Using 'Parallel Automaton' as a Single Notation to Specify, Design and Control small Computer Based Systems

H.G. Mendelbaum[1,2] , R.B. Yehezkael[1] (formerly Haskell)
[1]Jerusalem College of Technology - POB 16031 - Jerusalem 91160
[2]Univ. Paris V, Institut Universitaire de Technologie, 143-av. de Versailles, Paris 75016, France
Email : {mendel, rafi}@mail.jct.ac.il Fax 00-972-2-6751-200

Abstract

We present in this paper a methodology how to use 'Parallel Automaton' to set up the requirements, to specify and to execute small Computer Based Systems (CBS). A 'Parallel Automaton' is an extended form of the Mealy Machine. It handles a finite set of events (or variable conditions or clock conditions) which can occur in parallel, and performs a finite set of actions which can be done in parallel. In the 'Parallel Automaton' there is no concept of "global state" as in the Mealy Machine. Instead, to each action and each event, is associated a "private state" representing their occurrence in the application. Nevertheless, the number of events/actions private states is also finite.

This single notation ('Parallel Automaton' with Private States) can be used to describe in the same way requirements and specifications. More than that, these two descriptions can be connected. The aims of the application can be described using a 'Parallel Automaton', as a black-box with initial inputs and final outputs. This 'Parallel Automaton' can then be refined and enhanced with intermediate conditions and actions to obtain detailed requirements. By successive refinements and enhancements, a sufficiently detailed executable specification can be derived.

We present this methodology through a simple CBS example, for the requirements and the specifications using the 'Parallel Automaton' notation. We then give an architecture of a Virtual Machine that we have built to execute such a 'Parallel Automaton' on a network.

Keywords : *Extended Automata and Finite state Machines, EFSM, Parallel Automata, CBS methodology*

1. Introduction

Most CBS applications contain concurrent parts e.g. : hard/soft parts, network distribution, windows and data-bases software, etc.. Analysis and design methods (such as SFC/Petri-nets, UML, Corba, State-Charts etc.) contain notations and rules for specifications of concurrent parts. Modern languages (such as Ada, C++, Java etc.) now offer possibilities to program tasks or threads in current applications.

Many languages and tools have been proposed to describe the requirements, to specify the solution, to implement and execute systems. In fact different tools or languages or platforms have been proposed for each stage, and that introduces a supplementary degree of uncertainty : is the translation from one stage to the other correct ?

In our course of Real-Time Systems (at JCT-Jerusalem and at IUT-Paris), we teach how to use a simgle notation to specify, design and control CBS applications. This can be done using several methods cited above (SFC/Petri-nets, State-Charts), but this requires their translation into Ada, C++ or Java for execution.

We wanted a single tool which is able to describe all the stages of the development including the execution. For this we defined a new kind of generalized automata able to describe globally, applications which contain events/actions parallelism, synchronizations, and also timing or data constraints.

Here we propose this model ('Parallel Automaton') which can be used to describe the requirements, to specify, to design and also to control the CBS application. It can be implemented or simulated as a software scheduler for parallelism and synchronization, or as a hardware processor running a general parallel/synchronization automata table. It can be used to control applications or distribute algorithms in various changing conditions, by just changing the automata table.

2. A review of extended automata models

In the following, we shall present a bibliographic review and compare various proposals of extensions to sequential automata, in which parallelism, synchronization and timing features were introduced.

This review does not pretend to be exhaustive, but gives roughly the main kinds of possible extensions proposed until now.

Extensions to sequential automata were proposed in the literature as theoretical models, some authors study parallel extensions to automata to specify and/or analyze grammars describing sets of events (or symbols) e.g.[9][12][18] or set of processes e.g. [18][22]. Other authors view automata as specific kinds of geometrical spaces [17]. Others studied the rules of timed automata [19][20][21]. Other researchers studied the theoretical links between automata and temporal and/or linear logic[18][20].

All these theoretical models are important and we are interested in applying these models of extended automata in real parallel implementations. So, here, we shall review (not extensively) such extended automata which have been used for design of protocols, Web programming, Data-Bases, simulation graphics, chip-design, real-time or concurrent or distributed software etc.

2.1 A common representation for making comparisons

The problem is that each extended automaton proposed in the literature uses a different notation and a different mathematical abstraction. Each one has its own formalism and language.

So to facilitate a clearer comparison, we tried to express each kind of these proposed extended automata, using a common representation of automata, as extensions of Mealy State Machines, i.e. using sets of registers for events, states, actions etc... and a table containing transition functions of the minimal form :

event, state $\rightarrow$ action, newstate.

We shall try in this bibliographic review to classify the proposals into : extensions to state, time extensions and extensions to multiple events and actions.

