Behavior Description and Safety in Real Time Models

Reuven Greenberg

Technical Report #91-21

March 3, 1991

Abstract

This paper includes a survey on some modern methods that are used for describing and analyzing behavior of complex systems. It is believed that most safety problems arise in the interface between the controlling parts and other controlled subsystems. Therefore, a prerequisite for a good interface is an accurate definition of the system. Two objectives are evaluated: the description power and the analysis power for safety and timing properties. This is done by describing and analyzing a simple system that is composed of two doors, which are restricted by time and "safety" requirements. It is found that although good description methods do exist, their usefulness for analyzing safety timed properties is very limited.

Contents

List of Figures

List of Tables

1 Introduction

As the use of computers in controlling systems increased, new methods for system behavior description were introduced. The common denominator for those methods is their attempt to describe and reason about concurrent processes. This is different from early requirement models by both the objectives of the methods and their applicable areas. Not only do these methods specify the intended behavior of the system (i.e., *what* is needed), but they enable analysis of important properties as well, e.g., safety analysis.

The other difference, applicable areas, originated from the need to model concurrent processes and time restrictions that are almost inherent properties of reactive systems. Early methods were used for modeling environments that are *computer centralized*, operating systems in particular. In these environments the peripheral equipment serves the needs of the computer and not the converse. Reactive systems are entirely different; the computer services the other components (which may also be reactive in this sense). Moreover, the behavior of the environment in computer centralized systems is more or less predictable, whereas less is known about the environment in which a reactive system is installed. Here the environment is usually modeled and the knowledge is almost always incomplete.

There are two important outcomes of this distinction. First, the development methodology is reversed. In computer centralized systems the computer is the starting point and the requirements of the environment are derived from it. The environment has to fit the computer characteristics. The "design stream" is directed from the center to the periphery. In reactive systems the methodology is entirely different. The starting point is the environment out of which the system properties are derived and only then the computer is defined. The "design stream" is now directed from the periphery to the center. No longer does the computer define the environment, rather the environment defines the computer.

Another important outcome of this distinction is the way time and events are dealt with. In computer centralized systems the number of events is small and the order in which they are handled is determined by the computer. Therefore, the concept of "real time" is limited to the order of events and not to their exact physical time of occurrence. Obviously this is not the case in reactive systems. Such systems are required to operate in an uncontrolled environment, in which the number of events is usually very large and may occur in any order as well as simultaneously. The computer is supposed to respond to any type of event order by initiating activities that will comply with the environment.

Moreover, reactive systems are frequently safety critical as well. The system response to certain events **must** be correct and on time. Thus, performance considerations become not only a matter of convenience, (i.e., will the user receive a respond within one, two or three minutes), but the correct operation is dependent on the time. "Correct" results that arrive too late (or early) are often useless. Furthermore, a "correct" response but not on time may lead to hazardous situations and damage just as "incorrect" results could. A scram instruction for a nuclear reactor that is issued after core melt down has begun will bring the same consequences as if was not issued at all, and an early detonation of a bomb may destroy the aircraft from which it is released.

Many methods are considered to be "real-time" and indeed they were applied for real time systems. But in many cases this use was not concurrent or did not incorporate time properties. This survey is restricted to methods that provide for both time definitions and concurrent description.

In the next section general properties of description methods are presented and a simple "reactive system", *the clean room*, is portrayed. The remainer of the second section contains a detailed description of six methods, Statecharts [Harel86, HLN88], Modecharts [JM89, Mok85], ESM/RTTL [OW87, Ostroff88, Ostroff89], Timed Petri-nets [Merlin74, Peterson81], CIRCAL [Milne85, Milne82] and TAM [Zwarico88, LZ88]. The third section is composed of other three methods, Interval Logic [Ladkin86.1, Ladkin86.2, Ladkin87], CCS [Milner80, Milner89] and CSP [Hoare78, Hoare85] which are described in less detail. The fourth and last section is a summary.

2 Methods Presentation

Behavior description methods may be partitioned into two types: state based and process based. State based methods describe the behavior in terms of states and transitions. Usually they incorporate a pictorial representation of finite state automata aided by operators for parallel execution, time restrictions and features for avoiding a combinatorial explosion of states. The analysis is either based on a formal theory that includes basic axioms and inference rules, or on reachability graphs. Usually, the analysis is not tailored to a particular description, and it is possible to apply one analysis to a system described in a different method.

Process based methods are usually algebraic. The process is described as a sequence of events that "drives" the system. The analysis is usually done on the possible traces of the events in which the process can be engaged or by proving equivalence of the process and some "safe process". Here the analysis methods are a "built-in" part of the description, and usually it is not possible to apply an analysis method for a process that is described in a different description method.

In choosing the following methods we tried to reflect the present status in the area of specifications for reactive systems. As was mentioned we tried to include the three main approaches to the problem, algebraic, logical and semi-dynamic . Certainly there are other methods that are not included. We did not include here the most well-known algebraic methods CCS and CSP. However, we included two methods that are based on them. We start the presentation with four state-based methods and conclude with two that are process-based.

The usefulness of the methods is evaluated by describing and analyzing a simple "reactive system", called the clean room system, which is presented in figure 1. Although this is a simple system, it includes many time properties that can be found in more sophisticated systems. The fact that one system is analyzed by all the methods provides the reader with a convenient medium for comparison, so the evaluation can be done independently of the authors. In fact, the reader is encouraged to do so and challenge the conclusions of this paper. It is hoped that this paper provides enough tools for doing that independent evaluation, though some more details can be found in the cited references.

We divide the presentation of each method into three parts. The first part contains a general discussion and the second part brings the clean room example. In the third part the method is evaluated. In the evaluation we are going to stress the disadvantages of the method although we also assess its advantages. It is not that we think that disadvantages of a method are more important than its advantages, but we believe that the disadvantages of a method are going to determine whether it is used. Besides, the advantages can be found very easily in reading the original papers that describe each method. It is also true that as safety engineers we are used to thinking in a

The Clean Room Requirements

A clean room is a room that has to be kept dust free. In order to achieve it the air pressure in the room has to be kept above that of the environment. It is required to plan a control mechanism for a clean room containing two doors. Since the air-pressure mechanism can not keep the high pressure when the two doors are opened simultaneously, it is required that at most one door should be allowed to open at any time. Opening or closing a door should be done by a button. Also, each door should be closed after two minutes unless the open button is touched again. Furthermore, given the conditions for allowing an opening or closing a door exist, the door has to complete its movement within 5 seconds after the respective button has been touched. If the conditions do not exist the user should be notified to wait within 2 seconds.

Figure 1: The clean room example.

negative way.

2.1 State Based Methods

In this section four state based methods for behavior representation are evaluated: Statecharts [Harel86, HLN88], Modecharts [JM89, Mok85], ESM/RTTL [OW87, Ostroff88, Ostroff89] and Timed Petri-nets [Merlin74, Peterson81] . These methods represent a variety of approaches to the problem. Statecharts and Modecharts are similar in their concept but the second contains a logic part for proving time properties, whereas the first lacks any proof ability. ESM/RTTL defines a mathematical theory that is based on temporal logic as its proof scheme. Petri-nets is more suitable for the design stage and can be analyzed in a semi-dynamic way by reachability graphs.

2.1.1 STATECHARTS

General: Statecharts were first introduced by D. Harel from the Weizmann Institute in Israel [Harel86]. It is an attempt to supply a pictorial description for concurrent processes in reactive systems. The control of such systems is influenced by states, events and conditions. Events trigger transitions or start activities if given conditions (that guard the transition from occurring inadvertently) are satisfied. For example, upon sensing *power increase* (event) in *automatic control mode* (condition) the control rods *should be lowered* (activity) to the nearest safe point. As will be seen, Statecharts follows such representations.

Statecharts were developed for representing behavior in a visual manner. Previous attempts to represent the behavior of reactive systems were unable to overcome the "exponential explosion" of states and control lines in simple flat diagrams. It is well known that this problem must be handled by using abstraction structures, modularity and hierarchy. Such representations reduce the number of states and communication lines considered at any time.

Statecharts uses a Finite State Machine (FSM) formalism in which transitions are taken if specified events occur under certain conditions. Information about conditions is "transmitted and received" in a broadcast mechanism. Thus, no communication lines are specified. It is assumed that every FSM has access to all data items and thus "knows" the exact situation of the whole system. The exponential explosion is avoided by using abstraction techniques in which several states which are in the same level of abstraction and have common characteristics, are encapsulated into a Superstate. Statecharts uses the word "event" to represent time markers that consume no time. An "Activity", on the other hand, consumes time and therefore, can be captured as a state. Moreover, an activity is bounded by two events, one for the starting point and the other for its ending point. In general, transitions are represented by arrows which are labeled by *name(condition)/event*. A transition is taken when event *name* occurs if *condition* exists, thereby activating another *event*. Each of the *condition*s or *event*s or both may be missing. The clean room example will clarify these concepts.

The Clean Room Example: Figure 2 describes preliminary requirements for the clean room problem. States are represented by "rectangles with rounded corners". The states of each door are encapsulated in superstates.

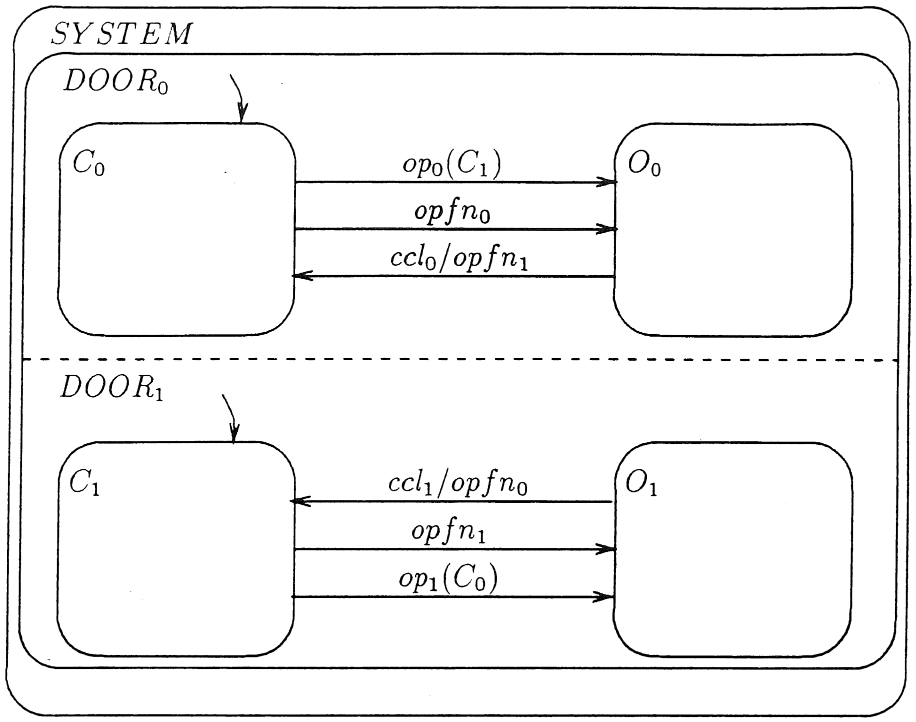


Figure 2: The behavior of the clean room problem in Statecharts.

The dashed line between the two subsystems represents two superstates that co-exist simultaneously, i.e., parallel or orthogonal processes. The small arrows that are attached to the close states (C_0 and C_1) represent the default state, i.e., the doors are closed. A door is considered "closed" only if it is completely closed. This means that a moving movement door, either to open or to close is considered open. Since a door can open either when an open button is pressed (if the other door is close), or after waiting for the other door to close, two open events exist. These events are represented in the figure by *op* ("open" button touched), or *opfn* ("open from notification" state). The third event *ccl* ("complete closing" activity) means that only when the door is completely closed, it is enters state C. Although two events can cause a door to close, pressing the close button or two minutes timeout, only one event represents the fact that a door is closed again. As can be seen a door changes states from C to O only if the other door is closed. This is expressed in the notations $op_0(C_1)$ and $op_1(C_0)$. The figure further shows that events $opfn_0$ and $opfn_1$ are activated by ccl_1 and ccl_0, respectively, which guarantees that the other door is closed.

It is obvious that many details are not addressed in this representation. For example, a notification state, the opening and closing activities, etc. These details are abstracted out of this figure and will be seen in the next level which is represented in figure 3. Since the states of both doors are equivalent only superstate $DOOR_0$ will be shown. As can be seen state C_0 contains three sub-states, FC_0 ("Fully Closed"), TNA_0 ("To Notification Activity") and NOT_0 ("NOTification" state). The transition from FC_0 to NOT_0 occurs as a result of event op_0 if $DOOR_1$ is in state O_1. State TNA_0 is an intermediate state that represents the activity before the user is notified. It ends when state NOT_0 is entered by event ent_0 ("end 'to notification' activity"). Notice that state TNA_0 may exist for 2 seconds at most. The superstate O_0 includes three states, OA_0 ("Opening Activity"), FO_0 ("Fully Opened") and CA_0 ("Closing Activity"). As is obvious state OA_0 that represents the movement of $DOOR_0$, is entered first when superstate O_0 is entered. State OA_0 is exited and state FO_0 is entered when event cop_0 ("completely open") occurs, i.e., the door is completely open. The closing activity in $DOOR_0$ (state CA_0) starts either by event cl_0 if the close button was pressed, or by event $2mn$ if two minutes elapsed with no op_0 event. Every occurrence of event op_0 at this stage will cause the two minutes timer to restart counting. The closing activity ends by event ccl_0 which (as mentioned earlier) activates an opening activity in $DOOR_1$ by event $opfn_1$, if $DOOR_1$ is in state NOT_1[1].

There are still requirements that are missing in figure 3. First, the five seconds restriction for the opening or closing process and second, the two seconds restriction for notifying a user that the other door is open. They can be represented by a timeout mechanism or timeout events. The idea is as follows: Suppose an exception handling state is defined for each time restriction (in this case three states). Such a state is entered if an event did not occur within its time restriction.

The idea of timeout can be further expended by incorporating exception handling states for unexpected events. Such events can occur even in a simple system as the clean room. For example, a button is out of order and is sensed

[1]This explanation hides some serious semantics issues. It is not clear when exactly does event $opfn_1$ occur relative to event ccl_0. Such fine problems may rise in real projects and should be carefully considered. More about these problems and generally about Statecharts semantics can be found in [HGdR88, HRdR88].

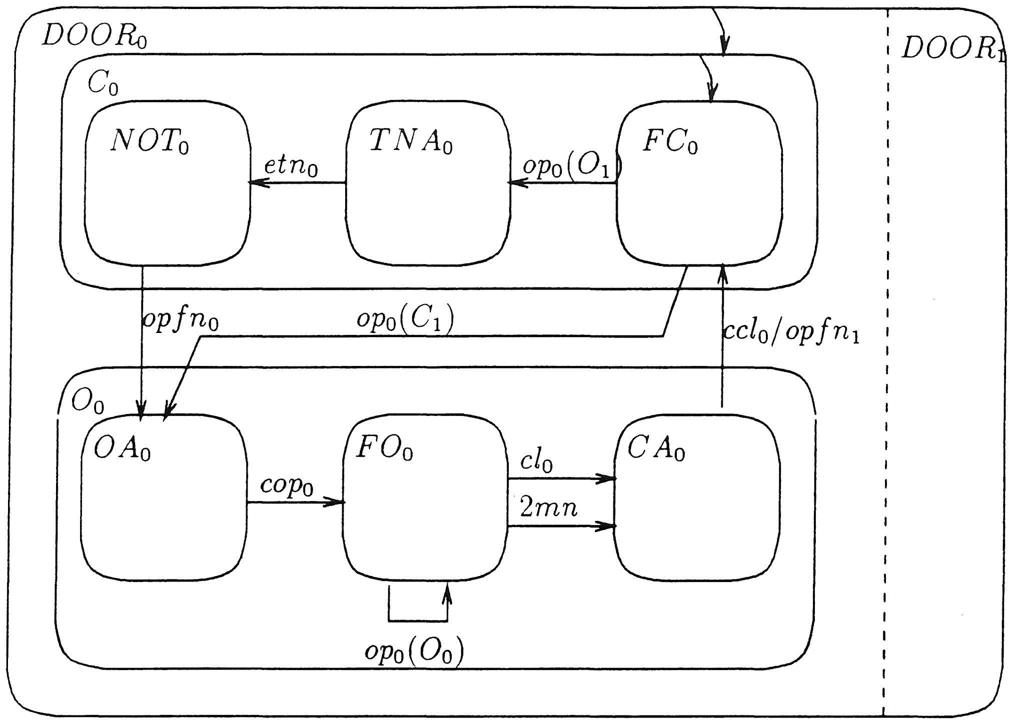


Figure 3: Abstraction features of Statecharts.

as always pressed, or a door does not close after waiting two minutes in open state and no button was pressed. Such events are usually not specified, but should be regarded in a safety critical system [JLHM91, Leveson86]. We do not include them in this paper in order to avoid confusion.

Statecharts has many other features for representing common requirements for reactive systems. History connective Ⓗ to indicate that upon entering to the superstate the assigned state is chosen. In case that the history connective is applied recursively until the most inner state is reached, the sign is changed to (H*). Conditions are represented by Ⓒ and selections, that allow transitions based on selected event, by Ⓢ . States that include periodical moves (for example, check temperature every 2 seconds) are represented by having jagged edge on the state rectangle and indicating the period time. Other time constraints are inserted as part of conditions. Recover procedures can also be handled in Statecharts by a "recover" state (or states) which is entered if unexpected events occur or malfunctioning is detected.

Evaluation: Statecharts is an evolving and promising method for representing system behavior. The variety of features that are included in it enables a good decomposition of complex systems. The fact that the system incorporates a visual facility makes the method easy to read and understand. Abstraction and hierarchy structuring avoid the exponential explosion of other diagrammatic methods.

The state/superstate visual relation and the parallel or orthogonal FSMs are the essence of Statecharts. Both make Statecharts intuitively understandable. The introduction of orthogonal states reduces the number of states and transitions drastically, thereby enabling a nice and appealing representation of complex systems. Even a simple example such as the clean room may end up with more than thirty states and transitions instead of figure 3. Another advantage of Statecharts is the ability to adapt it for many purposes. The method is very flexible and a user may decide what features to use or not.

The broadcast mechanism for information acquisition and access contribute to the representation simplicity but may become very confusing for analyzing. The problem arises when complex systems are described. Such projects usually deal with complex transitions in which the enabling conditions are very complex and dependent on many parameters. In these cases it becomes very difficult to track the origin of each parameter. It may become even more confusing when similar situations may exist in many different states. This is very similar to the problem that programmers encountered when complex programs were written in languages that did not have any scoping restrictions.

Another disadvantage of Statecharts is the lack of time representation. Time constraints are not represented in a natural and visual manner, rather they may be incorporated as conditions that do not allow tolerance assignments to subsystems and may add to the difficulty of the analysis. This can clearly be seen from the clean room example. No time restriction of the problem is represented. As was mentioned the only time restrictions that are included in Statecharts are periodical moves.

Statecharts were used successfully in many projects of various types and and scale. The Israeli Aircraft Industries used it manually for the development of the Lavi fighter. This project indicated the possibilities for using the method for complicated systems. In an experiment that was conducted in Microelectronics and Computer Technology Corporation (MCC) experienced software engineers were asked to learn Statecharts capabilities and use it for specifying an elevator system. The results showed great advantage of the method for the hierarchical decomposition of the system structure. It also indicated that some of the notations were difficult to understand. Recently the method was incorporated in a development environment called STATEMATE by i-Logix Inc. [HLN88]. Other developers used Statecharts for communication protocols [ZJ89], VLSI [VNG90], aircraft collision avoidance system [LHHRO91], etc. Each of these developers tailored the method to his specific needs.

2.1.2 RTL and MODECHARTS

General: Modecharts was introduced by F. Jahanian and A. L. Mok [JM89] as an implementation language for reasoning about time constraints in real-time systems. The language is based on the semantics of a timed first order calculus called RTL (Real-Time Logic) which was invented by the same authors [JM86]. The main reason for introducing modecharts was to enhance RTL with visual tools and to add hierarchical decomposition for large systems. The purpose of both RTL and Modecharts is to apply accurate reasoning to time constraints in real-time systems. The analysis is conducted by showing that safety assertions are not violated by the requirements, i.e., the negation of safety assertions and system specifications are inconsistent. The term *mode*, its use and representation, is very similar to the term *state* in Statecharts. There is a slight difference between these two terms. Mode refers to some way of operation, whereas state describes a situation of the system. A nuclear reactor may be in a maintenance mode in which the state of each subsystem is equal to the state in real operation.

