Defining Parallel Automata and their Conflicts

H.G. Mendelbaum[1], R.B. Yehezkael[1] (formerly Haskell), T. Hirst[1], A. Teitelbaum[1], S. Bloch[1,2]
[1]Jerusalem College of Technology - POB 16031 - Jerusalem 91160
[2]Univ. Reims, RESYCOM, Reims, France
Email: {mendel, rafi}@mail.jct.ac.il simon.bloch@univ-reims.fr

Abstract: *We define and classify a family of parallel automata (for Real-Time and Telecommunication modeling) in the context of a synchronous execution. First, an abstract form of Parallel automata is proposed as a generalization of various "Extended-Finite-states-Machines" found in the literature. Then, two implementable forms of Parallel Automata are presented : A "global Parallel automaton with private states" and sets of " Synchronous and Hierarchic Parallel automata with local states". An example of application is presented with these two formalism.*
We also define and classify various types of possible conflicts that can occur in Parallel automata. An example shows an application with various kinds of conflicts and their possible correction.
In a companion paper [17], we have shown that a-priori detection of actual conflicts for parallel automata is P-space hard. In view of this, an approach for a-priori potential conflict detection is developed. The complexity of detecting potential conflicts is shown to be possible in polynomial time, if all automata conditions are conjunctions.
An a-posteriori testing methodology is presented, using an execution platform for Parallel Automata that prevents conflicts at execution time.

Keywords : *Parallel Automata, Conflicts in Parallel Automata, Extended-Finite-states-Machines, Timed Automata*

1. Introduction

For parallel or distributed applications in Real-Time and Telecommunication modeling, 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 sequential interacting automata. But in this case, the problem is the complexity of describing the synchronizations between the various interacting simple sequential automata, and also the problem is the verification that their interactions produce an execution without conflicts corresponding to the global requirements of the application.
This brings to the idea of describing the parallel/distributed application by a parallel automaton with its parallel events (and their synchronizations), and all the parallel actions. This approach leads to define a new kind of automaton that allows describing 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 (or branches)
(3) to take into account all the possible values of variables and clocks.
In an earlier work Mendelbaum and Yehezkael [1] introduced the concept and a notation for “timed parallel automata” and it was conjectured that the conflicts of such automata could be detected a priori
In this paper, related works on proposed extended automata models are first compared. The concept of “abstract parallel automaton” is described as a generalization of the main kinds of extensions of finite state machines.
Then various kinds of conflicts that can occur in parallel automata are discussed, such as: events, conditions, actions or variable-updates, which should not arrive in the same cycle of the automata scanning; each scan is performed, supposing the synchronous hypothesis [2] for the execution of these automata: i.e. each scan is done when receiving a periodical tick of the central clock, so that all the events, conditions and actions are treated completely during each indivisible periodic cycle.
A classification of conflicts is proposed and their solution is handled a-priori using theoretical results. and a posterioriusing SPHAX, an execution platform designed by Teitelbaum [18], for executing timed parallel automata..

2. Introducing the concept of Parallel Automaton

First, let us compare, in the literature, various proposals of extensions to sequential automata, in which parallelism, synchronization and timing features were introduced. All these extensions can be viewed 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 → action, newstate.
Extensions to sequential automata were proposed in the literature as theoretical models, which are important, but we are interested in applying these models of extended automata in parallel applications for real-time and telecommunication.

2.1 Adding Conditions to the state:

In an early research (1974-77), Mendelbaum[3] proposed a generalized model of Mealy Machine associated with Petri-nets, for the scheduling of synchronized processes of chemical plants.
This extended model adds to the Mealy Machine, a finite set of boolean conditions c_n. The transition function of this extended automata is of the form $e_m,s_k,c_n,!c_{p...} \rightarrow a_i,s_r,!c_q,c_{t...}$
This kind of automata has a global state, like the classical Mealy Machine, it helps in describing synchronizations, using the boolean conditions c_n as semaphores. The receiving of parallel events is done by recording their arrivals each one in a given state of the machine or using these conditions.