2.2 Extending the state of the Automata

2.2.1 Adding Conditions to the state: In an early research (1974-77), we proposed a generalized model of Mealy Machine for the scheduling of synchronized processes for software, hardware and distributed systems [1-4]. This model has finite sets of events E{e_1,e_2,e_3...}, states S(s_1,s_2,s_3...}, actions A{a_1,a_2,a_3...} and boolean conditions C{c_1,c_2,c_3...}. The transition function of this extended automata is of the form

$e_m,s_k,c_n,!c_p$..etc.. $\rightarrow$ $a_i,s_r,!c_q,c_t$.. etc..

which means : <u>*when*</u> the event e_m arrives in the machine state s_k and if the condition c_n is true and the condition c_p is false ..etc.. <u>*then*</u> $\rightarrow$ perform the action a_i , put the automata in the state s_r, and the conditions c_q to false and c_t to true..etc..

This kind of automata has a global state, like the classical Mealy Machine, it helps in describing synchronizations, using the boolean conditions C. The receiving of parallel events is done by recording their arrivals each one in a given state of the machine, using conditions (boolean flags).

Associated with Petri-nets, this model of automata has been used for designing, describing and executing the process control of chemical plants. Nevertheless this model does not express the possibility of parallel execution of actions. And also, to take into account the parallel events, you need a condition flag c_n and a separate statement to record each event's arrival, and in addition, a statement to activate the action when all the events have already arrived, so it increases the number of states and statements.

2.2.2 Adding variables to the state EFSM

Other extensions to FSM have been proposed [e.g.: 5-8]. For instance, an *n*-dimensional linear space D of *n*-tuples can been added to the finite sets of events, states and actions. D is not necessary finite. An ordered pair <state,value of D> is called a *configuration* of the machine, a set of *configurations* is called a *region* .

The transition function of this automata is of the form

$e_m,s_k,d_n,\ldots \rightarrow a_i,s_r,d_t, \ldots$

For instance , in the case of a micro-controller [5], the space D can be made of a set of registers R, the events can be inputs I and the actions can be outputs O, so the transition-functions can be like :

$i_1, s_1,r_1<7 \rightarrow$ out:=r_1, s_0, r_1:= r_1+4

meaning that <u>*when*</u> the input i_1 is true and the machine is in the state s_1 , and if $r_1<7$ <u>*then*</u> $\rightarrow$ set the out:=r_1 , set the machine to the new state s_0, and increment r_1 such as r_1:= r_1+4.

This kind of automata too has a global state as regular FSM, but it helps in describing synchronizations using arithmetic conditions. It has been used in chip design, and in various protocol specification and analysis.

In this type of modelization, it seems difficult to describe the parallel receiving of events and the parallel execution of actions. Parallel events could be recorded using register values as the space D can be defined as a set of booleans, and then it would look like the preceding ‘automata with conditions’ model. In any way, we seek a model which easily expresses the possibility of parallel execution of actions.

2.2.3 Parallel graphs to represent multiple states

2.2.3.1 Stotts et al.[9] proposed a model of PFA (parallel finite automata) which is based on a modified interpretation of Petri-nets, it has a finite set of nodes (with initial and final nodes), a finite set of states (with initial states), a finite set of inputs that we call events in our common representation, a finite set of state transition-functions which are composed of node transitions, which can be written in the form : e_i,{n_2,n_5,..*etc*..} $\rightarrow$ {n_4,n_6,..*etc* }

This means that <u>*when*</u> the input e_i arrives in the state "where the nodes {n_2,n_5,..*etc*..} are active", <u>*then*</u> deactivate the nodes {n_2,n_5,..*etc*..} and then activate the nodes {n_4,n_6,..*etc* }.

In fact, this model (which is an extension of the Moore automata) seems to extend the concept of a unique machine state, but here the state is represented by several nodes which can be active in parallel, when an event occurs. The transition-functions perform a unique action, and switches the state of the machine by activating new node(s).

This model accepts a string of sequential inputs and treats it with a sequence of actions of parallel nodes activity. This 'Parallel Automaton' has been used [10-11] in multi-Web applications and in Hyperdocuments treatment. It is interesting, because it brings the idea of multiple-states, but it does not answer our search for parallel multiple events and actions.

2.2.3.2 Badler et al.[13-14] use also an extension of Petri-nets called PAT-NETS (Parallel Transition Networks) for the representation of the movements of human bodies in virtual reality. Each part of the bodies can move in parallel, but in synchronization. In this extension of automata, they represent the parallel moves using a parallel graph which shows also an extension of the global state concept to simultaneous states. This a good answer for graphical parallelism expression, but needs a translator to execute it. We prefer an 'automata description' which can be directly executed.