RTL formalism is based on four concepts and several notations that are applied to them. The basic concepts are:

ACTIONS: are defined as units of work that can be done either in parallel or in series, and are abbreviated as "$X||Y$" or "$X;Y$", respectively. Synchronization points are denoted by "$!N$", that is, "$X!N$" and "$!NY$" denotes that time point N represents the end of action X

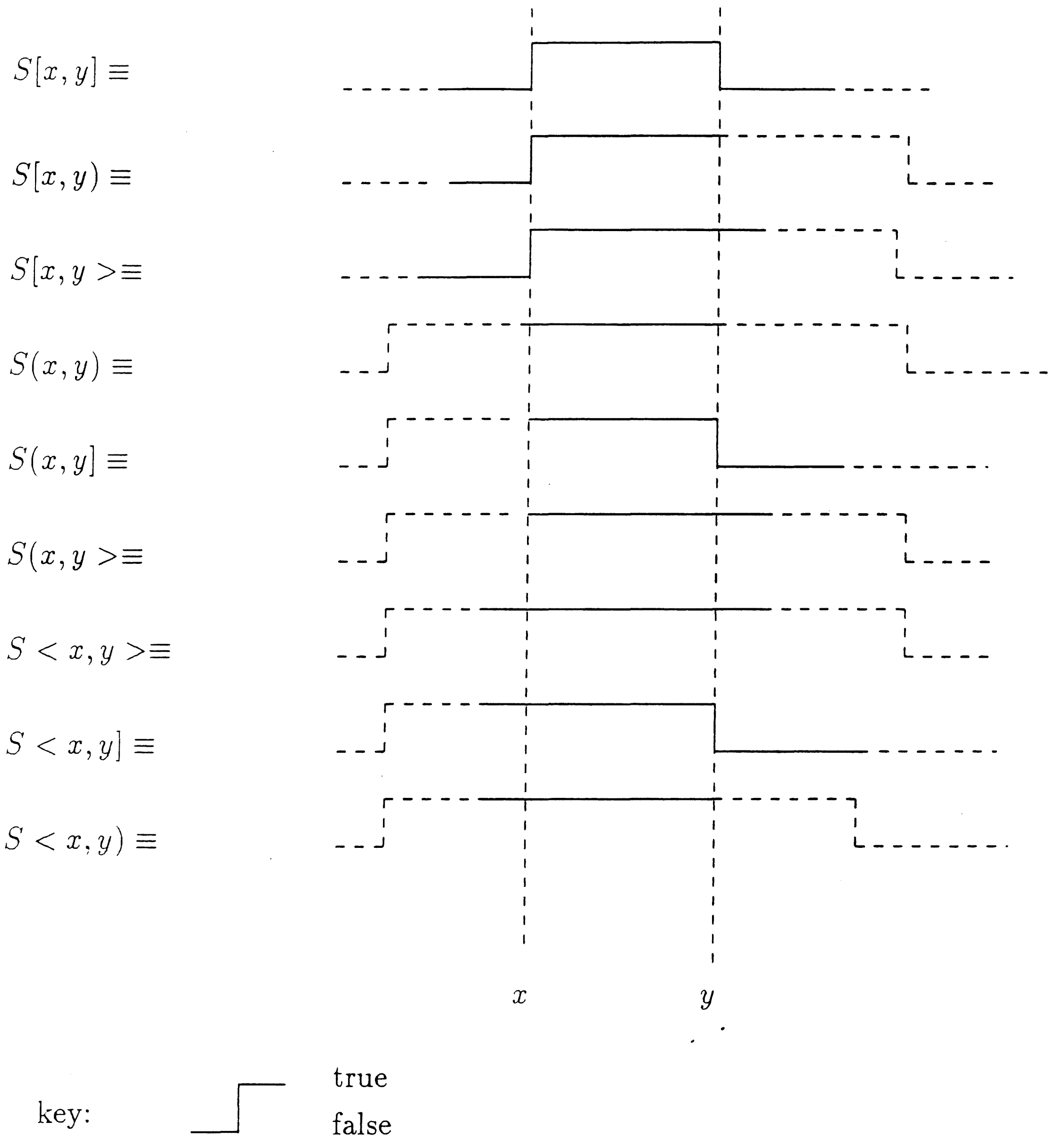


Figure 4: Schematic definitions of state predicates. x and y are events, dashed and solid lines mean respectively "unknown" and "known" time of occurrence.

and the beginning of action Y. The notations "$\uparrow A$" and "$\downarrow A$" denote markers (i.e., events, see below) for start and end of action A, respectively.

STATE PREDICATE: is an assertion about the physical state of the system. It is a boolean variable that is assigned values according to the physical system state. RTL supplies nine different forms for representing time assertions of a state predicate S, over a time interval T. Figure 4 defines them schematically.

EVENT: is a temporal marker that points of an occurrence which is significant for the system behavior.

TIMING CONSTRAINT: is an assertion about the absolute timing of certain events. RTL distinguishes between four different events: *(a)* External (denoted as "Ω"), *(b)* Start (i.e., action, etc. and is denoted as $\uparrow$), *(c)* Stop (i.e., action, etc. and is denoted as $\downarrow$) and *(d)* Transition (from mode to mode).

RTL uses an occurrence function, @, for capturing the event in the time domain. $@(e,i)$ represents the time of the i^{th} occurrence of the event e. Note that it is possible to define all nine state predicates with the occurrence function making use of the start and end action notations.

The visual representation of Modecharts is very similar to statecharts. In fact, the hierarchical structure in both methods is identical, i.e., modes that correspond to a lower hierarchy level are encapsulated in supermodes. Unlike Statecharts, Modecharts does not use special signs for representing history, selection or default modes. Those can certainly be applied, but since the only purpose of the method is to derive time constraints it does not need them. Other differences between the two methods concern the semantics that govern the procedures. Like Statecharts the concept *activity* is reserved for actions that take non-zero time. Unlike Statecharts, Modecharts associates each mode to *one action at most*[2]. In the clean room example that is

[2]The exact definitions of the concepts *action* and *activity* in both methods is different. Statecharts reserves the word *activity* for representing some work that consumes time and uses the word *action* to describe an event that starts or finishes the activity. Modecharts does not make any restrictions for the exact use, i.e., *action* is used for *activity* and vise-versa. As was mentioned, Modecharts associates events for starting and finishing *actions* and for mode transitions.

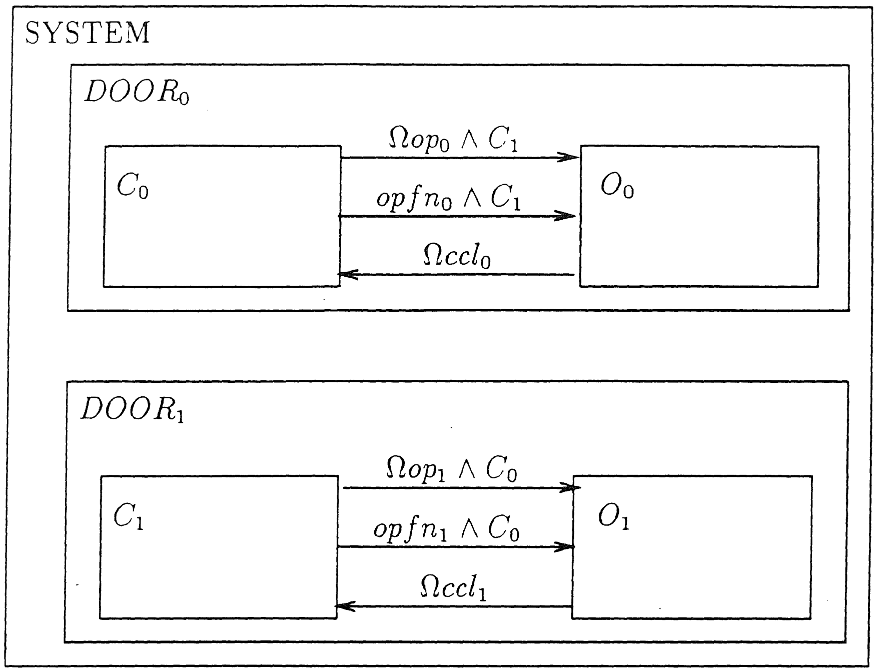


Figure 5: Representation of the clean room problem in Modecharts.

presented below, the exact moment of the mode transition from "close" to "open" (or vice versa) is decided according to some variable that tells the control system that the door is open or closed. The exact reasoning of the time constraints and other time implications is done by the RTL formalism. Both RTL and Modecharts have a syntactically and semantically accurate description [JM86, JM89], out of which only parts are to be presented hereafter using the clean room example.

The Clean Room Example: The description of the clean room problem in Modecharts language is represented in figure 5. The similarity to Statecharts representation is obvious and there is no need to explain each notation. It is worth while to stress some differences. First the symbol Ω precedes the *op* and *ccl* events, to indicate that they are external ones (respectively, as in Statecharts, touching the "open" button and complete closing). Note that event *ccl* is considered external since it causes some variable in the controller to change its value. Second, Modecharts does not use brackets to denote

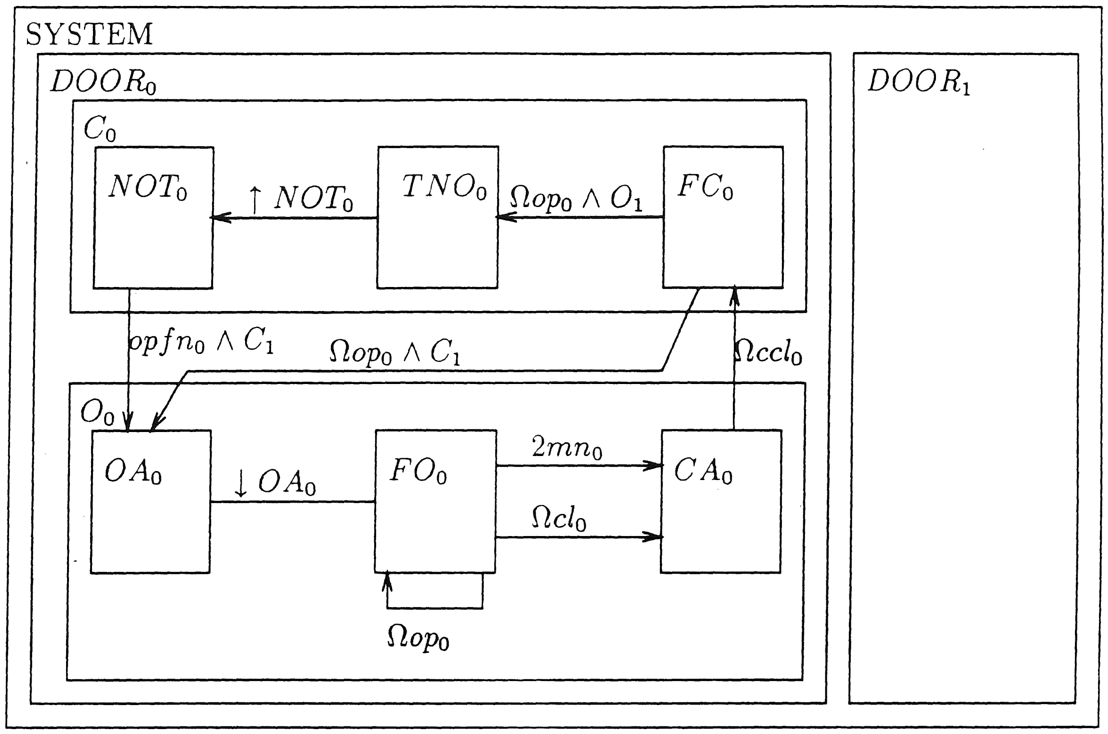


Figure 6: An expanded representation of the clean room problem.

conditions, it simply uses the $\wedge$ to express that the transition from mode C_0 to mode O_0 (respectively, from mode C_1 to mode O_1) will take place only if C_1 (respectively, C_0) exists at that moment. This difference has a deeper meaning. An event can be regarded as a condition and a transition *must* take place immediately as all conditions for its occurrence become true. In this case there are actually three conditions for the transition; $DOOR_0$ is in mode C_0, Ωop_0 occurs and $DOOR_1$ is in mode C_1 (respectively for the other door).

It is obvious from figure 5 that parts of the problem were "abstracted out" and it is necessary to expand it in order to include time details. This is done in figure 6. Each "supermode" (i.e., C_0, O_0, C_1 and O_1)[3] was expanded to include temporary modes. When the external event "Ωop_0" is sensed and

[3]The following description will be focused on door 0, but a similar description can obviously be made for door 1.

door 1 is closed (mode "C_1" is in effect) mode "OA_0", (Open Activity) is entered and remains in effect until the event "$\downarrow OA_0$" (i.e., end of Open Activity) causes the system to enter mode FO_0 (i.e., Fully Open). At this mode three events may take place; (a) the open button is pressed again (Ωop_0) causing the 2 minutes timer to restart, (b) the close button is pressed (Ωcl_0) causing the door to start closing and (c) start the closing process after two minutes $2mn_0$. The mode CA_0 (i.e., Close Activity) is entered if one or both the last events occur. This mode is in effect until the closing activity is completed (Ωccl_0) and and $DOOR_0$ is again in its initial configuration. The path that starts when an open button is pressed while the other door is open is similar to the one in Statecharts. Notice that also here we used the special Modecharts notation, $\uparrow NOT_0$, to show the start point of state NOT_0.

There are two points that are worthwhile to stress: First, the purpose of modes OA and CA is to stress the point that opening or closing a door is an activity and therefore, need to exist in a mode. And second, event $opfn_0$ that denotes the start point of the opening activity is caused by ccl_1 the end of the closing activity of the other door.

It is easy to see that Modecharts do not include any time consideration, they are merely a detailed representation of the modes. Time consideration and time analysis are conducted by RTL tools. Since RTL regards timing explicitly it is convenient to use a unified scale. We hereafter use seconds as that scale, i.e., 2 minutes=120. The first step is to represent the fact that the actual action of opening door-1 is less than 5 seconds.

$$\forall t, i\,(FC_0(t,t) \;\wedge\; C_1[t,t) \;\wedge\; t = @(\Omega op_0, i)) \rightarrow \exists j\; @(\downarrow OA_0, j) \leq t+5 \qquad (1)$$

$$\forall t, i\,(NOT_0(t,t) \;\wedge t = @(opfn_0, i)) \rightarrow \exists j\; @(\downarrow OA_0, j) \leq t+5 \qquad (2)$$

$$\forall t, i\; @(opfn_0, i) = t \rightarrow \exists t', j\; @(ccl_1, j) = t' \wedge t = t' + \epsilon \qquad (3)$$

Equations 1 and 2 express the fact that the opening activity (OA_0) may start either by pressing the open button (if $C_1 = true$) or by event $opfn_0$. The third equation (eq. 3) adds that $opfn_0$ must be initiated by a ccl_1 event. The ϵ preserves the causality relation and means that event $opfn_0$ must be preceded by ccl_1 by an infinitesimal small amount of time. The time restriction of the transition to mode NOT_0 after the open button was pressed can be written as.

$$\forall t, i\,(FC_0(t,t] \;\wedge\; O_1(t,t)) \rightarrow NOT_0[t',t') \wedge t' \leq t+2 \qquad (4)$$

Here we took full advantage of the redundant way for expressing time constraints by state predicates. $FC_0(t,t]$ means that the system exited mode FC_0 exactly on t while the other door was open $(O_1(t,t))$, entered mode NOT_0 at a time t' which is no more than $t+2$.

The closing activity is more complicated since there are two events that may lead to it, Ωcl_0 or $2mn_0$ and the second must have not preceded by Ωop_0 within the last 120 seconds. All these facts are expressed in the following equation.

$$\begin{aligned} FO_0(t,t] \quad \rightarrow \quad & (\exists j\ t \leq @(\Omega ccl_0, j) \leq t+5) \wedge \\ & [(\exists i\ @(\Omega cl_0, i) = t) \vee \\ & (\exists k\ @(2mn_0, k) = t \wedge \neg \exists l\ t - 120 \leq @(\Omega op_0, l) \leq t)] \quad (5) \end{aligned}$$

RTL contains other features that were not represented here. Such an important feature is its frame for expressing periodic behaviors. This feature is very useful since many systems include periodic behaviors. For instance, in the clean room problem one implementation possibility is by including a periodic process for checking which button was touched. In such cases, it is very common to find requirements for response time, thereby implying time constraint on the period time.

RTL supplies a powerful mathematical "toolkit" for expressing time constraints of system behavior. It enables a designer to compose the requirement accurately and analyze them with the same technique. This is very important for real time applications especially in safety related systems. RTL enables an analysis for consistency of the specifications with certain assertions. These assertions may be safety ones and therefore, it can be guaranteed to a certain extent that planned behavior does not contradict safety requirements. This is usually done by proving that a negation of the safety assertion is inconsistent with the behavior definition and is known as a resolution proof by refutation [LP81]. In the clean room problem a safety assertion is: $\forall t\ \overline{(O_0(t,t) \wedge O_1(t,t))}$[4], which means that there does not exist a situation in which both doors ore opened. But this situation may occur only if a door exited mode C and entered mode O while the other door is in mode O (provided that in the initial mode both doors are closed). Mode change is expressed in RTL as $M - M'$ and thus our safety assertion becomes:

$$O_1(t,t) \wedge \neg \exists j @((C_0 - O_0), j) = t \qquad (6)$$

[4]We shall use the signs $\neg x$ and $\overline{x}$ to denote the negation of x.

We again concentrate on the first door behavior. The negation of this assertion is obtained by removing the negation sign ($\neg$) from both above equations. Now, the transition $C_1 - O_1$ can occur only by events $op_0 \wedge C_1$ or $opfn_0$. Moreover, both events can not occur simultaneously, i.e., the or between them is actually an exclusive or ($\oplus$). This is written as:

$$\forall j @((C_0 - O_0), j) = t \rightarrow \exists l, k\ @(\Omega op_0, l) = t \wedge C_1(t,t) \oplus @(opfn_0, k) = t \quad (7)$$

Using the equivalences $x \rightarrow y \equiv \overline{x} \vee y$ and $x \oplus y \equiv (x \wedge \overline{y}) \vee (\overline{x} \wedge y)$, equations 6 and 7 can be rewritten in clauses of a conjunctive normal form after skolemizing.

$$\begin{array}{l}
O_0(t,t) \\
@((C_0 - O_0), j) = t \\
\overline{@((C_0 - O_0), j) = t} \\
\overline{@((C_0 - O_0), j) = t} \vee @(\Omega op_0, I) = t \vee @(opfn_0, K) = t \\
\overline{@((C_0 - O_0), j) = t} \vee C_1(t,t) \vee @(opfn_0, K) = t \\
\overline{@((C_0 - O_0), j) = t} \vee \overline{@(\Omega op_0, I) = t} \vee \overline{C_1(t,t)} \vee \overline{@(opfn_0, K) = t}
\end{array}$$

where K and I are skolem constants. The details of the resolution process are not shown here, but it is easy to see that the above equations are inconsistent, since for each clause its negation can be found.

Evaluation The connection between the visual representation and RTL language is "loosely coupled", and not natural. Modecharts gives no representation of the timing constraints, it merely helps in capturing the system structure. From this aspect any other structural representation could do as well. The attempt to combine the timing and structural representation caused many limitations on the ability to comprehend the structural behavior of the system. For example, the fact that a mode can be associated with at most one action forces partitioning of a system into parts that do not correspond to actual modules in the system. In order to represent the fact that mode transition can occur only after an action is completed, the mode is partitioned into three submodes; one represents the initial state, another represents the beginning of the action and yet another mode for representing the state before the action is completed. This partition is not natural and may cause difficulties in understanding the exact structural behavior of the system.

Another disadvantage of Modecharts and RTL is the inability to distinguish between deterministic and nondeterministic processes. Deterministic processes are those whose timing constraints are known to the external environment that uses them. In these cases the designer may make use of that knowledge and implement processes that are time scheduled, whereas in nondeterministic processes the order of events are unknown and the designer must insert check points in the external environment. These check points are time consumers and have to be considered. If, for instance, the clean room is implemented in a way that requires two consecutive open instructions to be separated with a time period larger than the time consumed for door movement, the design may save a check point of door "fully closed".

The applicability of RTL is also problematic. It is clear from the above description that its incorporation in big and complex systems is going to be very difficult, perhaps impossible. This fact was already observed by the authors [JM87]:

> ... *However, they* (i.e., RTL) *may be impractical for use in verifying an assertion against the full specification of a large and complex real-time system (see* [JM87] *page 963).*

One way of overcoming such difficulties is by isolating critical sections of the system from the rest of it and conducting RTL analysis only to these parts. This is not unusual. There are many methods that apply such "divide and concur" strategies in complex systems.

