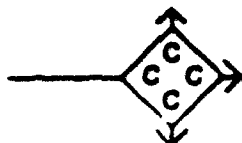


**Best
Available
Copy**



COMPUTER COMMAND AND CONTROL COMPANY

2300 CHESTNUT STREET, SUITE 230 • PHILADELPHIA, PA 19103
215-854-0555 FAX: 215-854-0665

①

Software Tools for Formal Specification and Verification of Distributed Real-Time Systems:

A Final Report

AD-A283 477



Submitted Under
Contract Number N00014-94-C-0081
Line Item Number 0001AB

July 29, 1994

Submitted to:
Scientific Officer
Attn: Gary M. Koob
Office of Naval Research
Ballston Tower One
800 North Quincy Street
Arlington, VA 22217-5660

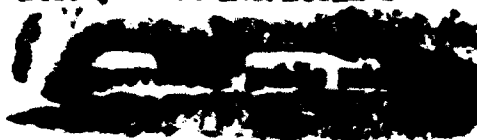
DTIC
ELECTE
AUG 17 1994
S G D

Prepared by:
J. Kim (Principal Investigator), J. Choi and I. Lee
Computer Command and Control Company
2300 Chestnut Street, Ste. 230
Philadelphia, PA 19103

708 94-24302
■■■■■■■■■■

94 8 01 041

DTIC QUALITY INSPECTED 1



Summary

The goals of Phase I of SBIR Contract N00014-94-C-0081 are to design in detail a toolkit environment based on formal methods for the specification and verification of distributed real-time systems and to evaluate the design. The evaluation of the design includes investigation of both the capability and potential usefulness of the toolkit environment and the feasibility of its implementation. To meet these goals, we have

1. Designed a graphical specification language based on ACSR (Algebra of Communicating Shared Resources) and develop a graph attribute grammar of the language.
2. Designed the overall implementation structure of the toolkit environment including a menu-driven graphical user interface, a graphical composition tool for specifications, analysis tools and the interface between the components.
3. Evaluated the implementations of existing graphical specification systems to see whether they can be used without major changes in implementing our graphical specification language.
4. Identified state minimization algorithms, analysis techniques (e.g., simulation), and verification techniques (e.g., equivalence test, state exploration, and model checking) that we plan to implement.
5. Identified a real-time logic and a model checking method as well as a proof system for our graphical specification language.

The two salient aspects of our graphical specification language are:

1. Its semantics is precisely the same as that of ACSR.
2. It includes features specially useful for scalability.

Since ACSR has a well-defined formal semantics, there is no semantics ambiguity in our graphical specification language.

Contents

1 Introduction	4
2 Background	9
2.1 ACSR: Algebra of Communicating Shared Resources	9
2.2 Modechart	14
2.3 Semantic Difference and Translation	17
3 GCSR: Graphical CSR	26
3.1 GCSR Syntax	26
3.2 Valid GCSR Specification	30
3.3 Informal GCSR Semantics	32
3.4 Formal GCSR Semantics	35
4 Examples	41
5 Automatic Analysis Techniques	53
5.1 State Minimization Algorithms	53
5.2 Additional Equivalence Relations for ACSR	55
5.2.1 LTS Based Equivalences	55
5.2.2 Trace Based Equivalences	62
5.3 Logic for Communicating Shared Resources	64
6 Phase II Implementation Plan	67
6.1 The Overall Structure of the Toolkit Environment	67
6.2 Implementation	69

Statement A per telecon Gary Koob
Office of the Chief of Naval Research
800 north Quincy Street Code 1133
Arlington, VA 22217-5000
NW 8/17/94

Accession For	
NTIS CRA&I	☒
DTIC TAB	☐
Unannounced	☐
Justification _____	
By _____	
Distribution /	
Availability Codes	
Dist	Avail and/or Special
A-1	

List of Figures

1	Syntax of ACSR Process Expressions	10
2	An example of modechart	15
3	An example of translation of ACSR to Modechart	21
4	Modechart specification of the HARM Missile example	23
5	GCSR types of nodes (top) and edges (bottom).	27
6	Extended BNF for GCSR	29
7	Extended BNF for GCSR (continued)	30
8	ACSR to GCSR translation.	36
9	ACSR to GCSR translation	37
10	GCSR simple graphical reductions and short hand notations.	38
11	GCSR simple graphical reductions and short hand notations (continued). . .	39
12	GCSR complex graphical reduction	40
13	Telephone Example.	42
14	Mouse Example.	44
15	Rail Road Crossing Example ACSR Specification	45
16	Railroad Crossing Example: GCSR Specification for ACSR terms in Figure 15	46
17	Example of graphical reductions applied to the example of Figure 16	47
18	Further unfolding applied on Train, Gate of the example of Figure 17. . . .	48
19	Further unfolding applied on Train, Gate and Control of the example of Figure 18.	49
20	Railroad Crossing Example: All reference nodes have been replaced.	50
21	Sensor Example. Reference nodes have been replaced.	51
22	Router Example	52
23	The Coarsest Partition	54
24	GCSR Toolkit Environment	68
25	Work Window: the HARM missile example	70

1 Introduction

This document is the final report for Phase I of SBIR Contract N00014-94-C-0081. It includes the results of Phase I as well as a brief plan for Phase II.

The result reported here represents an important development for implementing a graphical toolkit environment based on a formal method for the specification and verification of distributed real-time systems. Formal methods treat system components as mathematical objects and provide mathematical models to describe and predict the observable properties and behaviors of these objects. There are several advantages to using formal methods for the specification and analysis of real-time systems. They include (1) the early discovery of ambiguities, inconsistencies and incompleteness in informal requirements; (2) the automatic or machine-assisted analysis of the correctness of specifications with respect to requirements; and (3) the evaluation of design alternatives without expensive prototyping.

As computers become ubiquitous, they are increasingly used in mission critical environments. In particular, Navy systems rely increasingly on real-time software to accomplish their intended goals. Typical mission critical applications are control systems, monitoring systems and communication systems. Any failure of such computer systems may cause a great financial loss, environmental disaster or even the loss of lives. Potential high cost associated with an incorrect operation of these systems has created a demand for a rigorous framework in which various design alternatives can be formally specified and rigorously analyzed and tested before implementation.

It is commonly believed that future critical systems will be more complex due to increased demands on their functionalities as well as the size of the problem domain. Thus, it will be difficult to analyze and test correctness without computer-aided tools. In addition, these systems are costly to prototype, requiring careful prediction of timing properties before implementation and evaluation of design alternatives. Thus, it is important to develop a formal framework that supports automatic and computer-aided analysis as well as scalable specifications to effectively cope with increased complexity.

There has been significant progress in the development of real-time formal methods [36, 16, 32, 27, 26, 3, 9, 39, 2, 21, 10, 14, 12, 29, 28, 37, 22]. Much of this work falls into the traditional categories of untimed systems such as temporal logics, assertional methods, net-based models, automata theory and process algebras. While most formal models for real-time systems capture delays due to process synchronization, they abstract out resource-specific details by assuming idealistic operating environments. On the other hand, scheduling

algorithms and analyzers [25, 43, 34, 40, 41, 4, 38, 13, 42, 12, 35] developed for real-time systems capture contention on shared resources. In these approaches, however, the underlying computation model is generally limited to simple precedence relations between processes. Since complex interactions between processes are not captured, these approaches cannot be used to prove properties other than schedulability.

To bridge the gap between these two disciplines, we have developed a formal framework, called ACSR (Algebra of Communicating Shared Resources) [11, 23]. ACSR is based on a computation model that supports the notions of resources and priorities. These notions are necessary since the timed behavior of the system is affected not only by the time that actions take to execute but by delays introduced due to the scheduling of actions that compete to use the same resource. ACSR allows the specification of resource requirements and the verification of timing properties to take the availability and scheduling of resources.

One goal of this project was to develop a scalable graphical formalism that can be used to specify large real-time systems. For this, we developed an graphical formal specification language for real-time systems, called GCSR (Graphical Communicating Shared Resources). GCSR is a formalism developed by combining two real-time methods, Modechart and ACSR. GCSR is based on ACSR; that is, there is a well-defined mapping between GCSR and ACSR. This correspondence is very important since this implies that the semantics of GCSR is well-defined because the theory of ACSR has been completely worked out.

GCSR is similar to Modechart in a sense that it is a graphical specification language and it adopts basic graphical ideas such as node, directed edge and time of Modechart. GCSR, however, is different from Modechart because GCSR supports the notions of resource and priority. In addition, GCSR is action based and has constructs to build the modular and hierarchical specification of a real-time system. Because of the resource-based characteristics of ACSR, GCSR also supports the integrated specification of functional as well as resource aspects of a real-time system. In addition to modularity, this integrated specification capability also distinguishes GCSR from Modechart.

There are several advantages to our formal method GCSR:

- **Scalability:** For formal specifications to be practically useful, specifications must be scalable. GCSR is designed to support scalable specification: in particular, GCSR supports hierarchical structures through nesting, top-down and bottom-up development of specifications through naming and refinement, and modularity through limiting visibility of events.

- **Graphical and Textual Specification Languages:** It is easy to present and understand large scale systems at high-level using a graphical form. However, it sometimes is natural and faster to specify the details of a large system textually. In our toolkit environment, a real-time system can be specified through the mixed use of GCSR and ACSR, depending on whichever medium is more convenient.. Since there is a natural mapping between GCSR and ACSR (and vice versa), the mixed form of specifications does not result any ambiguity.
- **Equivalence:** The equivalence relation of ACSR indicates when two systems behave the same. Furthermore, equivalent processes can be substituted one another inside other process. This makes it possible for modular specification and analysis of large systems. In addition, process may be minimized with respect to the equivalence relation before analyzed, which often times simplifies verification. These techniques are obviously applicable to GCSR since there is a natural mapping between GCSR and ACSR. For example, one can check whether or not two GCSR specifications are bisimilar. This is remarkable since other graphical specification languages usually do support such an equivalence relation. When two specifications are equivalent, one specification inside a large system specification can be substituted by the other one. We note that we have identified many useful equivalence relations as part of the project (see Section 5.2).
- **Resources:** Since real-time systems consist of several shared resources such as cpu, memory and sensors, it is natural to model these resources in terms of the primitive notion of resource in GCSR. Furthermore, GCSR can be used to consider resource-induced constraints during the design stage of the development cycle and to eliminate unimplementable design alternatives without expensive prototyping.
- **Priority:** The notion of priorities is always present in real-time systems to arbitrate resource contention. Hence, every formal method developed for real-time systems should support the notion of priority; otherwise, it may not possible model system's behaviors correctly. It sometimes is possible to encode the notion of priority using boolean formulas. This, however, can result in a specification which is difficult to understand due to explosion on the number of Boolean conditions. For instance, Modechart model of the HARM missile system is quite complicate due to priority encoding [8], whereas the GCSR model of the same missile system is simple and clear. These examples are presented in later sections.

- **Formal Semantics:** As stated earlier, the semantics of GCSR is defined using ACSR, a real-time process algebra. The formal semantics determines the behaviors of a GCSR specification precisely and unambiguously. This in turn makes it possible to prove the properties of a GCSR specification rigorously and mathematically.
- **Executable:** Since ACSR has well-defined operational semantics, a GCSR specification is executable. There are a few advantages for executable specification. One is that it can be used as a fast prototype. Another is that may help to detect unintended behaviors of the specification, before attempting to prove its correctness.
- **Synchronous and Asynchronous Communication:** GCSR support both synchronous and asynchronous communications. Asynchronous broadcasting events are essential to describe events from the environment. Furthermore, we have experienced in many occasions the needs for broadcasting events during the specification of real-time systems.
- **Dense and Discrete Time:** Although GCSR is based on discrete time, it can easily be extended to dense time GCSR since we have already completed the development of dense time ACSR [6].

The toolkit environment we designed consists of a menu-driven graphical user interface, a graphics editor, analysis tools and the interfaces between the components.

The graphical user interface provides a set of commands for managing specifications, activating the graphics editor, performing formal analysis of a specification and terminating the environment.

A specification is represented in terms of graphs in the environment. A user creates, views and modifies a specification using the graphics editor which is based on icons and menus. The editor provides the following operations on graphical objects: copy, delete, paste, align, enlarge, shrink, etc. A large-scale system is represented by a set of graphs and their hierarchy. The editor provides tools of scrolling the graph window and navigating the hierarchy of the graphs.

A specification can be converted automatically to a state machine for analysis. Once a specification has been converted to a state machine, the analyst may execute the specification and test it to determine its reasonableness. The analyst may then apply optional state minimization algorithms to the specification. Though the effectiveness of state minimization will vary, successful application of this process can significantly reduce the computing resources required by later analysis phases.

The analysis of the state machine will be carried out using the following analysis tools: The first analysis tool is a simulation tool that demonstrates operational behaviors of a specification by executing the state machine. The second analysis tool is a model checking tool that will allow the specification to be tested against real-time logic propositions. The third analysis tool is a state exploration tool that can be used to generate valid traces of actions for the specification being analyzed. The fourth analysis tool will test for equivalence between two or more alternative specifications.

The rest of this report is organized as follows: In Section 2, we describe briefly ACSR and Modechart. The description of ACSR includes a set of extensions that we made to ACSR for this project. We also compare their semantics and motivate why we developed a new graphical language GCSR instead of using Modechart. Section 2.3 contrasts their expressiveness using the HARM missile system example. Section 3 details the syntax and semantics of GCSR. We explain how to translate from ACSR to GCSR and vice versa. In addition, we describe a set of graph reductions that are developed to enhance the readability of GCSR specifications. Section 4 provides example GCSR specifications for telephone system, mouse system, railroad crossing, sensor system, and router systems. Section 5 summarizes our investigation on automatic analysis techniques. Section 5.1 briefly explains the state minimization algorithm we plan to implement. Section 5.2 identifies a set of equivalence relations that are weaker than the ones we currently have. Section 5.3 describes logic that we have designed to facilitate the partial specification (i.e., requirement specification) and model checking of GCSR. Section 6 describes the overall structure of the toolkit environment that we proposed to implement during Phase II of this project and identifies existing software components that can be reused/adapted for the implementation.

2 Background

The proposed tools are based on our real-time algebra called ACSR (Algebra of Communicating Shared Resources). Section 2.1 briefly overviews ACSR. During Phase I, we have investigated a possibility of using Modechart as a graphical language for ACSR. In Section 2.2, we briefly explain Modechart. Section 2.3 contrasts the semantic differences between ACSR and Modechart. As shown in 2.3, it is possible to translate Modechart to ACSR. However, because ACSR has more features, it is not possible to capture all aspects of ACSR without extending Modechart. Our attempt to extend Modechart has resulted in a new graphical language, which is described in Section 3. As an illustration of the expressiveness of ACSR, we specify the HARM Missile example in ACSR and compare it with the Modechart specification 2.3.

2.1 ACSR: Algebra of Communicating Shared Resources

This section describes the syntax and operational semantics of ACSR (the Algebra of Communicating Shared Resources), a real-time process algebra that incorporates the notions of communication, concurrency, resources, and priorities into a single formalism. ACSR is completely described in [23].

Actions. When modeling a process with algebraic expressions, the progress of the process through its interactions with external agents is modeled by the execution of discrete "actions." ACSR uses two distinct action types to model computation: time and resource consuming actions, and instantaneous events.

Timed Actions.—We consider a system to be composed of a finite set of serially reusable resources, denoted by $\mathcal{R}$. An action that consumes one "tick" of time is drawn from the domain $\mathcal{P}(\mathcal{R} \times \mathcal{N})$ (the power set of $\mathcal{R} \times \mathcal{N}$), with the restriction that each resource be represented at most once. As an example, the singleton action, $\{(r, p)\}$, denotes the use of some resource $r \in \mathcal{R}$ running at priority level p . Priority values range over $\mathcal{N}$, with 0 being the lowest (least pressing) priority, and priority increasing with increasing p . The action $\emptyset$ represents idling for one time unit, since all resources are inactive.

We use $\mathcal{D}_R$ to denote the domain of timed actions, and we let A, B, C range over $\mathcal{D}_R$. We define $\rho(A)$ to be the set of resources used by the action A , e.g. $\rho(\{(r_1, p_1), (r_2, p_2)\}) = \{r_1, r_2\}$. We also use $\pi_r(A)$ to denote the priority level of the action A in the resource r , e.g. $\pi_{r_1}(\{(r_1, p_1), (r_2, p_2)\}) = p_1$. By convention, if r is not in $\rho(A)$, then $\pi_r(A) = 0$.

Instantaneous Events—We call instantaneous actions *events*, which provide the basic synchronization in our process algebra. An event is denoted by a pair (a, p) , where a is the label of the event, and p is its priority. Again, priority values range over $\mathbf{N}$ with 0 being the least pressing priority. Labels are drawn from the set $L \cup \bar{L} \cup \{\tau\}$, where if a is a given label, we say that $\bar{a}$ is its *inverse* label, i.e. $\bar{\bar{a}} = a$. A label and its inverse can be thought of as naming complementary ends of a communication channel. As in CCS[31], the special identity label, τ , arises when two events with inverse labels are executed in parallel.

We use $\mathcal{D}_E$ to denote the domain of events, and let e, f and g range over $\mathcal{D}_E$. We use $l(e)$ and $\pi(e)$ to represent the label and priority, respectively, of the event e .

Finally, the entire domain of actions is $\mathcal{D} = \mathcal{D}_R \cup \mathcal{D}_E$, and we let α and β range over $\mathcal{D}$.

ACSR Syntax and Semantics. Let P, P_1, P_2 , and P_3 range over the domain of terms, and let X range over the domain of term variables. Additionally, we assume an infinite set of free term variables, FV . ACSR's syntax is given by the grammar of Figure 1.

$$P ::= \text{NIL} \mid A : P \mid e.P \mid P_1 + P_2 \mid P_1 \parallel P_2 \mid \\ P \Delta_t^*(P_1, P_2, P_3) \mid [P]_I \mid P \setminus F \mid \text{rec } X.P \mid X$$

Figure 1: Syntax of ACSR Process Expressions

NIL is a process that executes no action (i.e., it is initially deadlocked). There are two prefix operators, corresponding to the two types of actions. The first, $A : P$, executes a timed, resource-consuming action A , consumes one time unit, and proceeds to the process P . The second prefix operator, $e.P$, executes the instantaneous event e , and proceeds to P . The Choice operator $P_1 + P_2$ represents nondeterminism - either of the processes may be chosen to execute, subject to the event offerings and resource limitations of the environment. The operator $P_1 \parallel P_2$ is the concurrent execution of P_1 and P_2 .

The Scope construct $P \Delta_t^*(P_1, P_2, P_3)$ binds the process P by a temporal scope[24], and incorporates both the features of timeouts and interrupts. We call t the *time bound*, where $t \in \mathbf{N}^+ \cup \{\infty\}$ (i.e., t is either a non-negative integer or infinity). P executes for a maximum of t time units. The scope may be exited in a number of ways. First, if P successfully terminates within time t by executing an event labeled with $\bar{a}$, then control proceeds to the "success-handler" P_1 (here, a may be any label other than τ). Second, if P fails to terminate within time t , then control proceeds to the "timeout exception-handler" P_2 . Lastly, at any

time while P is executing it may be interrupted by P_3 's execution of a timed action or instantaneous event, and the scope is then departed.

The Close operator, $[P]_I$, produces a process P that monopolizes the resources in $I \subseteq \mathcal{R}$. The Restriction operator, $P \setminus F$, limits the behavior of P : events with labels in F are permitted to execute only if they synchronize and become the internal event τ . The process $\text{rec } X.P$ denotes standard recursion, allowing the specification of infinite behaviors. The term X , without a "rec" binding, is a free variable that belongs to the infinite set FV .

The semantics of ACSR is defined in two steps [23]: unprioritized semantics that provides all behaviors ignoring priority information and then prioritized semantics that eliminates unprioritized behaviors through priority arbitration. Here, we only describe the unprioritized semantics of ACSR by developing the *unconstrained* transition system. A transition is denoted as $P \xrightarrow{\alpha} P'$, for P and P' processes and α an action. Within " $\rightarrow$ " no priority arbitration is made between actions.

The two rules for the prefix operators are *axioms*, i.e. they have premises of *true*. There is one rule for time-consuming actions, and one for events.

$$\text{ActT} \quad \frac{-}{A : P \xrightarrow{A} P} \quad \text{ActI} \quad \frac{-}{e.P \xrightarrow{e} P}$$

For example, the process $\{(r_1, p_1), (r_2, p_2)\} : P$ simultaneously uses resources r_1 and r_2 for one time unit, and then proceeds to P . Alternatively, the process $(a, p).P$ executes the event " (a, p) ," and proceeds to P .

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

$$\text{ChoiceL} \quad \frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'} \quad \text{ChoiceR} \quad \frac{Q \xrightarrow{\alpha} Q'}{P + Q \xrightarrow{\alpha} Q'}$$

As an example, $(a, 7).P + \{(r_1, 3), (r_2, 7)\} : Q$ may choose between executing the event $(a, 7)$ or the time-consuming action $\{(r_1, 3), (r_2, 7)\}$. The former behavior is deduced from rule **ActI**, while the latter is deduced from **ActT**.

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

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

Note that timed transitions are truly synchronous, in that the resulting process advances only if both of the constituents take a step. Thus, care must be taken to insure that every

step of a timed computation offers one or more timed alternatives, lest the lack of a timed step should "stop the clock." The condition $\rho(A_1) \cap \rho(A_2) = \emptyset$ mandates that each resource is truly sequential, and that only one process may use a given resource during any time step.