2.2.4 Time extension to the automata state

Alur and Dill [19] proposed to use "timed automata" to model the behavior of real time systems. Clocks are added to finite automata and timing constraints are put on the arcs of its state transition diagram. In our tentative common notation it could be represented by

$$e_q, s_p, \mathrm{cond}_n(\mathrm{clock}_m) \rightarrow a_k, s_j, \mathrm{reset}(\mathrm{clock}_i).$$

which means ***when*** the input e_q arrives in the machine state s_p and the condition$_n$ of the clock$_m$ is verified , ***then*** $\rightarrow$ do the action a_k ,put the global state to s_j, and reset the clock$_i$.

(in this interpretation, we do not use time-invariant conditions inside the states).

All clocks start at zero, they progress at the same rate but they may be independently reset to zero. So as to enable timed automata to be converted to classical untimed automata, restrictions are put on the timing constraints. (Clocks can only be compared against constants for the most part and adding clocks time together is not allowed etc.) The region and zone constructions are used for making this conversion [19]. Closure decidability and verification are issues discussed. As timed automata may be converted to untimed automata, existing minimization and testing techniques may be applied or adapted to timed automata - e.g. Bloch, Fouchal, Petijean[15, 23], Springintveld et al. [16]. Another approach for testing timed automata proposed by Laroussinie et al. [20] is to convert an automaton to a characteristic formula in a timed logic, and then use model checking techniques for verification. These proposals are only time-extension oriented, we want also to express and execute parallel actions responding to parallel events.

2.3 Extending Automata for several events and multiple actions (I/O automata)

Bob Harms [12] proposed an extended automaton that can take into account the arrival of several events, for this he used an extension of a Turing Machine which can read, each time, characters coming from several tapes in parallel. The machine has one global state, and a memory with statements of the form :

$$ev_{gr}, ev_{ph}, st_j \rightarrow act_i, st_k$$

He used such a machine to model the human language, in which you have to take into account both the grammar (ev_{gr}) and the phonology (ev_{ph}) of a sentence. This very simple and nice, but in our case (real-time systems), we need also an extension to take into account timing and variable condition and multiple actions.

Nancy Lynch [24] has used an extension of automata formalism using multiple inputs, timers and variable conditions, and multiple outputs. But it is rather a formalisation of distributed algorithms, than an executable automata specification.

2.7 Summary

In the litterature review that we have presented, we saw various kinds of extensions to the Mealy model, we found extensions to automata to express parallelism of events[12], parallelism of actions and synchronizations [1-4, 15-16, 24], expression of constraints on time [19-24] and data[5-8]. But we did not find <u>all</u> the extensions needed for our purpose of specification automata which also executable.

3. Our proposal : the 'Parallel Automaton'

For parallel or distributed applications, each branch of a parallel application or each processor of a distributed application has its own behavior, and can be described separately by a different automaton with its own local states. So a parallel application can be represented by a set of several (simple) interacting automata. But in this case, we risk coming back to the problem of the complexity of the description of the synchronizations between the various interacting automata, and to the problem of verifying that their interactions produce an execution corresponding to the global requirements of the application.

This brings us to the idea of a single global automaton describing globally the parallel/distributed application, its requirements, its groupings parallel events and their synchronizations, and all the parallel actions. So all these descriptions are done in one representation. It is likely to be easier to reason on it, because everything is described in one representation. Supplementary statements for handling the interaction between several independent automata are not needed (see our example in chapter 4).

But this obliges us to define a new kind of automaton which allows to describe the receiving of multiple events in parallel, and the activation in parallel of multiple actions. The problem is that the number of states of such a global automata would explode because of three causes :

(1) to take into account the synchronizations between several events

(2) to differentiate the same actions/events that occur in various different situations in the application

(3) to take into account all the possible values of variables and clocks.

3.1 The notion of a machine with private states

We propose that the global automaton would not have the feature of global state, but should take into account parallel events and actions *each one in their own private states* (to differentiate various branches or events or actions, or processors). This would avoid the explosion of the number of the states, since it would have the feature to deal separately with the 3 above points : event synchronization (in their private states), event/action differentiation (each state would then be just the occurrence number of the event/action), variable/clock values (each variable/clock would change its state only when they are required in a new case by the application).

This bring us to the idea of a single automaton describing globally the parallel/distributed application corresponding to its requirements, but with private states for its events and actions.

Remarks: It could be seen as a paradox, that a good way to describe a parallel/distributed application is to use a single (centralized) description, not several descriptions corresponding to the various parallel/distributed parts. But this way has advantages because it gives an overview of the global situation.

a) Regrouping the requirements allows to enumerate, reduce and solve, in an easier way, the interaction problems between the various branches (common events),

b) It can also reduce the necessary number of variables and clocks.

c) Finally there is no need to deal with the problem of differentiating the handling of the same events/actions which can occur in various (synchronized) branches, during the progress of the application.