2.2 Adding variables to the state EFSM

Other extensions to FSM have been proposed [e.g. 4-7] : an *n*-dimensional linear space D can been added to the finite sets of events, states and actions.
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 [4], the space D can be made of a set of registers.
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.

2.3 Adding clocks to the states

Alur and Dill [8] 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.
These transitions could be represented by $e_q, s_p, cond_n(clock_m) \rightarrow a_k , s_j , reset(clock_i)$.
Timed automata may be converted to untimed automata, existing minimization and testing techniques may be applied or adapted to timed automata -see for instance Bloch et al. [9,10], Springintveld et al. [11].

2.4 Parallel graphs to represent multiple states

2.4.1 Stotts et al.[12] 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,
The transition-functions (= node transitions of the graph), can be written: $e_i,\{n_2,n_5,..etc..\} \rightarrow \{n_4,n_6,..etc\ \}$
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 an action and switches the state of the machine by activating new node(s).
2.4.2 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.

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

2.5.1 Bob Harms [15] 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 such as : $ev_{gr}, ev_{ph},st_j \rightarrow act_i, st_k$
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.
2.5.2 Nancy Lynch [16] has used an extension of automata formalism using multiple inputs, timers and variable conditions, and multiple outputs. She uses this rather as a formalization of distributed algorithms, than for building executable automata specification.

In this short literature review, we saw the main kinds of extensions to the Mealy model, we found extensions to automata using data variables[4-7] or conditions [3], or states[12-14], to express parallelism of events[15], parallelism of actions and synchronizations [15-16], expression of constraints on time [8-11].

2.6 Synthesis and Generalization of these Automata Extensions into an abstract form

In the original Mealy sequential machines the state variable is unique and global to the whole machine, it represents the stage that the application has reached at a certain point of the execution, and allows to differentiate the occurrence of events in different situations.
In a generalized representation of parallel applications, what is the meaning of state variable(s) ?

- ✓ If we see the application as a collection of parallel branches, we can say that each branch will have a local state variable to represent its progression and to differentiate the occurrence of events in this branch. Each branch can be considered as a sub-automata in the main automata of the application. In this case, the application will have a collection of state variables which can be associated each one to a different branch.
- ✓ If we see the application globally as a collection of parallel independent transitions "conditions → actions", without explicit branches, the state variables will only differentiate similar conditions which provoke different actions according to various situations in the whole application.

Any way, in a generalized form we can represent the state variables as ordinary data that can be tested in the conditions as other data, events or clock values, they would not be necessarily coupled with events as in classical sequential automata.
What we see from all the reviewed proposals of extension, is that, from a formal point of view, all the transition-functions of the automata can be represented as : Boolean compound condition $\rightarrow \pi_k$ / $assign_k$ /
meaning that **when** the Boolean compound condition holds (testing of external input signals, events, state variables, values of data , clock values etc…), **then** the parallel assignment $assign_k$ of values to a set of variables will be performed. All the events, states, variables, actions, clocks will be represented as valued variables.

a. the states will be considered themselves also as conditions and will be represented as general variables,

b. the actions will be considered as assignment of values to variables (data or clock values (for instance clock-reset), set of calls of actions (functions), output of signals or events, change in state variables, etc…)

An example of an abstract parallel automaton description of an application can be :,

/event1=1/ /event2=0/ /state1=2/ /clock1>100/ → /action3:=1/ /output3:=3/ /state2:=5/ /clock1:=0/

meaning:

when event1 arrived and not event2 while state1=2 at time clock1>100

then do action3, send output3 with value 3, set state2 to 5, and reset timer clock1

remarks **:**

1) In the above example the left hand side of the rules should be understood as a conjunction.

2) In order to control the timing of execution, the 'Parallel Automaton' has to be executed in a synchronous way, 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.