The next three laws are for event transitions. As opposed to timed actions, events may occur asynchronously (as in CCS and related interleaving models).

$$\text{ParIL} \quad \frac{P \xrightarrow{e} P'}{P \parallel Q \xrightarrow{e} P' \parallel Q}$$

$$\text{ParIR} \quad \frac{Q \xrightarrow{e} Q'}{P \parallel Q \xrightarrow{e} P \parallel Q'}$$

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

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

The Scope operator possesses a total of five transition rules describing the various behaviors induced by a temporal scope. The first two rules show that as long as $t > 0$ and P does not execute an event labeled with $\bar{b}$, the executions of P continue.

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

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

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

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

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

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

Finally, **ScopeI** shows that the process S may interrupt (and kill) P while the scope is still active. Note that the interrupt step may be one of several options offered simultaneously. Also note that scopes may be nested arbitrarily deeply. When nested scopes use the same action to trigger an interrupt, the interrupting action will propagate upward after each interruption. This follows from rule **ScopeI**, which specifies that the interrupting action is preserved.

$$\text{ScopeI} \quad \frac{S \xrightarrow{\sigma} S'}{P \Delta_i^*(Q, R, S) \xrightarrow{\sigma} S'} \quad (t > 0)$$

The Restriction operator defines a subset of instantaneous events that are excluded from the behavior of the system. This is done by establishing a set of labels, F ($\tau \notin F$), and deriving only those behaviors that do not involve events with those labels. Note that while $P \setminus F$ restricts P from communicating with other processes using labels in F , concurrent sub-processes within P are free to interact with one another using these labels. Thus, restriction can be viewed as the assignment of dedicated channels for communication within a process. Note also that time-consuming actions are unaffected by restriction.

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

While Restriction assigns dedicated channels to processes, the Close operator assigns dedicated resources. When a process P is embedded in a closed context such as $[P]_I$, we ensure that there is no further sharing of the resources in I . Assume that P executes a time-consuming action A . If A utilizes less than the full resource set I , the action is augmented with $(r, 0)$ pairs for each unused resource $r \in I - \rho(A)$. The way to interpret Close is as follows. A process may idle in one of two ways: it may either release its resources during the idle time (represented by $\emptyset$), or it may hold them. Close ensures that the resources are held. (Instantaneous events are not affected.)

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

$$\text{CloseI} \quad \frac{P \xrightarrow{\sigma} P'}{[P]_I \xrightarrow{\sigma} [P']_I}$$

The operator $\text{rec } X.P$ denotes guarded recursion, allowing the specification of infinite behaviors.

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

$P[rec\ X.P/X]$ is the standard notation for substitution of $rec\ X.P$ for each free occurrence of X in P . By guarded recursion, we mean that all occurrences of X in $rec\ X.P$ are preceded by some action α . For example, $rec\ X.(A : X)$ is guarded, while $rec\ X.(X + A : X)$ is not.

As an example, consider $rec\ X.(A : X)$, which executes the resource-consuming action "A" forever. By ActT, $A : (rec\ X.(A : X)) \xrightarrow{A} rec\ X.(A : X)$, so by Rec, $rec\ X.(A : X) \xrightarrow{A} rec\ X.(A : X)$.

Extensions to ACSR. To make ACSR practical, we made two extensions. One extension is to include the notion of broadcasting. In ACSR, we introduce two symbols, ?? and !! for broadcasting: $a??$ denotes receiving an event through the broadcasting channel a , and $a!!$ sending an event. The operational semantics for $a??$ and $a!!$ are as follows:

$$\frac{P \xrightarrow{a!!} P', Q \xrightarrow{a??} Q'}{P \parallel Q \xrightarrow{a!!} P' \parallel Q'} \quad \frac{P \xrightarrow{a!!} P', Q \xrightarrow{a??} Q'}{P \parallel Q \xrightarrow{a!!} P' \parallel Q'}$$

$$\frac{}{a?? . P \xrightarrow{0} a?? . P} \quad \frac{Q \xrightarrow{a??} Q', P \xrightarrow{a??} P'}{Q \parallel P \xrightarrow{a??} Q' \parallel P'}$$

The above operational semantics are basic concepts of broadcasting communication which are based on the following two facts:

- Receiver waits indefinitely until sender sends a message.
- Sender proceeds to next process immediately after sending a message without checking the existence of receivers.

In order to model broadcasting communication using ACSR, $a!!$ should not be restricted, but $a??$ must be always restricted.

Another extension is to augment ACSR with operations that allow the use of ACSR as a more practical specification language. These operations include: a process name binding operation ($P = Q$), generalized parallel and choice operations, indexed processes ($P[i]$ for i from 1 to n), event ($((p, e[i]))$) and resource ($((p, r[i]))$) specifications. The detailed specification of these additions can be found in [7].

2.2 Modechart

Modechart is a graphical specification language developed to provide a compact and structured way to represent real-time systems. Modechart is a variant of the Statechart language.

The motivation of Modechart is given as follows [15]: (1) Statechart is too liberal in permitting the forms of predicates to appear in the conditions for enabling state transitions. As a result, it is difficult to prevent anomalies in defining a semantics for Statechart. (2) Statechart does not provide an adequate treatment of stringent timing constraints.

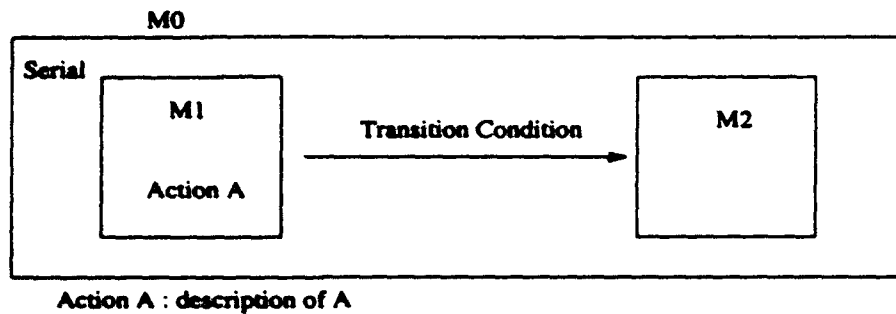


Figure 2: An example of modechart

We briefly review the notions of mode, transition and action in Modechart.

Mode. Modes partition the state space of a system. They can be viewed as the specification of control information that impose structure on the operation of a system.

Modes can be nested within a mode. Nested modes are related either serially or in parallel.

Serial modes—Serial modes specify a sequential relationship between modes. The system operates in at most one serial mode at a time.

Parallel modes—Parallel modes define a parallel relationship between modes. The system operates in all of the parallel modes simultaneously. There are following restrictions on parallel modes.

- A transition between parallel modes is not allowed.
- A transition out of one mode requires exit out of all other modes that are parallel to it.

Well-formed modes—In order to prevent ambiguous modechart specifications, the notion of well-formedness and the UDIM (Unambiguous Designation of Initial Modes) condition are developed. The semantics of modechart is only valid among modecharts of which modes are well-formed and satisfy the UDIM condition. Without the UDIM condition, a well-formed mode can still be ambiguous if initial modes are unspecified.

Transition Transition between two modes represents a change in the control flow of the system. A transition is an instantaneous event which takes zero time unit. Each transition is associated with a condition. The condition for a transition is of the form

$$c_1 \vee c_2 \vee \dots \vee c_n$$

where each c_i is either a triggering condition or a lower and upper time bound restriction on when the transition may be taken.

Triggering condition—Triggering condition c_i is of the form

$$p_1 \wedge \dots \wedge p_m$$

where the p_i 's specify a condition in terms of the occurrence of an event or the truth values of certain predicates. Each p_i is in one of the following three forms:

1. S (or $\bar{S}$) is true at time t if the the state variable S is true (or false) at time t .¹
2. $\{M_1, M_2, \dots, M_m\}$ is true at time t if the system is in at least one of the specified modes at a finite interval up to time t .²
3. E is the occurrence of an event E at the t^3 where E can be an
 - external event, e.g., ΩE ,
 - event denoting start of an action, e.g., $\uparrow A$,
 - event denoting completion of an action, e.g., $\downarrow A$,
 - event setting a state variable to true, e.g., $(S := T)$,
 - event setting a state variable to false, e.g., $(S := F)$,
 - event denoting entry into mode, e.g., $(M1 := T)$, or
 - event denoting exit from mode, e.g., $(M1 := F)$.

Lower/Upper Bound Condition—A condition c_i denoting a lower/upper bound restriction is of the form (r, d) , where r is a non-negative integer denoting a delay and d is a positive integer or ∞ denoting a deadline.

There are three special forms of lower/upper bound condition: *alarm* $r = (r, r)$, *delay* $r = (r, \infty)$, and *deadline* $d = (0, d)$.

¹In RTL, $S(t, t)$ (or $\bar{S}(t, t)$) is true.

²In RTL, for some M_i , $M_i(t, t)$ is true.

³In RTL, $\Theta(E, i) = t$ is true.

Actions. The basic difference between actions and mode transitions is that actions take nonzero time, whereas mode transitions take zero time. In Modechart, each action in a system must be associated with a mode. Furthermore, at most one action may be associated with a mode. Thus, if two or more actions need to be performed when a system is in a certain mode, then the designer should create a child mode for each action.

When a system designer wishes to specify that an action has to be performed when an event occurs, the action has to be specified in the destination mode of the transition. This is justified since a transition is instantaneous in Modechart, the action will be performed upon entering the destination mode.

The value of a state variable is changed explicitly after completing the execution of an action which takes non-zero units of time to perform. Therefore, the value of a state variable is changed at the end of action execution.

2.3 Semantic Difference and Translation

In this section, we identify semantic differences between ACSR and Modechart.

- *Non-delayed event vs. delayed event*—In Modechart, control can stay inside a mode waiting for the triggering condition to be true. That is, suppose we have a serial mode such that $M \xrightarrow{a} N$. The entering time of control into the mode M may differ from that into the mode N . On the other hand, in ACSR, process cannot wait for event, unless waiting possibility is explicitly specified. For instance, suppose Q is an ACSR process such that $Q \stackrel{def}{=} a.P$. Whenever Q is executed at time t , the event a must occur at time t , and the resulting process P is at the same time t . The delayed event in Modechart can be emulated by $Q \stackrel{def}{=} a.P + \emptyset : Q$.
- *Event expression*—In ACSR, events can not be composed using $\vee$ or $\wedge$, that is, events are atomic. Modechart can have an event expression as a triggering condition of a transition.
- *Resource*—ACSR has the notion of resources. The use of resources by a process requires the passage of time. The most natural Modechart notion to ACSR resource is the notion of action which is defined inside a mode. Note that MCTool does not support actions.
- *Priority*—The choice operator in ACSR is based on priority; a higher priority process is chosen. When priority comparison is not possible, the choice is done non-

deterministically. On the other hand, Modechart does not support the notion of priority. When several transitions are possible, one is chosen nondeterministically.

- *Broadcasting vs. synchronization*—The communication mechanism of ACSR is synchronous communication, whereas that of Modechart is broadcasting. The followings are basic concepts of broadcasting communication.

- Receiver process ($a??P$) waits until a sender sends a message.
- Sender process ($a!!Q$) proceeds to the next process immediately after sending a message without checking the existence of receivers. The same message is delivered to all waiting processes.

These communication behaviors can be emulated by ACSR as follows:

- $a??P$ by $recX.(a?.P + 0 : X)$
- $a!!Q$ by $recX.(a!.X + Q)$
- *Mode variable and value test*— Whenever control enters and exits a mode M , Modechart automatically generate events, $\rightarrow M$ and $M \rightarrow$, respectively. Also, there is a condition for each mode M such that $M == true$ or $M == false$ depending on whether control is in the mode M at the current time and is not in M , respectively. A mode variable is encoded using ACSR as follows: Suppose M is a mode.

$$\begin{aligned} M &\stackrel{def}{=} MF \\ MF &\stackrel{def}{=} false_M!.MF + enter_M?.MT + 0 : MF \\ MT &\stackrel{def}{=} true_M!.MT + exit_M?.MF + 0 : MT \end{aligned}$$

M signals $false_M!$ to whichever process needs it until M is entered. After M is entered, it signals $true_M$ until it is exited.

- *State variable*—Similar to mode variables, Modechart also supports state variables. For instance, action ($Set\ Var := F$) and boolean condition ($Var == true$). The behavior of a state variable can be encoded as follows: We assume that the initial value of the variable Var is $false$.

$$\begin{aligned} Var &\stackrel{def}{=} VarF \\ VarF &\stackrel{def}{=} false_Var!.VarF + set_VarT?.VarT + 0 : VarF \\ VarT &\stackrel{def}{=} true_Var!.VarT + set_VarF?.VarF + 0 : VarT \end{aligned}$$

- **Timeout**—ACSR has a powerful timeout which is whenever timeout occurs the timeout process must be executed. A similar notion of timeout in Modechart is *alarm t*, which is $[t, t]$. However there are two differences. 1) A mode can have several *alarm* transitions. For instance, a mode can have two transitions where for each triggering condition is *alarm 5* and *alarm 6*, respectively. 2) A mode may have two transitions: one with *alarm 5* and another with a condition *c*. Here, if the condition *c* is true at time 5, either transition can be executed nondeterministically.

Translation from Modechart to ACSR. It seems that the subset of Modechart which is supported by MCTool can be translated easily into ACSR. Such a subset of MCTool has the semantics of ACSR, hence, no ambiguity is possible. In this case, a real-time system can be specified using MCTool and the correctness of the system can be verified using ACSR based analysis tools.

There is one major problem with this approach because ACSR is based on synchronous communication, whereas Modechart is based on broadcasting. There are two ways to handle broadcasting mechanism: one way is to extend ACSR so that the new ACSR includes broadcasting events, and the other way is to emulate broadcasting mechanism using ACSR synchronous communication as seen in the previous section.

We now describe how a subset of Modechart may be translated into ACSR. We assume that the modecharts used here are well-formed.

1. **Mode names:** we define ACSR variables associated with the names of modes. Let P, P_i, Q and Q_i be such mode variables.
2. **Primitive mode:** A primitive mode can be encoded as $rec\ X.\emptyset : X$ in ACSR.
3. **Parallel mode:** Suppose P is a parallel mode with submodes, $P_1, \dots, P_n$ and no out-going transition. The ACSR translation for P is the following

$$P \stackrel{def}{=} P_1 \parallel \dots \parallel P_n$$

4. **Transition:** Now we translate mode transition such that

$$T : P \xrightarrow{con} Q.$$

First, we translate triggering condition. We have following three cases according to the type of enabling condition.

- Event denoting entry into mode, $-> M$:

$$T \stackrel{def}{=} \text{entry_}M?.\text{rec}X.(\text{exit_}P!.X + Q)$$

- Event denoting exit from mode, $M->$:

$$T \stackrel{def}{=} \text{exit_}M?.\text{rec}X.(\text{exit_}P!.X + Q)$$

- Event denoting status of mode, $\{M == \text{true}\}$ or $\{M == \text{false}\}$:

$$T \stackrel{def}{=} \text{true_}M?.\text{rec}X.(\text{exit_}P!.X + Q)$$

or

$$T \stackrel{def}{=} \text{false_}M?.\text{rec}X.(\text{exit_}P!.X + Q).$$

5. **Serial mode:** Now we describe how to translate a mode M along with translations $T_1, T_2, \dots, T_n, T_{out}$ which come from the mode M .

We assume that the mode M is the ACSR process. Also, we assume that T_i is the ACSR process associated with the transition T_i and T_{out} is the transition with timing constraint $[l, u]$.

$$\begin{array}{ll} M & \stackrel{def}{=} \dots \\ T_1 & \stackrel{def}{=} \dots \\ & \vdots \\ T_{out} & \stackrel{def}{=} \dots \\ MT & \stackrel{def}{=} M \Delta_l (T_{out}, T_1 + T_2 + \dots T_n) \\ & + M \Delta_{l+1} (T_{out}, T_1 + T_2 + \dots T_n) \\ & \vdots \\ & + M \Delta_u (T_{out}, T_1 + T_2 + \dots T_n) \end{array}$$

Where the transition with timing constraint, T_{out} is defined as

$$T_{out} \stackrel{def}{=} \text{rec}X.(\text{exit_}M!.X + Q)$$

A transition with the alarm t is translated into ACSR as follows:

$$MT \stackrel{def}{=} M \Delta_t (T_{out}, T_1 + T_2 + \dots T_n)$$

Note that in this ACSR transition, after passing t time units, timeout transition (a transition with alarm t condition) must be executed, regardless of other transitions.

Translation from ACSR to Modechart. Since a subset of ACSR can be regarded as a restricted form of Modechart in terms of structure, translation from an ACSR term to a Modechart specification seems straightforward as seen in Figure 3 except for synchronous communication, priorities, resources and resource closure. We are identifying on how to extend Modechart to support them. This study is to determine how MCTool can be used as a graphical front-end to ACSR based tools.

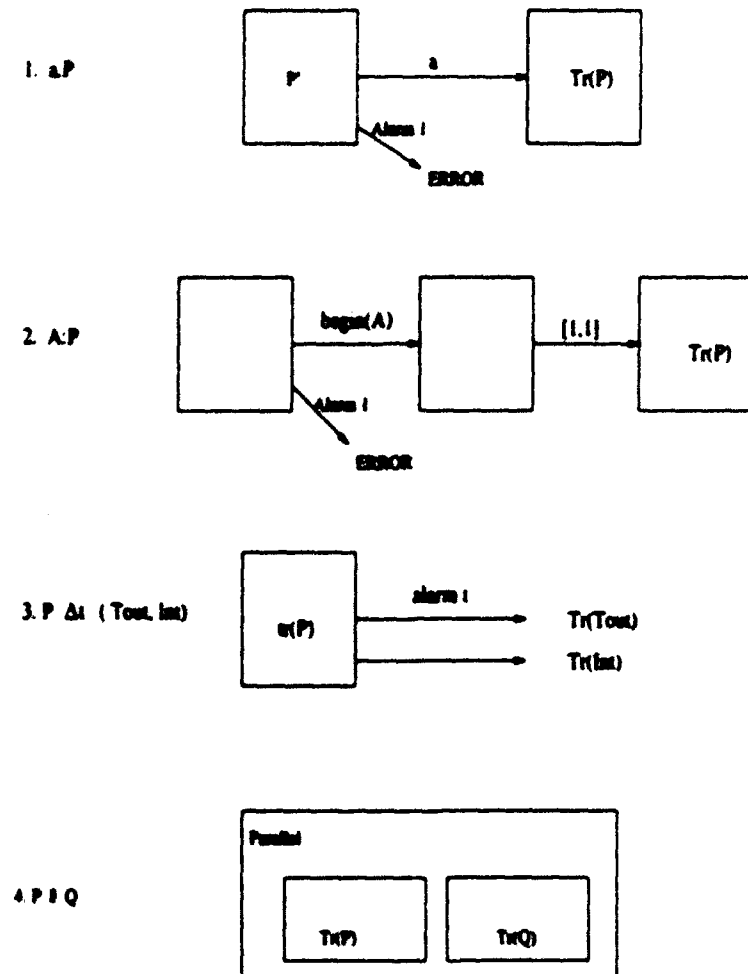


Figure 3: An example of translation of ACSR to Modechart

Figure 3 illustrates how ACSR terms can be translated into Modechart. It is clear that two prefix operators, $\cdot$ and $:$ should be translated into serial modes, since the semantics of prefix operators are identical to serial modes. Since Modechart semantics allows control to stay in a mode until a triggering condition is satisfied, the semantics of modechart corresponding the prefix $a.P$ is slightly different from that of ACSR. This is because the

event a should occur at the same time the mode P' is entered. This problem can be solved by adding an additional time-out transition from P' so that whenever the transition with the event a is not immediately chosen, the next choice is an error state. Similarly, timeout t in ACSR can be translated into serial mode with timing information $[t, t]$.

The above identifies a subset of ACSR which can be translated into the Modechart (defined in MCTool). To translate from the full feature of ACSR to Modechart, the following notions should be supported: prioritized choice, resource closure operator, event restriction operator, scope operator, resources. Section 3 describe a graphical language with these operators.