3.2 Definition of a 'Parallel Automaton'

Suppose we have a finite set of events (or conditions) which can be received in parallel, a finite set of actions which can be performed in parallel. Events can be valued or not, that is they can be Dirac signals (triggers) or conditions (i.e. changes in values of variables and timers).

To each action and each event, is associated a "private state" representing their occurrence number in the application. The number of states is also finite. This is a 'Parallel Automaton'.

The execution of a 'Parallel Automaton'can be viewed as a set of several threads, each one being a succession of actions and events, according to their private states, each thread is of the form :

$$\text{act1} \xrightarrow{\substack{\text{applic}\\ \text{produces}}} /\text{evt1,PS1}/ \xrightarrow{\substack{\text{automat}\\ \text{performs}}} \text{act2} \xrightarrow{\substack{\text{applic}\\ \text{produces}}} /\text{evt2,PS2}/ \xrightarrow{\substack{\text{automat}\\ \text{performs}}} \text{act3}$$

which means that an action act1 produces an event evt1 associated with its private state PS1, in the application, and the occurrence of the pair /evt1,PS1/ in the automaton will cause the activation of the action act2, and so on.

The private state will change only in two cases :

- Each time when the same pair act1 $\rightarrow$ /evt1,PS1/ occurs in different situations,
- When the same variable or when the same clock is reset and used in different situations of the application, so their respective private state will change.

This paradigm of state-change can also be suited to the applications containing several parallel branches of events and actions, each branch having its own private state. This comes in place of the Mealy model in which the execution is sequential with one unique global state.

On the other hand, synchronizations can be described by a product of pairs $/evt_i,PS_i/$ without changing the states to record the arrival of the events. And there is no need at all for a global state in the automaton, i.e. for the whole application, only private states for each couple event/action or for each branch in the whole application .

Each transition of the generalized 'Parallel Automaton'-table is written in a product form:

$$\boxed{\pi_i\ /cond_i, LocalState_i/ \rightarrow \pi_k\ /action_k, newLocalState_k/}$$

$\underline{cond_i}$ are boolean relations, it can be an event, a signal or an input flag (true or false) e.g "evt1" or "!evt2", it can be a variable condition e.g. "v != 10", or it can be a clock condition e.g. "100<=clock(x)<=200".

$\underline{action_k}$ are execution of actions, it can be an output flag e.g. "out3", it can be the execution of a function e.g. "sendEvt(to)" or " changeState(g)", it can be the setting of a value to a variable e.g. "setvar(v, 18)", or it can be the setting of a value to a clock e.g "setclock(b , 100)" or . "resetclock(t)".

Remarks :

1) In order to control the timing of execution, the 'Parallel Automaton' can be synchronous, in the sense that it is activated at intervals of time Δt , at each time Δt, all the events (variable conditions, clocks, in their respective states) are taken into account, the automata-table is scanned, all the corresponding actions are performed simultaneously and must finish before the next Δt. This means that there is one internal timer dealing with the scanning of the automaton, and external clocks used to measure the progression of the application.

2) For analysis purposes, we do not deal with the progressing of the clocks - except that when a clock is tested it gets a value (as in quantum physics, the particle are supposed to exist only when they are observed). In the same way variables get their values only when used, and the occurrence of events are tested only when you need them in specific situations. This curious assumption, does not change the reality, nor the control of the application, but it permits to considerably reduce the explosion of states, since you consider new states only when the controller needs them.

R3: A subset of the Parallel Automaton is the Mealy machine in which there is only <u>one state</u> register (this means that the machine is running only <u>one thread</u> of execution, and that it has one (global) state) :

$$/event_1, state_2/ \quad \rightarrow \quad /action_{10}, state_{12}/$$

On the other hand, the scheduling of parallel/distributed applications, which have a finite set of simultaneous branches of executions, can be written in such 'Parallel Automaton' forms, using a finite <u>set of state registers</u>, each one, for each parallel branch, or for each pair event/action.

4. An example using a parallel automaton

Let's take the Blood Test Machine, a simplified example, which contain parallelism and synchronization

4.1 Informal Description of the problem

Build a blood-test machine which can test Sugar, Cholesterol and Creatinin. The Nurse takes the blood and the data from the patient (name, address, doctor, date etc…,). The blood tube is introduced with an identification number in the machine, this blood is automatically divided (by a dispatcher) in 3 portions in order to perform the tests . The tests are made in parallel (but with various timings). The results are sent to the printer for the report, and to the cashier for the billing.