3) A subset of the Parallel Automaton is the Mealy machine in which there is only one state register, this means that the machine is running only one thread of execution, and that it has one (global) state. A typical rule of a Mealy automaton would be written in the form: /?event_received="$event_1$"/ / current_state="$state_2$"/ → /!$action_{10}$:=1/ /current_state:= "$state_{12}$"/

2.7 Implementable parallel automata derived from abstract parallel automata

There can be various implementations, for instance:

✓ **'Global Parallel Automaton with private states'** : an application can be represented globally as a unique parallel automaton which describes the whole parallel/distributed application corresponding to its requirements, and takes into account all the parallel events and actions *each one in their own private states* to differentiate various branches or events or actions, or processors. These events/actions private states enable to avoid the explosion of the number of states, since it allows to deal separately with the 3 above points: events synchronization (each one in its private state), event/action differentiation (each state would then be just the occurrence number of the event/action in its branch), variable/clock values (each variable/clock would change its state only when they are required in a new case by the application).

✓ **'Parallel Hierarchical Automata with local states' :** an application can be represented as a hierarchy of several parallel automata: *each one having a local automata state*, and representing a different parallel branch (or component) and a main automata synchronizing them. Each parallel automata can handle parallel events or actions using the automata local state. So, here also, there is no explosion of the number of states, since each hierarchical automata has his own local states and synchronizations.

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) There is no need to deal with the problems of differentiating the handling of the same events/actions that can occur in various (synchronized) branches, during the progress of the application.

c) Finally, it can also reduce the necessary number of variables and clocks.

2.7.1 Description of a 'Global Parallel Automaton with private states'

In the global parallel automata, each parallel action/event is coupled with a "private state" representing the action/event occurrence number in the application. The number of states is finite. There is no explosion of the number of states since the states are limited to each pair action/event.

Synchronizations can be described by a product of pairs /evt_i,privateS_i/ without changing the states to record the arrival of the events. 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 'Global Parallel Automaton'-table is written in a product form:

π_i /$cond_i$, $PrivateState_i$/ → π_k /$action_k$, $newPrivateState_k$/

which means that for each transition a set of parallel conditions $cond_i$ (each one in its own Private State) can provoke the execution of a set of parallel actions $action_k$ with new Private States.

Definitions :

$cond_i$ are boolean relations, it can be an event, an input signal or an input flag (true or false) noted for instance " ?evt1 ", it can be a variable condition e.g. "v >= 10", or it can be a clock condition e.g. "100<=clock(x)<=200".

$action_k$ are execution of actions, it can be an output flag e.g. " !out3 ", it can be the execution of a function , for instance send Event " ! go " 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(b)".

Translation to the abstract general form

For instance a transition such as / evt1 , 0 / /var1>3, 2/ → / send evt3 , 4/ /do act0, 5/
will be translated in an abstract form of : / ?evt1=1/ /s1=0/ /var1>3/ /s2=2/ → /evt3:=1/ /s3:=4/ /act0:=1/ /s1:=5/

2.7.2 Description of a set of 'Parallel Hierarchical Automata with local states'

if an application can be represented as a hierarchy of several parallel automata made of a main automata synchronizing sub-Automata. Each parallel automata has its own local state to manages its events/actions, and can activate sub-automata. So, here also, the number of states of each automata is finite. There is no explosion of the number of states, since each hierarchical automata has his own local states and synchronizations.

Here also, the local state of each automata will change, only in two cases :

- Each time when the same pair act1 → /evt1,S1/ occurs in different situations in the local automata,
- When the same variable or when the same clock is reset and used in different situations, in the local automata .

Each transition of a local 'Hierarchical Automaton'-table is written in a product form:

π_i /$cond_i$/ , LocalState → π_k /$action_k$/, π_n /$subAutom_n(subLocalState_m)$/ , newLocalState

which means that, for a local automata state, a set of parallel conditions $cond_i$ can provoke the execution of a set of parallel actions $action_k$ with a new Local automata State, and activate a set of sub-automata each one starting in its own local state $_m$.

The same definitions apply :

$cond_i$ are boolean relations, it can be an event, a signal or an input flag (true or false) , a variable condition or a clock condition .