Modechart Model of the HARM Missile System. Figure 4 is the modechart model of the HARM missile system developed by NRL [8]. The above modechart illustrates HARM missile system control flow but transition conditions are hidden. The following is a transition condition from mode CPU_idle to mode MFSP_awarded_cpu of the missile system[8].

```

CPU1_idle->MFSP_awarded_cpu: (MTS.wait==false & MTS.work==false
& PL.wait==false & PL.work==false & DLP.wait==false & DLP.work==false
& UT.wait==false & UT.work==false & MFSP.wait==true & !MFSP.suspend) |
(MTS.wait==false & MTS.work==false & PL.wait==false & PL.work==false
& DLP.wait==false & DLP.work==false & UT.suspend & MFSP.wait==true
& !MFSP.suspend) |
(MTS.wait==false & MTS.work==false & PL.wait==false & PL.work==false
& DLP.suspend & UT.wait==false & UT.work==false & MFSP.wait==true
& !MFSP.suspend) |
(MTS.wait==false & MTS.work==false & PL.wait==false & PL.work==false
& DLP.suspend & UT.suspend & MFSP.wait==true & !MFSP.suspend) |
(MTS.wait==false & MTS.work==false & PL.suspend & DLP.wait==false
& DLP.work==false & UT.wait==false & UT.work==false & MFSP.wait==true
& !MFSP.suspend) |
(MTS.wait==false & MTS.work==false & PL.suspend & DLP.wait==false
& DLP.work==false & UT.suspend & MFSP.wait==true & !MFSP.suspend) |
(MTS.wait==false & MTS.work==false & PL.suspend & DLP.suspend
& UT.wait==false & UT.work==false & MFSP.wait==true & !MFSP.suspend) |
(MTS.wait==false & MTS.work==false & PL.suspend & DLP.suspend
& UT.suspend & MFSP.wait==true & !MFSP.suspend) |

```

Scheduler (Parallel)

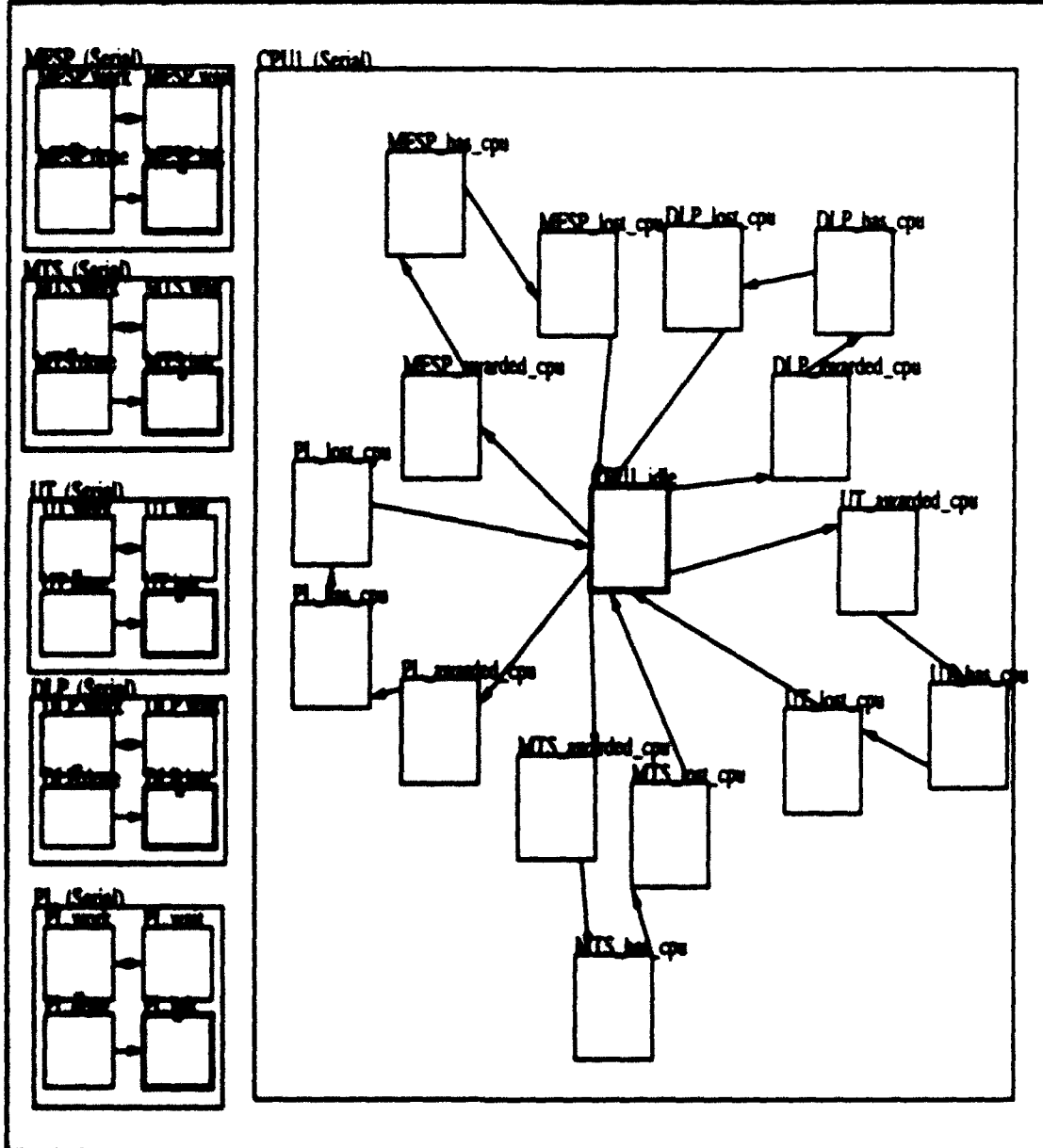


Figure 4: Modechart specification of the HARM Missile example

```

(MTS.suspend & PL.wait==false & PL.work==false & DLP.wait==false
& DLP.work==false & UT.wait==false & UT.work==false & MFSP.wait==true
& !MFSP.suspend) |
(MTS.suspend & PL.wait==false & PL.work==false & DLP.wait==false
& DLP.work==false & UT.suspend & MFSP.wait==true & !MFSP.suspend) |
(MTS.suspend & PL.wait==false & PL.work==false & DLP.suspend
& UT.wait==false & UT.work==false & MFSP.wait==true & !MFSP.suspend) |
(MTS.suspend & PL.wait==false & PL.work==false & DLP.suspend
& UT.suspend & MFSP.wait==true & !MFSP.suspend) |
(MTS.suspend & PL.suspend & DLP.wait==false & DLP.work==false
& UT.wait==false & UT.work==false & MFSP.wait==true & !MFSP.suspend) |
(MTS.suspend & PL.suspend & DLP.wait==false & DLP.work==false
& UT.suspend & MFSP.wait==true & !MFSP.suspend) |
(MTS.suspend & PL.suspend & DLP.suspend & UT.wait==false
& UT.work==false & MFSP.wait==true & !MFSP.suspend) |
(MTS.suspend & PL.suspend & DLP.suspend & UT.suspend & MFSP.wait==true
& !MFSP.suspend) ;

```

This is rather complex condition, which is resulted from trying to represent prioritized scheduling without the notion of priority.

ACSR Model of the Missile System. The following is ACSR specification for the previous modechart example of HARM missile system. Because of the notion of priority, the ACSR specification is quite simpler than that of Modechart.

<i>HARM</i>	$\stackrel{def}{=}$	$[MTS \parallel PL \parallel DPL \parallel UT \parallel MFSP] \{CPU\}$
<i>MTS</i>	$\stackrel{def}{=}$	$IDLE \Delta_{30} (NIL, MTS_Work, NIL)$
<i>MTS_Work</i>	$\stackrel{def}{=}$	$\{CPU, 5\}^\infty \Delta_8 (NIL, MTS, \{ \} : MTS_Work)$
<i>PL</i>	$\stackrel{def}{=}$	$IDLE \Delta_{20} (NIL, PL_Work, NIL)$
<i>PL_Work</i>	$\stackrel{def}{=}$	$\{CPU, 4\}^\infty \Delta_8 (NIL, PL, \{ \} : PL_Work)$
<i>DLP</i>	$\stackrel{def}{=}$	$IDLE \Delta_{25} (NIL, DLP_Work, NIL)$
<i>DLP_Work</i>	$\stackrel{def}{=}$	$\{CPU, 3\}^\infty \Delta_8 (NIL, DLP, \{ \} : DLP_Work)$
<i>UT</i>	$\stackrel{def}{=}$	$IDLE \Delta_{15} (NIL, UT_Work, NIL)$
<i>UT_Work</i>	$\stackrel{def}{=}$	$\{CPU, 2\}^\infty \Delta_8 (NIL, UT, \{ \} : UT_Work)$
<i>MFSP</i>	$\stackrel{def}{=}$	$IDLE \Delta_8 (NIL, MFSP_Work, NIL)$
<i>MFSP_Work</i>	$\stackrel{def}{=}$	$\{CPU, 1\}^\infty \Delta_8 (NIL, MFSP, \{ \} : MFSP_Work)$

3 GCSR: Graphical CSR

In this section, we present a graphical specification language (GCSR) that has a formal semantics based on the Algebra of Communicating Shared Resources (ACSR). GCSR is action based, and has constructs to build a modular (through limiting visibility of events) and hierarchical (through node nesting) specification of a real-time system. Its semantics, ACSR, is a formalism that supports the integrated specification of functional aspects of a real-time system and its run-time resource requirements. The outline of this section is as follows. Section 3.1 introduces the syntax of GCSR and validity criteria of GCSR specifications. Section 3.3 describes informally the semantics of GCSR, then gives the translations between ACSR and GCSR. Section 4 presents several examples.

3.1 GCSR Syntax

We now describe the building blocks of a GCSR graph, and how they are used to construct a GCSR specification.

Building blocks. The basic building blocks in GCSR are *nodes* that can be connected with directed *edges*. GCSR has three types of nodes (*NIL*, *Dot*, and *Box*) and two types of directed edges (*external exit*, and *internal exit*.) Figure 5 shows the graphical symbols of these building blocks.

Nodes. The *NIL* node is a special node that marks the end of a sequential execution. The *Dot* node is either a specification point (place in Petri net terminology) where an event must be instantaneously produced, or a specification point where the selection among multiple computations (work to do) must be instantaneously made. The necessity of the special *NIL* and *Dot* nodes will become clearer in Section 3.3 where we present the semantics of GCSR. A *Box* node represents a computation component that may consume resources and time. A *Box* can either be primitive, i.e. its internal structure is hidden, or complex, i.e. its internal structure is visible. A complex *Box* can contain one component or two (or more) components that are executed concurrently. A *reference Box* refers to another GCSR specification; such a node allows stepwise and compact graphical specifications.

As the extended BNF⁴ for the GCSR structures in Figures 6 and 7 shows, a node has a *name* that uniquely identifies it: a primitive box has a possibly empty set of resources

⁴We followed the notation of programming languages as used in [30, 19]

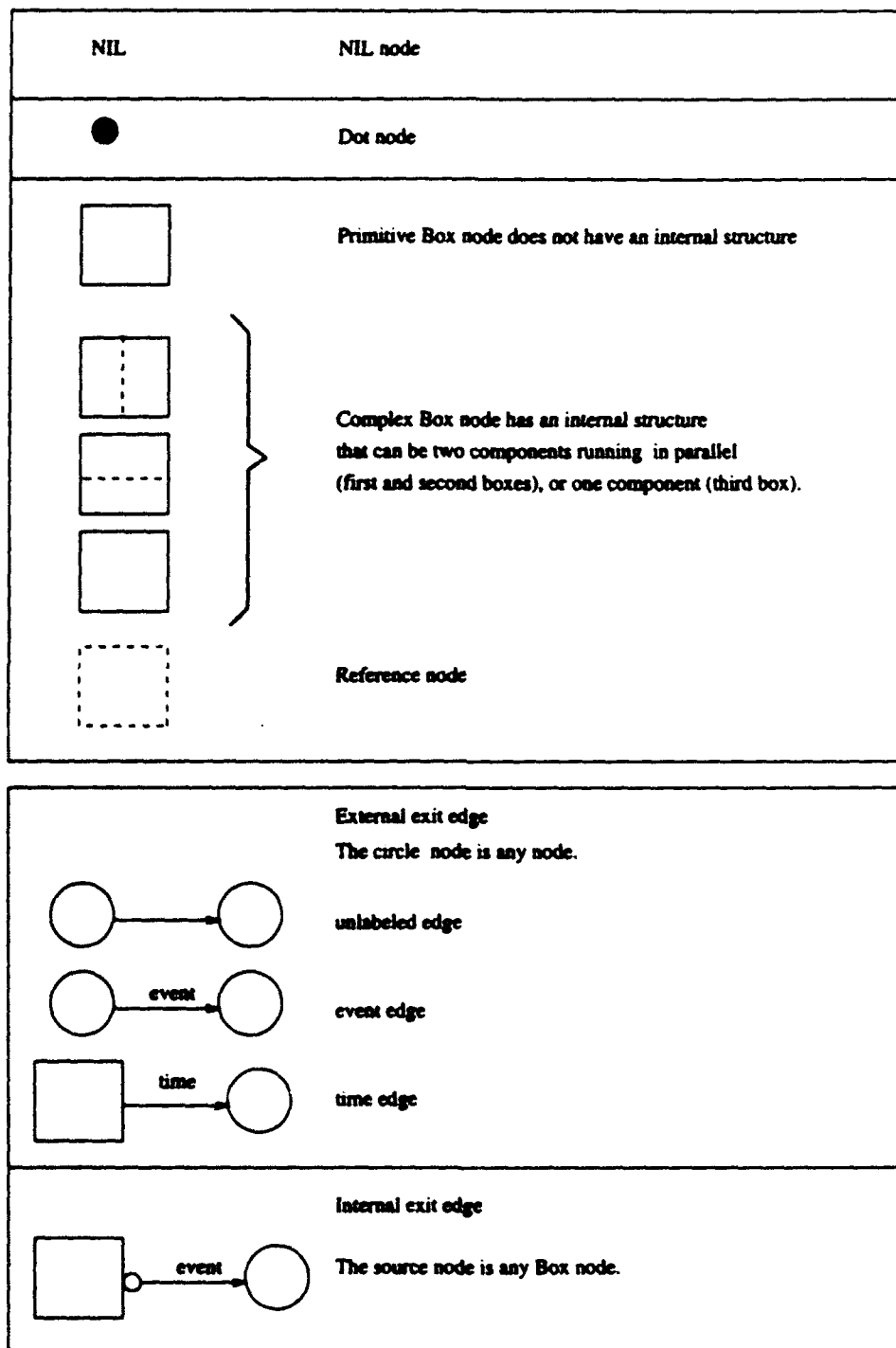


Figure 5: GCSR types of nodes (top) and edges (bottom).

(*Resource*); a complex box has in addition a set of dedicated resources (*Close*), a set of observable events (*Observe*), a set of local synchronization events (*Restrict*), and a list of GCSR components representing the internal structure it contains. In the case of a reference box, these attributes are deduced from the GCSR graph that defines it.

Edges. Nodes can be linked together with edges that do not cross node boundaries to form a GCSR Specification. There are two types of edges: *internal-exit* edges to reflect the end of execution from inside a Box node and to transfer control to another node at a higher level, and *external-exit* edges to reflect sequential flow of control among nodes at the same level. The two types of edges simplify a hierarchical graphical specification and make the transition semantics unambiguous. While the activation of an internal-exit edge depends on the source node, that of an external-exit edge depends on the environment. This semantic distinction between the two types of edges will be addressed in Section 3.3.

An internal-exit edge always has a Box source node, and is labeled with an event, "exit event" (which corresponds to "exception" in ACSR). An external-exit edge, on the other hand, can be either unlabeled or labeled with an event or time. An unlabeled edge and an event labeled edge can have either a Dot or a Box source node, while an edge labeled with time always has a Box source node. Although not shown, edge label $[t_1, t_2]$ is a short hand notation for multiple edges $t_1, \dots, t_2$.

GCSR Specification. Figures 6 and 7 show the extended BNF for GCSR. We use the following convention for extended BNF: A production (e.g. $A == B$) has a name (A) and a right hand side describing its components (B). The labels N, L, ... etc. are used as name tags to identify the component of the production. The asterisk is used to denote a list/sequence (e.g. $A == B^*$ means A is a list of B elements). Double asterisks are used to denote a set (e.g. $A == B^{**}$ means A is a set of B elements). The semi-colon (e.g. $A == B ; C$) is used to aggregate components in a production (A has two components B and C). The vertical bar (e.g. $A = B | C$) is used to denote choice in a production (A has either component B or C).

A GCSR specification is a *component*, which is a set of connected nodes, one of which is a designated *initial* node. A GCSR component also has attributes consisting of: an optional name, a set of resources, and a set of observable events.

Since a complex Box node has other nodes nested inside, we distinguish between levels at which a node resides and at which its components reside. A complex Box "encapsulates" the nodes it contains inside, and makes them inaccessible from outside its boundaries. That

GCSRComponent	==	N	: GCSRNode**;
		I	: GCSRNode;
		L	: Link**;
		A	: Attributes
Attributes	==	Name	: Identifier;
		Resource	: Identifier**;
		Observ	: Identifier
GCSRNode	==	Name	: Identifier;
		Nil	: NilNode
		Dot	: DotNode
		Box	: BoxNode
BoxNode	==	Prim	: PrimitiveNode
		Cmplx	: ComplexNode
		Ref	: ReferenceNode
NilNode	==		NULL
DotNode	==		NULL
ReferenceNode	==		NULL
PrimNode	==	Resource	: GeneralIdSet
ComplexNode	==	Resource	: GeneralIdSet;
		Close	: GeneralIdSet;
		Observ	: GeneralIdSet;
		Restrict	: GeneralIdSet;
		Inside	: GCSRComponent*
Link	==	S	: GCSRNode;
		T	: GCSRNode;
		Label	: InternalExit ExternalExit
InternalExit	==		Event
ExternalExit	==		Time Event NULL

Figure 6: Extended BNF for GCSR

Identifier	==	STRING IndexedVar;
IndexedVar	==	STRING; Index
GeneralIdSet	==	STRING** IndexedVar
Index	==	Min; Max; Step; Mask
Min	==	integer expression
Max	==	integer expression
Step	==	integer expression
Mask	==	boolean expression
Event	==	STRING
Time	==	INTEGER ∞

Figure 7: Extended BNF for GCSR (continued)

is, the nodes inside a complex Box node in a component belong to a different component from the one to which the complex Box belongs, one that is inside the complex Box node. Thus, the edges listed in the *Link* set (Figure 6) of a GCSR component connect only nodes that are at the same level as the initial node of the component.

Note that it is possible to transfer out of a nested node to a node adjacent to its ancestor node using an internal-exit edge. That is, our level restriction is “syntactic”, not “semantic”, restriction to prevent the crowding of edges in GCSR specifications.

3.2 Valid GCSR Specification

Obviously, not all possible GCSR specifications represent valid ACSR specifications. We now define a set of criteria for a GCSR specification to be valid.

Nil, Dot, primitive Box, and reference Box nodes are the basic nodes that do not contain an internal structure inside. A complex Box node contains internal components. A GCSR specification can be built from these nodes according to the following definition.

Definition 3.1 A GCSR valid component, $\langle N, I, L, A \rangle$, is a connected graph that consists of a set of nodes, N , one of which is a designated initial node, $I \in N$, and the connection edges, L , and attributes, A , satisfying the following rules:

1. Any Nil node does not have an outgoing edge; that is, a Nil node is always a sink node:
 $\forall l \in L : l.S.type \neq Nil$
2. Any Dot node has at least one outgoing event or unlabeled edge that connects it to

another node at the same level, i.e. without crossing any node's boundaries:

$\forall n \in N : \text{if } n.type = \text{Dot} \text{ then } \exists l \in L : l.S = n$

3. Any Box node can have at most one outgoing edge labeled with time, and possibly several outgoing event-labeled or unlabeled edges and external-exit edges that connect it to other nodes at the same level:

$\forall n \in N : \text{if } n.type = \text{Box} \text{ then } \forall l \in L : ((l.S = n \wedge l.type = \text{Time}) \rightarrow (\forall l' \in L - \{l\} : l.S \neq n \vee l.type \neq \text{Time}))$

4. For each complex Box node, the GCSR components inside must be valid and do not share nodes:

$\forall n \in N : \text{if } (n.type = \text{Cmplx} \wedge n.Inside = \{G_1, \dots, G_k\}) \text{ then } (\forall 1 \leq i \leq k : G_i \text{ is valid}) \wedge (\forall 1 \leq i \leq k : \forall 1 \leq j \leq k : (i \neq j \rightarrow N_i \cap N_j = \emptyset))$

where $\forall 1 \leq i \leq k : G_i = (N_i, I_i, L_i, A_i)$

5. Attributes are valid as follows:

Name: All component's names are unique.

Resource: The Resource set of a complex Box node is the union of the Resource sets of its subcomponents:

$\forall n \in N : \text{if } (n.type = \text{Cmplx} \wedge n.Inside = \{G_1, \dots, G_k\}) \text{ then } n.Resource = A_1.Resource \cup \dots \cup A_k.Resource$

where $\forall 1 \leq i \leq k : G_i = (N_i, I_i, L_i, A_i)$

The Resource set of a component is the union of all the Resource sets of its Box nodes.

$$A.Resource = \bigcup_{n \in N \wedge n.type = \text{Box}} n.Resource$$

Observ: The observable event set (Observ) of a complex Box node is a subset of the Observ sets of its subcomponents. (The additional observable events can be used to synchronize among the subcomponents inside the Box.)

$\forall n \in N : \text{if } (n.type = \text{Cmplx} \wedge n.Inside = \{G_1, \dots, G_k\}) \text{ then } n.Observ \subseteq (A_1.Observ \cup \dots \cup A_k.Observ)$

where $\forall 1 \leq i \leq k : G_i = (N_i, I_i, L_i, A_i)$

The *Observ* set of a component is the set of events labeling its edges (in L) union the *Observ* sets of its complex and reference Box nodes:

$$A.Observ = \bigcup_{n \in N \wedge n.type = Cmplx} n.Observ \cup \bigcup_{l \in L \wedge l.type = Event} l.Label$$

Restrict: The restrict event set (*Restrict*) of a complex Box node is a subset of the *Observ* sets of its subcomponents:

$\forall n \in N : \text{if } (n.type = Cmplx \wedge n.Inside = \{G_1, \dots, G_k\}) \text{ then } n.Restrict \subseteq (A_1.Observ \cup \dots \cup A_k.Observ)$

where $\forall 1 \leq i \leq k : G_i = \langle N_i, I_i, L_i, A_i \rangle$

Observ $\cup$ Restrict: All events inside a complex Box are accounted for:

$\forall n \in N : \text{if } (n.type = Cmplx \wedge n.Inside = \{G_1, \dots, G_k\}) \text{ then } (n.Observ \cup n.Restrict) = (A_1.Observ \cup \dots \cup A_k.Observ)$

where $\forall 1 \leq i \leq k : G_i = \langle N_i, I_i, L_i, A_i \rangle$

Reference node: The attributes of a reference Box are those of the GCSR component that defines it.