4.2 Using a 'Parallel Automaton' to describe the requirements

In order to translate the informal problem-description into a formal notation of requirements using a 'Parallel Automaton' form, we need to define sets of events, variables, conditions, actions, and clocks. Then describe the aims of the system in terms of 'Parallel Automaton'. At the first stage (aims and general requirements) you don't need to define all the events, variables, actions etc .., only the ones which are necessary to describe the initial inputs of the system (as a black-box), its main operations and its final outputs :

For example, for the informal description of the BloodTestMachine, we can define the following valued <u>Events</u> : *PatientEv* and *resultsEv*, to which can be associated the following <u>Variables</u> : *Data* (for the *PatientEv* event), *Crea, Chol* and *Sugar* (for the *resultsEv* event), and also we can define the following <u>Actions</u> : *CreaTest, CholTest* and *SugarTest, printer and cashier.*

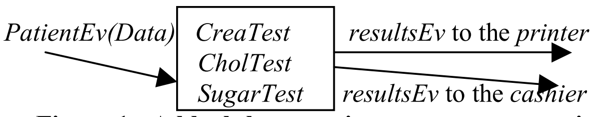

Figure 1: A black-box requirements representation

So, the Parallel Atomaton translation of the informal description (Figure 1) of the BloodTestMachine can be :

[1] /PatientEv(Data), 0/ → / *CreaTest*,1/ / *CholTest*,2/ / *SugarTest*,3/

[2] /*resultsEv*(*Crea*),1//*resultsEv*(*Chol*),2//*resultsEv*(*Sugr*),3)/ → /*printer*(*Data*,*resultsEv*(*Crea*),*resultsEv*(*Chol*), *resultsEv*(*Sugar*)),1/ / *cashier*(*Data)*,2/

The first line *[1]* means that when the Automaton receives the event *PatientEv* with the value *Data*, it will activate the 3 parallel tests *CreaTest, CholTest* and *SugarTest,* each one in its own private state. The second line *[2]* means that when the Automaton has received the 3 valued events *resultsEv* (each one in the private state corresponding to the respective tests), it will activate in parallel the *printer* and the *cashier* with the values of the results. This is in fact a behavioural "goal" description of the requirements, the declarative description is not covered in this paper.

4.3 Using a 'Parallel Automaton' to describe the Successive specifications

Now, by successive refinements, we can enrich this general description (goal requirements), adding more detailed events and intermediate actions, or variables, conditions and clocks. For instance, We can replace the line *[1]* by two lines such as :

[1] /PatientEv(Data), 0/ → /*divide*,0/

[1'] /divided,0/ → / *CreaTest*,1/ / *CholTest*,2/ / *SugarTest*,3/

To indicate that before performing the tests, you need to divide the blood in 3 parts.

Then you can add some timing to the tests by replacing the *[1']* line by

[1"] /*divided*,0/ → /*CreaTest*,1/ /*resetclock*(*Cr*),1/
/*CholTest*,2/ /*resetclock*(*Ch*),2/
/*SugarTest*,3/ /*resetclock*(*Su*),3/

And then add the time constraints to the line *[2]* :

[2] /*resultsEv*(*Crea*),1/ /*clock*(*Cr*)=5, 1/
/*resultsEv*(*Chol*),2//*clock*(*Ch*)=3, 2/
/*resultsEv*(*Sugar*),3//*clock*(*Su*)=2, 3/
→ /*printer*(*Data*,*resultsEv*(*Crea*),*resultsEv*(*Chol*),
resultsEv(*Sugar*)),1/ / *cashier*(*Data)*,2/

So, we shall get a full specification :

[1] /*PatientEv(Data)*, 0/ → /*divide*,0/
[1"] /*divided*,0/ → /*CreaTest*,1/ /*resetclock*(*Cr*),1/
/*CholTest*,2/ /*resetclock*(*Ch*),2/
/*SugarTest*,3/ /*resetclock*(*Su*),3/
[2] /*resultsEv*(*Crea*),1/ /*clock*(*Cr*)=5, 1/
/*resultsEv*(*Chol*),2//*clock*(*Ch*)=3, 2/
/*resultsEv*(*Sugar*),3//*clock*(*Su*)=2, 3/
→ /*printer*(*Data*,*resultsEv*(*Crea*),*resultsEv*(*Chol*),
resultsEv(*Sugar*)),1/ / *cashier*(*Data)*,2/
[3] /*print_end* , 1/ /*cash_end*,2/ → /finish, 0/

Notes: The private states of the clocks are changed only when these same clocks are tested in different situations.