$action_k$ are execution of actions, it can be an output flag or the execution of a function , or the setting of a value to a variable, or the setting of a value to a clock .

2.8 Examples of implemented parallel automata

Let's take the classical train/gate crossing control example: A railway is crossing a road, a gate lowers down when a train is passing to avoid accidents with cars on the road, and raises up, when the train has exited the crossing. The system was represented by Allur and Dill [8] as composed of three components (3 communicating independent timed-automata) working in parallel : the train, the gate and the controller.

To make the example more implentable and to separate clearly the events from the actions, we have added physical switches which send events to the automata (*Switch_TrainArrives, Switch_trainLeaves, Switch_gatedown, Switch_gateup*), orders which perform actions (for the train *let_in* , *don't_let_in* , for the gate *lower, raise*), and flags to synchronize the various automata (for the train *Approach,* i*sout, exited,* for the gate *isdown, isup*).

2.8.1 A solution using a set of "Synchronous Parallel hierarchical Automata with local states" representing the 3 automata corresponding to the components CONTROLLER, TRAIN, GATE, using three clocks x, y, z (one per component) :

π_i /$cond_i$/ , LocalState LSi → π_k /$action_k$/, π_n /$subAutom_n(subState_m)$/ /newLocalState LSj/

MAIN AUTOMATON :

1 /?*Init*/, LS 0 →/Train autom(LS 0)//Controller autom(LS 0)//Gate autom(LS 0)/,LS 0

TRAIN AUTOMATON

2 /?*Init*/, LS 0 → /! *don't_let_in* / LS 0
3 /? *Switch_TrainArrives*/, LS 0 → /! *Approach*/ /resetclock(x)/, LS 1
4 /?*isdown*/ /3<= clock(x)<=5/, LS 1 → /! *let_in*/, LS 2
5 /? *Switch_train leaves*/ /clock(x)>=5/, LS 2→ /! i*sout*/, LS 3
6 /?i*sout*/ / clock(x)>=5/, LS 3 → /! *exited*/ /! *don't_let_in* /, LS 0

CONTROLLER AUTOMATON :

7 /?*Approach*/, LS 0 → /resetclock(z)/, LS 1

8 /clock(z)=1/,	LS 1	→ /! *Lower/*,	LS 2
9 /?*exited*/,	LS 2	→ /resetclock(z)/,	LS 3
10 /clock(z)=1/,	LS 3	→ /! *raise/*,	LS 0
GATE AUTOMATON :			
11 /?*Lower/*,	LS 0	→ /resetclock(y)/,	LS 1
12 /?*Switch_gatedown*/ /clock(y)=1/,	LS 1	→ /! *isdown/*,	LS 2
13 /?*raise/*,	LS 2	→ /resetclock(y)/,	LS 3
14 /?*Switch_gateup*/ /1<= clock(y)<=2/,	LS 3	→ /! *isup/*,	LS 0

Explanation:

line 1: The MAIN AUTOMATON starts the 3 component-automata in their initial local states LS 0.

line 2: The TRAIN AUTOMATON starts by sending a *don't_let_in* event to the rail (meaning lighting a red semaphore, so that the trains cannot enter the crossing).

line 3: When a train arrives it activates on the rail the *Switch_TrainArrives*, this sends the *Approach* event to the CONTROLLER AUTOMATON and starts the clock x.

line 7: When the CONTROLLER AUTOMATON receives the *Approach* event , it starts its own clock z to wait for 1 minute, and then it sends the event *Lower* to the GATE AUTOMATON.

line 11: When the GATE AUTOMATON receives the *Lower* event, it starts its clock y to wait for 1 min and for the *Switch_gatedown* (meaning that the gate is already down). When these two conditions occur, it sends the event *isdown* to the TRAIN AUTOMATON.

line 4: When the TRAIN AUTOMATON detects the *isdown* event in a time between 3 min<= clock(x)<=5 min, it can send the event *let_in* to the rail (meaning lighting a green semaphore, so that the trains can now enter the crossing).

etc...