These above validity criteria are needed to define the semantics of a GCSR component using ACSR.

3.3 Informal GCSR Semantics

In this section we first describe informally the semantics of GCSR, then the translations between ACSR and GCSR. These translations assume that ACSR is augmented with a binding operator that binds a process term to a process name, generalized parallel and choice operators, as well as indexed variable names.

A GCSR (valid) component specifies the sequential flow of control among nodes representing units of work (Box nodes) and points of undelayed communication or control switch (Dot nodes). In our formalism, work is represented in terms of resource and time usage. Control flow in a GCSR component is indicated through directed edges connecting the nodes. Complex Box nodes allow compact specification of concurrent components as well as execution of a component under the control of the environment which can interrupt it and time it. Finally, nondeterministic execution is represented by multiple edges out of a (non Nil) node.

Nodes. A unit of work is represented by a Box node. It can represent either simple resource and time consumption (primitive Box), or multiple (or one) units of work that are executed in parallel (complex Box) and possibly under the (external) control of the environment which can time their activation and interrupt it. A complex unit of work can also relinquish control any time during its activation and indicates the direction of the control flow; we call this type of control flow internal-exit, since the complex unit of work internally ends its activation, as opposed to being interrupted by its environment. This explains the two types of edges in GCSR. In ACSR terminology, a simple unit of work is an action, while a complex unit of work is a process.

As an example, consider the railroad crossing example of Figure 20. The Box nodes with the empty set inside represent simple units of work, idling, where no resources are used. In the specification of the *Gate*, the complex work representing opening the gate can be interrupted by the event signaling to start closing the gate (*sdn*), or can finish its activation and proceeds to the initial state of *Gate* through the internal exit signal *open*.

The second type of nodes, Dot nodes, represent two specification scenarios. The first is a specification point where an event must be produced with no delay; this is represented by an event labeled outgoing edge. The second is a specification point where a unit of work must be started with no delay; this is represented by an unlabeled outgoing edge whose target node is a Box or Nil node. For example, consider again the *Gate* GCSR component in the railroad crossing example of Figure 20; the initial node is a Dot node with two outgoing edges. This represents a point where either the idling unit is started or the event *sdn* is produced; the choice among the two possibilities is made at the instant when the initial Dot node is activated, and it depends on the environment.

Node activation. When a GCSR component is activated, control enters its initial node. The flow of control stops when it reaches a Nil node, at which time the whole GCSR component is deactivated. The activation of a primitive Box node is subject to the availability of its resources. Finally, when a complex Box is activated all its subcomponents are simultaneously activated and remain active for the same period of time. In the railroad crossing example of Figure 20, the *Train*, *Control*, and *Gate* are all activated the instant the *Railroad* GCSR component is activated. Since there are no edges out of the initial node of *Railroad*, this latter remains active as long as its subcomponents remain so.

Unlabeled edge. An unlabeled edge is taken instantaneously if it leads to a node that can be activated. Such an edge allows sequential composition of different types of nodes; that is, it is used to describe a sequential execution where the next component is either a complex (complex Box node) or a simple resource and time consuming component (primitive Box node.) Unlabeled edges allow compact construction of specifications.

Event labeled edge. An event labeled edge that links a Dot node to another node is taken the instant the source Dot node is activated and if the environment allows the production of the labeling event. An event edge out of a Box node is taken any time the environment allows the production of the event and while the source Box node is active; this represents an interrupt and has a higher priority than edges inside the source Box node. The transition results in producing the event labeling the edge.

Time labeled edge. An edge labeled with time (t) represents the time-guarded execution of work represented by the source Box node. Once the source Box node is activated, control remains there exactly t time units after which it is instantaneously transferred to the target node of the time edge. It is important to note that the source Box node must remain constantly active for the t units. If it requires more than t time units, it is aborted; on the other hand, if it executes for less than t time units, the timed edge will not be taken. A timed edge is also a type of interrupt and hence has a higher priority than edges inside the source Box node.

Internal-exit edge. Any time during its activation, the source Box node can produce the exit event to signal internal end of execution. Control is then instantaneously transferred to the target node of the internal-exit edge. The internal exit event is a type of local synchronization between the internal structure of the Box source node and its interface; that is, this event is not visible anywhere else in the system.

It is important to note that control remains a non zero amount of time only inside Box nodes, and that all transitions are instantaneous. The activation of the target node of a taken transition might be subject to the environment. For example, if the target node is a primitive Box node whose resources are not available the instant the transition is taken, the system enters a deadlocked state after the transition is taken.

Modularity. Modularity is supported in GCSR through the visibility scope of the communication events. The set of observable events of a GCSR component, are visible everywhere.

On the other hand, the set of restricted events in a complex Box node are used for local synchronization among components inside the Box, and hence are not visible elsewhere outside the Box's boundaries. Such a scoping rule has the advantage of limiting dependencies among nodes in a GCSR specification. This advantage is important in a hierarchical design of a real-time system.

3.4 Formal GCSR Semantics

Figure 8 shows the translation of ACSR terms to a GCSR component based on structure of the ACSR terms. Figure 9 shows the translation of the binding operator, indexed variables as well as generalized parallel and choice operators. The function $g(\cdot)$ translates an ACSR term to a GCSR component.

Note that the direct translation of ACSR terms does not create a GCSR graph where a Box node has an edge labeled with an event or a Box node with multiple event edges. Such a graph can however be obtained through the simplifications described in Figure 10. Also, note that the translation of an ACSR term to a GCSR component does not introduce cycles. Graphical reductions as described in Figure 12 allow the simplification of the graph and creation of cycles.

Graphical reductions. We divide graphical reductions into two classes. Figures 10 and 11 show simple graphical reductions that:

1. merge "identical" portions of the graph;
2. remove unnecessary unlabeled edges that can be due to unnecessary parenthesis in the ACSR process;
3. remove edge labeled with infinity and its NIL target node;
4. merge consecutive "identical" activities; this applies the time additivity property of ACSR;
5. remove extra structure inside a Box, that denote recursive usage of resources; this is due to the fact that time is discrete in ACSR;
- 6-7. remove extra Box node nesting that can be due to resolving Reference Box nodes or unnecessary parenthesis in the ACSR process; (This simplification rule also makes use of the ACSR Close(5) and Rest(6) laws [6].)

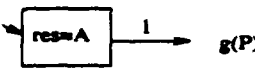
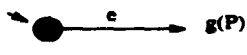
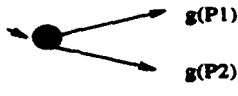
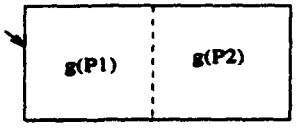
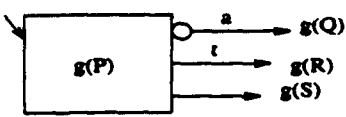
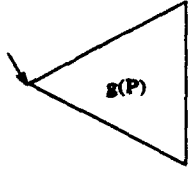
NIL		
A:P		
e.P		
P1 + P2		
P1 P2		Attributes: Restrict Observ Resource Close
P \ F		Attributes: Restrict: F
[P]I		Attributes: Close: I
P scope(a, t) (Q, R, S)		
X a variable node		
rec X. P Step1: create P Step2: initial node of g(P) is called X The triangle is a representation of P's GCSR component.		

Figure 8: ACSR to GCSR translation.

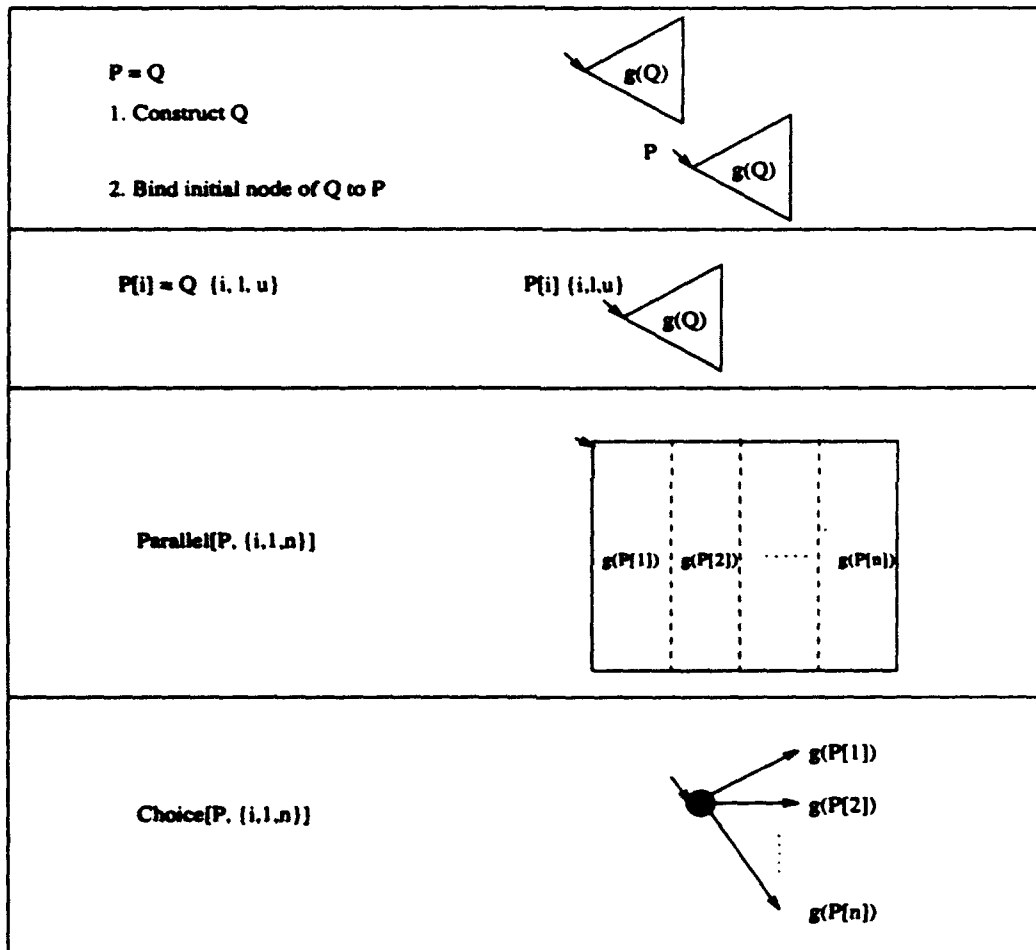


Figure 9: ACSR to GCSR translation

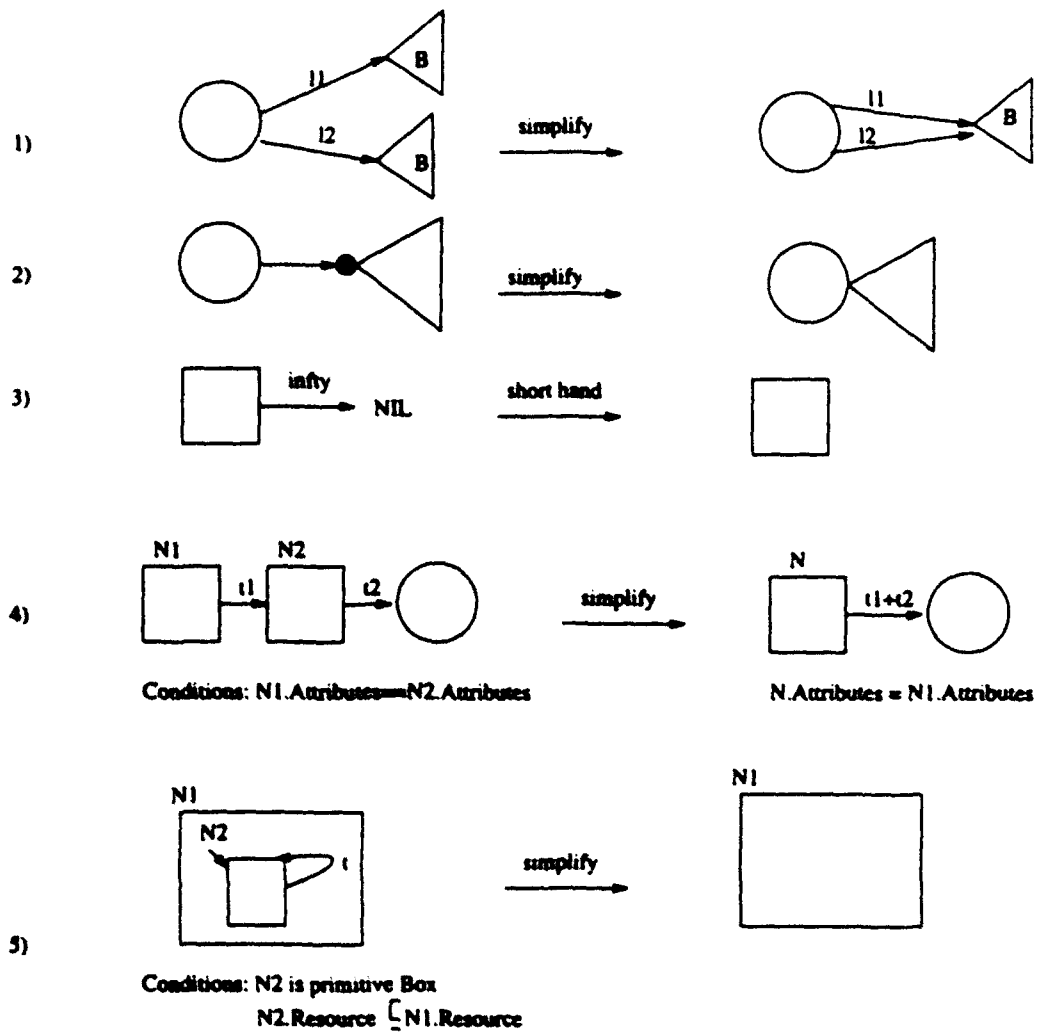


Figure 10: GCSR simple graphical reductions and short hand notations.

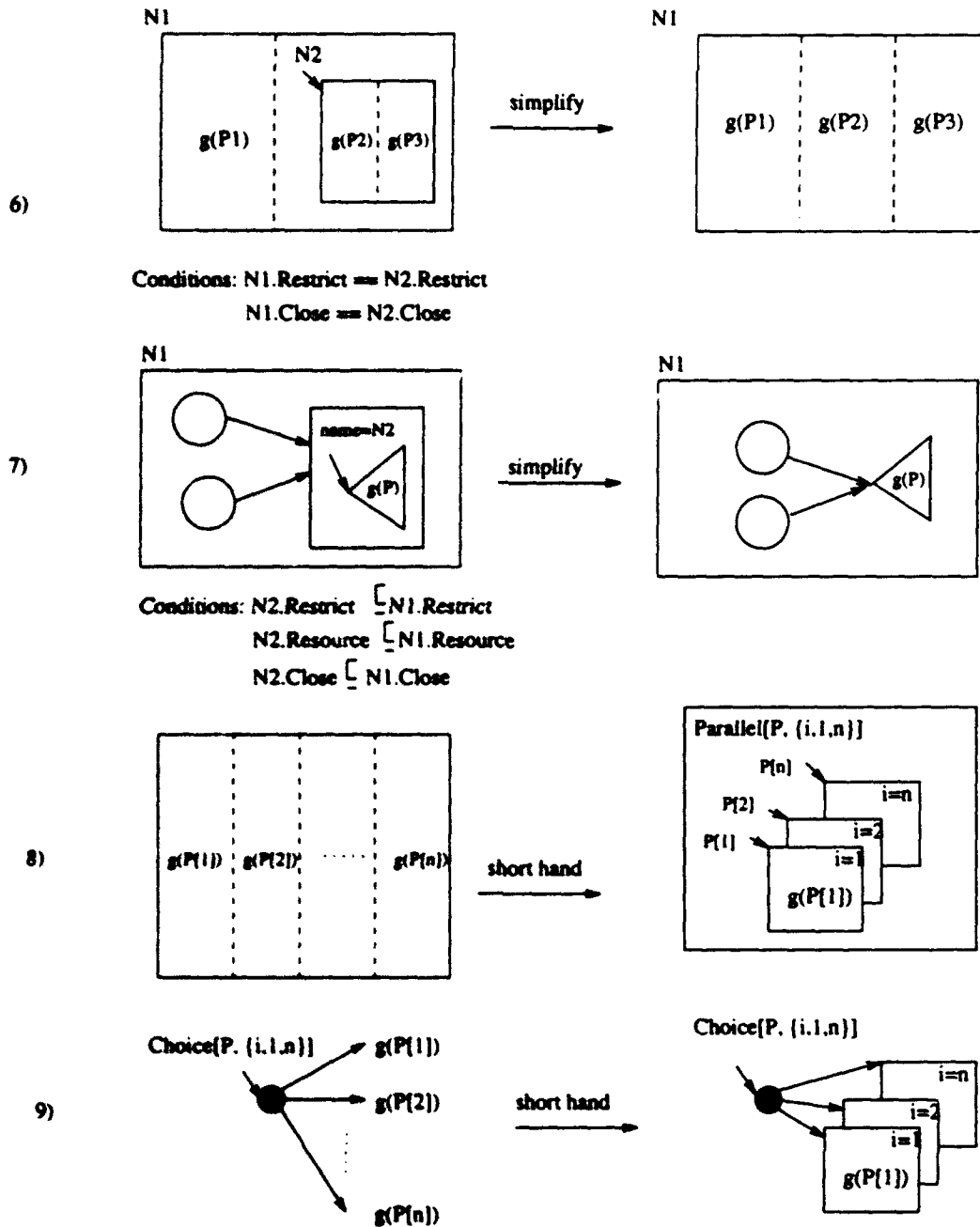


Figure 11: GCSR simple graphical reductions and short hand notations (continued).

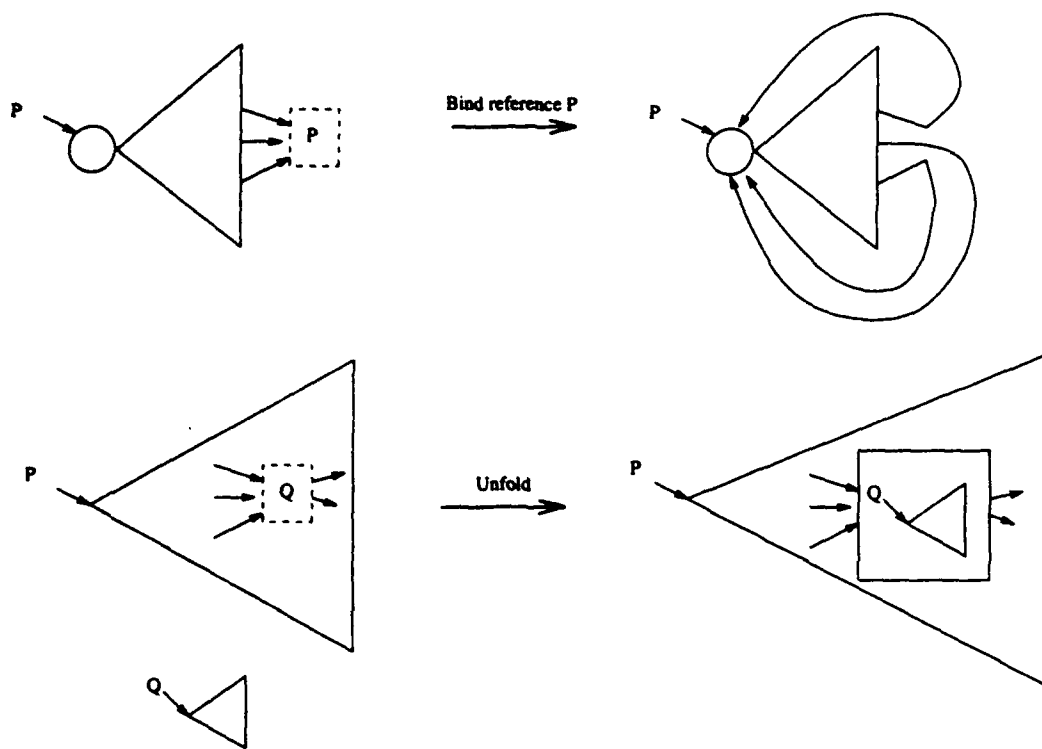


Figure 12: GCSR complex graphical reduction

- 8-9. give a template representation as a short hand notation for generalized ACSR parallel and choice operators.