Modecharts and RTL serve as a front end in a graphical specification tool called "SARTOR" (Software Automation for Real-Time OpeRation) which is developed at the University of Texas at Austin [Mok85]. SARTOR includes a set of tools for analyzing safety specifications and rapid prototyping.

2.1.3 ESM/RTTL

General: Extended State Machine (ESM) was introduced by J. S. Ostroff and W. M. Wonham [OW87, Ostroff88, Ostroff89] as a framework for real time discrete processes. The idea behind the method can be understood from its name. ESM is a finite state machine that uses a modular representation in which a system is partitioned into a "plant" and "controller". In the first step the plant is represented and the controller is then designed[5]. Both the plant and the controller are driven by discrete events. A system may consist of many ESMs that are interacting with each other through communication channels. The proof system is an extension of Temporal Logic (TL)[6] [MP83, Pnueli86] called Real Time Temporal Logic (RTTL). Systems are represented in this logic and by using inference rules, general and safety properties can be deduced or proven.

Formally, a basic ESM is a 5-tuple that consists of sets of *activity* ($\mathcal{X}$) and *data* ($\mathcal{Y}$) variables, *communication channels* ($\mathcal{C}$), *event labels* ($\mathcal{L}$) and *basic actions* ($\mathcal{A}$). Each ESM is identified by a name, its activity variable, which may be assigned with values that represent the ESM states. The data variables has the usual meaning. An *event* is an *operation*, conditioned by a *guard* (a boolean expression that has to be evaluated to *true* in order for the transition to take place), that causes the ESM to move from source to destination *activities* (a_s and a_d, respectively). This can be represented as:

$$\begin{tikzcd} a_s \arrow[rr, "guard \longrightarrow operation"] & & a_d \end{tikzcd}$$

The combination of the source activity and the guard can be referred to as the enabling condition of the transition. The combination of the *activity* variable and the ESM *events* is a *basic action*. An *operation* is either an *assignment* of a value to a *data* variable (denoted as $\alpha[y_1 : a_1, y_2 : a_2, \ldots y_n : a_n]$ where α is the event label and the a's are values assigned to the corresponding variables, y's), *send* (denoted $c!a$ where a is data or event label that is sent via communication channel c) and *receive* (denoted $c?a$ where c and a have similar meanings as for the send event). Two or more ESMs are interacting when all share any of their event labels. For example, a controlled two states switch (*ON* and *OFF*) may be represented by the 5-tuple: $(\{SWITCH\}, \emptyset, \{c\}, \{on, off\}, \mathcal{A}\})$, where:

$$\begin{aligned} \mathcal{A} \;=\; \{\; &\{(SWITCH, (OFF, true, c?on, ON))\}, \\ &\qquad\qquad \{(SWITCH, (ON, true, c?off, OFF))\}\; \}. \end{aligned}$$

[5]The word "designed" was chosen, since, as will be seen the method as described in the references is more suitable for the design phase. It can be used for behavior description with limited flexibility.

[6]The creators of ESM/RTTL use a notation that was introduced by Manna and Pnueli and so do we.

This means that ESM *SWITCH* may be found in one of the activities *ON* or *OFF*. In ESM terminology the activity variable *SWITCH* is of type $\{ON, OFF\}$. The transitions between the activities are always allowed (the guard is *true*) (later, we shall omit the guard if it is always *true*), given that it received a corresponding instruction via the communication channel ($c?on$ or $c?off$). The *on* and *off* are shared events labels.

The method further defines ways for parallel composition (denoted $||$) of action sets and ESMs. This definition is used for combining basic actions and ESMs into higher level, thereby taking care of abstracting out inner details. Thus, for example, common communication channels of basic ESMs, will not be seen after the ESMs are composed.

Transitions between states or activities are of great importance as they define the exact behavior of the system. Transitions may be labeled by the event names that causes them to occur or by symbols (say τ_3) when no name is assigned. The sign **n** means "the next state" and is used for identifying a transition, for example $\mathbf{n} = on$. Every transition must be associated with time restriction, i.e., the transition can occur between given lower and upper time bounds. If the upper time bound is infinite (not limited) the event is considered spontaneous (since it may always occur) and forced, otherwise. Time restrictions are introduced by defining a clock ESM, that has always a true guard, thus is always enabled to "tick". The parallel composition of the clock ESM and the component ESM share the "tick" events. A sequence of ESM transitions (within their time boundaries) is called "trajectory", and a sequence of allowed (designed) transitions is called the "legal trajectories" of the ESM.

As was mentioned, the proof scheme is based on temporal logic [MP83, Pnueli86]. TL uses temporal operators in conjunction with first order logic formulas that describe the system states (or activities, in ESM terminology). There are two basic temporal operators that can be used for deriving others. The basic operators are $\bigcirc$ (next) and $\mathcal{U}$ (until). If w is any state formula, the following four operators can be derived[7]:

[7]Manna-Pnueli original temporal logic is using four basic operators and many others are derived. These operators extend the expressive power of temporal logic and can be divided into two groups, past and future operators. The first group deals with state formulas that existed in the past and the other with future states. These details were not included here since they are less used.

- $\Diamond w = true \mathcal{U} w$ which means that eventually w will be true.
- $\Box w = (\neg(\Diamond(\neg w)))$ which means that henceforth, w will always be true.
- $w_1 \mathcal{P} w_2 = (\neg((\neg w_1)\mathcal{U} w_2))$ which means that if w_2 occurs then w_1 must precede.
- $w_1 U w_2 = ((\Box w_1) \vee (w_1 \mathcal{U} w_2))$ which means that henceforth, w_1 will be true unless w_2 will become true.

Notice that safety properties will be represented as $\Theta \rightarrow \Box safe$ where *safe* is some invariant property that must always exist and the symbol Θ is reserved to indicate the initial state.

Based on the above definitions, TL applies a set of axioms and inference rules for deducing system properties. RTTL uses the formal definition of TL transitions (as described above), which enables all TL axioms, theorems and inference rules to be valid in RTTL. The extension of RTTL is accomplished by adding a set of axioms and rules that are applicable to the ESM domain. It is out of this paper's scope to describe or even list all TL and RTTL axioms and inference rules, those can be be found in [Pnueli86, Ostroff89], some of those will be explained in the next section when the clean room example is described.

The heuristic of the proof scheme can now be described. The first step is to list all system transitions in a table. The table has to include the transition name, enabling condition, transformation (or destination activity) and the time boundaries. Proof diagrams, which are actually partial or complete reachability graphs, are drawn. These represent legal trajectories of the ESM. System properties are represented in RTTL so that the inference rules in conjunction with the proof diagrams can be used for proving correctness or incorrectness. The heuristics provide some basic approaches for expressions that are frequently used. In the case of safety properties, that are of the type $\Box safe$, reachability graphs are built and the following derived rule is used:

$$\textbf{if} \quad (\Theta \rightarrow \psi \text{ and } \{\psi\}\mathcal{T}\{\psi\}) \quad \textbf{then } \Box\psi$$

where ψ is any property and $\mathcal{T}$ is the set of all transitions. Intuitively this rule means that if a property is implied from the initial state and exists under all system transitions then it always exist (i.e., nothing can be changed within a state!). The problem of such proof heuristics is quite obvious. Usually, for big systems, reachability graphs are very complicated and difficult to build.

In an attempt to overcome this problem, the method supplies some efficient algorithms for constructing reachability graphs. The method even "pushes" towards an application of constraint logic programming languages such as Prolog or CLP($\Re$). Two algorithms are recommended: one for transitions whose lower time limit is always zero and another algorithm that does not contain this constraint. Even with these methods the reachability graph is hard to handle.

Another technique that is useful for safety properties can be used. This technique is based on a backward approach. In the backward approach unsafe states are identified and traced backward to find critical points, where choice between unsafe and safe paths is made. Regardless to the problem whether the unsafe choice can actually take place, an "interlock" that prevent it from being chosen is installed. This approach is the same as the one used in [LS87] in Petri-nets.

The Clean Room Example: The first step is to build the "plant" of two doors ESM. In [Ostroff89] it is further recommended to write "control code" and "translate" it into ESM form using some heuristic method (in [Ostroff89] Conic language [KMS84] is used). Such approach looks more like a design and not as a behavior description. As will be discussed in the evaluation this attitude limits the flexibility of the behavior description. We, therefore, proceed in a different way that is more behavior descriptive. This can be seen in figure 7 where $DOOR_i$ ESM ($i = i \ mod \ 2$) is represented. In this representation the two doors are connected via m_i channel which is used for transferring information about the state of the i^{th} door. In order to simplify the figure the transmitting source was omitted. In the complete description transitions that are initiated by $m_i!DOOR_i$ (m_i is the communication channel and $DOOR_i$ represents the state of the corresponding door) should be added to all states. The figure shows only the "receive" part of these events ($m_{i+1}?y$).

In the initial state (Θ) each door is closed and no button is pressed (state $CLOSE_i^0$). When an open button is pressed (event *open_but$_i$*) the ESM enters state $CLOSE_i^1$ where it waits for input information about the other door's state (event $m_{i+1}?y$). When this happens, the door enters state

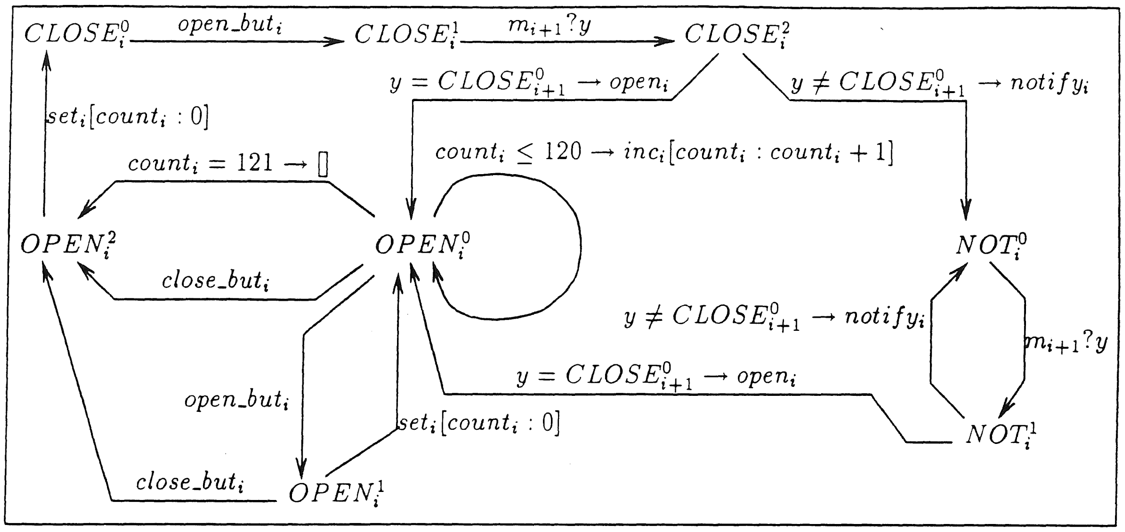

$CLOSE_i^0$
$open\_but_i$
$CLOSE_i^1$
$m_{i+1}?y$
$CLOSE_i^2$
$y = CLOSE_{i+1}^0 \rightarrow open_i$
$y \neq CLOSE_{i+1}^0 \rightarrow notify_i$
$set_i[count_i : 0]$
$count_i \leq 120 \rightarrow inc_i[count_i : count_i + 1]$
$count_i = 121 \rightarrow []$
$OPEN_i^2$
$OPEN_i^0$
NOT_i^0
$close\_but_i$
$y \neq CLOSE_{i+1}^0 \rightarrow notify_i$
$m_{i+1}?y$
$y = CLOSE_{i+1}^0 \rightarrow open_i$
$open\_but_i$
$set_i[count_i : 0]$
NOT_i^1
$close\_but_i$
$OPEN_i^1$

Figure 7: ESM $DOOR_i$.

$CLOSE_i^2$, where the content of variable y is checked. This variable holds the current state of the other door and is used to guard the transition. If the other door is closed $y = CLOSE_{i+1}^0$ the door opens (event *open$_i$* that causes the ESM to enter state $OPEN_i^0$). If the other door is not closed, $y \neq CLOSE_{i+1}$, the user is notified (event $notify_i$ to states NOT_i). While notifying, the door keeps track on the other's door state. This is done in a loop that is formed between states NOT_i^0 and NOT_i^1. The receive event, $m_{i+1}?y$, triggers the "forward" transition and the guarded event $notify_i$ triggers the reverse transition. Notice that states $CLOSE_i^2$ and NOT_i^1 are similar, i.e., both are entered and exited with the same transition.

The open state is a combination of three sub-states. Event inc_i represents the 120 seconds delay after which the door has to be closed automatically. As can be seen this is an assignment event $[count_i : count_i + 1]$ that is guarded by the condition $count_i \leq 120$. Obviously, this transition has to be initiated every second[8]. Event *open_but$_i$* represents an open button that is pressed while the door is open. This has to reset the counter and is represented by the

[8]This transition may be represented in many other similar ways. For example, $count_i \neq 60 \rightarrow inc_i[count_i : count_i + 1]$ every two seconds is equally correct or many others, as long as the event $count_i = 121 \rightarrow []$ is changed accordingly.

Table 1: Transitions in the clean room example

Transition	Label	Enabling condition	Transformation	Lower time	Upper time
τ_i^0	*open_but$_i$*	$DOOR_i = CLOSE_i^0$	$[DOOR_i : CLOSE_i^1]$	l^0	u^0
τ_i^1	m_{i+1}com	$DOOR_i = CLOSE_i^1$	$[DOOR_i : CLOSE_i^2]$	l^1	u^1
τ_i^2	$notify_i$	$DOOR_i = CLOSE_i^2 \wedge$ $y \neq CLOSE_{i+1}^0$	$[DOOR_i : NOT_i^0]$	l^2	u^2
τ_i^3	m_{i+1}com	$DOOR_i = NOT_i^0$	$[DOOR_i : NOT_i^1]$	l^3	u^3
τ_i^4	$notify_i$	$DOOR_i = NOT_i^1 \wedge$ $y \neq CLOSE_{i+1}^0$	$[DOOR_i : NOT_i^0]$	l^4	u^4
τ_i^5	$open_i$	$DOOR_i = NOT_i^1 \wedge$ $y = CLOSE_{i+1}^0$	$[DOOR_i : OPEN_i^0]$	l^5	u^5
τ_i^6	$open_i$	$DOOR_i = CLOSE_i^2 \wedge$ $y = CLOSE_{i+1}^0$	$[DOOR_i : OPEN_i^0]$	l^6	u^6
τ_i^7	inc_i	$DOOR_i = OPEN_i^0 \wedge$ $count_i \leq 120$	$[DOOR_i : OPEN_i^0,$ $count_i : count_i + 1]$	l^7	u^7
τ_i^8	*open_but$_i$*	$DOOR_i = OPEN_i^0$	$[DOOR_i : OPEN_i^1]$	l^8	u^8
τ_i^9	set_i	$DOOR_i = OPEN_i^1$	$[DOOR_i : OPEN_i^0,$ $count_i : 0]$	l^9	u^9
τ_i^{10}	*close_but$_i$*	$DOOR_i = OPEN_i^0$	$[DOOR_i : OPEN_i^2]$	l^{10}	u^{10}
τ_i^{11}		$DOOR_i = OPEN_i^0 \wedge$ $count_i = 121$	$[DOOR_i : OPEN_i^2]$	l^{11}	u^{11}
τ_i^{12}	*close_but$_i$*	$DOOR_i = OPEN_i^1$	$[DOOR_i : OPEN_i^2]$	l^{12}	u^{12}
τ_i^{13}	set_i	$DOOR_i = OPEN_i^2$	$[DOOR_i : CLOSE_i^0,$ $count_i : 0]$	l^{13}	u^{13}
τ_i^{13}	$m_i$com	any-state	same-state	l^{13}	u^{13}

assignment event $set_i[count_i : 0]$. State $OPEN_i^2$ is a temporary one that is reached before the door is completely closed. Either a *close_but$_i$* event or the elapsing of 120 seconds (guard $count_i = 121$) causes the transition. (Notice that the symbol [] is used to denote transitions that depend only on the guard and are not involved in any assignment or communication event.) Event set_i completes the circle and the door is closed, while the counter is ready for another use. As was mentioned, there is one more output transition that is not depicted (transition $m_i$com). This transition is initiated by output events and its source and destination activities are the same. It continuously reports its current activity through m_i channel.

Table 1 summarizes the transitions of each door. The time limits are not given explicitly, they will be shortly determined when the requirement are represented in RTTL notation. Also, notice that event labels may be overloaded (i.e., one event label can be used for more than one event). No confusion exists, since the source state is part of the enabling condition of the transition. This overloading enables the grouping of transitions with similar meaning. In the following discussion, whenever an overloaded label is used, all its transitions are referenced.

As is obvious, the initial state can be represented as:

$$\begin{aligned}\Theta &= (\,(\mathbf{n}=\text{initial}) \wedge (DOOR_0 = CLOSE_0) \wedge \\ &\quad (DOOR_1 = CLOSE_1) \wedge (count_0 = 0) \wedge (count_1 = 0)\,)\end{aligned}$$

Some other requirement are:

Rsafe: $\Box[\neg((DOOR_0 = OPEN_0) \wedge (DOOR_1 = OPEN_1))]$.
Henceforth the two doors should not be both opened.

R0: $(\mathbf{n} = open_i) \rightarrow ((\mathbf{n} = m_{i+1}\text{com}) \, \mathcal{P} \, (\mathbf{n} = open_i))$.
Every open event must be preceded with an input event.

R1: $(\mathbf{n} = open\_but_i \wedge DOOR_{i+1} = CLOSE^0_{i+1} \wedge DOOR_i = CLOSE^0_i \wedge t = T) \rightarrow \Diamond(DOOR_i = OPEN^0_i \wedge t \leq T+5)$ seconds.
If an open button is pressed the corresponding door should be opened within 5 seconds, given that the other door is closed. This imposes $u^0 + u^1 + u^6 \leq 5$ seconds.

R2: $(\mathbf{n} = open\_but_i \wedge DOOR_{i+1} \neq CLOSE^0_{i+1} \wedge DOOR_i = CLOSE_i \wedge t = T) \rightarrow \Diamond(DOOR_i = NOT^0_i \wedge t \leq T+2)$.
If an open button is pressed and the other door is not closed, the user has to be notified within 2 seconds. This imposes $u^0 + u^1 + u^2 \leq 2$ seconds.

R3: $((\mathbf{n} = close\_but_i \vee \mathbf{n} = \tau^{11}_i) \wedge DOOR_i = OPEN_i \wedge t = T) \rightarrow \Diamond(DOOR_i = CLOSE^0_i \wedge t \leq T+5)$.
As R1 for the closing process. This imposes that $u^{10} + u^{13} \leq 5$ seconds, or $u^{11} + u^{13} \leq 5$ seconds, or $u^{12} + u^{13} \leq 5$ seconds.

R4: $(DOOR_i = NOT_i \wedge \mathrm{n} \ = \tau_{i+1}^{13} \wedge t = T) \rightarrow \Diamond(DOOR_i = OPEN_i^0 \wedge t \leq T + 5)$.
If a user is notified and the other door closes, the first door should be opened within 5 seconds. This imposes $u^4 + u^3 + u^5 \leq 5$ seconds. This takes into account the most critical situation in which τ_{i+1}^{13} occurs simultaneously with τ_i^4.

R5: $(\mathrm{n} \ = inc_i \wedge t = T)\mathcal{P}(\mathrm{n} \ = inc_i \wedge t = T') \rightarrow (T' - T = 1 \text{ second})$.
Exactly one second has to elapse between two successive inc_i transitions. This imposes $l^7 = u^7 = 1$ second.

Now, it is required to show that the design fulfills the above requirements. We will not show the whole proof but rather concentrate on the first two requirements **Rsafe** and **R0** as both are important to safety. Also, these will be done very briefly. The proof of **R0** is obvious, $m_{i+1}\mathrm{com} \ = \{\tau_i^1, \tau_i^3\}$, and $(m_{i+1}\mathrm{com}\ ) \rightarrow \bigcirc(DOOR_i = NOT_i^1 \vee DOOR_i = CLOSE_i^2)$ and $(DOOR_i = NOT_i^2 \vee DOOR_i = CLOSE_i^2) \rightarrow \bigcirc(\mathrm{n} \ = notify_i \vee \mathrm{n} \ = open_i)$.