2.8.2 A solution using a "global parallel automaton with private states" representing the synchronizations of all the 3 components in one automata, using 2 clocks (*x* for the *train* and *y* for the *gate*):

	π_i /$cond_i$, PrivateState PS_i/	→ π_k /$action_k$, newPrivateState PS_k/
1	/? *Switch_TrainArrives*, PS 0/	→ /newState(*Switch_TrainArrives*), PS 1/ /! *don't_let_in*, PS 0/ /resetclock(x), PS 0/ /resetclock(y), PS 0/
2	/clock(y)=1, PS 0/	→ / ! *lower* , PS 0/ /resetclock(y), PS 1/
3	/?*Switch_gatedown*, PS 0/ /clock(y)=1, PS 1/ /3<= clock(x)<=5, PS 0/	→ / ! *let_in* , PS 0/
4	/? *Switch_train leaves*, PS 0/ /clock(x)>=5, PS 0/	→ /! is*out,* PS 0/
5	/? is*out*, PS 0/ /clock(x)>=5, PS 0/	→ /! *exit,* PS 0/ /! *don't_let_in* , PS 0/,
6	/? *exit*, PS 0/	→ /resetclock(*x*)), PS 1/
7	/clock(*x*)=1, PS 1/	→ / ! *raise* , PS 0/ /resetclock(*y*), PS 2/
8	/?*Switch_gateup,* , PS 0/ /1<= clock(y)<=2, PS 2/	→/newState(*Switch_TrainArrives*),PS 0/

Explanation:

In this type of notation, we don't need inter-automata synchronization events, such as *Approach, isdown, isup***, since the synchronizations are made by the global automaton itself.**

Line 1: When a train arrives (signal *Switch_TrainArrives)*, the system blocks the *Switch_TrainArrives* of another train by changing the private reception state to 1 , furthermore it prepares two clocks for measuring the timings of the train and of the gate , and it sends a *don't_let_in* event to the rail (meaning lighting a red semaphore, so that the trains cannot enter the crossing).

Line 2: the system waits for 1 min to send the event *lower* to the gate, the clock y is reset to prepare new gate timing (in its new private state 1). The gate lowers.

Line 3: after 1min more and when the gate sent *Switch_gatedown*., the system verifies if the train clock x is between 3min<= clock(x)<=5min, if yes, it sends the event *let_in* to the rail (meaning lighting a green semaphore, so that the trains can now enter the crossing).

etc...

3. Conflicts in Parallel automata

A major problem in the verification phase, is to detect and analyze conflicts caused by the parallelism. Parallel Automata are executed in a synchronous mode : So that at each cycle of the synchronous clock, all the lines of the parallel automata are scanned and eventually executed during this clock cycle, depending of the arrived events which occurred in the precedent cycle, or depending on the conditions (of variables or of time) which are true at this cycle. So, conflicts can appear during a given same cycle of time:

- when several contradictory events or conditions occur,
- when contradictory actions are performed or different values are assigned to the same variables.

In fact, in each case the conflict is detected at execution time when reading flags (events flags or condition flags), or when writing value to flags (actions or timers), or writing values in variables. So, we can analyse these conflicts as read/write (RW) and write/write (WW) conflicts. In the general case, it is doubtful if algorithms can be designed for detecting conflicts in parallel systems, but in the case of finite state parallel automata, we can give algorithms which can be used as a basis for developing verification and testing tools.
The main conflicts are:
1) RW conflict.
2) WW conflicts are of two kinds.
(i) A strong conflict occurs when two or more assignments of *different* values are made (pseudo) simultaneously to the same location.
(ii) A weak conflict occurs when two or more assignments of the *same* value are made (pseudo) simultaneously to the same location *We now consider three kinds conflict free parallel automata:*
1) Very strict conflict free parallel automata, do not have any RW and WW conflicts and do not need any external synchronization mechanism to ensure freedom from these conflicts. *We will not discuss this notion further here.*
2) Strict conflict free parallel automata do not have any (strong or weak) WW conflicts, and it is assumed that a RW conflict never occurs because of the external synchronization used, e.g. read and write cycles never overlap.
3) Lenient conflict free parallel automata, are like the strict ones, but may have weak conflicts. Weak conflicts should be reported as a warning, as it is up to the programmer or designer to decide whether or not it is possible for the application and hardware to run correctly with weak conflicts.