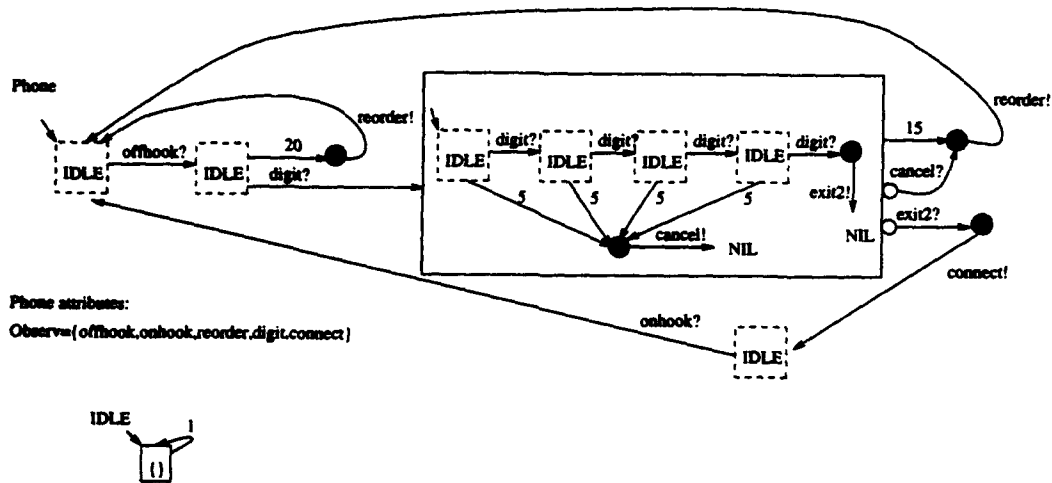
Figure 12 shows more complex graphical reductions to resolve Reference Box nodes by 1) binding them to a node, or 2) replacing them with the process definition (unfolding). Top, sink reference nodes in Figure 12 are bound to the initial node of the corresponding process. This reduction is allowed only if the reference node and the initial node of the corresponding process are at the same level. Bottom, a reference Box is replaced by the GCSR component defining it. Further reductions would be based on the ACSR laws.

To define the semantics of GCSR, we have developed an algorithm that translates a GCSR component to an ACSR term. This translation assumes some validity criteria of GCSR components. These criteria are described in Section 3.2.

4 Examples

In this section we present several examples of GCSR specifications and their corresponding ACSR processes. We use the railroad crossing example to illustrate how some of the graphical reductions described in the previous sections can be applied. We do not show all the attributes to make the figures more readable, and we list the resource set inside primitive boxes.

Telephone. Figure 13 shows the GCSR specification of a telephone system and its corresponding ACSR specification. The phone is initially idle until it receives the signal that the receiver is *offhook*. At this time, the user has 20 time units to dial the first digit (signaled by the reception of *digit*.) If the user fails to dial the first digit within the 20 time unit deadline, a the phone produces a *reorder* signal and returns to its initial state. If the user dials the first digit within the deadline, he/she must dial the remaining four digits within 15 time units relative to dialing the first one. An additional timing constraint on the last four digits is that any two consecutive digits must be dialed in less than five time units apart. Any time this timing constraint is violated, the phone issues a *reorder* signal and returns to its initial state. On the other hand, if the last four digits are dialed according to this deadline and within the 15 time units from the first digit, the phone issues a *connect* signal and allows the user to talk. When the user puts the receiver on the hook again (signaled by the reception of *onhook*) the phone returns to its initial state.



$$Phone = IDLE \Delta_{\infty} (NIL, NIL, (offhook, 1).OH)$$

$$OH = IDLE \Delta_{20} (NIL, (\overline{reorder}, 1).Phone, (digit, 1)Digits)$$

$$Digits = D[1] \Delta_{15}^{cancel, exit2} ((\overline{reorder}, 1).Phone + Talk, (\overline{reorder}, 1).Phone, NIL)$$

$$D[i] = IDLE \Delta_5 (NIL, (\overline{reorder}, 1).(\overline{cancel}, 1).NIL, (digit, 1).D[i+1]) \quad 1 \leq i \leq 4$$

$$D[5] = (\overline{exit2}, 1).NIL$$

$$Talk = (\overline{connect}, 1).(IDLE \Delta_{\infty} (NIL, NIL, (onhook, 1).Phone))$$

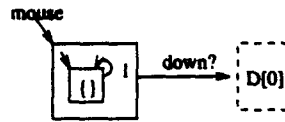
Figure 13: Telephone Example.

Mouse. Figure 14 shows the GCSR and corresponding ACSR specification of a mouse example. The mouse is initially idle until its button is pressed down (*down* event is received.) This is the beginning of a click that might be followed by another click to signal a double click. A single click consists of pressing the mouse button *down* then *up*. A double click consists of two consecutive single clicks that happen within 500 time units and such that the *down* and *up* of each single click are less than 200 time units apart. Each time the mouse button is pressed down for 200 time units or more, the mouse signals *hold*, then when the button is pressed up the mouse signals *release* and returns to its initial state. The mouse also signals either *click* or *doubleclick* according to whether a signal click or double clicks happened.

Railroad Crossing. Figure 15 shows the ACSR specification of the standard Railroad crossing system with fixed parameters. The system consists of a train (*Train*), a controller (*Control*) and a gate (*Gate*), that coordinate their activities through the synchronization events *srn*, *sdn*, *sup*, and *srp*. The detailed description of this example can be found in [7]. Figure 16 shows the direct translation of ACSR specification to GCSR, and Figures 17-19 show how the GCSR specifications can be simplified using graph reductions. The steps are: Figure 17 shows 1) Unfolding applied to *Train*, *Train'*, *Gate* and *GD'*; the reference nodes were replaced by their corresponding process graph. 2) Unfolding applied to all references to *IDLE*, then the simple reduction where the recursive consumption of no resources is replaced by a primitive *Box* with no resources (reduction number 4). 3) A simple reduction in *GU* where the infinity time edge with target *NIL* was removed (reduction number 2). In Figure 18, further unfolding step is applied on *Train* and *Gate* of the railroad crossing example of Figure 17, Figure 19 demonstrates further unfolding applied on *Train*, *Gate* and *Control* of the railroad crossing example of Figure 18. The reference nodes have names of nodes that are part of the graph and at the same level. They are removed and their incoming edges are bound to the actual node. Finally, all reference nodes are removed in Figure 20.

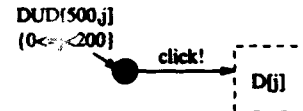
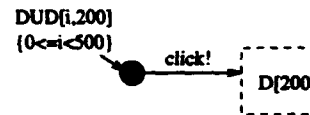
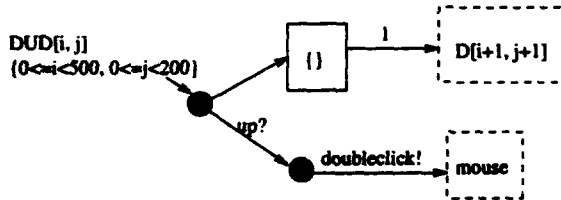
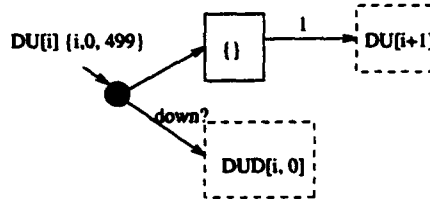
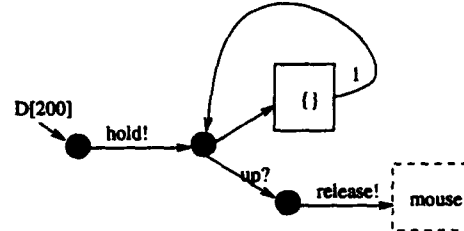
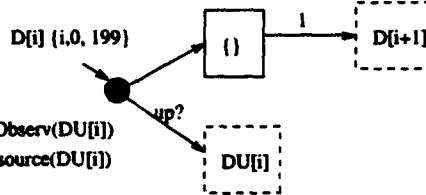
Sensor. Figure 21 shows the GCSR specification of a *Senor* system. It consists of a data reading unit (*ReadSensor*) and a data processing unit (*Server*), that communicate through a shared buffer (*B*). Mutual exclusive access to the buffer is enforced by a binary semaphore (*Semaphore*.) *ReadSensor* accumulates data for three time units, then averages it in one time unit, and stores it into the buffer in two time units. *Server* is initially for six time units, to avoid reading out of an empty buffer. Once active, *Server* waits for a *read* signal

mouse's attributes:
 Observ = {down} U Observ* $D[0]$
 Resource = Resource($D[0]$)
 Close = {}



$D[i]$'s attributes:

Observ = {up} U Observ($D[i+1]$) U Observ($DU[i]$)
 Resource = Resource($D[i+1]$) U Resource($DU[i]$)
 Close = {}



$mouse = IDLE \Delta_{\infty} (NIL, NIL, (down, 1).D[0])$
 $D[i] = \emptyset : D[i+1] + (up, 1).DU[i] \quad 0 \leq i < 200$
 $D[200] = (\overline{hold}, 1).rec X.(\emptyset : X + (up, 1).(\overline{release}, 1).mouse)$
 $DU[i] = \emptyset : DU[i+1] + (down, 1).DUD[i] \quad 0 \leq i < 500$
 $DU[500] = (\overline{click}, 1).mouse$
 $DUD[i, j] = \emptyset : DUD[i+1, j+1] + (up, 1).(\overline{doubleclick}, 1).mouse \quad 0 \leq i < 500, 0 \leq j < 200$
 $DUD[i, 200] = (\overline{click}, 1).D[200] \quad 0 \leq i < 500$
 $DUD[500, j] = (\overline{click}, 1).D[j] \quad 0 \leq j < 200$

Figure 14: Mouse Example.

```

// ACSR program and specification for standard railroad
// crossing problem subject to the following assumptions:
// - Train may return before gate is fully up
// - All parameters are fixed
// Implementation

Train      = (Nc,1).( $\overline{\text{srn}}$ ,1).(IDLE  $\Delta_{60}$ (NIL,Train',NIL));
Train'     = (Ic,1).(IDLE  $\Delta_{20}$ (NIL,Train'',NIL));
Train''    = (Pc,1).( $\overline{\text{srp}}$ ,1).(IDLE  $\Delta_{30}$ (NIL,Train''',NIL));
Train'''   = IDLE  $\Delta_{\infty}$ (NIL,NIL,Train);

Control    = {}:Control + (srn,1).( $\overline{\text{sdn}}$ ,1).Control + (srp,1).( $\overline{\text{sup}}$ ,1).Control;

Gate       = {}:Gate + (sdn,1).GD;
GD         = (Bdn,1).(IDLE  $\Delta_{40}$ (NIL,(Dn,1).GD',NIL));
GD'        = {}:GD' + (sup,1).(Bup,1).GU;
GU         = (IDLE  $\Delta_{40}$ (NIL,(Up,1).( $\overline{\text{open}}$ ,1).NIL,NIL))  $\Delta_{\infty}^{\text{open}}$ (Gate, NIL, (sdn,1).GD);
Railroad   = (Train || Control || Gate) \ {srn,srp,sdn,sup};

```

Figure 15: Rail Road Crossing Example ACSR Specification

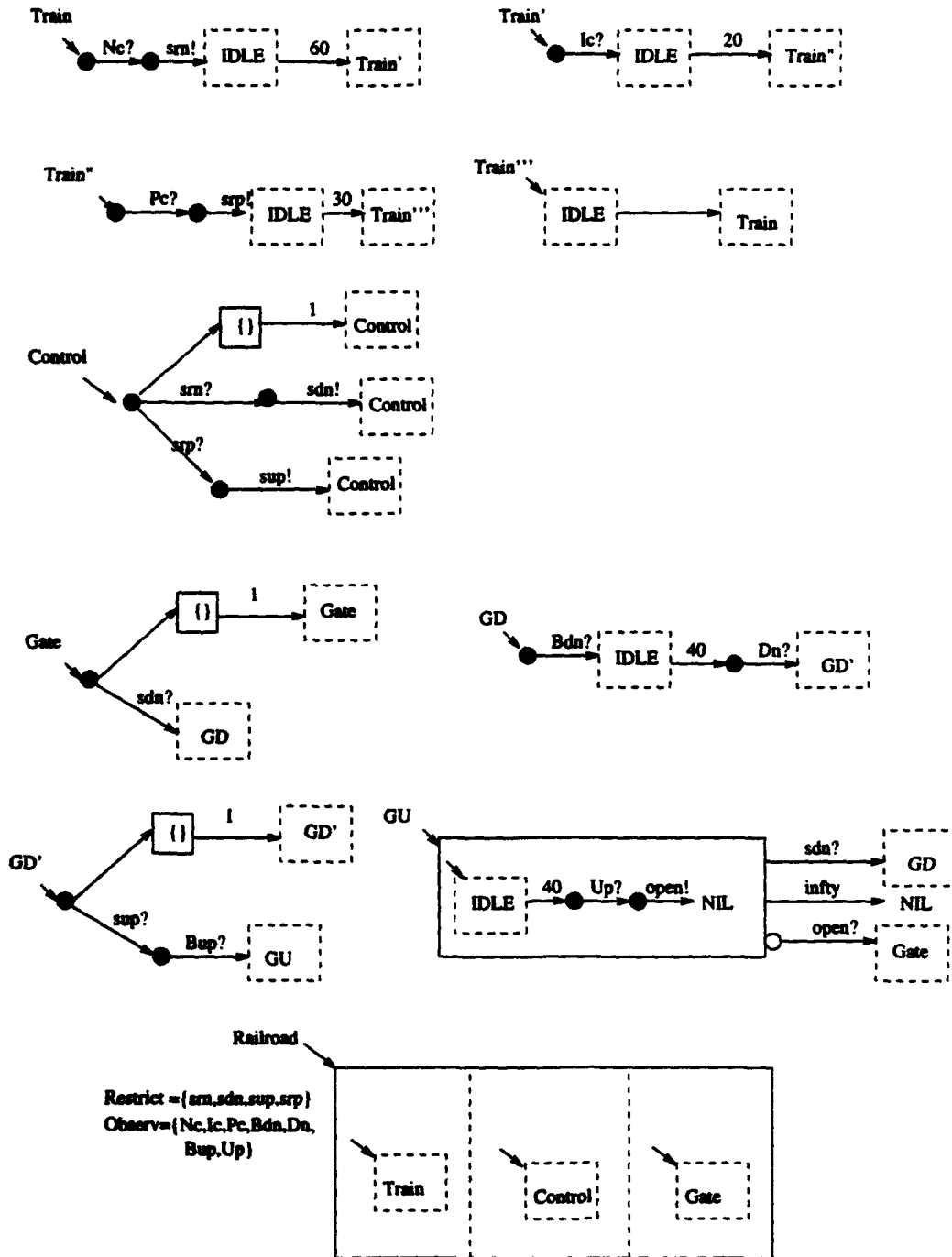


Figure 16: Railroad Crossing Example: GCSR Specification for ACSR terms in Figure 15

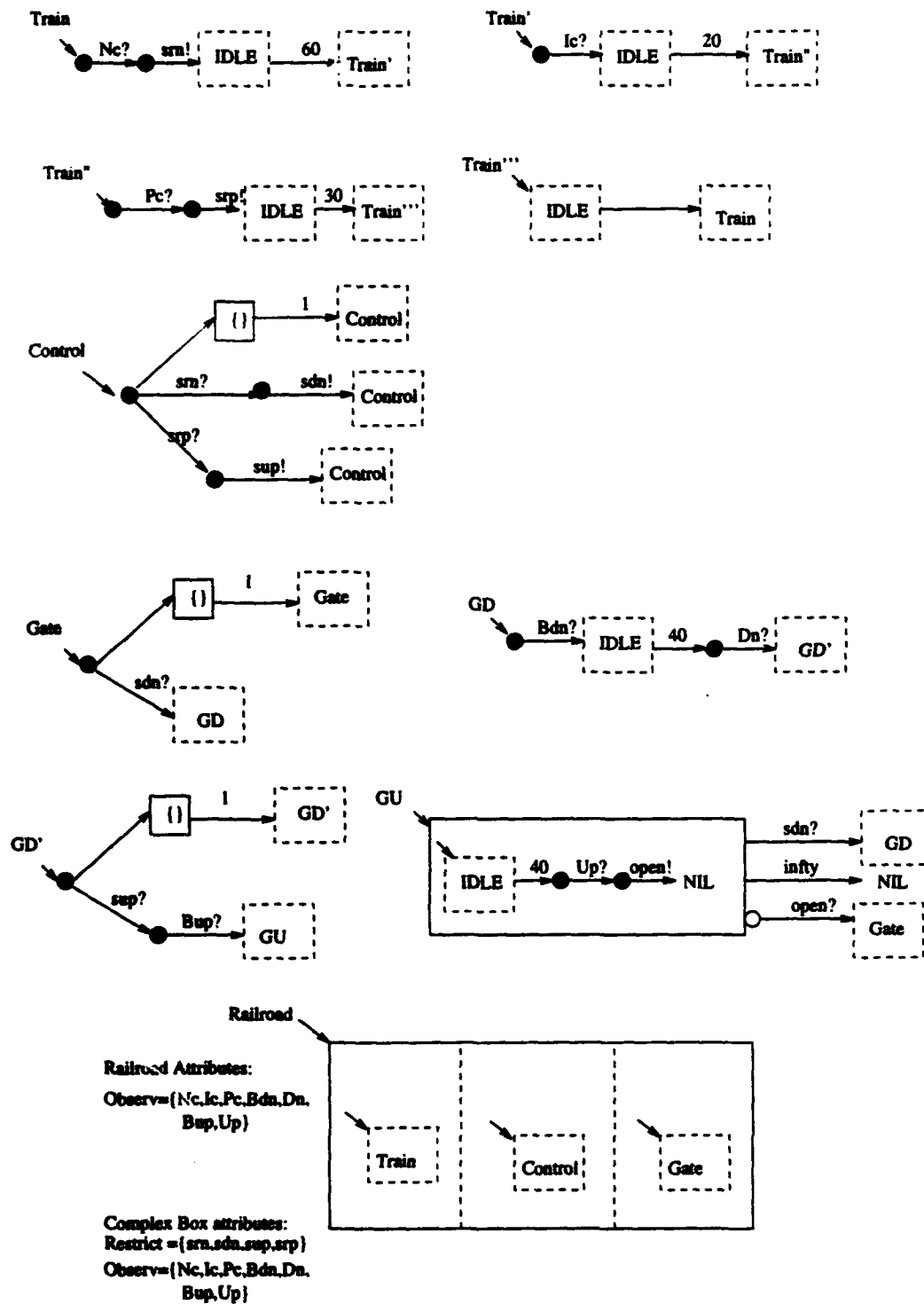


Figure 17: Example of graphical reductions applied to the example of Figure 16

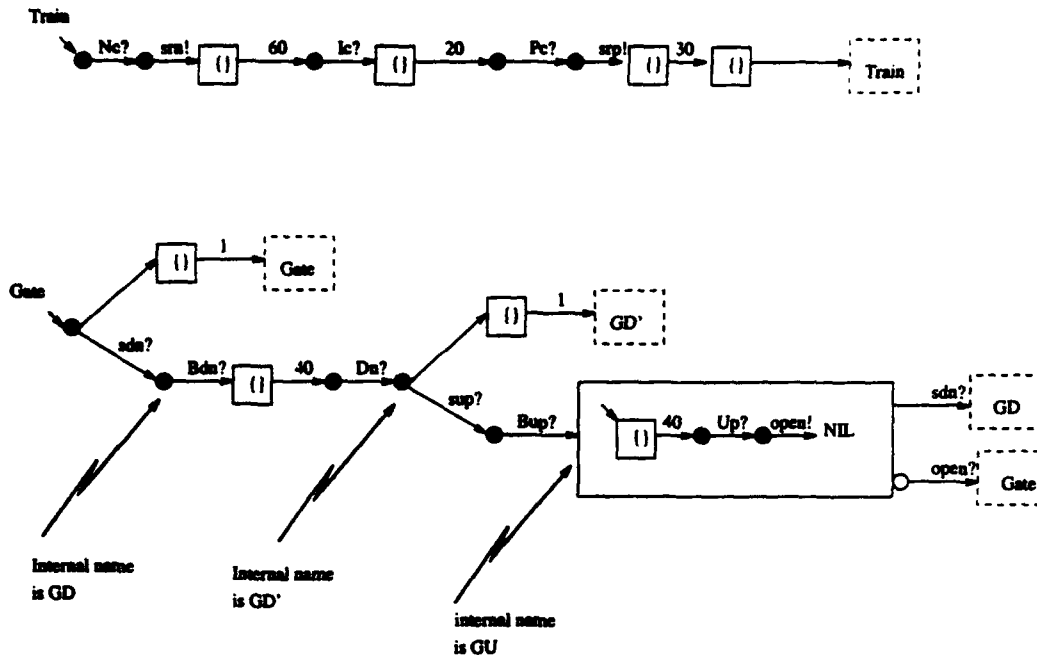


Figure 18: Further unfolding applied on Train, Gate of the example of Figure 17.

at which time it tries to get access to the buffer to copy the value from the buffer to a display (*D*); copying and displaying the data takes each one unit unit. Further simplifications on the initial nodes of Server and ReadSensor are possible.

Router. Figure 22 shows the GCSR and corresponding ACSR specifications of a router's specification *RouterSpec* and implementation *RouterImp*. The router is initially idle until it receives *dataIn* signal. It then reads the data in one time unit, tries to send in two time units at which time it signals an acknowledgement *ack*, then waits one time unit before returning to its initial state. The implementation of the router, *RouterImp*, shows the details of the sending unit. It consists of a unit to prepare the datagram for transmission, synchronizing with a unit that gets transmission permission. If the router has permission to forward the datagram, it carries the send in one time unit and signals a *success* acknowledgement, otherwise it executes an error handler routine and signals *failure*.