As was mentioned, safety requirements such as **Rsafe** are proven by showing that an unsafe situation is never reached in a reachability graph. This may turn out to be very difficult, especially in concurrent systems (as ours) where time constraints of one process are independent of the others[9]. The transitions of one door may happen when the other door is in any state. We, therefore, are going to discuss some of the conclusions that may be reached even without a rigorous graph representation.

Figure 8 is a partial reachability graph in which only part of the first transitions and nodes are shown. The sign $\leftarrow$ means that both transitions to its left occurred simultaneously. Notice that state ψ_2 can be reached either when *open_but*$_0$ and *open_but*$_1$ are initiated simultaneously or consequently before any communication event takes place. As we proceed with the process and reach any of the states ψ_8, ψ_9 or ψ_{10} the value of variable y will not be $CLOSE^0$ for both doors. This will cause both doors to take the $notify$ transitions and eventually reach a deadlock[10] (dotted arrow), since henceforth $y \neq CLOSE^0$ for both doors. Now, although the deadlock state

[9]If a "design" of plant and controller were used, the task might have become easier. The controller synchronizes the processes so that simultaneous events can not occur.

[10]This state may be called divergence instead of deadlock, since the loop NOT^0 NOT^1 is always active.

STATE	DESCRIPTION
ψ_0	$DOOR_0 = CLOSE_0^1 \wedge DOOR_1 = CLOSE_1^0$
ψ_1	$DOOR_0 = CLOSE_0^1 \wedge DOOR_1 = CLOSE_1^1$
ψ_2	$DOOR_0 = CLOSE_0^1 \wedge DOOR_1 = CLOSE_1^1$
ψ_3	$DOOR_0 = CLOSE_0^2 \wedge DOOR_1 = CLOSE_1^0$
ψ_4	$DOOR_0 = CLOSE_0^0 \wedge DOOR_1 = CLOSE_1^2$
ψ_5	$DOOR_0 = OPEN_0^0 \wedge DOOR_1 = CLOSE_1^0$
ψ_6	$DOOR_0 = CLOSE_0^2 \wedge DOOR_1 = CLOSE_1^1$
ψ_7	$DOOR_0 = OPEN_0^0 \wedge DOOR_1 = CLOSE_1^1$
ψ_8	$DOOR_0 = CLOSE_0^2 \wedge DOOR_1 = CLOSE_1^1$
ψ_9	$DOOR_0 = CLOSE_0^1 \wedge DOOR_1 = CLOSE_1^2$
ψ_{10}	$DOOR_0 = CLOSE_0^2 \wedge DOOR_1 = CLOSE_1^2$

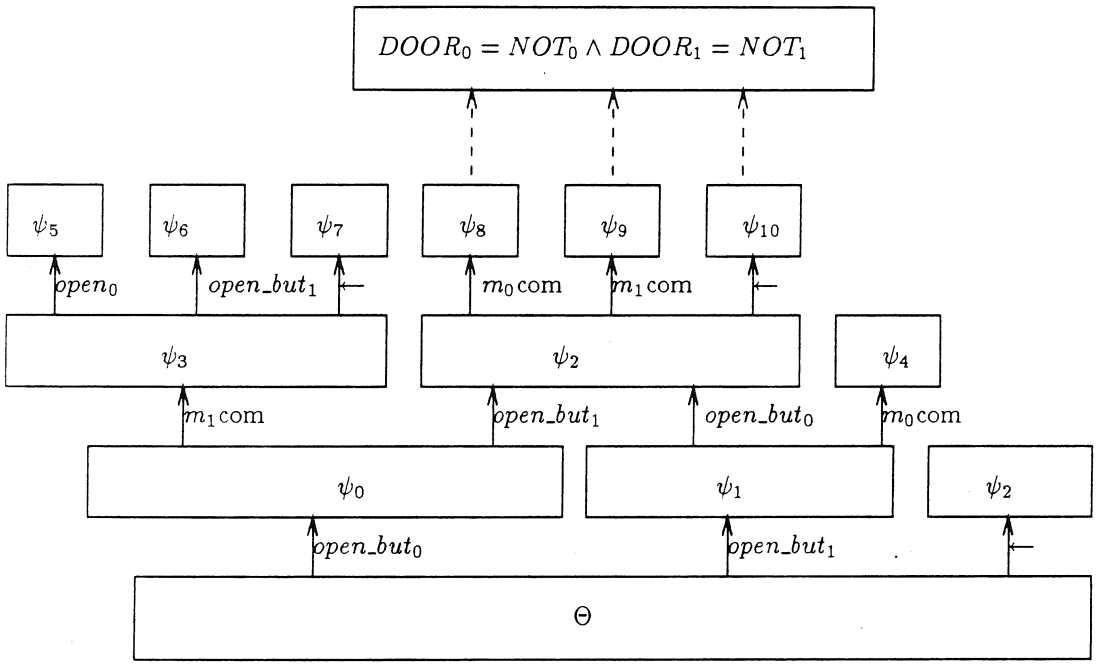


Figure 8: A partial reachability graph.

is not inconsistent with requirement **Rsafe** (after all the two doors are kept closed), it is still a situation that has to be avoided. Also, we are unable to justify such a situation and ignore the simultaneous *open_but* transitions by saying that there is no specific requirement for that case. State ψ_2 may be reached in a sequential process. But, what is even more crucial, it is hard to handle simultaneous events in the method. The point is that information is transferred in a point-to-point fashion and no broadcasting is allowed. This means that if we were given a specific requirement about priority for the simultaneous *open_but* events, the information about the "button state" ("pressed", "not-pressed") had to be transferred in a communication event that is not $m_i$com or m_{i+1}com .

One way to overcome the sequential deadlock is to impose time constraints on the transitions. Notice that the sequential process that gives rise to the deadlock requires that both *open_but* transitions have to be completed before any input event may take place. This can be prevented if the upper time limit of a communication transition is less then the lower time limit of an *open_but* transition, or $u^1 < l^0$.

In order to conduct timing analysis, some assumptions must be made. For example, there is no reason to have two m_{i+1}com transitions with different time constraints. Therefore, we may assume that $l^3 = l^1$ and $u^3 = u^1$. The same assumption may be made for the two *open_but*$_i$, *open*$_i$ and *notify*$_i$ transitions. Furthermore, it may be assumed that *open_but*$_i$ and *close_but*$_i$ transitions are subjected to similar time constraints. Therefore, $l^0 = l^8 = l^{10} = l^{12}$, $u^0 = u^8 = u^{10} = u^{12}$, $l^2 = l^4$, $u^2 = u^4$, $l^5 = l^6$ and $u^5 = u^6$.

Other constraints can be concluded by using the requirement $u^1 < l^0$ in conjunction with the above assumptions and the constraints which were derived from the requirements **R1-R4**. For example, u^1 may be omitted from the equations in **R1** and **R2**, and u^3 can be omitted from the equation in **R4**. This will lead to interesting conclusions.

Evaluation: ESM/RTTL provides a complete scheme for designing and analyzing systems. It includes a homogeneous structure that enables the application of the description and the analysis in the same terms. The definitions of events and actions is done according to the base TL theory and thus remain unchanged in the analysis and as subsystems are joined to form the entire system. TL and the added ESM tier supplies the designer and analyzer with a rich set of rules that is of great help if used properly. From a safety point of view, this is of great importance. Safety properties are usually defined in a unique way by the henceforth ($\Box$) operator for which a proof scheme is outlined.

Although the method description seems more suitable for the design stage, it may be adopted to the early requirement stage. This enables a smooth transition from the behavior description to the design and even implementation phases. The theory, upon which the proof scheme is grounded, is kept unchanged and valid. Moreover, since the design and the implementation add constraints to the system, the analysis may become more efficient.

Although, ESM/RTTL is an event driven model, conditions and time constraints can be expressed and analyzed. As such it is suitable for modeling reactive system. The method supplies tools for hierarchy buildup that is coherent with the basic terms. For example, actions remain unchanged as basic actions are becoming interactions, or when actions and ESMs are combined in a parallel composition. This makes the method a powerful tool in future applications.

The drawbacks of the method can be deduced from the clean room description and the discussion that preceded. The hierarchy and abstraction are not done easily, they are rather oriented towards component buildup and not on behavior buildup. This means that subsystems are actually real components and it is hard (sometimes impossible) to construct a hierarchical structure of behavior. Such properties are more suited for the design or implementation stages.

In addition the method is organized in a way that becomes most efficient when a system is partitioned to plant and controller. This is really a design decision. After all, a controller has to accomplish what is required from the plant and it is not a stand alone subsystem. By doing so in the early stages of the requirements, the developer is almost "forced" to build a centralized system. It forces the developer to partition the system and assign logic tasks to actual components. This may strongly limit future flexibility. This may become more clear if someone tries to solve the clean room example in a "plant" and "controller" fashion. For example, the buttons must be represented as ESMs with communication channels to the controller. This, does not allow two simultaneous *open_but* events to occur, since the controller can be designed to handle one request at a time, at most. Also, the controller must be connected to each door by two communication channels. One for controlling and another for data.

Although the method is event driven, there are no facilities to allow handling simultaneous multiple events. This is partly due to the controller plant approach and partly to the point-to-point communication scheme. In the clean room example, two data types "door state" and "button state" had to be dealt with as two distinguished events.

The ESM/RTTL proof scheme is a very sophisticated task that requires mastery of the various rules. This task is very dependent on the analyzer skills and experience. The set of TL and RTTL rules is complicated by itself and its application may becomes unsolvable for large scale systems. A proof could exist, but could not be attained. Human factors may become even worse since TL and RTTL are new concepts that most analyzers are unfamiliar with. This means that almost any application of the method must be preceded by a significant learning period.

Proofs of safety properties are even more problematic. Building reachability graphs is a very tedious and error prone task, especially during the first stages of the project, when there are almost no limitations. Concurrent systems are even worse, since time can not be ignored (which limits the efficiency of the algorithms for building reachability graphs). This could be seen in the clean room example. A simple calculation could show that the number of "state-maps"[11] in the clean room example is on the order of 3^8! In this case more efficient tools, other then reachability graphs, are needed.

The usefulness of the method can hardly be evaluated since there are no records of "real-life" use. The only records are small scale problems similar to the clean room example.

2.1.4 Timed Petri-nets

General: Petri-nets were used to model systems and reason about properties such as deadlocks, reachability and safety. Petri-nets enable a system approach that incorporates one language for hardware, software and human behavior [AVD76, Peterson81] and its dynamic properties make it appropriate for real-time reactive systems. The lack of timing information in the

[11]The term "state-maps" is used in the method to describe similar states that may be reached in different event order or time frames. For example, the ψ_2 states in figure 8 are one state map.

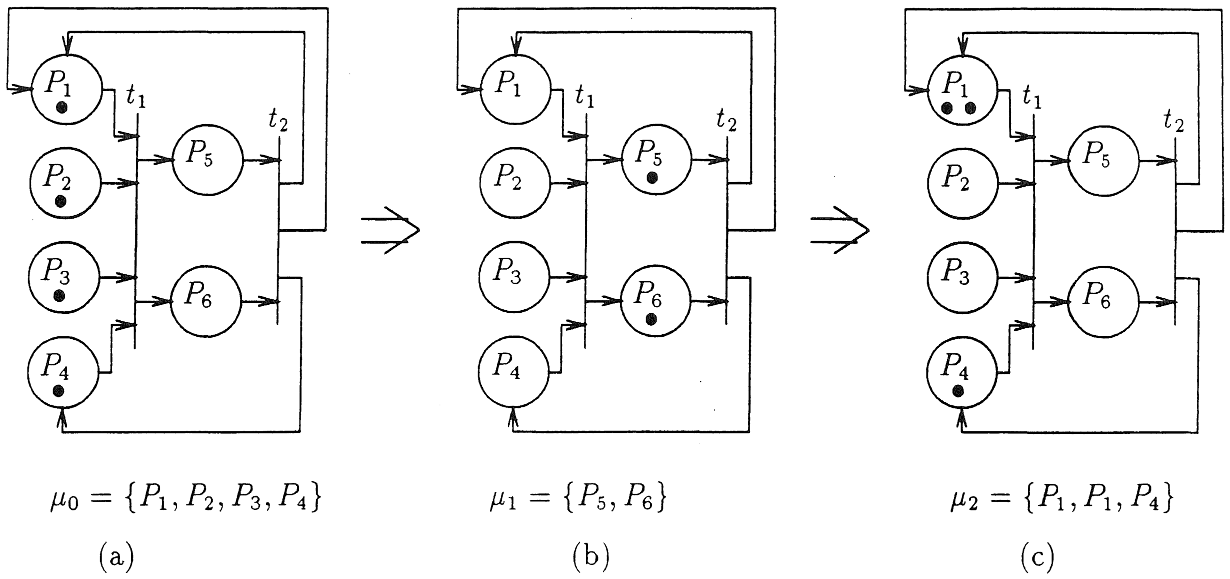
original Petri-nets was a major disadvantage of the method that was overcome only lately by introducing time in various ways[12]. In this paper we adopt the Merlin [Merlin74] and Merlin and Farber [MF76] scheme. This scheme was shown to be efficient for safety analysis by Leveson and Stolzy [LS87].

Regular Petri-nets are composed of sets of *places* (P) and *transitions* (T) that are connected by *input* (I) and *output* (O) functions. The dynamics of the net is represented by movements of tokens according to specified rules. The bag of places that contain tokens is called the *marking* (μ) of the net. It is very convenient and useful to represent a Petri-net in a graph structure in which places are represented by circles "◯" and transitions by bars "|". The input and output functions are represented by arrows from places to transitions or form transitions to places, respectively. The tokens are represented by black dots that are contained in places. A transition is *enabled* if and only if each of its input places contains at least as many tokens as there exists arrows from that place to the transition. An enabled transition *may fire* and remove all enabling tokens from input places, and deposit a token in each output places. Figure 9 shows an execution of a Petri-net. Part (d) of the figure shows the input and output functions. Part (a) shows the initial marking (μ_0). Transition t_1 is enabled since there are enough tokens in each of its input places. In this state, transition t_2 is not enabled. After transition t_1 fires, four tokens are removed and two new ones are assigned to places P_5 and P_6, which enables transition t_2 (figure 9 (b)). After transition t_2 fires the net is in a deadlock state (C) since places P_2 and P_3 do not contain tokens.

One significant conclusion that can be drawn from the above description is that a system structure is totally represented by a Petri-net. The dynamic execution of the system is represented by the various markings that are encountered. This means that the system states are represented by the markings.

As was mentioned, "original" Petri-nets do not include time restrictions. We add time restrictions by assigning two numbers to each transition. These numbers define a time interval in which the transition must fire after it was enabled ("enabled" in the sense of regular Petri-nets). In the graph structure

[12]Ghezzi et al. [GMMP89] have shown that all timed Petri-nets can be unified into a general single net, called ER nets. However, also these researchers agree that in practice it might be more convenient to use a different notation, tailored to the specific use. We are following this recommendation.



$I(t_1) = \{P_1, P_2, P_3, P_4\}$ $\qquad$ $O(t_1) = \{P_5, P_6\}$

$I(t_2) = \{P_5, P_6\}$ $\qquad$ $O(t_2) = \{P_1, P_1, P_4\}$

(d)

Figure 9: Execution of Petri-nets

they are represented in brackets at the side of the transition. A transition *must* fire during this interval and the firing does not consume time (example: $t_1(2..5)$ means that transition t_1 *must* fire in the time interval that starts 2 time units and ends 5 time units after t_1 was enabled). A transition may not fire during the assigned interval if and only if it was disabled before firing and during that interval. We will use two conventions: *(a)* no numbers are assigned if the firing must occur immediately after the transition is enabled, and *(b)* one number (n) denotes that the transition must fire exactly n time units after the transition was enabled.

The Clean Room Example: Now we are ready to analyze the clean-room problem with Petri-nets. Figure 10 describes the problem in a Petri-net graph

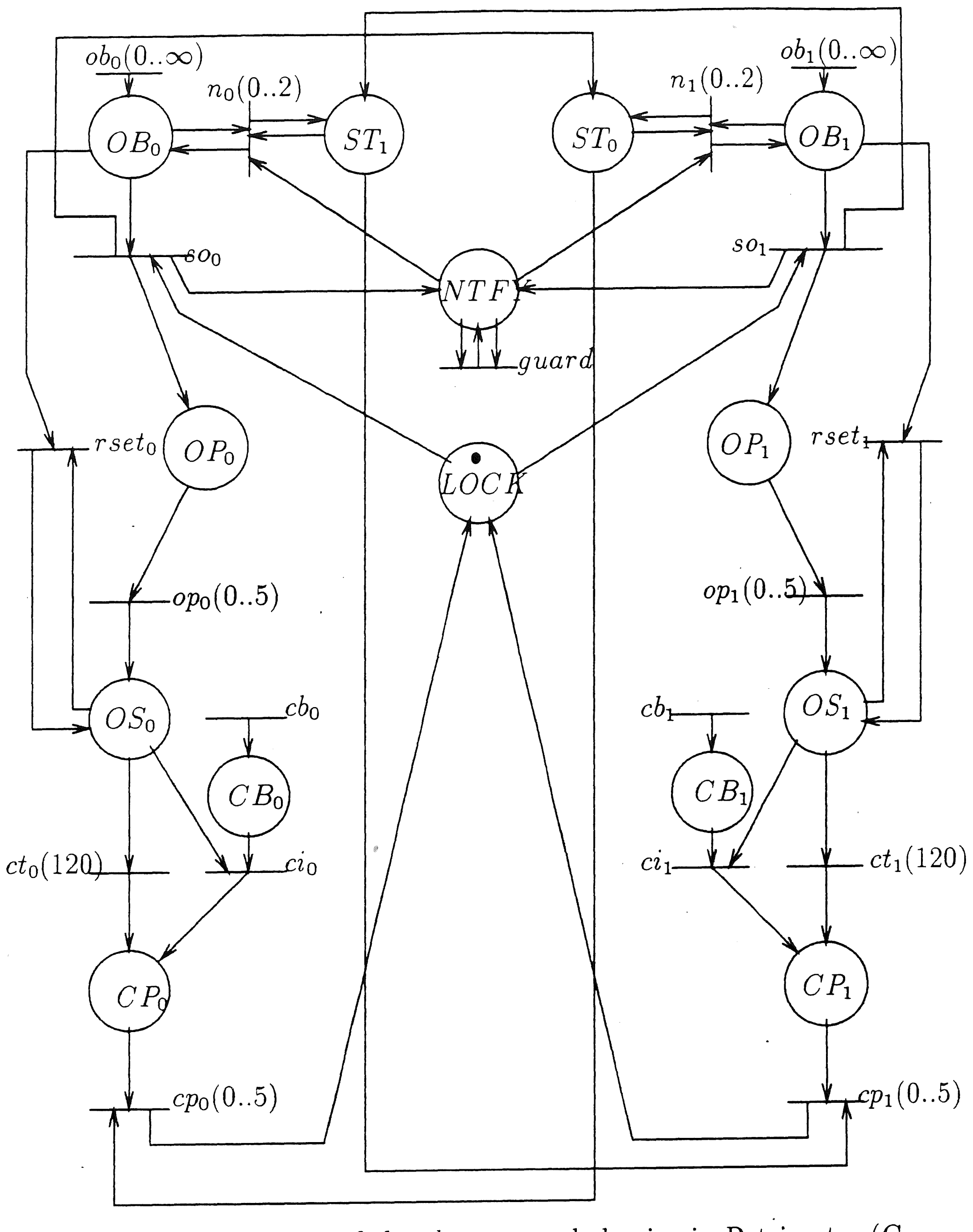


Figure 10: Representation of the clean room behavior in Petri-nets. (Courtesy of Jon Reese)

representation. The places and transitions of each door are distinguished by a subscript number that is attached to each label. Places are labeled by capitals and transitions by regular letters. Places $LOCK$ and $NTFY$ are the only ones that lack subscripts as they are used for mutual purposes. $LOCK$ serves as an interlock that enables an opening process for one door at most, and $NTFY$ is used for enabling an automatic opening of a door when the other ended its closing process. In the initial marking only place $LOCK$ contains a token (i.e., $\mu_0 = \{LOCK\}$).