4. An example of an automaton with conflicts

Let us define a coffee, milk automatic vending machine in which :

- -there is a place to insert coins , the machine can recognize the coins to sum them up, each time a coin is inserted it produces the event *coin(A)* with the value A;
- -there are two buttons that give the events *coffee* or *milk* to choose the desired beverage, and a button that gives the event *cancel* to return the sum inserted;
- -there are two functions *pourCoffee* and *pourMilk* to give the desired beverage, and a function *returnMoney* to return the (remaining) inserted sum

Here are the rules of a parallel automaton without conflict error for it:

1 /? *coin(A)*, PS 0/ → /sum:=sum + A, 0/
2 /? *coffee*, PS 0/ /sum>=coffeeprice, PS 0/→ / *pourCoffee,* PS 0/ */putSugar,* PS 0/
/returnMoney(sum - coffeeprice), PS 0/ /sum:=0, PS 0/
3 /? *milk*, PS 0/ /sum>= milkprice, PS 0/ → / *pourmilk,* PS *0/ /putSugar,* PS *0/*
/returnMoney(sum - milkprice), PS 0/ /sum:=0, PS 0/
4 /? *cancel* , PS 0/ → / *returnMoney*(sum), PS 0/

the first line can be activated several times, each time a coin is inserted,
the 2nd or 3rd line can be activated when there is enough money,
the 4th line can be activated at any time, but will give back money or not according to the value of sum.

Let us modify the machine and add a button *capuccino* to obtain a coffee-milk beverage.
As a first idea, let us add a single supplementary line in the automaton specification, in order to activate in parallel the lines 2 and 3 :

5 /? *capuccino* , PS 0/ → / ! *coffee* , PS 0/ / ! *milk* , PS 0/

This shows how lack of attention to detail can introduce weak and strong conflicts in the execution of lines 2 and 3, which will run in parallel. Strong conflicts : wrong conditions will be tested on the prices, and wrong sums will be returned, weak conflict : putSugar will be activated simultaneously twice (depending on the hardware, it will actually put one or two sugar(s).
A better idea is to modify the initial automaton to avoid all the conflicts, by adding 3 lines:

5 /? *capuccino* , PS 0/ /sum>=capucprice, PS 0/→ / ! *coffee* , PS 1/ / ! *milk* , PS 1/
/ returnMoney(sum - capucprice), PS 0/
6 /? *coffee*, PS 1/ → / *pourCoffee, PS 0/ /putSugar, PS 0 /*
7 /? *milk*, PS 1/ → / *pourmilk, PS 0/* /sum:=0, PS 0/

by changing the private states of the *coffee* and *milk* events we can activate the 6th and the 7th lines in parallel instead of the lines 2 and 3 which will remain only sequential . So, changing (in line 5) the price condition to test and returning money only once. Then putting sugar and resetting the sum variable are made afterwards in line 6 and 7.

5. An execution platform for parallel automata for handling conflicts at run-time

Another way for conflict handling is building a parallel automata based executor, which prevents/alerts at the running stage, the execution of conflicts.

Definition:

Synchronized Parallel Hierarchic Automata executor (SPHAX)[18] is a robust execution platform for parallel-automata running. The **automata** and their interconnection are defined by a set of functions, and saved in a system table. The system **executor** scans the system table one time in a **cycle**, and reacts according to the rules of transitions defined. The reactions are executed sequentially, but the automata run in **parallel**, since all their transitions are executed during the same cycle.

The system structure is **hierarchic**. This means that an automaton may contain sub-automata in a state. The lifetime of the sub-automata is equal to the time delayed in the super-automata container state.

Implementation:

For an implementation, a Virtual Machine is designed as a platform to run the 'Parallel Automaton'.