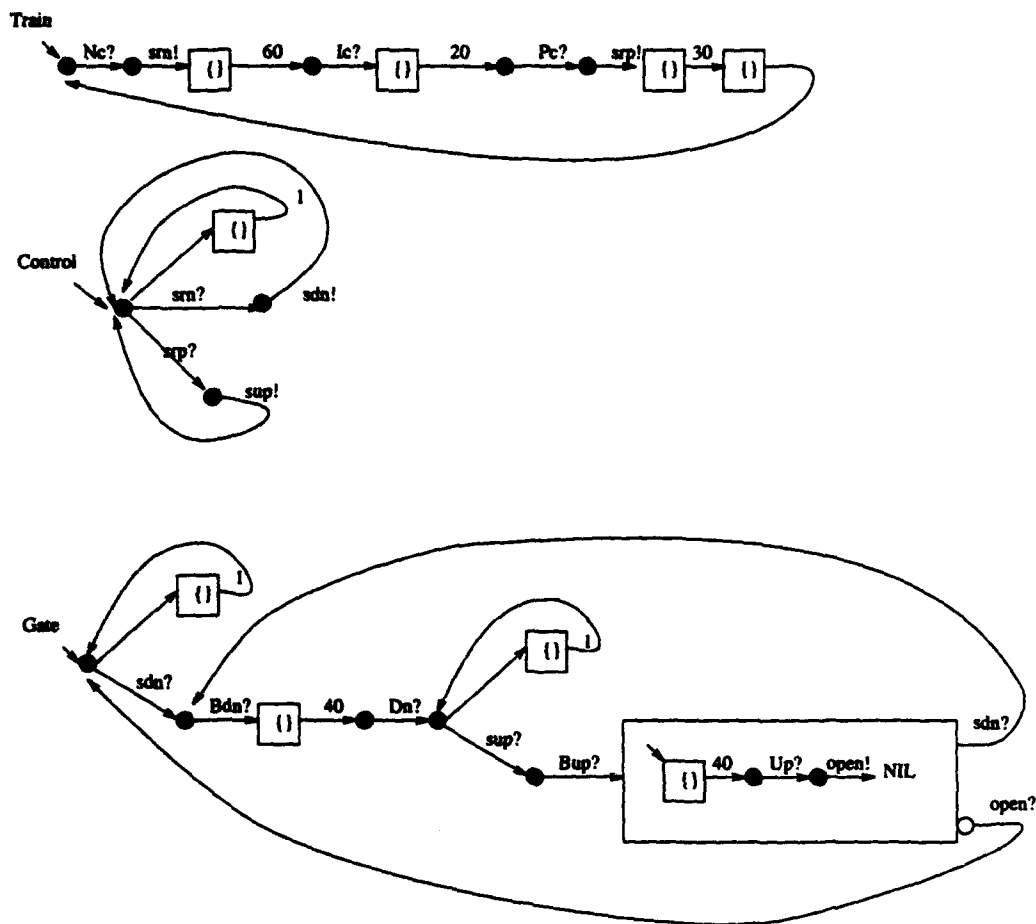


Figure 19: Further unfolding applied on Train, Gate and Control of the example of Figure 18.

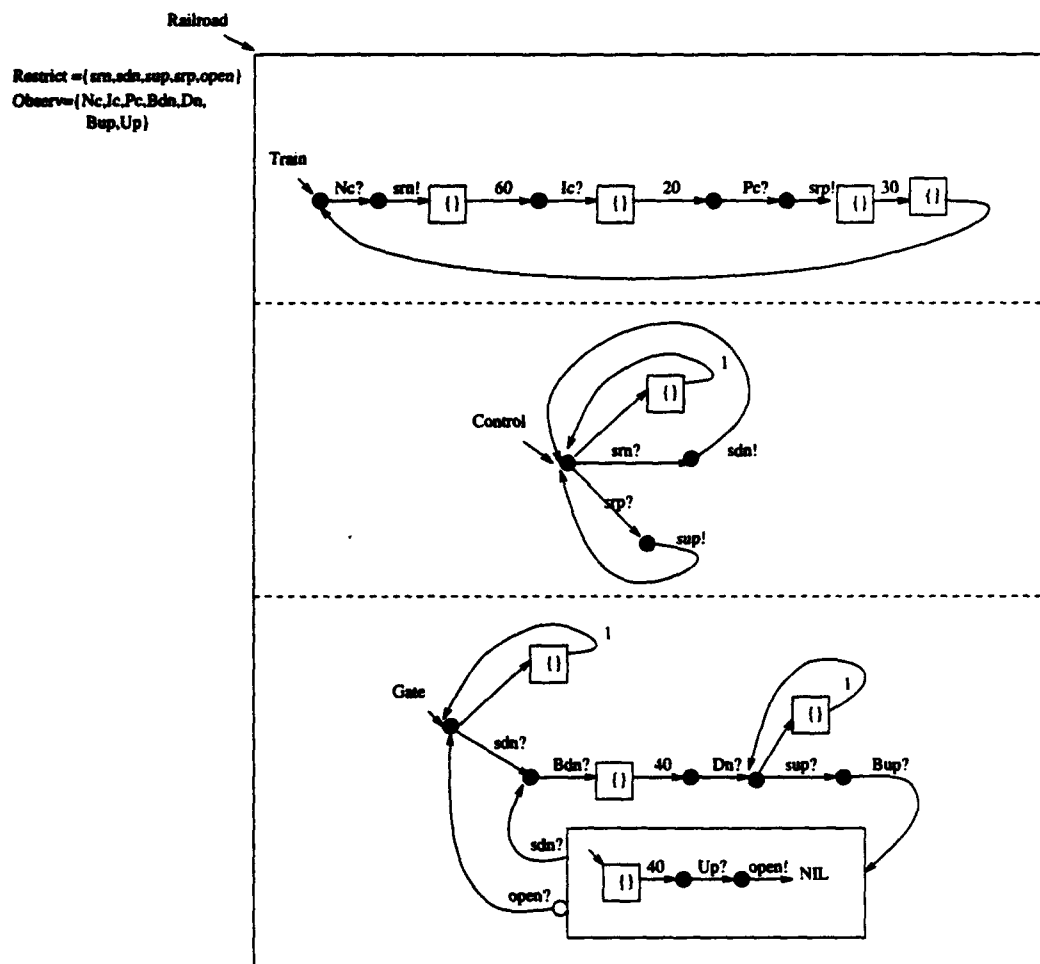


Figure 20: Railroad Crossing Example: All reference nodes have been replaced.

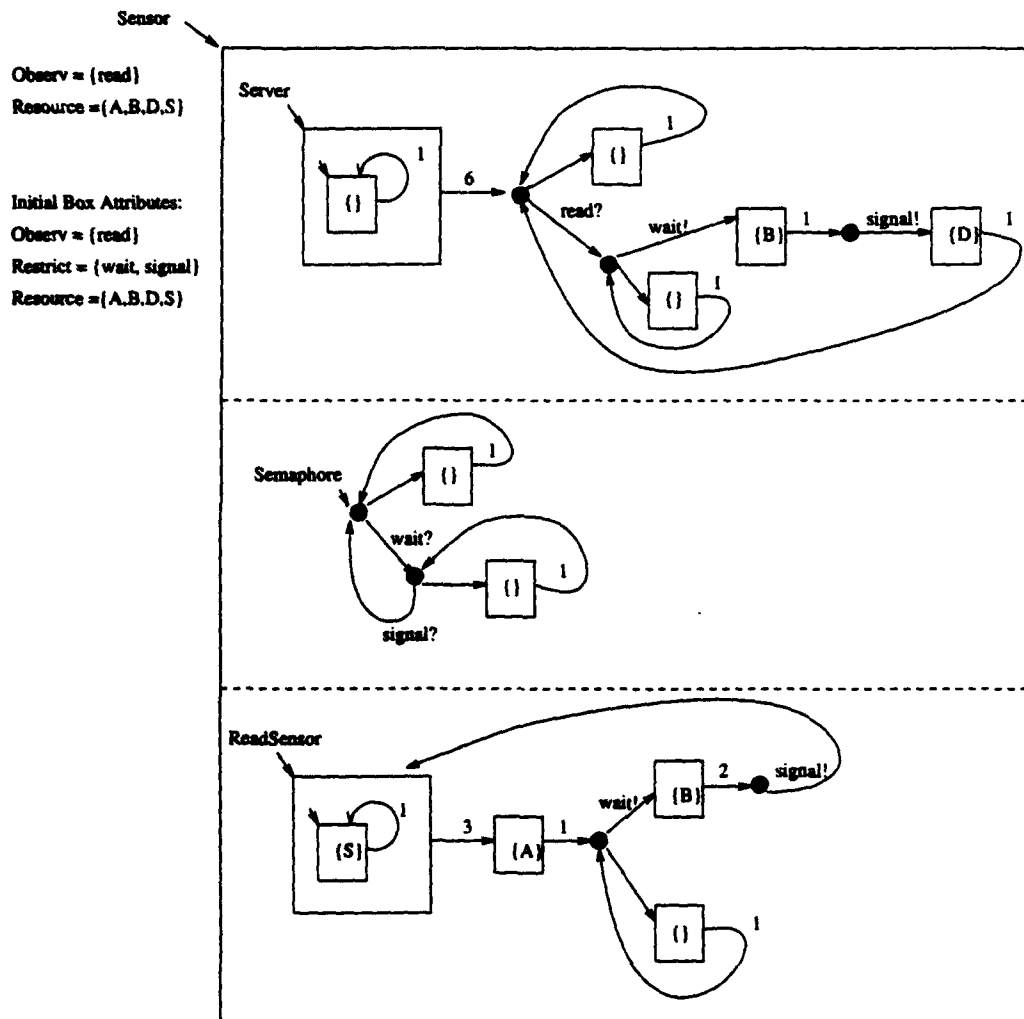
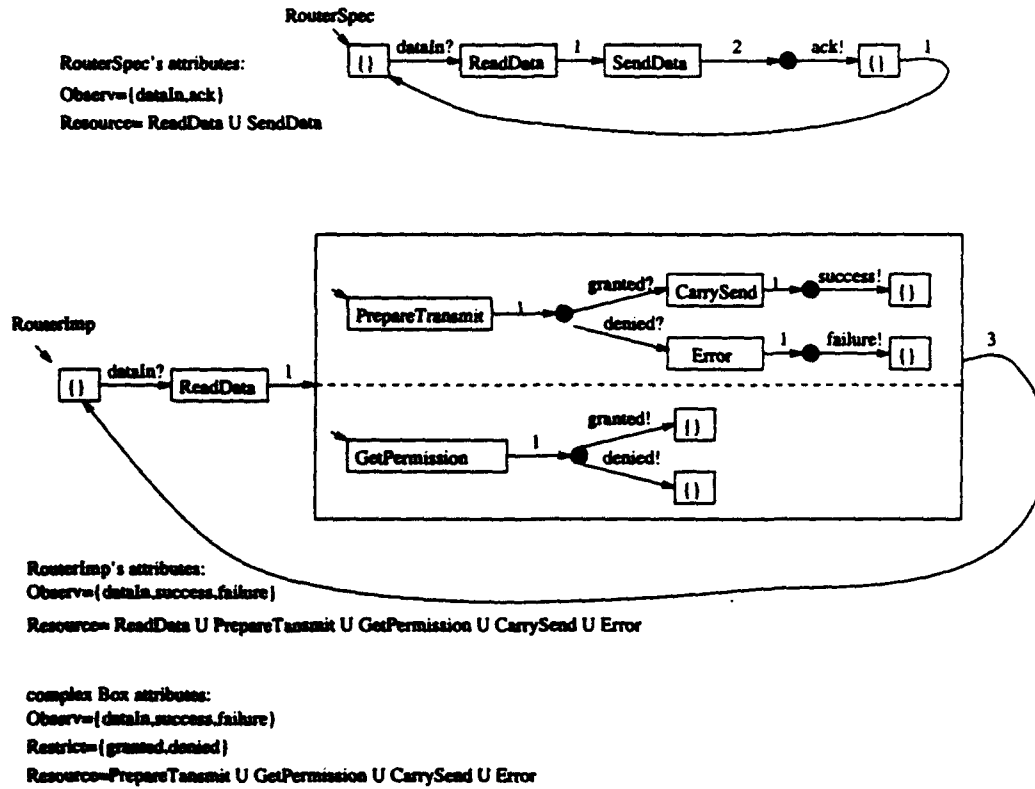


Figure 21: Sensor Example. Reference nodes have been replaced.



$\text{RouterSpec} = \text{IDLE } \Delta_{\infty} (\text{NIL}, \text{NIL}, \text{dataIn}.(1, \text{ReadStep}).(2, \text{SendStep}).\overline{\text{ack}}.\{\}: \text{RouterSpec})$
 $\text{RouterImp} = \text{IDLE } \Delta_{\infty} (\text{NIL}, \text{NIL}, \text{dataIn}.(1, \text{ReadStep}).\text{Process})$
 $\text{Process} = \text{Process}' \Delta_3 (\text{NIL}, \text{RouterImp}, \text{NIL})$
 $\text{Process}' = ((1, \text{PrepareTransmit}): (\overline{\text{granted}}.(1, \text{CarrySend}).\overline{\text{success}}.(\infty, \{\}): \text{NIL} + \overline{\text{denied}}.(1, \text{Error}).\overline{\text{failure}}.(\infty, \{\}): \text{NIL}))$
 $\parallel$
 $(1, \text{GetPermission}): (\overline{\text{granted}}.(\infty, \{\}): \text{NIL} + \overline{\text{denied}}.(\infty, \{\}): \text{NIL})) \setminus \{\text{granted}, \text{denied}\}$

Figure 22: Router Example

5 Automatic Analysis Techniques

To support the automatic analysis of GCSR specifications, we have investigated state minimization algorithms and verification techniques that we plan to implement. Section 5.1 briefly explains the state minimization algorithm we plan to implement. Section 5.2 identifies a set of equivalence relations that are weaker than the ones we currently have. We believe that these weaker notions are much more practical than our current notions of prioritized strong and weak bisimulations. Section 5.3 describes logic that we have designed to facilitate the partial specification (i.e., requirement specification) and model checking of ACSR.

5.1 State Minimization Algorithms

One of the goals of this project is to identify efficient algorithms for the automated verification of distributed real-time systems based on state space exploration. There exist several automatic verification techniques for finite state systems. Such techniques are usually based on *state space exploration*. That is, they first identify a set of states that are reachable from the initial states and then analyze this set for verification. They are used for proving absence of deadlock or livelock, for proving properties expressed in propositional temporal logic or real-time logic, and for determining trace equivalence, testing preorder or bisimulation equivalence, etc.

There exist several state minimization algorithms for a labeled transition system that collapse a set of states that are bisimilar into an equivalent class [17, 33]. These algorithms require the generation of the entire state space, including unreachable states. Thus, they can be applied only to systems with a finite, relatively small state space. It would be desirable to explore only the reachable portion of the state space. Bouajjani *et al.* have developed such an algorithm to find the minimal reachability graph for unlabeled transition systems [5]. The algorithm performs reachability analysis and minimization simultaneously. This algorithm is very effective when the reachable portion is much smaller than the full state space. But, unlabeled transition systems used by them are not suitable for describing concurrent systems since they cannot capture internal actions and communication actions. Our algorithm is an extension of the algorithm by Bouajjani *et al.* [5] to a labeled transition system.

The basic idea of minimizing a transition system is to find a partition of states such that all the states in each class of the partition are bisimilar and all bisimilar states are in the same class. The following describes the basic idea of our algorithm [18]:

repeat

```

pick a reachable class  $X$ ;
if  $s \xrightarrow{a} Y$  and  $s' \not\xrightarrow{a} Y$  for  $s, s' \in X$  then
    split  $X$ 
until no more splits possible

```

Starting from the class consisting of the entire states as the sole member of the initial partition (that is, $\rho_0 = \{Q\}$), the algorithm tries to iteratively split classes in the current partition until it is no longer possible. The splitting procedure keeps states in the same class until they are shown to be non-bisimilar. Such a class is called *stable* with respect to the current partition in the algorithm. In other words, for a given initial partition ρ_0 , the algorithm repeatedly split classes that are not stable with respect to the current partition until the coarsest stable partition is found. The coarsest stable partition is equal to the greatest bisimulation. We note the algorithm may not always terminate. It terminates only when the greatest bisimulation has finite number of equivalence classes.

Figure 23 gives the comparison of the results of the Paige and Tarjan's algorithm [33] and our algorithm. The partitions are the same except for unreachable state space. But, Paige and Tarjan's algorithm explores the whole set of states including unreachable states. Thus, our algorithm is based on a more efficient algorithm for the algorithm developed by Paige and Tarjan.

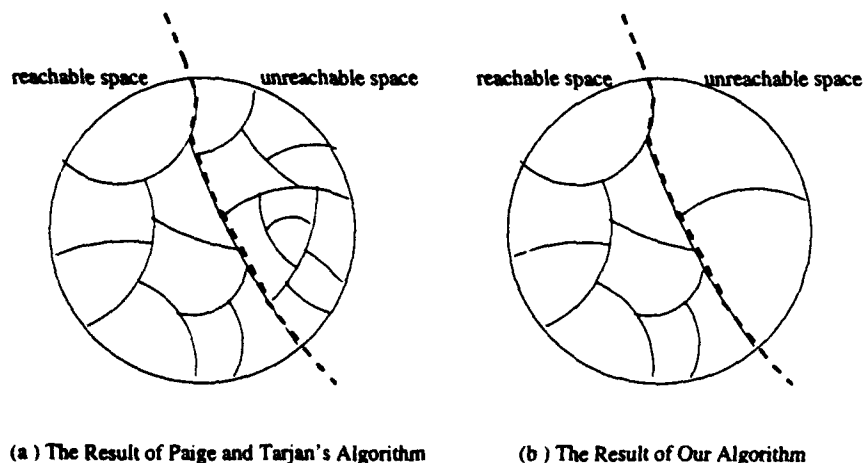


Figure 23: The Coarsest Partition

5.2 Additional Equivalence Relations for ACSR

Strong equivalence is a useful notion for comparison of agent expressions, but for many practical applications weaker forms of equivalence suffice. The need to weaken the requirement regarding exact matching of τ actions is well known, but ACSR introduces several other areas where requiring strict equivalence between expressions is cumbersome. For example, although precise priority values are required for each event and resource in an action, frequently it is the case that the relative priority levels are more important than the exact values.

The sections that follow will detail a number of equivalence relations weaker than strong bisimulation, but perhaps more useful for comparison of realistic implementations and specifications. In the remainder of this section, terms used throughout the definitions of equivalence are presented.

Definition 5.1 (Substitution of Terms) *The agent expression $A\{E/X\}$, where A and E are agent expressions and X is a process variable, shall denote the agent resulting from the syntactic substitution of agent expression E wherever X occurs free in A .*

Definition 5.2 (Sequential ACSR) *Full ACSR's sequential combinators form an algebra that we shall refer to as sequential ACSR. It has the following syntax:*

$$P ::= NIL \mid A : P \mid (a, n).P \mid P + P \mid \text{rec}X.P \mid X$$

The semantics of each of the operators is identical to the meaning assigned in full ACSR, including the definitions of preemption and prioritized transitions.

5.2.1 LTS Based Equivalences

Unprioritized Congruence. Although priorities are important for mediating competition for resources and implementing preemption based synchronization, the final determination as to whether or not two agent expressions are equivalent is likely to have more to do with the sequence of event labels and resource allocations generated than with the precise priority value associated with each event and resource. For example, consider the following processes:

$$\begin{aligned} P &= (a, 1).(\tau, 1).(b, 1).NIL \\ Q &= ((a, 1).(\text{sync}, 1).NIL \parallel (\overline{\text{sync}}, 1).(b, 1).NIL) \setminus \{\text{sync}\} \end{aligned}$$

Since P and Q generate the same traces of event labels (a , then τ , and then b), it is appealing to consider them equivalent in some sense. However, the semantics of synchronization require that the resulting τ event be assigned the sum of the priorities of the complementary events. Therefore $P \not\sim_\pi Q$ since the priority of the τ step in P must be 2 if P and Q are to be strongly bisimilar.

One way to address this problem would be to ignore priorities altogether in making the comparison, but since priorities play an important role in defining behaviors within an agent, the equivalence relation that resulted would be of little use. Instead, we define the following equivalence that tempers the notion of priority-free equivalence with the preservation of relative priority within agents.

Definition 5.3 (Unprioritized Equivalence: $=_?$) $P =_? Q$ iff, for all $\alpha \in \mathcal{D}$,

- (i) Whenever $P \xrightarrow{\alpha}_\pi P'$ then, for some Q' and $\beta \in \mathcal{D}$, $Q \xrightarrow{\beta}_\pi Q'$, $l(\alpha) = l(\beta)$ (for instantaneous events) or $\rho(\alpha) = \rho(\beta)$ (for timed actions), and $P' =_? Q'$.
- (ii) Whenever $Q \xrightarrow{\alpha}_\pi Q'$ then, for some P' and $\beta \in \mathcal{D}$, $P \xrightarrow{\beta}_\pi P'$, $l(\alpha) = l(\beta)$ (for instantaneous events) or $\rho(\alpha) = \rho(\beta)$ (for timed actions), and $P' =_? Q'$.

Proposition 5.1 *Unprioritized equivalence is an equivalence relation.*

Proposition 5.2 *Unprioritized equivalence is not a congruence relation for sequential ACSR.*

It follows directly from proposition 5.2 that unprioritized equivalence is not a congruence over full ACSR, since any counterexample formulated in sequential ACSR will also hold for full ACSR.

Unprioritized equivalence fails to be a congruence for sequential ACSR because of the interaction between initial prioritized transitions and transitions they may be combined with in an unprioritized choice. We can obtain a weaker equivalence that is a congruence for sequential ACSR by requiring equivalent priorities in initial prioritized transitions.

Definition 5.4 (Simple Unprioritized Congruence: $\sim_?$) $P \sim_? Q$ iff, for all $\alpha \in \mathcal{D}$,

- (i) Whenever $P \xrightarrow{\alpha}_\pi P'$ then, for some Q' , $Q \xrightarrow{\alpha}_\pi Q'$ and $P' =_? Q'$.
- (ii) Whenever $Q \xrightarrow{\alpha}_\pi Q'$ then, for some P' , $P \xrightarrow{\alpha}_\pi P'$ and $P' =_? Q'$.