The process starts when an "open" button is pressed, say door 0. This is represented by the firing of transition ob_0 which puts a token in place OB_0 ("Open Button"). Since, only transition so_0 ("Start Open") is enabled, it fires immediately. This removes the tokens from places $LOCK$ and OB_0, and deposits tokens in places OP_0 ("Opening Process"), ST_0 ("Status") and $NTFY$ ("Notify") (i.e., $\mu_1 = \{OP_0, ST_0, NTFY\}$). Notice that by this transition so_1 is disabled, even if a token is placed in place OB_1. In terms of the system, it prevents door 1 from opening if its "open" button is pressed. The opening process is represented by transition op_0 that may fire within 5 seconds (0..5). This removes the token from place OP_0 and assigns one to place OS_0 ("Open State"). Now three events may take place: (a) nothing is done so the door has to be closed automatically after 2 minutes [transition ct_0 ("close on time") fires], (b) the close button is pressed [transition cb_0 ("close button")] which starts an immediate closing process by putting a token in place CB_0 which, in turn, causes transition ci_0 ("close immediately") to fire immediately, or (c) the open button is pressed again resetting the 2 minute timer. The last event is represented by transition $rset_0$ ("reset") that removes the tokens from places OB_0 and OS_0 and returns a token to place OS_0. This disables and reenables transition ct_0 which restarts the counter for that transition. The closing process is represented by place CP_0 and transition cp_0. Upon completion of the closing process a token is returned to place $LOCK$ while places ST_0 and CP_0 are emptied.

If a door 1 "open" button is pressed while door 0 is open the system has to notify the user within 2 seconds. This is represented by transition n_1. This transition is enabled since all its input places $\{NTFY, ST_0, OB_1\}$ have tokens. The transition fires within 2 seconds giving rise to a marking that includes places ST_0 and OB_1 but not $NTFY$. This symbolizes the notify state. Notice that as soon as transition cp_0 fires it enables transition so_1, i.e., door 1 starts the opening process from the notify state. It also empties place

ST_0.

There is only one point that is left to explain and that is the purpose of transition *guard*. Notice that place $NTFY$ is not cleared if a door is opened and closed while no open request arrives from the the other door. This may cause this place to be filled with any number of tokens. If place $NTFY$ contains more than one token, it will prevent a notify state to be reached (remember that this state is represented by a marking that includes OB and ST places but not $NTFY$). To avoid this situation, transition *guard* fires whenever there are more then one token in place $NTFY$ until only one is left. This will allow an empty $NTFY$ place when either transition n_1 or n_2 fires.

Places OB_0, CB_0, OB_1 and CB_1 represent "open" or "close" requests from a human user. Therefore, they may contain more then one token as well (who is not acquainted with multiple pressing of an "open" elevator button). This may cause to a strange behavior of our Petri-net. In order to avoid it "*guard*" transitions has to be added to all those places. Without explicitly representing this situation we will assume that all the above places are guarded by *guard* transitions.

Any type of Petri-net analysis consists of building reachability graphs. This means executing the net and tracing the markings. Despite the fact that time restriction may reduce drastically the number of reachable markings, this number may still remain too large to be analyzed [HV87]. Leveson and Stolzy [LS87] offered backward reachability analysis that is usually sufficient for safety analysis where only certain, known states have to be investigated (i.e., hazardous states). In this case a marking that includes OS_0 and OS_1 is such a state. This marking corresponds to a state in which both doors are open. Figure 11 is a backward reachability graph for this state. In this figure we use the sign $*$ to denote any marking and rectangles to denote "bottom marking" that are not further developed.

Two observations are needed to understand the graph. First, since both doors are equal, we may follow either of them along similar paths. Therefore, whenever such a situation occurs door 0 is chosen leaving a circled $=$ symbol for door 1. The second observation regards the reachable path to place OS_0 and in particular its input transition $rset_0$. Notice that this transition may fire only if place OS_0 is already with a token. Such a firing will cause place OB_0 to loose its token, but will not change anything in place OS_0. From a reachability point of view, this firing brings us back to the "top

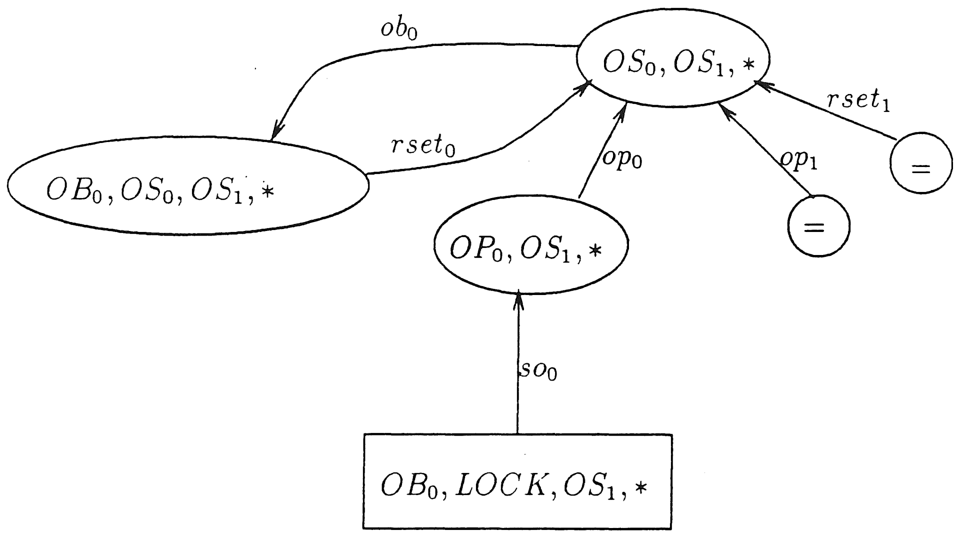


Figure 11: Backward reachability graph for OS_1, OS_2 marking

marking". This is shown as a loop in the reachability graph and needs no further elaboration.

It is not difficult to see from figure 11 that in order for the system to reach the unsafe marking $\{OS_0, OS_1, *\}$ a previous critical marking must precede. This marking includes both $LOCK$ and one of the OS places. This situation can occur if an "interlock" device changes states because of external interference or malfunctioning.

The solution for this case follows the recommendation of Leveson and Stolzy [LS87]. No matter if a critical marking can be reached, it must be avoided. Figure 12[13] shows how this is achieved. Two transitions $safe_0$ and $safe_1$ are added. Transition $safe_1$ will fire whenever places $LOCK$ and OS_1 have tokens. This will immediately remove the tokens from these places and deposit a token in place CP_1. Recalling that this place represents the closing process of the door and noticing that since transition ob_1 fired place ST_1 must possess a token (figure 11, one of the * places), transition cp_1 is enabled. Thus, the $safe$ transitions represent a recovery process, that sense an unsafe situation and "fixes" it. Notice that there is still another change

[13]We show the critical marking and discuss the solution, with respect to door 1, but obviously a similar description can be made for door 0.

in figure 12. A *guard* transition was attached to place $LOCK$. The purpose of this *guard* is to avoid a situation in which place $LOCK$ starts a process with more than one token, which eventually will enable the unsafe state.

Evaluation: The above discussion shows clearly the Petri-net interpretation. Places represent system states while transitions stands for events. The marking can be captured as the state that is in effect. As in other methods actual states are used as predicates or conditions for future events. The execution of a Petri-net follows very simple rules and thus is very easy to trace. This, in turn, makes the analysis simple. The cost of this simplicity is paid with large graphs that are hard to understand and interpret. Reachability, which is the main tool for analyzing Petri-nets, may become complex even for small system. The interpretation of markings as states is also not straightforward. As an example, what is the interpretation of more then one token in place SO_1? Is it a failure of the control that interpreted a close state as open and thus issued a second open order?

Petri-nets model distributed control systems. Events (transitions) occur (fire) according to some local arrangement ("partial marking") and are not dependent on the entire state (marking) of the system. Information is transferred in a point to point method. This increases the complexity of the net especially when conditions are not simple. The only way to represent conditions on events using Petri-nets is by connecting transitions with desired places. This means that the number of edges is proportional to the number of predicates and their complexity. This can be seen in the clean room representation. Whenever events are dependent on states and may have different meaning, more edges and transitions are present. The "open" request may have three meanings that are state dependent, i.e., (a) start opening if the other door is closed, (b) reset the timer if the door is already opened, and (c) notify the user if the other door is not closed. These requirements are "responsible" for more then half the transitions, places and edges.

The above discussion highlights other properties of Petri-nets. Many details that were left unnoticed in other methods had to be dealt with in Petri-nets. This has advantages and disadvantages. Details may turn out to be very important as the development continues. It is well known that the earlier a problem is identified, the less costly it is in fixing it. Many projects suffer from small unnoticed problems that were discovered too late. Petri-net

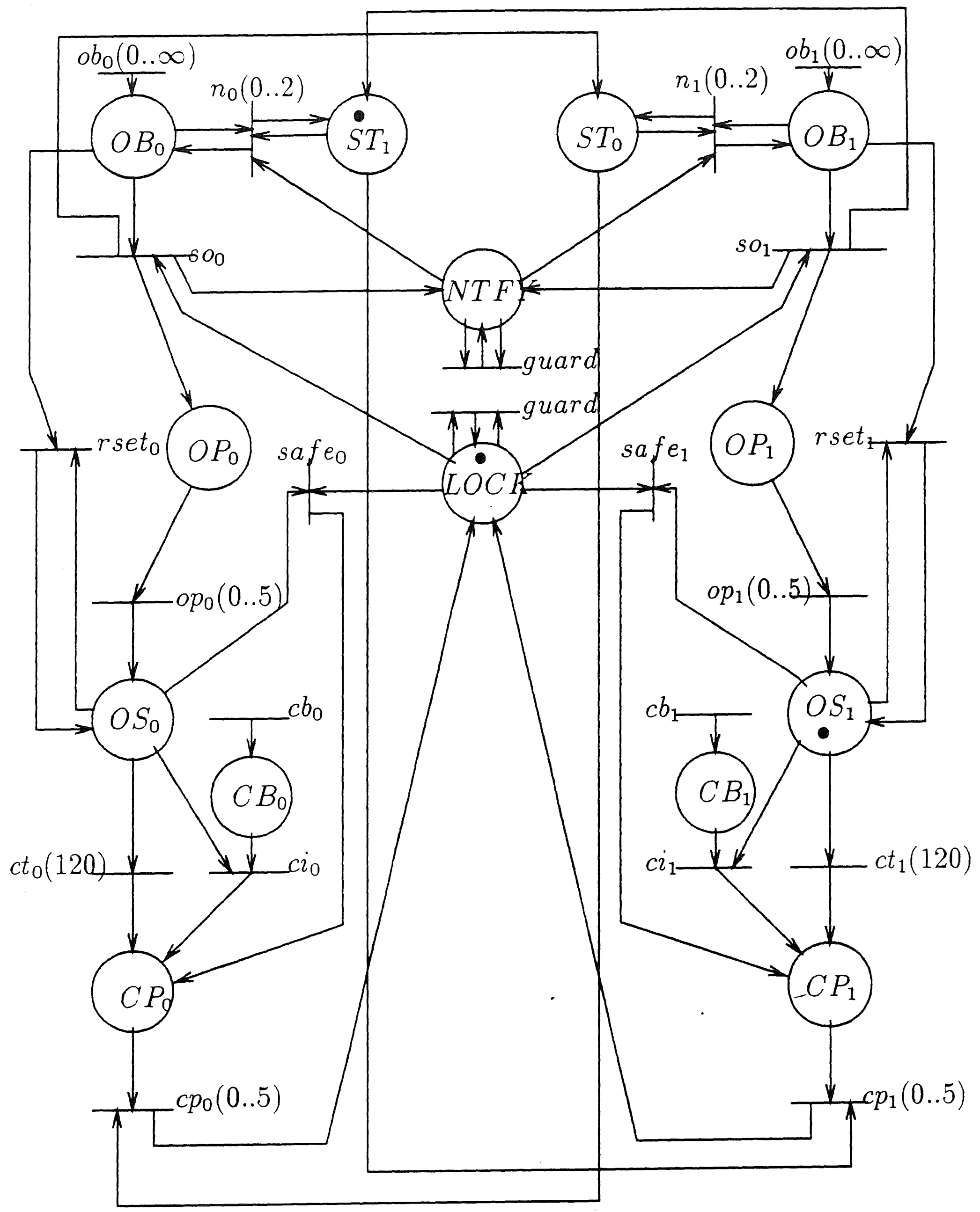


Figure 12: The “fixed” representation of the clean room problem.

representations forces the developer to trace every detail, analyze it carefully and decide on the best way to handle it. As for the disadvantages, too many details at the beginning of the development may cover major properties. The first stages of the development process are characterized by abstracting out small details of the system in order to understand the basics of its behavior. This is difficult to do in Petri-nets.

This leads to the conclusion that was already mentioned. Petri-nets are a good design method but are less efficient for behavioral description. It is recommended to use them during the late stages of the development process.

2.2 Process Based Methods

In this section two process based methods are presented, CIRCAL [Milne85, Milne82] and TAM [Zwarico88, LZ88]. CIRCAL and TAM are algebraic methods that include some basic axioms and definitions above which more rules can be built. Systems are described and analyzed by the processes they may execute, and properties are proven by applying those rules.

2.2.1 CIRCAL

General: CIRCAL or CIRcuit CALculus was introduced by G. J. Milne as a framework for modeling of asynchronous and simultaneous behavior, mostly for integrated circuits [Milne83, Milne85]. The method is based on, and is an extension of a calculi for representing concurrency and intercommunication called *dot calculus* [Milne80, Milne82]. CIRCAL is also an extension of CCS [Milner80] in its ability to handle simultaneous events. Other properties of CIRCAL are borrowed from Hennessy and Milner's acceptance semantics [HM80, HM85, Hennessy85]. The idea behind this semantics is to use system processes for determining whether an action is accepted and thus initiate a change or not. When this decision is based on external processes, the decision is said to be deterministic and nondeterministic otherwise.

In CIRCAL terminology, computing *agents* which are completely described by *actions* they wish to perform in cooperation with their environment. The environment, in turn, is also described by agents and actions. This means that actions is the interacting media among agents. This can be depicted as boxes (representing agents) connected by arcs via ports (representing actions). Both ports and agents may be labeled. Ports with the

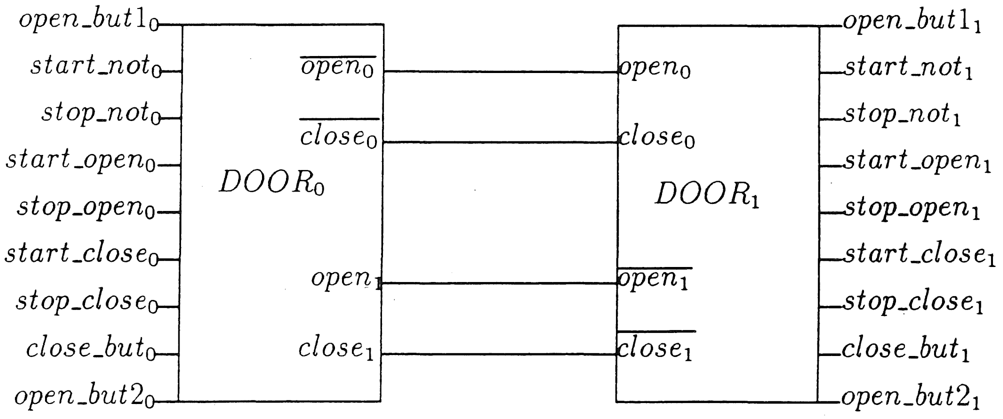


Figure 13: Clean room problem represented in CIRCAL.

same label are connected.

There are two more concepts in CIRCAL *term* and *sort*. The first represents agents or actions and the second is a set of actions. Properties of agents are described by their sort, that is, according to the set of actions they can be engaged in. The clean room can be depicted in this manner as is shown in figure 13. The two doors have common ports $open_0$, $open_1$, $close_0$ and $close_1$. In the example, the agent $DOOR_i$, $(i = 0, 1)$ is of sort[14]:

$$\{open\_but1_i, start\_not_i, stop\_not_i, start\_open_i, stop\_open_i, close\_but_i,$$
$$start\_close_i, stop\_close_i, open\_but2_i, \overline{close_i}, open_{i+1}, close_{i+1}\}.$$

The process is described as a sequence of all possible ordered events (the traces) that may occur during the term's life. This will become clearer as we continue in the discussion.

Each term is characterized by its reaction to a given *stimulus*. There are two possible reactions; the labeled stimulus is either *accepted* or *rejected*. In the first case the term will evolve to a new one and in the other case a $*$ will be produced. The first case is denoted by $TERM \overset{stimulus}{\longmapsto} TERM'$ and the second $TERM \overset{stimulus}{\longmapsto} *$.

In CIRCAL there are four primitive operators:

[14]The exact meaning of each port will be explained later, when we show the clean room representation.

- Guarding which is denoted as mP and means that the term mP desires to perform the action m and evolve to the new term P,

$$mP \overset{n}{\longmapsto} \begin{cases} P & \text{if } n = m \\ * & \text{if } n \neq m. \end{cases}$$

- Deterministic choice between two or more terms is denoted $+$ or $\sum$ respectively.

- Nondeterminism choice between two or more terms is denoted by $\oplus$ or $\bigoplus\!\!\!\!\sum$, respectively.

- Termination of a process is denoted as Δ.

Concurrent composition of two or more processes is represented by a $\bullet$. For example, the expression $A \bullet B$ means two agents A and B executing in parallel. The formal definition of this operator is:

$$\begin{aligned} \text{for:} \quad A \Longleftarrow \quad & \sum_i \lambda_i A_i \text{ of sort } L, \; B \Longleftarrow \sum_j \mu_j B_j \text{ of sort } M \\ A \bullet B \Longleftarrow \quad & \sum_{\lambda_i \cap M = \emptyset} \lambda_i [A_i \bullet B] + \sum_{\mu_j \cap L = \emptyset} \mu_j [A \bullet B_j] \\ + & \sum_{(\lambda_i \cap M) = (\mu_j \cap L)} (\lambda_i \cup \mu_j)[A_i \bullet B_j] \end{aligned} \tag{8}$$

Where $\Longleftarrow$ is used for defining new terms. This operator captures both independent and synchronized execution of the two processes. This is done in a rather sophisticated way. When two terms are executing in parallel, they may interact with identical ports. The first two clauses represent independent execution, i.e., guards of different sorts. If one of the λ_is that does not intersect with M occurs first, the process A will evolve to A_i and continue to execute in parallel with B. This is represented by the first clause. The second clause is similar and corresponds to a μ_j that does not intersect with L.

The third clause represents simultaneous actions of two types. The first type regards actions that are required to execute simultaneously, since they have the same label (i.e., $(\lambda_i \cap M) = (\mu_j \cap L) \neq \emptyset$). The second type is composed of actions that are included in only one sort and thus are not required to, but may execute simultaneously, "if it so happens" (i.e., $(\lambda_i \cap$

$M) = (\mu_j \cap L) = \emptyset$). In both cases the simultaneous actions are written in parentheses. For example, $(\alpha\ \beta)P \overset{(\alpha\ \beta)}{\longmapsto} P'$ means that α and β must occur simultaneously in order for P to change into P' (a combination of two or more simultaneous actions is considered a new action).

The $\bullet$ operation can capture deadlocks as well. For example, if P and Q are of sort $L = \{a, b\}$ and $P \Longleftarrow a\ bP'$, $Q \Longleftarrow b\ aQ'$, than $P \bullet Q$ deadlocks on the first action. This happens because $(a \cap L) \neq (b \cap L) \neq \emptyset$, which means that a and b must execute simultaneously (only the third clause of equation 8 exists). But, this is in contrast with both process P and process Q, since each requires a sequential execution.

CIRCAL includes also several axioms that are used for deriving properties and analyzing terms. There are axioms for commutative and associative combination that are quite obvious and others that are more complicated. Examples of axioms are: $X \bullet [Y \oplus Z] = [X \bullet Y] \oplus [X \bullet Z]$, $[A \bullet B] \bullet C = A \bullet [B \bullet C]$ or $\alpha P + \alpha Q = \alpha P \oplus \alpha Q$.