The following components are defined for the SPHAX implementation:

- An *Automata-Table*, where all transitions of all automata are defined. The transitions are grouped by their trigger (a combination of events and conditions) at different states. Each group is called *Entry* and each *Entry* becomes a line in the *Automata-Table*.
- A set of S*ystem-Clocks*, that they automatically increments between the cycles. The user can only read or reset these clocks.
- An *Interrupt-Handler* that can receive either external events or data. The External events/data causes to emission of internal events or putting data to internal pipes. Internal events are saved in the *Global-Event-Flag-Array*.
- An *Automata-Processor*, which is activated at the beginning of each cycle. . The *Automata-Processor* scans the *Automata-Table* for each automaton A_i , and records for each *Entry* of the current state S_b , the incoming relevant events into the entry's *Local-event-Flag-Array* (ef_i). When all expected events arrived (not necessary in the same cycle) the *Condition* ($C_j \wedge v_k$ see following definitions) is checked and if is it true the reaction of the *Entry* is activated. The *Condition* may be combined by: a condition based on clocks (C_j) and a condition based on variables (v_k) . The entry's *Local-event-Flag-Array* is reset even if the *Condition* was not true.
- When the Entry is activated, the reaction (function r_n)) is executed. The *Entry* reaction is followed by the output of events (e_q) , and by the automaton current state update to S_d. If the new or old state holds sub-automata, sub-automata will be activated ($+A_z$) or deactivated ($-A_y$) respectively.

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

event-flags,	conditions	state	→	reaction,output	new subAutom	new state
$\pi(ef_i)$	$C_j \wedge V_k$	S_b	→	$r_n \wedge \pi(e_q)$	$-A_y+A_z$	S_d

And it is read as following:

When all the expected events $\pi(f_i)$ arrived and the conditions $c_j \wedge v_k$ is true, **then** the automaton reacts according to its current state S_b by executing a reaction function r_n, emitting output-events $\pi(e_q)$, updating its state S_d and activating and deactivating automata $-A_y+A_z$.

Robustness :

In order to ensure correct execution, the system makes efforts to stabilize the environment during the cycle, and making all changes between the cycles.

During the cycle all incoming events are delayed to the next cycle. The clock values are in cycle units, and they maintain their values during the cycle. The automata activation or deactivation is delayed to the end of the cycle.

The value of the variables may be changed during the cycle by the actions. Actions may be called in the same cycle many times. The automata state may change during the cycle.

The incoming relevant events are saved in flags by each automata entry, in order to avoid flag erasing by another automata. All variables are initiated to "no value". Reading a variable that contains "no value", turns the System Status Variable to "instable".

According to these definitions, the reaction time of an event is no more than 2 cycles, and the completion of all reaction must be before the next cycle starts. If the execution of the reactions overflows the cycle, the System Status Variable turns to "instable".

Conflicts management:

Related to conflicts, the system avoids simultaneous access to variables by the serialization of actions. Moreover, the system detects also possible conflicts, by checking for double assignment of variables in a cycle. The same procedure is made for double calling of actions in a cycle. In case of a detection of possible conflict in a variable or in an action, the System Status Variable is turned to "possible conflict". A stricter way for handling variable conflict is to assign a "no value" to the conflicted variable. In order to avoid transition conflict, the transitions are saved in the system table in an

ascending priority order. This method provides chance for execution of high priority transitions first, since the executor scanning is made in a pre-defined order.

6. A-priori detection of actual conflicts

In a companion paper [17], we have shown that a-priori detection of actual conflicts for parallel automata is P-space hard. We therefore develop an approach based on potential conflicts.

6.1 Potential conflict

If the conjunction of the conditions of two rules is satisfiable and their right hand sides may cause a write/write conflict we say that the two rules are involved in a potential conflict. Note that the run time behavior is not considered at all, and that there may be no actual conflict even though there are potential conflicts. It is clear however, that if there are no potential conflicts, there are no actual conflicts.