Proposition 5.3 *Simple unprioritized congruence is an equivalence relation.*

Proposition 5.4 *Simple unprioritized congruence is a congruence relation for sequential ACSR.*

For full ACSR the situation is complicated by the existence of operators that can change priority relationships that exist deep within an agent. For example, consider the following processes:

$$\begin{aligned} P &= (e_1, 1).(e_2, 1).(\{(r_1, 0)\} : R_1 + \{(r_2, 3)\} : R_2) \\ Q &= (e_1, 1).(e_2, 1).(\{(r_1, 3)\} : R_1 + \{(r_2, 0)\} : R_2) \\ S &= [X]_{\{r_1, r_2\}} \end{aligned}$$

That $P \sim_f Q$ follows from the definition of simple unprioritized congruence since the initial event steps are identical and the same nondeterministic choice between r_1 and r_2 exists in both P and Q . However, $S\{P \setminus X\} \not\sim_f S\{Q \setminus X\}$ since closure with r_1 and r_2 introduces preemptions into P and Q eliminating the possibility of $P \Rightarrow R_1$ and $Q \Rightarrow R_2$.

Proposition 5.5 *Simple unprioritized congruence is not a congruence relation for full ACSR.*

Proof: By the previous counter-example. $\square$

To simplify the definition of a more robust unprioritized congruence we first define the following *close* operator for sets of resource, priority pairs. We then define unprioritized congruence in two steps, the first addressing potential preemptions that could arise from application of the ACSR close operator, and the second addressing the need for an exact match of priorities for the initial step of a process.

Definition 5.5 *Given two timed actions α and β , $\text{close}(\alpha, \beta)$ will represent the addition of resource and priority pairs to α that is necessary to insure that $l(\alpha) \supseteq l(\beta)$. Formally,*

$$\text{close}(\alpha, \beta) = \{(r, p) \mid (r, p) \in \alpha \vee (r \notin l(\alpha) \wedge r \in l(\beta) \wedge p = 0)\}.$$

Definition 5.6 $P \approx_f Q$ iff, for all $\alpha \in \mathcal{D}$,

- (i) Whenever $P \xrightarrow{\alpha} P'$ and $\neg \exists \alpha' \in \mathcal{D}$ such that $\alpha \prec \alpha'$ or $\text{close}(\alpha, \alpha') \prec \alpha'$ and $P \xrightarrow{\alpha'}$ then, for some Q' and $\beta \in \mathcal{D}$, $Q \xrightarrow{\beta} Q'$, $\neg \exists \beta' \in \mathcal{D}$ such that $\beta \prec \beta'$ or $\text{close}(\beta, \beta') \prec \beta'$ and $Q \xrightarrow{\beta'}$, $l(\alpha) = l(\beta)$ (for instantaneous events) or $\rho(\alpha) = \rho(\beta)$ (for timed actions), and $P' \approx_f Q'$.

- (ii) Whenever $Q \xrightarrow{\alpha} Q'$ and $\neg \exists \alpha' \in \mathcal{D}$ such that $\alpha \prec \alpha'$ or $\text{close}(\alpha, \alpha') \prec \alpha'$ and $Q \xrightarrow{\alpha'}$ then, for some P' and $\beta \in \mathcal{D}$, $P \xrightarrow{\beta} P'$, $\neg \exists \beta' \in \mathcal{D}$ such that $\beta \prec \beta'$ or $\text{close}(\beta, \beta') \prec \beta'$ and $P \xrightarrow{\beta'}$, $l(\alpha) = l(\beta)$ (for instantaneous events) or $\rho(\alpha) = \rho(\beta)$ (for timed actions), and $P' \approx_{\tau} Q'$.

Definition 5.7 (Unprioritized Congruence: $\approx_{\tau}$) $P \approx_{\tau} Q$ iff, for all $\alpha \in \mathcal{D}$,

- (i) Whenever $P \xrightarrow{\alpha}_{\pi} P'$ then, for some Q' , $Q \xrightarrow{\alpha}_{\pi} Q'$ and $P' \approx_{\tau} Q'$.
(ii) Whenever $Q \xrightarrow{\alpha}_{\pi} Q'$ then, for some P' , $P \xrightarrow{\alpha}_{\pi} P'$ and $P' \approx_{\tau} Q'$.

Proposition 5.6 Unprioritized congruence is a congruence relation.

τ -Free Congruence. The development of τ -free equivalence for ACSR parallels the development of weak bisimulation (observation equivalence) and observation congruence in CCS[31]. We begin by introducing a relation which allows all τ actions to be ignored, offering an equivalence that is useful for comparing complete agents, but fails to satisfy the requirements for a congruence.

As in CCS, we define an operator to “hide” the τ events in a trace, and a new transition relation that allows τ actions to be added at will.

Definition 5.8 If $t \in \mathcal{D}_E^*$, then $\hat{t} \in \mathcal{D}_E^*$ is the sequence obtained by deleting all occurrences of τ from t .

Definition 5.9 If $t = \alpha_1 \cdots \alpha_n \in \mathcal{D}^*$, then $E \xRightarrow{t} E'$ if

$$E(\xrightarrow{\tau})^* \xrightarrow{\alpha_1} (\xrightarrow{\tau})^* \cdots (\xrightarrow{\tau})^* \xrightarrow{\alpha_n} (\xrightarrow{\tau})^* E'.$$

We shall also write $E \xRightarrow{\hat{t}}$ to mean that $E \xRightarrow{t} E'$ for some E' .

Definition 5.10 (τ -Free Equivalence: $=_{\tau}$) $P =_{\tau} Q$ iff, for all $\alpha \in \mathcal{D}$,

- (i) Whenever $P \xrightarrow{\alpha}_{\pi} P'$ then, for some Q' , $Q \xrightarrow{\alpha}_{\pi} Q'$ and $P' =_{\tau} Q'$.
(ii) Whenever $Q \xrightarrow{\alpha}_{\pi} Q'$ then, for some P' , $P \xrightarrow{\alpha}_{\pi} P'$ and $P' =_{\tau} Q'$.

Proposition 5.7 τ -free equivalence is an equivalence relation.

Proposition 5.8 τ -free equivalence is not a congruence relation for sequential ACSR.

The way in which the preemption relation is defined for comparisons between τ events and time consuming actions greatly simplifies the task of recasting τ -free equivalence as a congruence (as compared to the work required above to form an unprioritized congruence from unprioritized equivalence). Just as in CCS, initial τ steps must be preserved if nondeterministic choice is to preserve τ -free congruence. But unlike unprioritized equivalence, the possibility that application of the ACSR close operator may introduce preemptions where none existed before is immaterial, since any nondeterministic choices that exist between time consuming action steps in τ -free equivalent processes must match exactly, so introduction of new preemptions through the close operator will effect all τ -free equivalent processes identically.

Definition 5.11 (τ -Free Congruence: $\approx_\tau$) $P \approx_\tau Q$ iff, for all $\alpha \in \mathcal{D}$,

- (i) Whenever $P \xrightarrow{\alpha}_\pi P'$ then, for some Q' , $Q \xrightarrow{\alpha}_\pi Q'$ and $P' =_\tau Q'$.
- (ii) Whenever $Q \xrightarrow{\alpha}_\pi Q'$ then, for some P' , $P \xrightarrow{\alpha}_\pi P'$ and $P' =_\tau Q'$.

Proposition 5.9 τ -free congruence is a congruence relation.

Time-Triggered Congruence. In [20] real-time systems are subdivided into time-triggered and event-triggered models, based on the way in which a system responds to its inputs. Briefly, time-triggered systems are polling systems, in which inputs are accumulated between polling intervals and acted upon as a group at the end of the polling interval without regard to the exact ordering of received events. In contrast, event-triggered systems are event driven, responding immediately upon receiving an input. Event-triggered systems can receive and respond to a potentially infinite stream of inputs between cycles of the system clock.

Although ACSR's underlying model is event-triggered, it is still possible to use ACSR to represent time-triggered systems. To do so, the author of a specification need only restrict their processes so that received events are not acted upon until a unit of time has elapsed, and the order in which events are received does not change the result computed. The result of this restriction is that the order of event occurrences between any given pair of time consuming actions becomes immaterial, since the events that happen between time intervals are predetermined by the end of the preceding time consuming action. For example, consider the following processes for handling out-of-paper alarms from a printer process:

$$PrintMonitor = recX.((PaperJam, 1).\emptyset : (\overline{SignalOperator}, 1).(\overline{WriteLog}, 1).X +$$

$$\begin{aligned}
& \emptyset : X \\
&) \\
\text{PrintMonitor}' &= \text{rec}X.((\text{PaperJam}, 1).\emptyset : (\overline{\text{WriteLog}}, 1).(\overline{\text{SignalOperator}}, 1).X + \\
& \emptyset : X \\
&)
\end{aligned}$$

No notion of equivalence that has been presented in the preceding sections could be used to claim that these two processes are equivalent, yet from a time-triggered point of view they must be, since the ordering of *SignalOperator* and *WriteLog* will not change the operation of any time-triggered agent *PrintMonitor* or *PrintMonitor'* could be composed with. Their behaviors are equivalent, from a time-triggered point of view.

We begin the definition of time-triggered equivalence with two preliminary definitions that will simplify the formal definitions.

Definition 5.12 If $t = \langle (e_1, p_1), (e_2, p_2), \dots, (e_n, p_n) \rangle \in \mathcal{D}_E^*$, then when $E \xrightarrow{(e_1, p_1)}_\pi \dots \xrightarrow{(e_n, p_n)}_\pi E'$ we write $E \xrightarrow{t}_\pi E'$. We shall also write $E \xrightarrow{t}_\pi$ to mean that $E \xrightarrow{t}_\pi E'$ for some E' .

Definition 5.13 Given an event trace $t \in \mathcal{D}_E^*$, the set of unique label and priority pairs (e_i, p_i) in t is denoted $\text{events}(t)$.

The definition of time-triggered equivalence addresses three types of process behaviors. For two processes to be equivalent, their initial untimed event sequences must match, any event sequences between corresponding time consuming actions must match, and any sequences of untimed events that lead up to a deadlocked state (*NIL*) must match. (To "match" two untimed event sequences means that they must be made up of the same set of unique (*label, priority*) pairs, although order and duplication is not significant.) The definition of *Action Equivalence* addresses the second and third conditions, while the definition of *Time-Triggered Equivalence* addresses initial sequences of untimed events, and deadlocked traces with no time consuming actions, while relying on the definition of *Action Equivalence* to address the remaining portions of traces.

Definition 5.14 (Action Equivalence: $=_{ae}$) $P =_{ae} Q$ iff, for all $A_1, A_2 \in \mathcal{D}_R$ and $t \in \mathcal{D}_E^*$,

- (i) Whenever $P \xrightarrow{A_1}_\pi \xrightarrow{t}_\pi P' \xrightarrow{A_2}_\pi$ then, for some Q' and $t' \in \mathcal{D}_E^*$, $Q \xrightarrow{A_1}_\pi \xrightarrow{t'}_\pi Q'$, $\text{events}(t) = \text{events}(t')$, and $P' =_{ae} Q'$;

- (ii) Whenever $Q \xrightarrow{A_1} \xrightarrow{t} P' \xrightarrow{A_2}$ then, for some P' and $t' \in \mathcal{D}_E^*$, $P \xrightarrow{A_1} \xrightarrow{t'} P'$, $\text{events}(t) = \text{events}(t')$, and $P' =_{ae} Q'$;
- (iii) Whenever $P \xrightarrow{A_1} \xrightarrow{t} P' \not\xrightarrow{A_2}$ then, for some Q' and $t' \in \mathcal{D}_E^*$, $Q \xrightarrow{A_1} \xrightarrow{t'} Q' \not\xrightarrow{A_2}$, and $\text{events}(t) = \text{events}(t')$.
- (iv) Whenever $Q \xrightarrow{A_1} \xrightarrow{t} Q' \not\xrightarrow{A_2}$ then, for some P' and $t' \in \mathcal{D}_E^*$, $P \xrightarrow{A_1} \xrightarrow{t'} P' \not\xrightarrow{A_2}$, and $\text{events}(t) = \text{events}(t')$.

Proposition 5.10 Action equivalence is an equivalence relation.

Definition 5.15 (Time-Triggered Equivalence: $=_t$) $P =_t Q$ iff, for all $A \in \mathcal{D}_R$ and $t \in \mathcal{D}_E^*$,

- (i) Whenever $P \xrightarrow{A} \xrightarrow{t} P' \xrightarrow{A}$ then, for some Q' and $t' \in \mathcal{D}_E^*$, $Q \xrightarrow{A} \xrightarrow{t'} Q'$, $\text{events}(t) = \text{events}(t')$, and $P' =_{ae} Q'$;
- (ii) Whenever $Q \xrightarrow{A} \xrightarrow{t} Q' \xrightarrow{A}$ then, for some P' and $t' \in \mathcal{D}_E^*$, $P \xrightarrow{A} \xrightarrow{t'} P'$, $\text{events}(t) = \text{events}(t')$, and $P' =_{ae} Q'$;
- (iii) Whenever $P \xrightarrow{A} \xrightarrow{t} P' \not\xrightarrow{A}$ then, for some Q' and $t' \in \mathcal{D}_E^*$, $Q \xrightarrow{A} \xrightarrow{t'} Q' \not\xrightarrow{A}$, and $\text{events}(t) = \text{events}(t')$;
- (iv) Whenever $Q \xrightarrow{A} \xrightarrow{t} Q' \not\xrightarrow{A}$ then, for some P' and $t' \in \mathcal{D}_E^*$, $P \xrightarrow{A} \xrightarrow{t'} P' \not\xrightarrow{A}$, and $\text{events}(t) = \text{events}(t')$.

Proposition 5.11 Time-triggered equivalence is an equivalence relation.

Proposition 5.12 Time-triggered equivalence is not a congruence relation.

Proof: Consider the following process specifications:

$$\begin{aligned}
 P &= (e, 1).(f, 1).NIL \\
 Q &= (f, 1).(e, 1).NIL \\
 S &= X + (e, 2).NIL
 \end{aligned}$$

That $P =_t Q$ follows from the definition, since they include sequences of untimed actions that differ only in the order of events. But substitution of P into S ($S\{P/X\}$) results in a preemption of the P sub-process by the $(e, 2)$ event of S that will not occur in the purely nondeterministic $S\{Q/X\}$. $\square$

The problems involved in creating a time-triggered congruence for full ACSR are more severe than for any of the previous relations. The difficulty arises from the restriction operator, which can terminate a process at any specified event label. Since allowing permutations of event labels is basic to time-triggered equivalence, and early termination of permuted event sequences may not produce the same set of *(label, priority)* pairs, there is no meaningful way to reformulate the definition so that restriction will preserve congruence. Instead, we settle for congruence over sequential ACSR.

Definition 5.16 (Partial Time-Triggered Congruence: $\sim_{tt}$) $P \sim_{tt} Q$ iff, for all $\alpha \in \mathcal{D}$,

- (i) Whenever $P \xrightarrow{\alpha}_{\pi} P'$ then, for some Q' , $Q \xrightarrow{\alpha}_{\pi} Q'$ and $P' =_{tt} Q'$.
- (ii) Whenever $Q \xrightarrow{\alpha}_{\pi} Q'$ then, for some P' , $P \xrightarrow{\alpha}_{\pi} P'$ and $P' =_{tt} Q'$.

Proposition 5.13 *Partial time-triggered congruence is an equivalence relation.*

Proposition 5.14 *Partial time-triggered congruence is a congruence relation over sequential ACSR.*

In fact, something stronger could be proved, since restriction is the only operator that must be eliminated from full ACSR to allow a congruence to be defined.

5.2.2 Trace Based Equivalences

Definition 5.17 (Traces) For an ACSR process P with action domain $\mathcal{D}$ a trace of P is a sequence of zero or more actions $\langle \alpha_1, \alpha_2, \dots, \alpha_n \rangle \in \mathcal{D}^*$ such that $P \xrightarrow{\alpha_1}_{\pi} P_1 \xrightarrow{\alpha_2}_{\pi} \dots \xrightarrow{\alpha_n}_{\pi} P_n$. The set of all traces of P is denoted $tr_{\pi}(P)$.

Definition 5.18 (Operators on Traces) For $t = \langle \alpha_1, \alpha_2, \dots, \alpha_n \rangle$ a trace of an ACSR process, $hd(t) = \alpha_1$ (the head of t) and $tl(t) = \langle \alpha_2, \alpha_3, \dots, \alpha_n \rangle$ (the tail of t).

Definition 5.19 (Trace Equivalence: $=_T$) $P =_T Q$ iff $tr_{\pi}(P) \subseteq tr_{\pi}(Q)$ and $tr_{\pi}(Q) \subseteq tr_{\pi}(P)$

Proposition 5.15 *Trace equivalence is an equivalence relation.*

Proposition 5.16 *Trace equivalence is a congruence relation for full ACSR.*

Trace equivalence is defined in terms of trace inclusion, which is simply a subset relation between trace sets. In the spirit of the LTS based equivalence relations presented in Section 5.2.1, there are other interesting equivalence relations between trace sets that can be defined to yield new trace based notions of equivalence. The first such relation ignores priority levels on actions and external events. It is analogous to unprioritized equivalence.

Definition 5.20 (Unprioritized Trace Inclusion: $\subseteq_{\neq}$) For ACSR processes P and Q , $tr_{\pi}(P) \subseteq_{\neq} tr_{\pi}(Q)$ iff for all $\langle \alpha_1, \alpha_2, \dots, \alpha_n \rangle \in tr_{\pi}(P)$, $\exists \langle \beta_1, \beta_2, \dots, \beta_n \rangle \in tr_{\pi}(Q)$ such that for all i , $l(\alpha_i) = l(\beta_i)$ (for events) or $\rho(\alpha_i) = \rho(\beta_i)$ (for actions).

Definition 5.21 (Unprioritized Trace Equivalence: $=_{T\neq}$) $P =_{T\neq} Q$ iff $tr_{\pi}(P) \subseteq_{\neq} tr_{\pi}(Q)$ and $tr_{\pi}(Q) \subseteq_{\neq} tr_{\pi}(P)$

In a system with resources dedicated to each process, the specific resources used by a process may be of little interest when comparing alternative implementations. The following equivalence relation captures that notion by matching only synchronization actions and the placement of timed steps, ignoring the resources consumed by timed steps.

Definition 5.22 (Resource Neutral Trace Inclusion: $\subseteq_{\neq}$) For ACSR processes P and Q , $tr_{\pi}(P) \subseteq_{\neq} tr_{\pi}(Q)$ iff for all $\langle \alpha_1, \alpha_2, \dots, \alpha_n \rangle \in tr_{\pi}(P)$, $\exists \langle \beta_1, \beta_2, \dots, \beta_n \rangle \in tr_{\pi}(Q)$ such that for all i , $l(\alpha_i) = l(\beta_i)$ (for events) or α_i and β_i are actions.

Definition 5.23 (Resource Neutral Trace Equivalence: $=_{T\neq}$) $P =_{T\neq} Q$ iff $tr_{\pi}(P) \subseteq_{\neq} tr_{\pi}(Q)$ and $tr_{\pi}(Q) \subseteq_{\neq} tr_{\pi}(P)$

The following equivalence relation extends the notion of τ -free equivalence to equivalence based on traces.

Definition 5.24 (τ -Free Equivalence of Traces: $=_{\neq}$) For traces s and t , $s =_{\neq} t$ iff

1. $s = \langle \rangle$ and $t = \langle \rangle$; or
2. $hd(s) = hd(t)$ and $tl(s) =_{\neq} tl(t)$; or
3. $hd(s) = \tau$ and $tl(s) =_{\neq} t$; or
4. $hd(t) = \tau$ and $s =_{\neq} tl(t)$.

Definition 5.25 (τ -Free Trace Inclusion: $\subseteq_{\tau}$) For ACSR processes P and Q , $tr_{\pi}(P) \subseteq_{\tau} tr_{\pi}(Q)$ iff for all $s \in tr_{\pi}(P)$, $\exists t \in tr_{\pi}(Q)$ such that $s =_{\tau} t$.

Definition 5.26 (τ -Free Trace Equivalence: $=_{\tau}$) $P =_{\tau} Q$ iff $tr_{\pi}(P) \subseteq_{\tau} tr_{\pi}(Q)$ and $tr_{\pi}(Q) \subseteq_{\tau} tr_{\pi}(P)$

Although there are many other interesting relations that can be defined, we believe that the above ones are most intuitive and thus probably practical.

5.3 Logic for Communicating Shared Resources

In this section, we describe briefly on LCSR, logic for communicating shared resources. LCSR is a temporal logic which is dedicated to ACSR. LCSR is well suited for expressing properties about temporal ordering of events. The temporal operators are useful for specifying ACSR program behavior. A LCSR formula, containing temporal operators, is interpreted over a structure of ACSR program. LCSR is event based and interval logic. Hence the semantics is based on intervals on time in ACSR program.