The 1st line *[0]* initializes the machine for the first time. In the 2nd line *[1]*, when the *PatientEv* event occurs with its *Data* , the machine divides the blood into 3 parts. In the 3rd line *[1"]*, when the event *divided* arrives, the machine starts in parallel the 3 tests each one with its respective clock, we mark each parallel branch by a different private state. In the 4th line *[2]*, when the parallel events *resultsEv* come with their respective timing conditions (each one in its private state), the automaton activates the *printer* and the *cashier* passing them their parameters. The last line *[3]* re-initializes the machine when the *printer* and the *cashier* have both sent their *end* events.

We can use this PARALLEL AUTOMATON as a unifying notation along all the stages of the development process :

[i] to define its requirements,
[ii] then to specify the solution,
[iii] then as an execution tool (hard and/or soft) which implements this solution,

5. 'Parallel Automaton' as an execution tool

The specification as a Parallel Automaton can also serve as a solution to build the application. Furthermore this automaton can help in simulating it in a rapid prototyping.

For this, a Virtual Machine can be designed as a platform to run the 'Parallel Automaton'.

This can be a platform for a single processor and in that case the parallel actions will be handled as time-shared threads, or the platform can be designed to run the application on a network with several machines, and in this case we can distribute the parallel actions on various processors and run them as real parallel processes.

We have built such a network platform to emulate the 'Parallel Automaton' on several machines. Here is the architecture of the emulation.

The 'Parallel Automaton' can be used as an interpreter to execute the applications, for this it is necessary to build a Virtual Machine as a run-time platform.

There are several ways how to implement such a Virtual Machine , let us give one that we have developped on a star-syle network:

The Virtual Machine is made of 4 parallel threads for controlling the 'Parallel Automaton' on a main computer + *N* threads on *N* satellite computers (each one for each parallel action) :

4 parallel threads for the PARALLEL AUTOMATON VM on a main computer :

1- An "event Handler" receives the events from the network, each one with its private state, the Handler transforms the pair /E,S/ into a flag (using a reference-table: /E_m,S_n/ →F_k) and updates this flag in the "Arrived-Flag Vector".

2- A "clocks-conditions Handler" manages a set of clock registers (each one associated to a private state), each time the real-time clock ticks it increments the clock-registers and updates the "Arrived-Flag Vector" if a condition is reached (according to a reference-table

/$clock_m$(condition), S_n /→F_k).

3- A "variables-conditions Handler" manages a set of Variables (each one associated to a private state), each time an action-function modifies a variable, it updates the "Arrived-Flag Vector" if a condition is reached (according to a reference-table /$Variable_m$(condition), S_n /→F_k).

4-an "Automaton Processor" is activated each Δt by a timer when it searches in the "Automaton table" to activate the transitions for which the events arrived and the (variable and clock) conditions are true. The "Automaton table" is like a reference-table

($Mask_n$ → /$list_n$(action-functions,S/). Each Mask corresponds to the Arrived-Flag Vector relevant for each transition, if a Mask corresponds to the present "Arrived-Flag Vector", the "Automaton Processor" executes the action-functions indicated in the "Automaton table" for this transition, and updates the acknowledged flags of the Arrived-Flag Vector (then the satellite computers receive the acknowledgement of their events).

N action-threads on *N* satellite computers

The action-functions are distributed on the network on various computers working in parallel. The action-functions are activated when receiving (from the

"Automaton Processor") a message with its private state. When an action-function finishes its work, it sends back to the event-Handler computer an event with its private state.

Notes

1) In a fault-tolerant view, all the computers of the star-network have the same Virtual Machine and actions Threads, only one computer plays the role of the Main VM. If the Main Parallel Automaton Processor fails, the first Satellite computer, which sends an end-action event and receives back a net-error, will become the new main VM and will send a notify-message to all the other satellites. The other satellites will send him again their last not acknowledged events to update the actual lost Arrived-Flag Vector. If a satellite fails after receiving the order to perform an action, the main VM computer will know it by a time out exception, and will send the same action to do on another satellite.

2) So at the beginning, the global PARALLEL AUTOMATON of an application is split and coded in several reference-tables :

*> the event/flag table $/E_m,S_n/ \rightarrow F_k$

*> the clock/flag table $/clock_m(condition), S_n /\rightarrow F_k$

*> the variable/flag table

$/Variable_m(condition), S_n /\rightarrow F_k$

which are made of all the pairs $/cond_i,S_j/$ of the application, and serve to detect the occurrences of the various conditions and translate them in Flags of the "Arrived-Flag Vector".

*> And finally the PARALLEL AUTOMATON table

$Mask_n \rightarrow /list_n(action-functions),S/$

which serves to compare the various transition-Masks with the actual "Arrived-Flag Vector", in order to activate the corresponding relevant action-functions.

6. Conclusion

In this paper, we define a PARALLEL AUTOMATON and show how we use it in order to describe formally the requirements of Computer-Based Systems, to specify it, and to execute it. Our Engineering students used this tool on a lot of examples since 1995. They have built platforms to emulate a Virtual Machine running 'Parallel Automaton'. And we intend to build a prototype of co-processor based on this model.