There are two abstraction operators that are of particular interest. One deals with connected ports and the other with isolated ports. Both are used very often for representing systems in different levels of detail and both introduce nondeterminism. The first operator is called the abstraction operator and is defined as:

$$(\sum_i \mu_i P_i) - \alpha \quad \Longleftarrow \quad \left[\sum_{\alpha \in \mu_i, \mu_i \neq \alpha} [\mu_i - \alpha][P_i - \alpha] + \sum_{\mu_i \neq \alpha, \alpha \notin \mu_i} \mu_i [P_i - \alpha] + S \right] \oplus S$$
$$\text{where} \quad S \Longleftarrow \sum_{\mu_i = \alpha} [P_i - \alpha] \tag{9}$$

The events that are in the set α are "internalized" and thus concealed from the environment which can not control the choice or even know when it took place. This introduces nondeterminism. The meaning of this equation is as follows: If $\alpha = \mu_i$, then one of the events in α must occur first but because of the abstraction, the environment lost control of it. This explains the nondeterminism and the right most clause, S. The first summands within the big rectangle brackets represents an occurrence of an event that is in μ_i but not in α. The second summand is similar to the first but represents a case where $\alpha \cap \mu_i = \emptyset$. The third summand, S, is equal to the one outside the brackets but represents a case where the agent is not able to interact with the environment at all. In this case, only the internal events in α may occur, but the environment can not know which one. For example,

$$[(b\ c)P + aQ] - a = [(b\ c)[P - a] + [Q - a]] \oplus [Q - a]$$

If $(b\ c)$ can occur, than either it occurs first giving rise to the term $P - a$ or a occurs first giving rise to the first $Q - a$. If $(b\ c)$ can not occur, the only possibility left is $Q - a$. Since abstraction introduces nondeterminism, it distributes over nondeterminism, i.e.,

$$[\sum_i P_i] - \alpha = \sum_i [P_i - \alpha] \tag{10}$$

The other abstraction operator, the "hiding" operator (denoted $\backslash$), is used to hide isolated ports. Since isolated ports are members of one agent, they may influence only that agent. Other agents may be affected as a result of eventual simultaneous occurrence. The hiding operator causes all affected ports, including simultaneous ones, to disappear. For example, $[(a\ b)P + cQ]\backslash b = c[Q\backslash b]$ and $[(a\ b)P + bQ]\backslash b = \Delta$ but $[(a\ b)P + cQ] - b = a[P - b] + c[Q - b]$ and $[(a\ b)P + bQ] - b = [a[P - b] + [Q - b]] \oplus [Q - b]$.

Another important feature of CIRCAL is *value passing.* CIRCAL can represent passage of values between ports. This is done by using as many as needed CIRCAL ports for each physical port. If a physical port is used for passing boolean values, than only two CIRCAL ports are needed, however if no limit is known, infinite number of CIRCAL ports are required. For example, an agent E with one physical port α is used for passing integers from 1 to N (denoted as $E \Longleftarrow \sum_{i \in N} \alpha i\ E'$) will need N CIRCAL ports.

In order to distinguish between output and input values or ports, CIRCAL uses the convention that output values are denoted by on "overbar". For example, $\overline{\alpha i}\ P \bullet \beta i\ Q$ means that the value i is passed from process P as output through port α to process Q via port β.

Time passage is modeled in CIRCAL by attaching each port to a clock agent of sort t which represents time units. By including t in the sort of every agent a clock synchronization is forced on the agent but not on the clock. This is represented in equations of the form $A \Longleftarrow (\alpha\ t)A' + tA$ where α is the port label.

The Clean Room Example: To simplify the problem, we shall assume that the only delays in the clean room are those specified by the requirements.

This means for example, that checking the state of a door takes no time, but the time needed for opening a door can get any value between 0 and 5 seconds[15]. We also assume that time units are seconds. The following CIRCAL definitions will be used:

$$\begin{aligned} t^i &\Longleftarrow \overbrace{tt\ldots t}^{i \text{ times}} \\ \alpha(0,n)\,\beta &\Longleftarrow [\alpha \bigoplus (t\alpha) \bigoplus t(t\alpha) \bigoplus \cdots \bigoplus \overbrace{tt\ldots t}^{n-1 \text{ times}}\,(t\alpha)]\,\beta \end{aligned}$$

The first definition denotes passage of i seconds. The second definition uses $\alpha(0,n)\beta$ to denote that event α occurs within n seconds and is followed by event β. The nondeterministic choice, $\bigoplus$, is used since the procedure is internal and the environment never knows exactly when event α is going to execute.

The clean room process, CR, is combined of two sub-processes, one for each door. We will call them $DOOR_i$ for $i = 0, 1$.

$$\begin{aligned} CR &\Longleftarrow DOOR_0 \bullet DOOR_1 && (11) \\ DOOR_i &\Longleftarrow open\_but1_i\; CHK_i\; OPEN\_STATE_i\; DOOR_i. && (12) \end{aligned}$$

where:

$$\begin{aligned} CHK_i &\Longleftarrow open_{i+1}\; start\_not_i(0,2)\; CHK_i' + close_{i+1} \\ CHK_i' &\Longleftarrow open_{i+1}\; CHK_i' + close_{i+1}\; stop\_not_i && (13) \end{aligned}$$

and,

$$\begin{aligned} OPEN\_STATE_i &\Longleftarrow \overline{open_i}\; start\_open_i\; stop\_open_i(0,5) \\ &\qquad FUL\_OPEN_i\; start\_close_i\; stop\_close_i(0,5)\; \overline{close_i}\; DOOR_i && (14) \\ FUL\_OPEN_i &\Longleftarrow [open\_but2_i\; FUL\_OPEN_i] \bigoplus \\ &\qquad [close\_but_i] \bigoplus [t^{120}] && (15) \end{aligned}$$

In these equations $i = i \; mod\; 2$.

[15]We can make this simplification even stronger. The time restrictions are given only to events that involve interaction with the external environment. Usually, these interactions take much more time than those required for internal electronic interactions. Thus, we may "safely" neglect internal interactions.

The process starts in equation 12 when port $open\_but1_i$ is executed, that is, the open button is touched. The suffix "1" distinguishes this CIRCAL port from the other in equation 15 that represents the same physical operation when the door is open.

CHK represents the checking process that must precede the open state. In this process, the state of the other door is checked. If it is closed ($close_{i+1}$) the door is allowed to open, otherwise ($open_{i+1}$) the user has to be notified within 2 seconds ($start\_not_i(0,2)$). Process CHK' is similar: the process checks for the other door state, once $close_{i+1}$ executes the door is allowed to open, so the notification is removed ($stop\_not_i$)[16].

Events $start\_open_i$ and $stop\_open_i(0,5)$ ($start\_close_i$ and $stop\_close_i(0,5)$) represent the door's opening (closing) process. Process FUL_OPEN_i (equation 15 represents the various possibilities when the door is fully opened. Closing may start if the close button is touched ($close\_but_i$) or if 120 seconds elapsed (t^{120}) with no $open\_but2_i$ execution. Touching the open button while the door is open will cause the whole FUL_OPEN_i process to re-start. Since no external involvement can occur while the door is opened, these possibilities are chosen nondeterministically.

The processes $OPEN\_STATE_i$ (equation 14) represent critical atomic sections. These sections must not be broken, i.e., once a section is started, the other should not start unless the first ends. This is the safety specification. The *open* and *close* events can be used as start and end points for these sections and the safety specification in CIRCAL terminology will look like:

$$CR \Leftarrow [(\overline{open_0}\ \overline{close_1})CR] \oplus [(\overline{open_1}\ \overline{close_0})CR] \oplus [(\overline{close_0}\ \overline{close_1})CR] \quad (16)$$

What is left is to show that the safety specifications are consistent with equations 11 to 15. This is done by using the definition of the dot operation (equation 8) and hiding (equations 9 and 10) all ports except those specified in equation 16. This process is very tedious and demands evaluation of long and complicated CIRCAL expressions. Instead of showing the whole procedure, only the first steps are described. This will give some more insight to the use of the CIRCAL definitions and axioms and allow some more "feelings" about the proof scheme. Using the dot operation on equations 11 and

[16]Notice that each door needs four different CIRCAL ports for the CHK process. Each port is designated to transfer (receive) either *open* or *close* for each door. Also, we preferred not to use value passing, as it does not reduce the number of CIRCAL ports and may cause confusion.

12 we get:

$$\begin{aligned} CR \;\Longleftarrow\; & open\_butl_0 \left[[CHK_0\, OPEN\_STATE_0\, DOOR_0] \bullet DOOR_1\right] \\ + \;& open\_butl_1 \left[DOOR_0 \bullet [CHK_1\, OPEN\_STATE_1\, DOOR_1]\right] \\ + \;& (open\_butl_0\, open\_butl_1)\Big[[CHK_0\, OPEN\_STATE_0\, DOOR_0] \\ & \qquad \bullet[CHK_1\, OPEN\_STATE_1\, DOOR_1]\Big] \quad (17) \end{aligned}$$

This equation shows how powerful CIRCAL is. It represents all possible combinations of open requests; either in series, as in the first and second summands (first $open\_butl_1$ and then $open\_butl_2$ or vice-versa), or simultaneously, as in the third summand. Now, the third summand pinpoints a problem that was already mentioned. The system behavior in a situation where both buttons were pressed simultaneously, is not specified. This property of CIRCAL is very important, especially when compared with other algebraic methods such as CCS [Milner80].

Now applying the hiding operator (equation 10) on the *open_butl* ports of equation 17, and using again the dot operator results in:

$$CR \Longleftarrow \sum_{i=0,1} CHK_i\, OPEN\_STATE_i\, CR\backslash\{open\_butl_0, open\_butl_1\}$$

In a further development of this equation (using the abstraction operator (equation 9)) the nondeterministic choice operator, $\tilde{\sum}$ will replace the deterministic one, $\sum$, or the $+$ will be replaced by $\oplus$, which is consistent with the end result, i.e., equation 16.

Evaluation: CIRCAL is an event driven method. It models systems by tracing their possible sequences of events. As was shown, CIRCAL can capture and analyze synchronous and asynchronous processes. This is a real advantage over previous algebraic methods. This ability was demonstrated in equation 17 by pinpointing a missing requirement. More than that, this result did not demand special treatment, it merely required the use of the method primitives.

CIRCAL was found to be suitable for analyzing procedures in computer operating systems and VLSI. Such systems are basically event driven and incorporate multilevel design, concurrent and sequential processes. These properties can be represented in CIRCAL; multilevel design by using abstraction and the other properties by applying the dot calculus [Milne80, Milne82]. Timing is achieved implicitly by defining an agent clock and using its "ticks" as the driving events.

CIRCAL is very restrictive in its abilities to express many features that complex systems contain. For example, it is difficult to express periodical processes with time restriction and tolerances. Also, periodic processes for a limited time followed by a different path of activities are not included in the CIRCAL grammar. CIRCAL expressions for such processes are clumsy and hard to comprehend. In general, CIRCAL may fit for describing accurate processes without any tolerances. This is very restrictive and makes the method almost useless for real time reactive systems.

Conditional expressions are also very clumsy in CIRCAL. Both the deterministic and the nondeterministic choice operators are efficient for selecting different paths according to the events that may occur, but become very awkward when the choice depends on a situation or value in another process. In cases where the condition is a combination of many clauses the expression may become very complicated and totally unmanageable. The reason for this is the attempt to use only very primitive operators. In the clean room example, only one way was presented; assigning a special port to every possibility (actions $open\_but1$ in equation 12 and $open\_but2$ in equation 15, as well as, the *open* and *close* actions in the CHK sub-process (equation 13)).

Another way for representing conditional expressions by value passing can be used. But, this may become even more clumsy. As an example, in the CHK sub-process two pairs of connected ports (e.g., labeled *check*) had to be used, each for passing "open" or "close" values according to the specific door state. As can be seen this does not reduce the number of CIRCAL ports but may clarify that the selected path depends on a particular state and not on the events.

Safety analysis in CIRCAL is also difficult and tedious. Long and sophisticated expressions are usually a result of even simple processes. This is error prone and exposes the analyzer and the designer to similar problems. Such problems may bring into question the usefulness of CIRCAL in its present form, as a general purpose method.

CIRCAL can be used as good starting point for developing and defining other methods that will include, apart from CIRCAL primitives, more powerful expression tools. Such tools will allow better description and enhance the analysis power. This may be done in two complementary ways. The first by proceeding with the algebraic approach and defining tools that are derived from the primitives of CIRCAL and probably used in special areas rather then for general purpose. The second way is by applying some pictorial aids that may have more comprehensive power, and thus enable a better analysis.

2.2.2 Time Acceptance Model

General: TAM or Time Acceptance Model was developed by Lee and Zwarico [LZ88, GLZ88, Zwarico88] as a direct time description tool. By the word "direct" we mean that time is represented explicitly and not only through synchronization heuristics. This is an advantage over other methods in which time has to be represented as synchronization points with some imaginary clock. The method is an extension of Hennessy's *Acceptance Trees* model for describing and ordering nondeterministic processes [Hennessy85] and Hoare's *Communicating Sequential Processes (CSP)* [Hoare78, Hoare85]. This is done by adding explicit time stamps to each event and requiring it to occur within a specified time interval. As other algebraic methods, TAM is event driven. It incorporates notations for representing time intervals and process decisions that are taken accordingly. Periodic and recursive processes are described in a natural way that is easy to understand. Although the basic theory of TAM is complicated, the syntax and the semantics of the model are clear and can be used for evaluating, analyzing and proving time assertions of the requirements.

The model is based on a *process* that is fully characterized by the sequences of events it can execute and their time constraints. In the language terminology, the events are the *alphabet* and the execution sequences are the *acceptance set.* Every process execution consists of *time traces*, that represents the time progress, and *state set*, that represents the set of possible paths. Syntactically, a process is represented by a sequence of event time pairs. These concepts are natural in the sense that system operation is dependent not only on the state of the system, but also on the time when events occur. In other words an occurrence of an event in identical states may cause different consequences depending on the occurrence time.

The model includes basic concepts that are combined into a *domain of time dependent processes* (denoted as $\mathcal{TD}$) over which time operators execute.

The basic concepts are:

Time Trace: is a finite sequence $\langle(a_1, n_1), (a_2, n_2), \ldots, (a_i, n_i)\rangle \in (\Sigma \times N)^*$ where Σ represents the alphabet and N the natural numbers including ∞. Each pair (a_i, n_i) represents the occurrence of event a_i, n_i time units after the previous event.

States: are deterministic[17] choices that a process may make in order to decide its next action. A state is represented by event-time pairs $\{(A_1, n_1), \ldots, (A_m, n_m)\}$. For example the process:

$$\{(\{a\},1),(\{a\},2),(\{b\},1),(\{b\},2),(\{c\},3),(\{c\},4),(\{c\},5),(\emptyset,7)\}$$

(which may be denoted as $(\{a\ b, [1,2]), (c, [3,5]), (\emptyset, 7)\})$ may choose to execute event a or b during the interval $[1,2]$ or c during $[3,5]$ or stop $(\emptyset)$ at time 7.

State Sets: represent nondeterministic decisions of a process. For example, $\{\{(a,[1,2])\}, \{(a,[1,2]),(c,3)\}, \{(c,[3,5])\}\}$ represents the nondeterministic choice of one of the states: $\{(a,[1,2])\}$, $\{(a,[1,2]),(c,3)\}$ or $\{(c,[3,5])\}\}$. (It is clear that in order to prove behavior of nondeterministic processes some *extra knowledge* is needed. This knowledge can be provided by some other sources or can be determined when more details of the process are revealed.) State sets are denoted by $\bar{\sigma}$.

Acceptance: $(s, \bar{\sigma} \neq \emptyset)$ represents possible execution of a process, where $\bar{\sigma}$ is the set of states that can be reached after executing the trace s. For example, the process $(\langle(a,1),(b,3),(c,4)\rangle, \{\{(d,[1,3])\}, \{(d,2)\}\})$ can execute either $\{(d,[1,3])\}$ or $\{(d,2)\}$ after executing the trace $\langle(a,1),(b,3),(c,4)\rangle$ (i.e., a at 1, b at 4 and c at 8).

$\mathcal{TD}$ is defined in a way that guarantees a coherent operation of the process. For example, every process that is in $\mathcal{TD}$ contains the empty trace $\langle\rangle$ and must be prefix closed, that is, if process P executes a trace s, then it must have executed all its prefixes first. Processes that are in $\mathcal{TD}$ can be shown to be in a complete order that is bounded from below and above and all their set states can be arranged in a descending (ascending) order. For

[17]Deterministic and nondeterministic processes, have the same meaning as in CIRCAL.

every alphabet $\mathcal{A}$ the two boundaries are defined as $STOP_{\mathcal{A}}$ and $CHAOS_{\mathcal{A}}$. The first does nothing, and the second represent every possible trace of $\mathcal{A}$.

TAM includes a set of primitive and derived operators. We shall use the clean room example to represent most of them.

The Clean Room Example: As in CIRCAL we simplify the problem by assuming that every information transfer is done promptly, so there are no "electronic" delays. That is, only required and "Mechanical" delays are taken into consideration. For example, a checking procedure takes zero time but notifying a user about the state of the other door may be differed for 2 seconds.

The clean room behavior (CR) may be divided into two parallel processes; $CR = DOOR_0 || DOOR_1$. As before we will use $i = i \; mod \; 2$ to stand for any door.

$$\begin{aligned} DOOR_i \;=\;& \epsilon \overset{(0,\infty)}{\rightsquigarrow} open\_but_i \overset{0}{\rightsquigarrow} \\ & [(open_{i+1} \overset{(0,2)}{\rightsquigarrow} not_i \overset{(0,\infty)}{\rightsquigarrow} close_{i+1}) \Box (close_{i+1})] \\ & \overset{0}{\approx\!\!>} start\_open_i \overset{(0,5)}{\approx\!\!>} stop\_open_i \overset{0}{\approx\!\!>} OPEN\_STATE_i \\ & \overset{0}{\approx\!\!>} CLOSE\_STATE_i \overset{0}{\rightsquigarrow} DOOR_i \qquad (18) \end{aligned}$$

where:

$$\begin{aligned} OPEN\_STATE_i \;=\;& (\epsilon \overset{(0,120)}{\approx\!\!>} open\_but_i \overset{0}{\approx\!\!>} OPEN\_STATE_i) \\ & \sqcap \; ((\epsilon \overset{(0,120)}{\approx\!\!>} close\_but_i) \sqcap (\epsilon \overset{120}{\approx\!\!>} close\_on\_time_i)) \qquad (19) \end{aligned}$$

and

$$CLOSE\_STATE_i = \epsilon \overset{0}{\approx\!\!>} start\_close_i \overset{(0,5)}{\approx\!\!>} stop\_close_i \qquad (20)$$

The process starts when an open button is touched ($open\_but_i$). This event's time restriction is denoted by the "deterministic time action operator", $\rightsquigarrow$. This operator represents a deterministic sequential execution of a process. The expression $\epsilon \overset{(0,\infty)}{\rightsquigarrow} open\_but_1$ means that event $open\_but_i$ may happen at any moment during the interval $(0, \infty)$. The symbol ϵ represents a time mark but is not an actual event (i.e., $a \overset{n}{\rightsquigarrow} \epsilon \overset{m}{\rightsquigarrow} b = a \overset{n+m}{\rightsquigarrow} b$).

The second line of equation 18 represents a choice between two sub-processes and depends on the first event, $close_{i+1}$ or $open_{i+1}$. The choice is deterministic which is represented by $\Box$. If the other door is open (event $open_{i+1}$) the process notifies the user within two seconds, $open_{i+1} \overset{(0,2)}{\leadsto} not_i$. When $DOOR_i$ determines that $DOOR_{i+1}$ is closed (event $close_{i+1}$), it starts the opening process. If event $close_{i+1}$ (i.e., $DOOR_{i+1}$ is closed) occurs when the open button is touched, the opening procedure starts immediately.

The opening process can take any time between zero and five seconds, which is represented by the "nondeterministic time action operator", $\approx\!\!>$. The expression $start\_open_i \overset{(0,5)}{\approx\!\!>} stop\_open_i$ means that event $stop\_open_i$ may happen nondeterministically within five seconds. This represents the restriction on the door movement. This process is nondeterministic since the external environment can not influence it.

While the door is open, state $OPEN\_STATE_i$, three sub-processes may take place (equation 19). Since the events in this process are not influenced by the environment, the time operators, $\approx\!\!>$, and the choice operator, $\sqcap$, are nondeterministic. $OPEN\_STATE_i$ may remain for an arbitrary time period. This is represented in the first part of equation 19 and may happen if the close button is not touched (event $close\_but_i$) and the open button is touched again and again, such that the time interval between two successive touches is always less than 120 seconds. Process $CLOSE\_STATE_i$ (equation 20) can start if either the the close button is touched or no button is touched in a 120 second time interval (event $close\_on\_time_i$). This process is simple and needs no further explanations.

The safety analysis can be conducted by specifying the safe operation of the processes. The usual way of doing it is by defining a "SAFE" process (a process that represents a safe operation) and showing that it is consistent with the system process. In TAM this is done by showing that SAFE is "contained" in the system process. The contained relation (denoted "$\sqsubseteq$") was not discussed here. Generally speaking it deals with the "amount" of the "nondeterminism" property. $P \sqsubseteq Q$ if P is more nondeterministic than Q. In a more accurate manner (but still without the exact definition which can be found in [LZ88] and [Hennessy85]) a process P is more nondeterministic than process Q if they both *may* accept the same alphabet (have the same events) and Q *must* accept at least the alphabet that P *may* accept. Such a relation means that SAFE is actually a part of the system process and therefore, is consistent with it.