The syntax of LCSR is following:

$$\begin{aligned} E &::= 1 \mid e \mid R \mid e^k \mid R^k \mid \neg E \mid E_1 \vee E_2 \\ F &::= \text{true} \mid \xrightarrow{E_1}_I \langle E_2 \rangle F \mid \xrightarrow{E_1}_I \{E_2\} F \mid \neg F \mid F_1 \vee F_2 \end{aligned}$$

1 stands for disjunction of all events and all actions. e is instantaneous events and R is time consuming actions. Informally, $\xrightarrow{E_1}_I \langle E_2 \rangle F$ means that for some computation path, E_1 will occur contiguously until, within time I , event E_2 occurs at which point F is true. $\xrightarrow{E_1}_I \{E_2\} F$ means that there is for all computation paths, E_1 will occur contiguously until, within time I , event E_2 occurs at which point F is true. Those two operators can be expressed in terms of TCTL [1]:

$$\begin{aligned} \xrightarrow{e_1}_I \langle e_2 \rangle F &\implies \exists e_1 \mathcal{U}_I (e_2 \wedge F) \\ \xrightarrow{e_1}_I \{e_2\} F &\implies \forall e_1 \mathcal{U}_I (e_2 \wedge F) \end{aligned}$$

The followings are some interesting real-time properities and corresponding LCSR formulas.

- After the reaction process (denoted by the events "beg", "end") starts, it cannot be interrupted (denoted by the events "intBeg", "intEnd") for more than 12 time units unless the reaction process stops:

$$[beg] \xrightarrow{\neg end}_{[intBeg]_{[0,12]}} \{end \vee intEnd\}$$

- Once a job starts execution (denoted by "beg"), it will run to completion (denoted by "end") without missing deadlind (denoted by "out"):

$$[beg] \xrightarrow{\neg out} \{end^1\}$$

- Any sampling operations (denoted by "s") and injection operations (denoted by "p") in the gas container should be at least 7 time units apart from each other:

$$([s] \xrightarrow{\neg p}_{[7,7]} \{1\}) \wedge ([p] \xrightarrow{\neg s}_{[7,7]} \{1\})$$

- If the right button has been clicked (denoted by "b") for three times within 5 time units, there will be a menu popped up (denoted by "m") within 5 time units from the last click:

$$[b]_{[0,5]} [b^2]_{[1,5]} \{m^1\}$$

- Any interval of a train passing the cross (denoted by "enter" and "exit") should be contained strictly in an interval of the gate being fully down (denoted by "down" and "up"), and securing at least t time units beyond each end:

$$\begin{aligned} & [up] \xrightarrow{\neg enter} \{down^1\} \\ & \wedge [down] \xrightarrow{\neg enter}_{[t,t]} \{1\} \\ & \wedge [enter] \xrightarrow{\neg up}_{[t,\infty)} \{up^1\} \end{aligned}$$

1. There is no train entering the crossing from the gate being up to being down;
2. There is no train entering the crossing t time units from the gate being down;
3. Once a train enters the crossing, the gate will not be up until after t time units after it exits.

LCSR can naturally describe various desirable properties of a ACSR (i.e., GCSR) specification. The reasons for this new logic is because we could not use RTL as our requirement specification logic and other real-time (temporal) logics does not naturally describe properties of ACSR specifications. We believe that a LCSR formula can be efficiently model checked for ACSR specifications.

6 Phase II Implementation Plan

One goal of Phase I is to evaluate existing graphical specification systems to check whether they can be used in implementing the toolkit environment. In Section 2.2 and Section 2.3, Modechart was evaluated and compared with ACSR. It was shown that ACSR terms can be translated into Modechart by extending the graphical specification language. Also, GCSR, a graphical specification language for ACSR, was defined in Section 3. It follows that MCTool, a graphical specification system based on Modechart, can be extended to be a graphical front-end of the toolkit environment. On the other hand, analysis tools of the toolkit environment can be implemented by reusing source code from VERSA, an ACSR-based tool for the algebraic analysis of real-time systems [7].

The overall structure of the toolkit environment is discussed in Section 6.1. The implementation plan of reusing and extending MCTool and VERSA are presented in Section 6.2.

6.1 The Overall Structure of the Toolkit Environment

A high level view of the toolkit environment is presented in Figure 24.

The environment has a main menu window which contains commands of opening a specification, creating a specification and terminating the environment. Once a user opens or creates a specification, a menu window for the specification appears. The menu window contains commands that apply to a specification as a whole. The commands include saving, reverting to a saved specification, printing, closing and performing analysis. A user creates, views and modifies a GCSR specification using a graphics editor. It has icons of nodes and edges of a GCSR graph. A user creates a graphical object by clicking one of the icons. Then he annotates the created object using a corresponding template for its attributes. Also, he can manipulate graphical objects by copying, deleting, pasting, aligning, enlarging, shrinking, etc. A GCSR specification of a large-scale system can be represented by a set of graphs and their hierarchy. A user can view the specification by scrolling the windows and navigating the hierarchy of the graphs. The graphics editor provides a navigation map for traversing the whole specification expressed in GCSR.

A GCSR specification can be converted automatically to an ACSR specification that can also be automatically translated to the CSR state machine for analysis. Once a specification has been converted to the CSR state machine, the analyst may execute the specification and test it to determine its reasonableness. The analyst may then apply optional state mini-

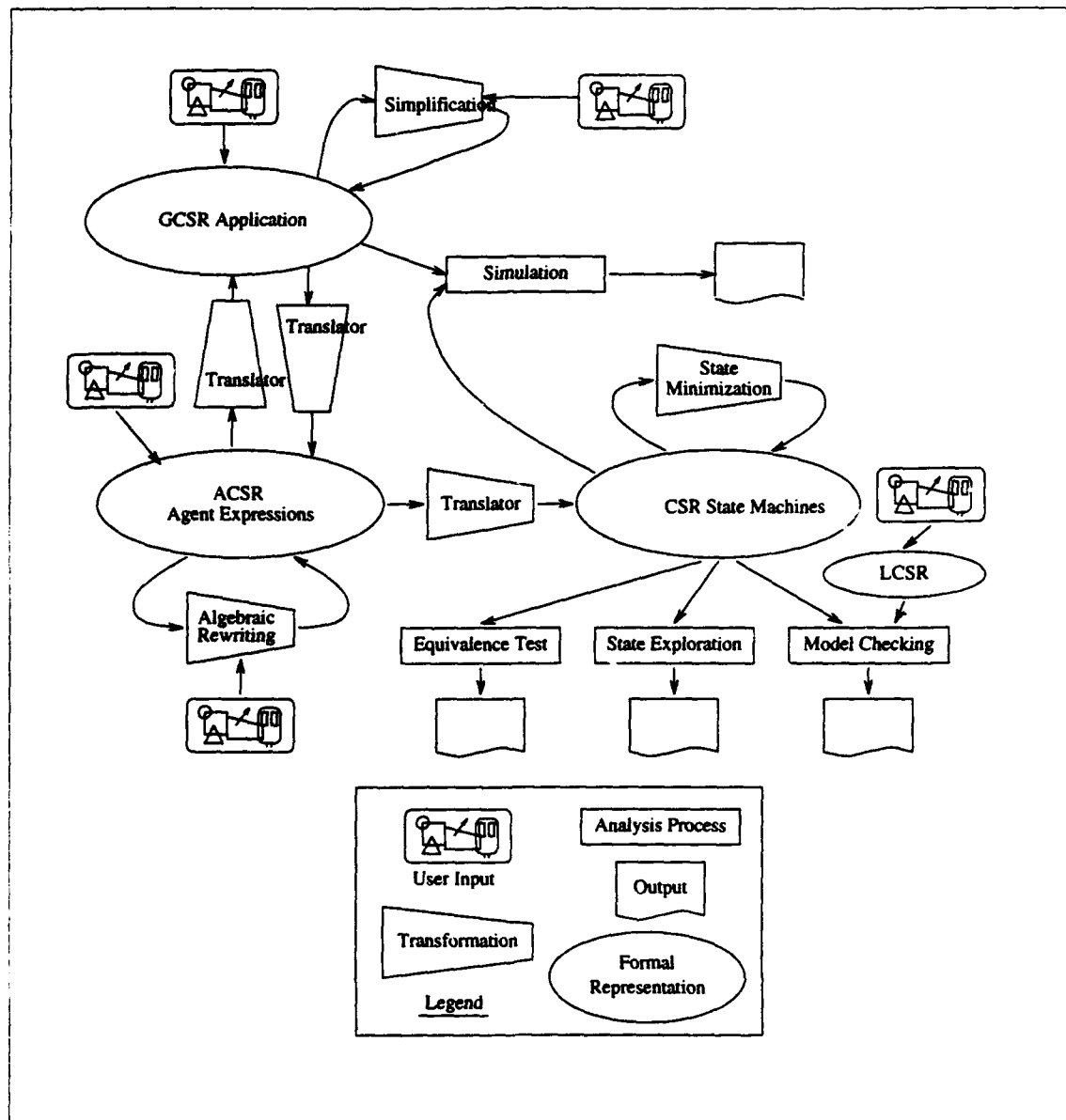


Figure 24: GCSR Toolkit Environment

mization algorithms to the specification. Though the effectiveness of state minimization will vary, successful application of this process can significantly reduce the computing resources required by later analysis phases.

Analysis of the CSR state machine will be carried out using four basic analysis tools. The first analysis tool is a simulation tool that demonstrates operational behaviors of a specification by executing the CSR state machine. The GCSR graphs of the specification being simulated are used as a graphical user interface. The analyst uses the GCSR graphs as input ports for entering simulation parameters. The results from the simulation are displayed through the graphs.

The second analysis tool is a model checking tool that will allow the GCSR specification to be tested against LCSR propositions. LCSR propositions can be formulated to assert the truth or falsity of various properties of the system, and the model checking algorithms can be used to verify whether the CSR state machine (and consequently the GCSR specification modeled), satisfies the given property.

The third analysis tool is a state exploration tool that can be used to generate valid traces of actions for the system being analyzed. For finite systems it will be possible to examine all valid traces of the system in question. For both finite and infinite systems the state exploration tool will allow interactive exploration of traces, the selective enumeration of subtraces, generation of all traces of a given finite length, and generation of fixed length traces on the basis of random selection, or statistical weights of alternatives.

The fourth analysis tool will test for equivalence between two or more alternative GCSR specifications. The type of equivalence verified may be relatively crude, such as trace equivalence, or it may be a more sophisticated equivalence, such as weak bisimulation as described in Section 5.2. For models that incorporate probability weights to characterize nondeterminism, the analyst can use probabilistic bisimulation.

6.2 Implementation

The toolkit environment will be implemented in C++ and Unix/X windows environment. The source code of MCTool and VERSA can be reused during the implementation.

The graphical user interface of the toolkit environment can be implemented by reusing the source code of MCTool: The main menu window can be constructed by reusing the main window of MCTool. The Specification window of MCTool can be used for implementing the specification menu window of the environment.

The graphics editor of the environment will be implemented by extending the Work

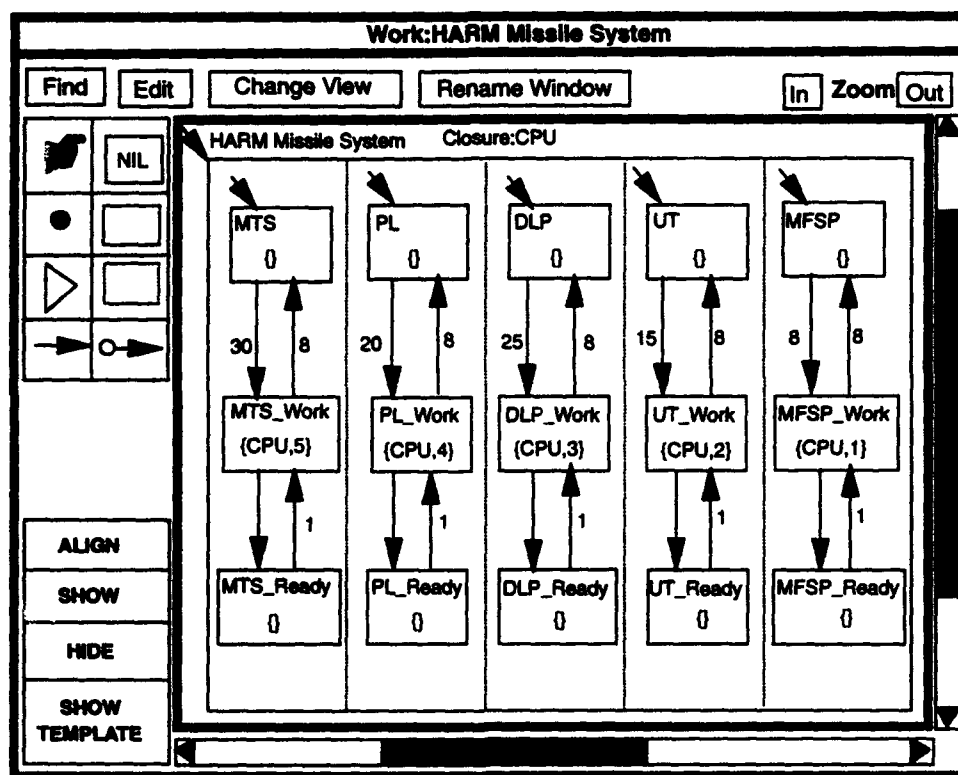


Figure 25: Work Window: the HARM missile example

window of MCTool. New icons for the NIL node, the Dot node, the Recursion node, the Reference node, the External Exit edge, and the Internal Exit edge should be added as to the Work window as shown in Figure 25. The templates of the nodes and the edges must be created so that a user can enter and update the values of the attributes of the associated graphical objects. Figure 25 illustrates a sample Work window which contains the HARM missile example in GCSR. The navigation map of the graphics editor will be implemented by reusing the Locator window of MCTool. The Locator window displays the entire specification. A Work window indicator of the Locator window denotes the portion of a specification displayed in the Work window. The indicator is used in rapid navigation around specification. The indicator can be dragged and resized.

The current version of VERSA has tools for algebraic rewriting, equivalence testing (for strong and weak bisimulation) and interactive execution [7]. The algebraic rewriting tool and equivalence testing tool can be reused for the toolkit environment. The interactive execution tool can be extended to be the GCSR simulator of the environment. VERSA is now being extended by augmenting tools of model checking and state exploration. Those tools can also be reused for the toolkit environment.

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. Alur and D. Dill. Automata for modeling real-time systems. In *Proc. of 17th ICALP, LNCS 443*, pages 322-335. Springer Verlag, 1990.
- [3] H. Attiya and N. Lynch. Time Bounds for Real-Time Process Control in the Presence of Timing Uncertainty. In *Proc. of IEEE Real-Time Systems Symposium*, pages 268-284, December 1989.
- [4] C. Belzile, G. MacEwen, and G. Marquis. RNet: A Hard Real-Time Distributed Programming System. *IEEE Transaction on Computers*, C-36(8):917-932, August 1987.
- [5] A. Bouajjani, J.-C. Fernandez, and N. Halbwachs. Minimal Model Generation. In *Proc. of the second Workshop on Computer-Aided Verification*, 1990.
- [6] Patrice Brémont-Grégoire. *Aprocess Algebra of Communicating Shared Resources with Dense Time and Priorities*. PhD thesis, Department of Computer and Information Science, The University of Pennsylvania, Philadelphia, PA 19104, 1994. Tech. Report MS-CIS-94-24.
- [7] Duncan Clarke, Insup Lee, and Hong liang Xie. VERSA: A Tool for the Specification and Analysis of Resource-Bound Real-Time Systems. Technical Report MS-CIS-93-77, Dept. of CIS, Univ. of Pennsylvania, Sept 1993.
- [8] P.C. Clements, C.L. heitmeyer, B.G. Labaw, and A. T. Rose. Mt: A toolset for specifying and analyzing real-time systems. In *Proc. of IEEE Real-Time Systems Symposium*, pages 12-22, Raleigh-Durham, North Carolina, December 1-3 1993.
- [9] M.K. Franklin and A. Gabrielian. A Transformational Method for Verifying Safety Properties in Real-Time Systems. In *Proc. of IEEE Real-Time Systems Symposium*, pages 112-123, December 1989.
- [10] A. Gabrielian and M.K. Franklin. Multilevel Specification of Real-Time Systems. *Comm. of ACM*, 35(5):51-60. 1991.
- [11] R. Gerber and I. Lee. A Layered Approach to Automating the Verification of Real-Time Systems. *IEEE Trans. on Software Eng.*, 18(9):768-784, 1992.

- [12] T. Henzinger, Z. Manna, and A. Pnueli. Temporal Proof Methodologies for Real-Time Systems. In *Proc. of ACM Principles of Programming Languages*, 1991.
- [13] K. Hong and J. Leung. Preemptive Scheduling With Release Time and Deadlines. *Real-Time Systems: The International Journal of Time Critical Computing Systems*, 1(3), December 1989.
- [14] J. Hooman. *Specification and Compositional Verification of Real-Time Systems*. PhD thesis, Eindhoven University of Technology, 1991.
- [15] F. Jahanian, R.S. Lee, and A. Mok. Semantics of Modechart in Real Time Logic. In *Proc. 21st Hawaii Int. Conf. on System Sciences*, Jan. 88.
- [16] F. Jahanian and A.K. Mok. Safety analysis of timing properties in real-time systems. *IEEE Transactions on Software Engineering*, SE-12(9):890-904, September 1986.
- [17] P. C. Kanellakis and S. A. Smolka. CCS Expressions, Finite State Processes, and Three Problems of Equivalence. *Information and Computation*, 86:43-68, 1990.
- [18] Inhye Kang and Insup Lee. State Minimization for Concurrent System Analysis Based on State Space Exploration. to appear in *Proc. of COMPASS*, June 1994.
- [19] M.F. Kleyn and J.C. Browne. A high level language for specifying graph based languages and their programming environments. In *15th International Conference on Software Engineering, I.E.E.E. Proc.*, pages 16-21, Baltimore, Maryland, May 1993.
- [20] H. Kopetz and K. Kim. Temporal uncertainties in interactions among real-time objects. In *Proc. 9th Symposium on Reliable Distributed Systems*, pages 165-174, October 1990.
- [21] R. Koymans. Specifying real-time properties with metric temporal logic. *Real-Time Systems*, 2(4):255-299, 1990.
- [22] B. Krämer, Luigi, and V. Berzins. Compositional Semantics of a Real-Time Prototyping Language. *IEEE Trans. on Software Eng.*, 19(5):453-477, May 1993.
- [23] I. Lee, P. Brémont-Grégoire, and J. Berber. A Process Algebraic Approach to the Specification and Analysis of Resource Bound Real-Time Systems. *Proceedings of the IEEE*, pages 158-171. Jan 1994.

- [24] I. Lee and V. Gehlot. Language Constructs for Distributed Real-Time Programming. In *Proc. IEEE Real-Time Systems Symposium*, 1985.
- [25] C.L. Liu and J.W. Layland. Scheduling algorithms for multi-programming in a hard-real-time environment. *Journal of the ACM*, pages 46 – 61, January 1973.
- [26] N. Lynch and H. Attiya. Using Mappings to Prove Timing Properties. Technical Report MIT/LCS/TM-412b, Laboratory for Computer Science, Massachusetts Institute of Technology, 1988.
- [27] 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.
- [28] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, 1992.
- [29] M. Merritt, F. Modugno, and M. Tuttle. Time-Constrained Automata. In *CONCUR '91*, August 1991.
- [30] B. Meyer. *Introduction to the Theory of Programming Languages*. Prentice Hall, 1990.
- [31] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.
- [32] J.S. Ostroff and W.M. Wonham. Modelling, Specifying and Verifying Real-time Embedded Computer Systems. In *Proc. of IEEE Real-Time Systems Symposium*, pages 124–132, December 1987.
- [33] R. Paige and R.E. Tarjan. Three Partition Refinement Algorithms. *SIAM J. Comput.*, 16(6), December 1987.
- [34] D. Peng and K.G. Shin. Modeling of Concurrent Task Execution in a Distributed System for Real-time Control. *IEEE Transactions on Computers*, pages 500–516, April 1987.
- [35] J.L Redondo. Schedulability Analyzer Tool. Technical Report UILU-ENG-93-1706, University of Illinois at Urbana-Champaign, Feb 1993.
- [36] G.M. Reed and A.W. Roscoe. A Timed Model for Communicating Sequential Processes. In *Proc. of Int. Conf. on Automata, Languages and Programming*. LNCS 226, Springer Verlag, 1986.

- [37] F.B. Schneider, B. Bloom, and K. Marzullo. Putting Time into Proof Outlines. Technical Report TR-93-1333, Cornell University, March 1993.
- [38] L. Sha, R. Rajkumar, J. Lehoczky, and K. Ramamritham. Mode change Protocols for Priority Driven Preemptive Scheduling. *Real-Time Systems: The International Journal of Time Critical Computing Systems*, 1(3), December 1989.
- [39] A.C. Shaw. Reasoning About Time in Higher-Level Language Software. *IEEE Transactions on Software Engineering*, 15(7):875-889, 1989.
- [40] J.A. Stankovic, K. Ramamritham, and S. Cheng. Evaluation of a Flexible Task Scheduling Algorithm for Distributed Hard Real-Time Systems. *IEEE Transactions on Computers*, pages 1130-1143, December 1985.
- [41] A. D. Stoyenko. A Schedulability Analyzer for Real-Time Euclid. In *Proc. of IEEE Real-Time Systems Symposium*, pages 218-227, December 1987.
- [42] H. Tokuda and M. Kotera. A Real-Time Tool Set for the ARTS Kernel. In *Proc. of IEEE Real-Time Systems Symposium*, pages 289-298, December 1988.
- [43] W. Zhao, K. Ramamritham, and J. Stankovic. Preemptive Scheduling under Time and Resource Constraints. *IEEE Transactions on Computers*, pages 949-960, August 1987.