The main advantages of this notation are :

- same notation at all steps of development: requirements, specifications, execution etc…
- it reduces the problem of validation when passing from each step to the next one,
- parallelism and private states reduce the number of automata statements,
- this notation is close to the way of work of automation-engineers (on controllers and PLC)

Acknowledgements : We want to thank for their work, help and useful remarks Dr Reuven Gallant, Ariel Lubish, Chaim Sulam and Arye-Chaim Teitelbaum, and the JCT's President for a special research grant.

7. Bibliography on related works

Automata with conditions

[1] H.G.Mendelbaum, F.Madaule "Automata as structured tools for real-time programming" IFAC/IFIP workshop on real-time programming, Griem Ed.,Boston, USA, 1975

[2] H.G.Mendelbaum, F.Madaule "a class of structured real-time systems centered on a descriptive nucleus" 1st IFAC/IFIP Symposium on software for computer control (SOCOCO-76), Tallinn,USSR,1976

[3] F.LeCalvez, F.Madaule, H.G.Mendelbaum " compiling Gaelic, a global real-time language" IFAC/IFIP workshop on real-time programming, Smedema Ed, Eindhoven, Holl, 1977

[4] R.Samuel, H.G.Mendelbaum, F.Madaule "a fault-tolerant distributed real-time machine" EUROMICRO, North-Holland Publ.,p.229, 1977

EFSM and ECFSM

[5] K.T.Cheng, A.S.Krishnakumar " automatic generation of functional vectors using extended state machine model" ACM trans on design automation of electronic systems, vol 1, n#1, jan 1996,p57-79

[6] M. Higushi " a study on verification methods for communication protocols modeled as ECFSM", PhD thesis, (Osaka univ), nov 1994, http://www-fujii.ics.es.osaka-u.ac.jp/~higuchi

[7] Teruo Higashino et al. "Deriving concurrent synchronous EFSM from protocol specifications in LOTOS", Trans. IEICE of Japan, 1999, http://www-fujii.ics.es.osaka-u.ac.jp/~higashino

[8] D.Cypher, D.Lee, W.Martin-Villalba, C.Prins, D.Su : "Formal specification, Verification and automatic test generation of ATM routing protocol: PNNI" Proc FORTE/PSTV'98,nov 1998, Paris

Parallel finite automata and parallel graphs

[9] D. Stotts,, W. Pugh "parallel finite state Automata for modeling concurrent software systems" Journal of systems and software, Elsevier science, vol 27, 1994, p27-43

[10] B.Ladd, M.Capps, D. Stotts, R. Furuta " MMM, Multi-head, Multi-tail, Mosaic, adding parallel automata to Web" Proc. 4th WWW conf, Boston, MA, dec 1995, p433-440

[11] D. Stotts, R. Furuta, JC.Ruiz " Hyperdocuments as automata", ACM Trans. On information systems", 1996 (http//:www.cs.unc.edu/~stotts)

[12] Bob Harms " two-level morphology as phonology (parallel automata, simultaneous rule application)", Texas linguistic Forum 35, fall '95 (harms@mail.utexas.edu)

[13] N.I. Badler et al "Behavioral control for real-time simulated human agents" Proc. 1995 Symp. on interactive 3D-Graphics, ACM press, New-York, USA, p.173-180

[14] R. Bindiganavale, B.J. Douville "C++ and Lisp PAT-nets (Parallel Transition Networks)",1995 ftp://ftp.cis.upenn.edu/pub/graphics/rama/patnets

Timed Automata

[15] S. Bloch, H. Fouchal et al. "Timed and Untimed Testing", univ. reims, 1999 Simon.Bloch@univ-reims.

[16] fr J. Springintvelt, F. Vaandrager, P.R. D'Argenio "Testing Timed Automata" cath. univ. Nijmegen, Netherlands, CSI-R9712, aug. 1997, fvaan@cs.kun.nl

Theory of extended automata

[17] pratt@cs.stanford.edu " Chu-spaces a model of concurrency"

[18] F. Moller,G.Birtwistle "Logics for concurrency : structure versus automata" Lecture notes in comp. Sc., Springer Publ., vol. 1043, ISBN 3-540-60915-6, 1996
[19] Rajeev Alur and David Dill " a theory of timed automata" Theoretical Computer Science 126:183-235, 1994
[20] F. Laroussinie, K.G. Larsen,.C. Weise "from timed automata to logic and back" BRICS, univ Aarhus, RS-95-2, ISSN 0909-0878, 1995 ftp ftp.brics.dk (cd pub/BRICS)
[21] Gupta/Henzinger/Jagadeesan : "Robust timed automata", Proc. intern. workshop HART'97, Maler ed.,Lecture Notes in Comp. Sc., vol 1201,p.331-345, Springer-Verlag, 1997
[22] Z. Manna, A.Pnueli "specification and verification of concurrent programs by ∀-automata" Proc. 14th ACM POPL, 1987
[23] Eric Petijean and Hacene Fouchal, "From Timed Automata to Testable UntimedAutomata", RESYCOM lab., Univ. Reims, France
[24] Nancy Lynch : "Distributed Algorithms", Morgan Kaufmann Publ.,1996