A safe operation of CR means that both doors are not open simultaneously. This means that once a door starts to open, the other door may do so **only** after the first door is closed. An external viewer will sense the sub-processes that take place from *start_open* to *stop_close*, as atomic. Moreover, he will not be able to influence this process. In TAM terminology, the process SAFE is represented as:

$$\begin{aligned} SAFE &= (close_1 \overset{0}{\approx\!>} start\_open_0 \overset{(0.5)}{\approx\!>} stop\_open_0 \overset{0}{\approx\!>} \\ &\qquad OPEN\_STATE_0 \overset{0}{\approx\!>} CLOSE\_STATE_0 \overset{0}{\rightsquigarrow} SAFE) \\ &\sqcap \; (close_0 \overset{0}{\approx\!>} start\_open_1 \overset{(0.5)}{\approx\!>} stop\_open_1 \overset{0}{\approx\!>} \\ &\qquad OPEN\_STATE_1 \overset{0}{\approx\!>} CLOSE\_STATE_1 \overset{0}{\rightsquigarrow} SAFE) \end{aligned}$$

where the $OPEN\_STATE$ and $CLOSE\_STATE$ are defined in equations 19 and 20, respectively.

This can be represented as:

$$\begin{aligned} SAFE &= (DOOR_0 \backslash \{open\_but_0, open_1, not_0\}) \sqcap \\ &\qquad (DOOR_1 \backslash \{open\_but_1, open_0, not_1\}) \end{aligned}$$

The concealment operator, $\backslash$, means the part of the process that does not include the events "after the operator". In this case $DOOR_i$ process without events $open\_but_i$, $open_{i+1}$ and not_i. The mutual exclusiveness is guaranteed by the $\sqcap$. In this case it can easily be seen that the relation $SAFE \sqsubseteq CR$ does NOT exist since their alphabets are different ($\bar{\alpha}SAFE \neq \bar{\alpha}CR$). This can be overcome by proving that:

$$SAFE \sqsubseteq CR \backslash \{open\_but_0, open_1, not_0, open\_but_1, open_0, not_1\}$$

since $\bar{\alpha}SAFE \subseteq \bar{\alpha}CR$. The exact proof is omitted but here are some intuitive explanations. As can be seen, process $SAFE$ is a combination of the two doors processes for which the non-critical part was removed. This means that every event that $SAFE$ **may** execute is also in CR and therefore, CR **must** execute, which is exactly what is meant by the containment property. The nondeterministic choice, $\sqcap$, guarantees that no external involvement is possible, or whenever a process of one "side" of this operator starts it must end before the other "side" starts. This means that once a door starts opening it must close before the other door moves.

Evaluation: TAM represents a procedural approach to the problem. It stresses the events driven process as a part of the control stream. This enables an easy and natural representation of passage of time along with process relationships. Safety analysis with respect to time can be conducted using the "contains" relationship. The superiority of TAM over other methods is in the fact that it can represent events regardless of their synchronization with the system clock. Time is captured and measured directly and thus enables a natural approach that needs no further elaboration. Also, time tolerances, that was found difficult to express in CIRCAL, can be easily represented.

As in other algebraic methods abstraction techniques are applied by using nondeterminism. This is used for both the representation and proof scheme. For this purpose, the method is built on accurate definitions of both time and events.

One great disadvantage (that was already recognized in CIRCAL) of the method and probably the weakest part of it is its inability to account for states, or values. In particular, conditional expressions are represented in a clumsy way that makes the readability and comprehension difficult, or even impossible if the condition is complicated. This could be seen in the deterministic choice between a notification or an open process. In the simple "if $open_{i+1}$ then not_i else $start\_open_i$" expression the $open_{i+1}$ was captured as an event and not as a state.

Another difficulty in applying TAM is the complicated and long terms that the user may encounter. Such equations are difficult to understand and error prone. This can easily be seen in the equations presented here as well as those presented in the original paper [LZ88] (for example, p. 19). This is probably the reason that no real experience with the method is reported.

The above difficulty seems hard to overcome, at least by defining more elaborated operators. TAM is built in a very delicate and connected structure that makes necessary changes hard or even impossible to make. Therefore, attempts to define useful operators in order to tailor the method for particular areas may require major changes in the language itself. For example, a real-time formalism for communicating processes called Communicating Shared Resources (CSR) [GL89] that was recently introduced, suffers from such restrictions and although it incorporates many of the original TAM ideas, its structure is totally different. Since tailoring a method for specific uses is done regularly, this is a very big deficiency.

3 Other methods

In this section we briefly describe three other methods that were suggested in the literature. The first method, Interval Calculus, is state based but represents a different approach from those described below. The two others, CCS and CSP, are process based and were widely discussed in the literature and implemented in many versions. CIRCAL and TAM are among these versions.

3.1 Interval Calculus

Interval Calculus (IC) is a general approach that was introduced in artificial intelligence as a scheme for temporal knowledge representation. This approach was adapted for design and behavior description method. We bring here a method that was first introduced by J. F. Allen [Allen83] and then improved by P. Ladkin [Ladkin86.1, Ladkin86.2, Ladkin87]. This method seems to be the most progressive one in that context. The basic model uses convex time intervals that represent time "slices" during which processes are continuously operating, or system states continuously exist, i.e., there are no gaps in which the process stops or the state changes. The basic argument of the calculus is that it is possible to represent all temporal binary relations between system states.

Thus, IC is a method that uses time intervals as its basic concept. It defines states in terms of time intervals. This is in contrast to most other methods (as seen above) in which states and events determine time. IC exhaustively represents all binary relations between time intervals. The binary relations are then used for reasoning about multiple state relations.

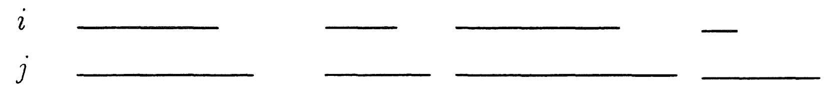
A system state can be represented by a predicate that evaluates to true in the time interval during which the state exists, and is false otherwise. Thus, for example, if door 0 of the clean room example is opened in interval d, the predicate may be:

$$open(door_0, d) = \begin{cases} true & \text{if } door_0 \text{ is open in interval } d \\ false & \text{if } door_0 \text{ is not open in interval } d \end{cases}$$

IC provides "operators" that describe binary relations between intervals. These relations are actually predicates of the form i **[always-]relation** j

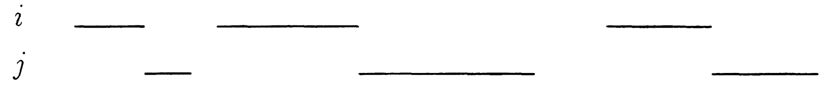
where i and j are sets of convex time intervals. The term **always** may or may not exist depending on the specific case. Here is a partial list of predicate relations:

- i **always-starts** j



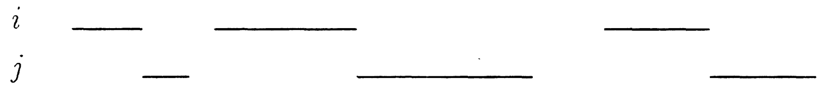
  i and j start simultaneously but i is included in j.

- i **always-meets** j



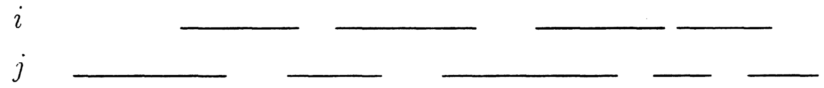
  i ends and j starts simultaneously.

- i **always-(precedes-or-meets)** j



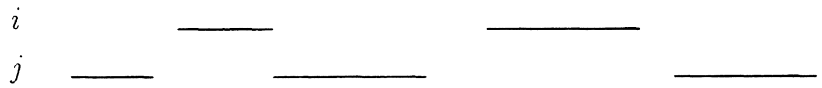
  i either ends before j starts, or i ends and j starts simultaneously.

- i **bars** j



  Combining the i and j will result a convex interval.

- i **disjoint-from** j

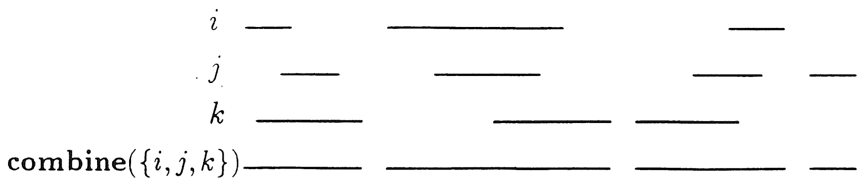


  i and j have no common subintervals

If we take the clean room as an example, the basic "safety" property can be described as:

interval-of($open(door_0)$) **disjoint-from** interval-of($open(door_1)$)

IC uses the operator **combine** to define the union of interval sets. **combine** $(\{i, j, k\})$ is the union of the interval sets $\{i, j, k\}$, as follows:



The strategy that is used in IC for specifying a system is basically top-down. First global properties are described as time assertions, than as the design proceeds more details, such as guarding intervals, synchronization points, etc. are defined. Also, this refinement procedure applies a rich language of statements.

IC represents a unique approach for time description. It attempts to represent time as intervals and uses predicate calculus for reasoning. Thus, it lacks the structure of process flow. This has advantages and disadvantages that are dependent on the nature of the developed system.

As for the the advantages, time intervals are intuitive and easy to understand and represent. This makes the analysis and design easy and less exposed to errors. Time intervals can be thought of "time pieces" during which processes take place, so the description provides an indirect representation of the system states. Moreover, since time intervals are directly represented, the states are described as a function of time. This gives the designer and the analyzer a clear and intuitive picture of the system state, as a function of time, out of which the process flow can be inferred.

The disadvantages of this approach may become more clear if we try to think of the complementary part of states, the transitions. IC does not provide any tools for representing transitions. Recalling that part of the logic of the clean room example was concentrated in the transitions, makes this disadvantage clear. A complete picture of a system is a combination of its states and transitions, which means that IC itself is inherently unable to provide a complete description of a system.

The above discussion may lead us to a conclusion about the system dependency. The description of systems, in which most of the logic is concentrated in the transitions, is less efficient and hard to understand in IC. On the other side, systems in which the transition are simple and easy to understand may be represented in IC. Also, systems that are strongly time dependent may be found easy to be represented in IC.

3.2 CCS

CCS or Calculus of Communicating Systems was introduced by Milner in 1980 [Milner80, Milner89] as a theoretical basis for process control. The ideas of this theory were later used for devising methods in particular areas. For example, CIRCAL is one outcome of the theory, which is used for defining VLSI installments.

The basic terminology of the theory is similar to the one described in CIRCAL. *Agents* are defined in terms of the *actions* they can be engaged in. As in CIRCAL, the actions are CCS *ports* that are labeled with a finite alphabet. "Overbared" labels denote output ports and are combined with regular same labels (e.g., ℓ and $\overline{\ell}$) in a synchronization line. The restrictions on the actions define the processes that may take place. Agents are interacting and may be combined to form larger systems. The interaction and composition is subjected to a set of six rules or *combinators* in CCS terminology. Here is a set of rules written in a form of $\frac{\text{assumption}}{\text{result}}$.

Action: which is the basic response of an agent to stimulus in its alphabet.
$\frac{}{\alpha.E \xrightarrow{\alpha} E}$ term $\alpha.E$ changes into E as a response of α.

Summation: which is equivalent to the deterministic choice in CIRCAL.
$\frac{E_j \xrightarrow{\alpha} E'_j}{\sum_{i \in I} E_i \xrightarrow{\alpha} E'_j}\{j \in I\}$ if term E_j changes into E'_j as a result of stimulus α, then it will choose this term out of a choice set $\{E_i\}$.

Composition: which acts as a parallel combinator among agents. The mathematical notation includes three parts.

$$\text{(a)}\ \frac{E \xrightarrow{\alpha} E'}{E|F \xrightarrow{\alpha} E'|F} \qquad \text{(b)}\ \frac{F \xrightarrow{\alpha} F'}{E|F \xrightarrow{\alpha} E|F'} \qquad \text{(c)}\ \frac{E \xrightarrow{\ell} E' \quad F \xrightarrow{\overline{\ell}} F'}{E|F \xrightarrow{\tau} E'|F'}$$

The first two rules show that this operator is commutative and that each agent is independent as long as the specific port is not combined with the other agent. The third rule shows that the two ports were combined so their activation synchronizes the operation of the two agents. τ is an internal silent action in the composed agent and thus is hidden from the environment. Note that this definition gives meaning to the

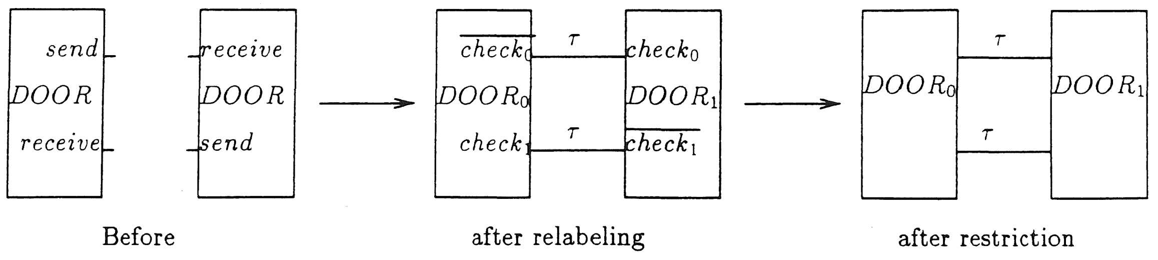


Figure 14: Composition of agents in CCS.

empty string in classic automata theory. τ is also a complete action in the sense that it incorporate both input and output.

Restriction: which internalizes some actions. $\frac{E\xrightarrow{\alpha}E'}{E\backslash L\xrightarrow{\alpha}E'\backslash L}(\alpha,\overline{\alpha}\notin L)$. No change is caused if the action labels are not in the internalized ones.

Relabeling: which assigns new labels to actions according to some relabeling function, f. $\frac{E\xrightarrow{\alpha}E'}{E[f]\xrightarrow{f(\alpha)}E'[f]}$

Constants: which is used for defining agents. $\frac{E\xrightarrow{\alpha}E'}{A\xrightarrow{\alpha}E'}(A \overset{\text{def}}{=} E)$.

Only these rules are defined, other properties may be derived from them. For example, as in CIRCAL, value passing may be derived by defining CCS ports, each for a value.

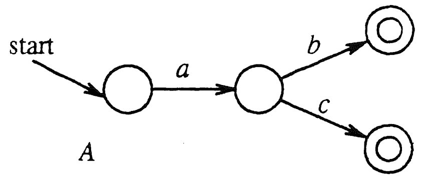
To see how these rules can be used, consider the two door example. Since, each door agent has to check the state of the other door agent before opening, suppose that each door has two ports *send* and *receive*. Using the relabeling, composition and restriction operators the problem can be represented as:

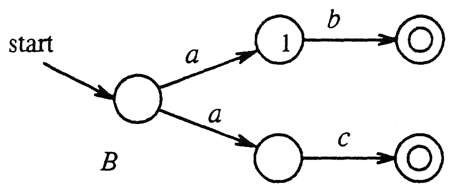
$$\begin{aligned}&(\ (DOOR_0[\overline{check_0}/send, check_1/receive])|\\&\quad(DOOR_1[\overline{check_1}/send, check_0/receive])\ )\backslash\{check_0, check_1\}\end{aligned}$$

Each *send* port becomes $\overline{check_i}$ and each *receive* port becomes $check_{i+1}$ ($i = i\ mod\ 2$). This means that a synchronization line combines two pairs of the former *send* and *receive* ports, each port from a different door. According to the third composition rule, each action on this line is internalized and is no longer observed by the environment. This procedure is described in figure 14.

It is easy to see from the basic definitions that simultaneous events can not be represented. The process evolution as captured in the two first parts of the composition rule, allows either process E or F to proceed. (Silent or complete actions (ℓ and $\overline{\ell}$) are actually one action.) There are several ways to overcome this disadvantage. CIRCAL, for example, overcome it by using dot calculus as a concurrent combinator instead of CCS composition.

The theory focuses on equivalence properties of processes. The basic term is *bisimulation* which was introduced by Park [Park80] and gives meaning to the ability of distinguishing between processes according to their external properties. Bisimulation is defined over a range of "equivalence" properties recursively over all system states. This means that two processes may be simulated if every state in one process can be matched to similar states in the other. The main distinction between the bisimulations concentrates on capturing the silent action, τ, i.e., whether τ is taken into account or not. In *Strong bisimulation*, τ is considered as a regular action whereas, in *weak bisimulation* or *observance equivalence* it is not. This allows some of the states in one to be matched with fewer in the other. Notice that this distinction makes a difference in the external behavior of the agent. For example, the following two agents:





are equal in the traditional automata theory, but are different even according to the weak bisimulation, since agent A will always accept the strings ab or ac, whereas B may not! (For example, if B is in state "1" after receiving a it will not receive c.) This means that an external observer can distinguish between the two agents behavior. Another type of bisimulation, *observation congruence,* requires each action in one process to be matched by at least one action in the other. There are other types of bisimulation but they are outside the scope of this paper.

The importance of CCS is not in its practical use (which is very limited). It is rather a theoretical handling of process control. It lays the foundation for many other methods that may be defined above it. In fact, many methods were developed in this way and applied successfully in various areas. CIRCAL is only one of them. LOTOS, [Brinksma88] which is a language for communication protocols is another example. Another study that refines the theory and introduces convenient ways for abstraction can be found in [BK85].

The bisimulation relations supply a basis for analyzing processes. For example, in CIRCAL, the proof that equation 16 is consistent with equations 11 to 15 means that the process that is represented in the first equation is a bisimulation with the process represented in the last ones. These relations are not restricted to the requirement stage but may be used for consistency checks later during the design. The power of this concept and its applicability is not restricted to a particular method. It is of general interest. For example some properties of CSP can be shown to be bisimulation.

The theoretical power of CCS is wide enough to accept general properties of processes. This is of great importance since many systems are specified in such terms. This has also safety significance. In fact, safety invariants or general invariants are part of *process logic* ($\mathcal{PL}$) which was developed for this purpose. Also, Manna and Pnueli temporal logic [MP83, Pnueli86] can be incorporated in $\mathcal{PL}$ [HS85].

3.3 CSP

CSP or Communicating Sequential Processes was first introduced in 1978 by Hoare [Hoare78] as a programming language for concurrent processes. Since then it was used intensively in many areas and evolved in many versions [OH86][18]. It is probably the most cited algebraic method. TAM, which was discussed above is one of these (indirect) versions. A more general discussion on the basics of the method can be found in [Hoare85].

As with other algebraic methods, a process is a sequence of possible events. All the events that a process can be engaged with, or the process *alphabet* (denoted αP for some process P) is a finite set. A process can be analyzed according to its *traces*, or its sequence of symbols recording the

[18]It is interesting to mention the differences in the basic approaches between CCS and CSP. The former is a theoretical approach for handling processes which is applicable to almost any environment, software or hardware. In contrast CSP evolved as a programming language which was later adapted to hardware areas. This has other implication as well. For example, CCS contains only basic definitions and leaves the specific refinements to be introduced at the application area. On the other hand, CSP has a very rich set of process operators and special events definitions only part of which are applied in any area.

events in which the process has engaged up to some moment. CSP includes two sets of operators for handling processes and traces. These operators form a convenient structure for expressing both the process and its specifications.

It is easy to see that traces form a partial order. In fact, it can be proved that for a given alphabet A a lower and upper bound can be defined. The lower bound is the empty string, i.e., the state before the machine was turned on. The trace of this is $\langle\rangle$. An upper bound of a process is one that can do everything but without any external influence or awareness. Such a process is actually "chaotic" and thus was given a name $CHAOS_A$.

