling of the language and configuration issues. In those earlier versions, we did not provide a separate configuration language; instead, we used two parallel operators: "&" denoted interleaving concurrency, while true parallelism was represented by the "||" symbol. Also, connections were declared using global naming conventions; specifically, a unique event "$ch1$?" was forced to synchronize with its counterpart event "$ch1$!". We found these approaches clumsy for several reasons. First, the events "$ch1$?" and "$ch1$!" had unique, system-wide names, which prevented other processes from using different "instances" of them. Second, it enforced one-to-one communication, which precluded broadcasting and other, more sophisticated schemes. But of most importance, by distinguishing at the functional level between the two types of parallelism, we found it difficult to minutely alter the specification without rewriting it completely. For example, suppose we were to determine whether the sensor-monitor system required a dedicated resource for each monitor module. This could quite easily be accomplished within a new configuration schema, while the same functional specification could be used.

Our long-term goal is to develop techniques and tools that can be used to build a real-time system with predictably correct behavior. As a step toward meeting our goal, we have implemented a translator from the CSR specification language to the CCSR process algebra. We are currently investigating the implementation of the CCSR proof system, as well as extending the CSR specification language to support variables and probabilistic timing behavior.

References

[1] R. Alur, C. Courcoubetis, and D. Dill. Model-Checking for Real-Time Systems. In *Proc. of IEEE Symposium on Logic in Computer Science*, 1990.

[2] R. Cleaveland and M. Hennessy. Priorities in Process Algebras. *Information and Computation*, 87:58–77, 1990.

[3] M.K. Franklin and A. Gabrielian. A Transformational Method for Verifying Safety Properties in Real-Time Systems. In *Proc. IEEE Real-Time Systems Symposium*, pages 112–123, December 1989.

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

[5] R. Gerber and I. Lee. Communicating Shared Resources: A Model for Distributed Real-Time Systems. In *Proc. 10th IEEE Real-Time Systems Symposium*, 1989.

[6] R. Harper. Introduction to standard ML. Technical Report ECS-LFCS-86-14, Department of Computer Science, University of Edinburgh, The King's Buildings—Edinburgh EH9 3JZ—Scotland, 1986.

[7] J. Hooman. *Specification and Compositional Verification of Real-Time Systems.* PhD thesis, Eindhoven University of Technology, 1991.

[8] C. Huizing, R. Gerth, and W.P. de Roever. Full Abstraction of a Denotational Semantics for Real-time Concurrency. In *Proc.* 14th *ACM Symposium on Principles of Programming Languages*, pages 223–237, 1987.

[9] M. Joseph and A. Goswami. What's 'Real' about Real-time Systems? In *IEEE Real-Time Systems Symposium*, 1988.

[10] J.E. Coolahan Jr. and N. Roussopoulos. Timing Requirements for Time-Driven Systems Using Augmented Petri Nets. *IEEE Trans. Software Eng.*, SE-9(5):603–616, September 1983.

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

[12] N. Lynch and M. Tuttle. An Introduction to Input/Output Automata. Technical Report MIT/LCS/TM-373, Laboratory for Computer Science, Massachusetts Institute of Technology, 1988.

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

[14] J.S. Ostroff and W.M. Wonham. Modelling, Specifying and Verifying Real-time Embedded Computer Systems. In *Proc. IEEE Real-Time Systems Symposium*, pages 124–132, December 1987.

[15] G.M. Reed and A.W. Roscoe. Metric Spaces as Models for Real-Time Concurrency. In *Proceedings of Math. Found. of Computer Science, LNCS 298*, 1987.

[16] A.C. Shaw. Reasoning About Time in Higher-Level Language Software. *IEEE Transactions on Software Engineering*, 15(7):875–889, 1989.

[17] W. Yi. Ccs + time = an interleaving model for real time systems. In *ICALP*, 1991.