6.2 Detecting potential conflicts is possible in polynomial time when all conditions are conjunctions

Detecting potential conflicts appears to be straightforward as can be seen from the following algorithm, which is equally applicable to untimed and timed automata.

```
potential_conflict:=false;
for every pair of rules
loop
if  there exist values for making the left hand sides of the pair of rules
    true and the same variable is assigned (different values) on the right
    hand sides of the pair of rules
then potential_conflict:=true; exit for loop;
end if;
end for;
```

Even though checking for potential conflict appears to be straightforward, let us now investigate its complexity. Checking pairs of conditions contributes a quadratic term to the complexity, and then we need to determine whether the conjunction of a pair of conditions is satisfiable.

As all the conditions are conjunctions of primitive conditions, then so too is the conjunction of pairs of such conditions. Fortunately, determining the satisfiability of a conjunction of primitive conditions is easily done in polynomial time as follows.

```
for each variable in the condition
loop Form the intersection of the ranges of values this variable takes.
end for;
if all these intersections of are not empty
then the condition is satisfiable
else the condition is unsatisfiable
end if;
```

Remark

In the companion paper dealing with theoretical properties of abstract parallel automata [17], we show that any abstract parallel automaton can be converted in polynomial time into a nearly equivalent automaton with no potential conflicts, with size proportional to the size of the original automaton. The new automaton does not execute any assignments involving conflicts. This means of course that if the original automaton is free of conflicts, the new automaton has no potential conflicts and is equivalent to the original automaton.

7. Conclusion

1) A general form of extended finite-states-machine was proposed, and two implementations were presented.

2) Several forms of conflicts were identified, and several ways of dealing with this problem were proposed. An a-priori conflict prevention approach, based on potential conflicts, seems to be useful for dealing with these problems. It is of importance that conversion to a form with no potential conflicts is possible in polynomial time, and the new automaton is proportional in size to the original automata.

3) An important advantage of working with conflict free automata is easier testing and debugging. When transition rules are 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.

4) Parallel automata notation was easy to use for building an execution platform and presenting proofs. It was hard to use for programming purposes, since the default behaviour is infinite looping (all rules are always active). A better default behaviour is that a rule fires once only, and on the right hand side we should indicate which rule(s) if any should

Bibliography on related works

[1] " H.G. Mendelbaum & R.B. Yehezkael " Using 'Parallel Automaton' as a Single Notation to Specify, Design and Control small Computer Based Systems, 8th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS), Washington D.C., IEEE April 2001
[2] Gérard Berry and Georges Gonthier "The Esterel Synchronous Programming Language: Design, Semantics, Implementation",. Science of Computer Programming vol. 19, n°2, pp 87-152, 1992.
[3] 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
[4] 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
[5] 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
[6] 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
[7] 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
[8] Rajeev Alur and David Dill " a theory of timed automata"
Theoretical Computer Science 126:183-235, 1994
[9] S. Bloch, H. Fouchal et al. "Timed and Untimed Testing",univ. reims, 1999 Simon.Bloch@univ-reims.fr
[10] Eric Petijean and Hacene Fouchal, "From Timed Automata to Testable UntimedAutomata", RESYCOM lab., Univ. Reims, France
[11] J. Springintvelt, F. Vaandrager, P.R. D'Argenio "Testing Timed Automata" cath. univ. Nijmegen, Netherlands, CSI-R9712, aug. 1997, fvaan@cs.kun.nl
[12] 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
[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
[15] Bob Harms " two-level morphology as phonology (parallel automata, simultaneous rule application)", Texas linguistic Forum 35, fall '95 (harms@mail.utexas.edu)
[16] Nancy Lynch : "Distributed Algorithms", Morgan Kaufmann Publ.,1996
[17] "Some Theoretical Results on Parallel Automata, Conflict, Complexity", T. Hirst, R.B. Yehezkael, H.G. Mendelbaum, JCT research report 2003, available at http://sukka.jct.ac.il/~rafi
[18] A.H. Teitelbaum, "A unified methodology for the formal design and execution of Real-Time applications", JCT research Seminar, 5/2/2002