To get more feeling about the use of the operators, consider the clean room example. Each door process can be divided into three sub-processes, $CLOSE_i$, $OPEN_i$ and NOT_i $(i = 0, 1)$, that represent the sequences of events that may occur while the door is closed, opened or notifying a user respectively. This can be represented as:

$$\begin{aligned}
&(DOOR_0 \| DOOR_1) = \mu X : A. \\
&\quad (\,(open\_but_0 \rightarrow (OPEN_0 \sqcap NOT_0; OPEN_0); CLOSE_0 \rightarrow X)| \\
&\quad\;\, (open\_but_1 \rightarrow (OPEN_1 \sqcap NOT_1; OPEN_1); CLOSE_1 \rightarrow X)\,)
\end{aligned}$$

This equation describes a recursive process that is composed of two $DOOR$ processes executing in parallel (denoted '$\|$'). The environment supplies the first event either $open\_but_0$ or $open\_but_1$ which initiate a sub-process in the corresponding door. This choice is deterministic and is denoted by '$|$'. The door might either open or notify the user that it can not do so. This is a nondeterministic choice which is denoted by '$\sqcap$'. The '$\rightarrow$' denotes the prefix of the process that is to its right side. The notation $\mu X : A$ shows that the process is recursive with alphabet $A = \alpha(DOOR_0 \| DOOR_1)$. X is a dummy process variable and the semicolon ';' means a successful termination of the process to the left and an immediate initiation of the process to the right. The safety requirement of the clean room problem can be represented as:

$$\mathcal{SPEC} \equiv (traces OPEN_i) \downarrow \{\alpha OPEN_{i+1}\} = 0 \quad (i = i \, mod \, 2)$$

This equation requires that the number of occurrences of events (denoted $\downarrow$) that are in the alphabet of process $OPEN_{i+1}$ in the trace of process $OPEN_i$ will be zero.

CSP operators are subjected to application laws. These laws are actually definitions of the operators. Operators may be used for showing that a process satisfies its specifications. In the clean room example this will be written as $DOOR_0 || DOOR_1$ **sat** $\mathcal{SPEC}$.

The parallel composition operator is of great importance. It allows two or more processes to execute concurrently as long as their alphabets are disjunctive. Mutual events have the same symbols and must executed simultaneously. If this does not occur, the processes deadlock.

The nondeterminism was already represented in the clean room representation. It is interesting to mention that CSP has two nondeterminism choices. The one shown means that the external environment is totally unaware of the internal process and thus can not influence it. The other operator (denoted $\Box$) gives the external environment limited influence by allowing it to choose the first event and then give up control.

Nondeterminism may be introduced by application of a hiding operator (denoted $\backslash$). This is similar to the restriction operator in CCS though there is a slight difference between the two. CCS leaves a τ sign in place of the restricted label, whereas nothing is left in CSP after the hiding takes place. This leads to another major difference in the possible evolution of a process as described in the two methods. In CCS The left τ can be used to guard the process and thus the process is still bounded even when it is completely internalized. A complete internal CSP process is not guarded at all and thus necessarily reduces to *CHAOS*.

CSP is a well developed method that has been used in many areas. Its application has been extended far beyond the original programming language and is still applied to many areas. The original operators form a wide basis for defining new special purpose ones (e.g., [Moore90]). The theory behind CSP is simple and enables these new definitions.

CSP is event driven and is concentrated around the actions and not system states. This implies that system requirements have to be specified in terms of events and not in terms of states. On the other hand many properties of reactive systems are specified in terms of states. In these cases an event interpretation may be hard to understand. For example, the safety specifications of the clean room are clearly understood in state terms "the two doors should not be opened simultaneously". This interpretation in event terms as shown in the $\mathcal{SPEC}$ definition is not straightforward.

Another problem that is not unique to CSP but exists in many algebraic methods is time application. Time has to be introduced through some external clock component. This is not natural. It is more common to find requirements specified in usual time units and not in term of computer cycles.

4 Summary

In this paper we presented several modern methods for behavior description and discussed their abilities in practical use. Some of the methods were discussed in more detail and others were briefly represented.

We restricted this survey to methods that can be used for behavior description that include time and concurrent capabilities. The presented methods include all three properties, i.e., they can be used for a black box description, they include capabilities for parallel composition of many processes and they can represent time passage and time restrictions that are imposed on the system. We are aware to the fact that there are other methods that are considered "behavior description", however, many of them do not include one or more of the other properties. It is also true that it is hard to draw a clear line between behavior description and design. In fact, two of the described methods, Petri nets and ESM/RTTL, may also be considered as design tools. However, we used them for describing the clean room behavior. In view of the above, almost every programming language that includes concurrent processes such as Ada, OCCAM, Concurrent Pascal, etc. can be used for behavior description. Therefore, our choice could not be independent of subjective discretion.

We tried to concentrate on methods that represent different approaches to the problem. Statecharts is probably the first method that coped successfully with the problem of exponential explosion of states and transitions. Modecharts and RTL represent an approach that maps time restrictions into the domain of predicate calculus in an attempt to prove existence or nonexistence or certain properties in a system. ESM/RTTL uses a different approach and makes use of temporal logic for establishing a mathematical theory which, in turn, serves as a basis for the proof scheme. Petri-nets is special in its representation of states and transitions and was widely used since its introduction. TAM and CIRCAL are algebraic methods that represent two different approaches. TAM uses a CSP structure enhanced with direct time representation, while CIRCAL is based on CCS and needs to define a "clock" agent for representing time. IC that was briefly discussed, uses time as its only parameter for determining interrelations among states or events.

It is almost self evident from the discussion that no method is a "silver bullet" for general use. It seems that more than one method are needed to describe and analyze complex reactive systems. If a set of methods is used other questions arise. For example, what combination of methods is best for a given purpose? how should they integrate?

Although many methods are more powerful than finite state machines their practical use is almost always restricted to handle that machine type. Further, since every **built** system is finite, it can be described as a finite state machine. Therefore, a developer can avoid being entangled in complex (high descriptive power) models and restrict himself to simple ones. The use of simple models does not imply that plain finite state machine should be used for now and ever. Ways for focusing attention, abstracting out detail and enabling analyses should be incorporated in each method.

Most of the methods claim to be "event driven". Is this an advantage in reactive systems? The answer to this question is not sharp. It is also not clear to what extent is a method event driven. It seems that behavior of reactive systems is governed by a combination of events and conditions. This means that both, events and conditions, matter and both need to be represented and analyzed. In many cases the conditions are very complicated and thus are more problematic than the pure events. As shown, state based methods can represent such conditions, but the analysis is still difficult.

Process based methods are almost pure event driven. Systems are represented as sequences of events executing sequentially or concurrently. As a result, the description of branching becomes cumbersome when it is state dependent. (Even a simple systems, such as the clean room, branching became less comprehensible in TAM and CIRCAL representation.) This limits the practical use of these methods.

Process based methods are usually less pictorial[19] or at least their developers did not attempt to introduce pictorial tools. Although pictorial representation is not always superior to programming-like languages [Green77, FG79, Green86] it seems that in reactive systems it is superior. It might be that the reason for this is the high interaction among interdisciplinary personal that are used to pictorial representations. Almost every engineering discipline uses pictorial tools for solving difficult control problems. For exam-

[19]"Pictorial" is used here in a very wide meaning. For example, tables, decision trees, etc. are considered to be pictorial.

ple, figures of system parts and their projections are used as communication media in mechanical engineering, "blue prints" or block diagrams are used in electrical engineering. Even "pure physics" uses Finmam diagrams to reason about quantum transitions. Notice also that control of reactive systems is basically a mapping of the behavior of each subsystem or component. This means that the control flow matches the system structure to a certain extent. Such structures are almost always given in a pictorial description.

Another difficulty in using algebraic methods is their own complexity. When many operators are used (as in CSP) the expressive power increases, but the ability to analyze decreases. This is also the reason that methods that apply many operators (many "symbols") tend to be more human dependent. The more experienced user, the better the results.

Despite the above critique on process based methods they seem to be promising at least in safety critical systems. In such systems it is desirable to prove that the safety property is consistent with the system behavior. Process-based methods have the property that proofs may be generated from the requirements specification. However, the expressive power of process-based methods is relatively weak, making the proofs overly complicated. A combination of process-based and state-based methods may increase the expressive power of the representation, thus making simpler safety proofs possible.

Another factor that was briefly discussed in Statecharts is the way a method handles unexpected events. There are two problems with these events. The first is their representation. It is obvious that handling such events will increase the number of states and transitions. As was shown, even methods as Statecharts or Modecharts are unable to cope with this problem efficiently. We are not aware of any method that handles unexpected events efficiently.

The other problem with unexpected events is their generation. Jaffe at. al. [JLHM91] use a plain finite state machine for listing and categorizing unexpected events. Others use simple check lists [MMW84, Tuma84]. These approaches are general and obviously help the user in creating unexpected events. The problem with such approaches is that the user can very easily overlook many of the events. Even if we consider the above approaches as complete ones, i.e., containing all the possible unexpected events of a system, we still miss the incorporation of these ideas into a development method that will oblige their consideration. Not that the user will have to specify every unexpected event (this is certainly impractical in large systems), but he will be obliged to consider them.

The above discussion gains importance when dealing with safety critical systems. In such systems, catastrophe has to be avoided at all costs. It is well known [Leveson86] that unexpected events highly contribute to catastrophe development, and therefore, they have to be modeled into the system.

Time restriction is another point that is worth discussion. Every method has its own difficulties in representing and analyzing time constraints. RTL and temporal logic use operators that transfer temporal markers into another mathematical domain. As a result, time restrictions and other requirements are handled similarly, i.e., as predicates in the calculus, or axioms in a theory. This has an obvious advantage since a single methodology is used for all system constraints. The disadvantage of this attitude is the high complexity and high sensitivity to errors.

The intuition and simplicity of interval calculus has a great appeal for being used for time description. It can certainly be analyzed. The disadvantage of this method is that only time descriptions can be handled, and it is difficult to generalize it. It might be used in cases where every requirement has time constraints. It is also possible to use this method in combination with others.

There is still another factor that may be considered, simulation capabilities. This property is of particular importance in areas where a strong human machine interface exists (but is not restricted to this area). In view of this, it is also of significant safety importance. Many famous catastrophes were related to human errors. Three mile island and Chernobyl are two examples. Statecharts and Petri nets can be easily simulated and may be used for this system type.

References

[Allen83] J. F. Allen, **"Maintaining Knowledge about Temporal Intervals"**. *Communication Of The A.C.M.*, vol. 26, pp. 832-843, 1983.

[AVD76] P. Azema, R. Valette and M. Diaz, **"Petri-nets as a Common tool for Design Verification and Hardware Simulation"**. In

Proceedings of the 13-th IEEE Design Automation Conference, pp. 109-116, June, 1976.

[BK85] J. A. Bergstra and J. W. Klop, "**Algebra for Communication Processes with Abstraction**". *Journal of Theoretical Computer Science,* vol. 37. pp. 77-121, 1985.

[Brinksma88] E. Brinksma, "**Information processing Systems – Open Systems Interconnection – LOTOS – A Formal Description Technique Bases Upon the Temporal Ordering of Observational Behavior**". *Draft International Standard*, ISO8807, 1988.

[FG79] M. J. Fitter and T. R. G. Green, "**When Do Diagrams Make Good Computer Language?**" *Int. Journal of Man-Machine Studies,* vol. 11, pp. 235-261, 1979.

[GL89] R. Gerber and I. Lee, "**Communicating Shared Resources: A Model for Distributed Real-Time Systems**". Proceedings of the 1989 Real-Time Systems Symposium.

[GLZ88] R. Gerber, I. Lee and A. Zwarico, "**A Complete Axiomatization of Real-Time Processes**". Technical Report MS-CIS-88-88 GRASP LAB 162, Department of Computer and Information Science, University of Pennsylvania, Philadelphia.

[GMMP89] C. Ghessi, D. Mandrioli, S. Morasca and M. Pezze, "**A General Way to Put Time in Petri Nets**".

[Green77] T. R. G. Green, "**Conditional Program Statements and Their Comprehensibility to Professional Programmers**". *Journal of Occupational Psychology,* vol. 50, pp. 93-109, 1977.

[Green86] T. R. G. Green, "**Design and Use of Programming Language**". *Software System Design Methods,* vol. F22, Springer-Verlag, 1986.

[Harel86] David Harel, "**Statecharts: A Visual Formalism for Complex Systems**". *Science of Computer Programming,* 8, 231-274, 1987.

[Hennessy85] M. Hennessy, "**Acceptance Trees**". J. ACM, vol. 32, pp. 896-928, 1985.

[HGdR88] C. Huizing, R. Gerth and W. P. de Roever, "**Modelling Statecharts Behaviour in a Fully Abstract Way**". *Technical Report CSN88/07,* Eindhoven University of Technology, Department of Mathematics and Computer Science, The Netherlands. July, 1988.

[HLN88] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman and A. Shtul-Trauring, "**STATEMATE: A Working Environment for the Development of Complex Reactive Systems**". In Proceedings of *The 10-th International Conference on Software Engineering,* April 11-15, 1988, Singapore, pp. 396-406, 1988.

[HM80] M. Hennessy and R. Milner, "**On Observing Nondeterminism and Concurrency**". *Lecture Notes in Computer Science,* vol. 85, Springer-Verlag, New-York, 1980.

[HM85] M. Hennessy and R. Milner, "**Algebraic Laws for Nondeterminism and Concurrency**". *Journal of the ACM,* vol. 32, pp. 137-161, 1985.

[Hoare78] C. A. R. Hoare, "**Communicating Sequential Processes**". *Communications of the ACM,* vol. 21(8), pp. 666-667, 1987

[Hoare85] C. A. R. Hoare, "**Communicating Sequential Processes**". Prentice Hall, 1985.

[HRdR88] J. Hooman, S. Ramesh and W. P. de Roever, "**A Compositional Axiomatization of Statecharts: Soundness and Completeness**". *DESCARTES, Document D4-2-2-2, Project 937, Package 4,* October, 1988.

[HS85] M. C. Hennessy and C. P. Stirling, "**The Power of Future Perfect in Program Logics**". *Information and Control,* vol. 67, pp. 23-52, 1985.

[HV87] M. A. Holliday and M. K. Vernon, "**A Generalized Petri-Net Model for Performance Analysis**". *IEEE Transactions on Software Engineering,* SE-13(12) pp. 1297-1310, 1987.

[JLHM91] M. S. Jaffe, N. G. Leveson, M. Heimdahl and B. Melhart, "**Software Requirement Analysis Control for Real-Time Process-Control Systems**". *IEEE Transactions on Software Engineering,* to appear March, 1991.

[JM86] F. Jahanian and A. K. Mok, "**Safety Analysis of Timing Properties in Real-Time Systems**". *IEEE Transactions on Software Engineering,* SE-12, pp. 890-904, September, 1986.

[JM87] F. Jahanian and A. K. Mok, "**A Graph-Theoretic Approach for Timing Analysis and its Implementation**". *IEEE Transactions on Computers,* C-36, pp. 961-975, August, 1987.

[JM89] F. Jahanian and A. K. Mok, "**Modechart: A Specification Language for Real-Time Systems**". *IEEE Transactions on Software Engineering,* 1989.

[KMS84] J. Kramer, J. Magee, and M. Sloman, "**A Software Architecture for Distributed Control Systems**". *Automatica,* vol. 20(1), pp. 93-102, 1984.

[Ladkin86.1] P. B. Ladkin, "**Primitives and Units for Time Specifications**". . In *Proceeding of the AAAI-86*, pp. 354-359, Morgan Kaufmann, 1986.

[Ladkin86.2] P. B. Ladkin, "**Time Representation: A Taxonomy of Interval Relations**". In *Proceeding of the AAAI-86*, pp. 360-366, Morgan Kaufmann, 1986.

[Ladkin87] P. B. Ladkin, "**Specification of Time Dependencies and Synthesis of Concurrent Processes**". In *Proceeding of the 9-th International Conference on Software Engineering,* Monterey California, USA, pp. 106-115, 1987.

[Leveson86] N. G. Leveson, "**Software Safety: Why, What and How**". *Computing Surveys,* vol. 18(2), pp. 125-163, 1986.

[LP81] H. R. Lewis and C. H. Papadimitriou, **"Elements of the Theory of Computation"**. Prentice-Hall, Inc., 1981.

[LHHRO91] N. G. Leveson et. al. **"Experiences Using Statecharts on a Complex System Requirements Specification".**

[LS87] N. G. Leveson and J. L. Stolzy, **"Safety Analysis Using Petri Nets".** *IEEE Transactions on Software Engineering,* SE-13(3), pp. 386-397, 1987.

[LZ88] I. Lee and A. Zwarico, **"Timed Acceptances: A model of Time Dependent Processes".**

[MF76] P. M. Merlin and D. J. Farber, **"Recoverability of Communication Protocols–Implications of a Theoretical Study".** *IEEE Transactions of Communication,* vol. COM-24, pp. 1036-1043, 1976.

[MMW84] P. Middleton and S. McWeathy, **A Methodology For Improving Software Safety Assurance".** *Minutes of the Third Software System Working Group S^3WG,* Crystal-City (Arlington), USA, September, 1984.

[Moore90] A. P. Moore, **"The Specification and Verified Decomposition of System Requirements Using CSP".** *IEEE Transactions on Software Engineering,* SE-16(9), pp. 932-948, 1990.

[MP83] Z. Manna and A. Pnueli, **"How to Cook a Temporal Proof System for your Pet Language".** In *Proceedings of the 10-th Annual ACM Symposium on Principles of Programming Languages,* Austin, Texas, pp. 141-154, 1983.

[Merlin74] P. M. Merlin, **"A Study of the Recoverability of Computing Systems".** Ph.D. Dissertation, Department of Information and Computer Science, University of California, Irvine. 1974.

[Milne80] G. J. Milne **"The Representation of Communication and Concurrency".** *Rep. 4088,* Computer Science, California Institute of Technology, 1980.

[Milne82] G. J. Milne **"Abstraction and Nondeterminism in Concurrent Systems"**. In *Proceedings of 3rd International Conference on Distributed Computing Systems,* IEEE Computer Society Press, New-York, pp. 358-364, 1982.

[Milne83] G. J. Milne **"CIRCAL: A Calculus for Circuit Description"**. *INTEGRATION, the VLSI Journal 1,2 and 3,* pp. 121-160, 1983.

[Milne85] G. J. Milne **"CIRCAL and Representation of Communication Concurrency and Time"**. *ACM Transactions on Programming Languages and Systems,* vol. 7, pp. 270-298, April 1985.

[Milner80] R. Milner **"A Calculus of Communicating Systems"**. *Lecture Notes in Computer Science,* vol. 92, Springer-Verlag, New-York, 1980.

[Milner89] R. Milner **"Communication and Concurrency"**. Prentice Hall, 1989.

[Mok85] A. K. Mok, **"SARTOR-a Design Environment for Real-Time Systems"**. *Proceedings of 9-th IEEE COMPSAC,* Chicago, Illinois, pp. 174-181, 1985.

[OH86] E. R. Oldberg and C. A. R. Hoare, **"Specification–oriented Semantics for Communicating Processes"**. *Acta Informatica,* vol. 23(1), pp. 9-66, 1986.

[Ostroff88] J. S. Ostroff, **"Modular Reasoning in the ESM/RTTL Framework for Real-Time Systems"**. *Technical Report CS-88-03,* York University, North York, Ontario, Computer Science Department.

[Ostroff89] J. S. Ostroff, **"Temporal Logic for Real-Time Systems"**. Research Studies Press Ltd., John Wiley & Sons Inc., 1989.

[OW87] J. S. Ostroff and W. M. Wonham, **"State Machines, Temporal Logic and Control: a Framework for Discrete Event Systems"**. In *Proceedings of the 26th IEEE Conference on Decision and Control,* Los Angeles, California, pp. 681-686, 1987.

[Park80] D. M. R. Park, **"Concurrency and Automata on Infinite Sequences"**. *Lecture Notes in Computer Science,* vol. 104 Springer-Verlag, 1980.

[Peterson81] J. L. Peterson, **"Petri Net, Theory and Modeling of Systems"**. Englewood Cliffs, NJ: Prentice-Hall, 1981.

[Pnueli86] A. Pnueli, **"Applications of Temporal Logic to the Specification of Reactive Systems: A Survey of Current Trends"**. In J. de Bakker, W. P. de Roever and G. Rozenburg, editors, *Current Trends in Concurrency,* LNCS 244. Springer Verlag, 1986.

[Tuma84] F. A. Tuma, **"Verifying Software System Safety"**. *Minutes of the Third Software System Working Group S^3WG*, Crystal-City (Arlington), USA, September, 1984.

[VNG90] F. Vahid, S. Narayan and D. D. Gajski, **"Synthesis from Specifications: Basic Concepts"**. *Technical Report 90-03*, Information and Computer Science, UCI, 1990.

[ZJ89] P. Zave and D. Jackson, **"Practical Specification Techniques for Control-Oriented Systems"**. In G. X. Ritter, editor *Information Processing*, Elsevier Science Publishers, 1989.

[Zwarico88] A. Zwarico, **"Timed Acceptance: An Algebra of of Time Dependent Computing"**. PhD Thesis, Department of Computer and Information Science, University of Pennsylvania, Philadelphia.