APPENDIX : Converting 'Parallel Automaton' to Sequential Ones

For verification purposes, it is useful to convert a PARALLEL AUTOMATON into a sequential one. This we describe informally.

1) Removal of "NOT".

A transition rule such as: $/!e_1 , s_1/ ...... \rightarrow /.. , ..//.. , ../$ .is replaced by a collection of transition rules:

$/E , S/ ...... \rightarrow /.. , ..//.. , ../ ....$

for all events (or conditions) and state pairs $/E , S/$ such that $/E , S/ \neq /e_1 , s_1/$.

2) A state of the sequential automata is a tuple of private states of the components of the 'Parallel Automaton'. An event of the sequential automata is a tuple of events handled by components of the 'Parallel Automaton', and similarly for actions/outputs.

The symbols "n_e" and "n_a" are used in the sequential automata to denote "no event received" or "no event produced" by components of the 'Parallel Automaton'. With this in mind, a transition rule such as:

$/e_1 , s_1//e_2 , s_2/ \rightarrow /a_1 , s_1'//a_2 , s_2'//a_3 , s_3'/$

is replaced by a collection of transition rules:

$(e_1 , e_2 , E_3 , E_4 ...... E_n) (s_1 , s_2 , S_3 , S_4 ...... S_n)$

$\rightarrow (a_1 ,a_2 , a_3 , n\_a ...... n\_a) (s_1' , s_2' , s_3' , S_4 ...... S_n)$

where S_3 to S_n range over all private states of components 3 to n of the 'Parallel Automaton' respectively, and E_3 to E_n range over all events including the null event "n_e". (Note there is no change to private states S_4 to S_n .)

3) Regarding timed 'Parallel Automaton', the above construction can be used to deal with all components of the transition rules *except* for those dealing with the time. But once this has been done, we will have a sequential timed automata and the algorithms described for example in [19] can be used to convert such a sequential timed automata into an untimed one.

Reducing the number of states

The above construction can produce a very large sequential finite automata from a 'Parallel Automaton'. Using only relevant private states and relevant event sets, the automata can be reduced significantly.

A relevant state is one which may be read at its next access. If the next access is a write or if it will be never be accessed in the future, its value is not relevant and so may be set to zero or omitted from the tuple. Relevant states may be determined by a path analysis of the parallel automaton. Initially we assume all private states are zero (not relevant).

Similarly, relevant event sets are those sets which are formed from the events on the left hand side of a single transition. Event sets which would activate more than one transition, are replaced by a sequence of the event sets of the left hand sides of the transitions. For verification, all orders of events arrival need to be tested, so event sets which activate several transitions may be ignored.

With this in mind, a transition rule such as:

$/e_1 , s_1//e_2 , s_2/ \rightarrow /a_1 , s_1'//a_2 , s_2'//a_3 , s_3'/$

is replaced by a collection of transition rules:

$\{e_1 , e_2\} (s_1 , s_2 , S_3 , S_4 ...... S_n)$

$\rightarrow (a_1 ,a_2 , a_3 , n\_a ...... n\_a) (s_1' , s_2' , s_3' , S_4 ...... S_n)$

where for a relevant state, S_k ranges over all private states of component k, and for a state which is not relevant, S_k is zero. (Note there is no change to private states S_4 to S_n .)

Relevant state and event sets are important in other settings, for example, to reduce the state explosion in the construction of product automata [19].

Notion of Conflict Free Automata

A conflict free automaton is one in which there can be no simultaneous read and write, or two writes (of different values) to the same location (private state, output, etc.). By examining the left hand sides of pairs of transition rules, we can determine if they may be active simultaneously. If they can be active simultaneously, the right hand sides are examined to see if they involve a read/write or write/write conflict. In this way, potential conflicts can be identified. (This is a strict approach and perhaps these conditions can be relaxed).

Testing conflict free automata should be simpler, since if transition rules can be active simultaneously, the end result does not depend on the order of activation. (In this way they bear similarities to deterministic sequential automata.) Thus all possible interleavings of concurrent activities need not be considered, one is enough.

The parallel automaton example in this paper is conflict free according the strict approach, even though we did not have this in mind when defining it.