Some Theoretical Results on Parallel Automata, Conflict, Complexity

T. Hirst, R. B. Yehezkael (formerly Haskell), H. G. Mendelbaum
Jerusalem College of Technology - POB 16031 - Jerusalem 91160
Email: erst@jct.ac.il, rafi@jct.ac.il, mendel@jct.ac.il
Fax: +972 2 6432145

Abstract: *The purpose of this work is to classify parallel automata conflicts, to develop techniques for handling conflicts a-priori and a-posteriori, and to analyze the complexity of conflicts and other problems concerning parallel automata. Synchronous parallel automata having inputs and outputs, and which are useful for designing real time systems will be discussed.*

A-priori conflict detection for parallel automata is shown to be P-space complete. In view of this, an approach based on potential conflicts, is developed. The complexity of detecting potential conflicts is shown to be NP complete, but if automata conditions are in disjunctive normal form, potential conflict detection is possible in polynomial time.

We show that any 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 is equivalent to the original automaton, when the original automaton is free of conflicts.

The complexity of liveness, looping and deadlock in parallel automata is discussed.

Apart from parallel execution, rules are given ensuring well-defined execution in any sequential order. Some methods of handling conflicts a-posteriori at run time are discussed.

Prioritized execution is discussed and concerning prioritized execution, conversion of a parallel automaton into a form where all conditions are conjunctions of primitive conditions is possible in polynomial time.

The representation of parallel automata and conflict in the predicate calculus is discussed.

Certain extensions to abstract parallel automata are also briefly discussed.

Keywords: *Parallel Automata, Conflict, Complexity, Deadlock, Liveness, Looping, Flexible execution*

1. Introduction

A major challenge concerning parallel systems is to understand the conflicts caused by the parallelism, and to identify ways of avoiding conflict, so as to ensure well behaved execution. Various techniques exist for dealing with conflicts and the following table lists some of these techniques according to area and conflict type.

AREA	CONFLICT TYPE(S)	TECHNIQUE(S)
Operating Systems	Read-write Write-write	Critical regions, semaphores, monitors
Synchronous systems	Read-write Write-write	Synchronous execution prevents read-write conflicts. Write-write conflicts are handled by compositions, which combines all values written in parallel, into a uniquely defined value.
Data bases	Read-write Write-write	Atomic transactions.
Language definition using automata	Non determinism	Non-determinism is tolerated. The language defined is based on accepting all sequential execution possibilities.
Syntax Analysis	Shift-reduce Reduce-reduce Replace-replace	Backtracking can be used. Preferably, the grammar is rewritten to avoid these conflicts. (Syntax analysis conflicts would cause a write-write conflict when carried out in parallel. So in a sense a write-write conflict is more fundamental.)

Our study of conflict is carried out in the framework of synchronous parallel automata having inputs and outputs, and which are useful for designing real time systems. We shall also study complexity issues concerning parallel automata conflicts.

1.1 Types of conflict

The main types of 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.

Other kinds of conflicts are:

3) A *simultaneous conflict* is a kind of conflict, which causes an error if the operations are executed simultaneously, but gives uniquely defined end-result if executed sequentially **in any order**. For example the statements x+=3; x+=7 gives a uniquely defined end-result if executed sequentially **in any order** (in the same cycle of the synchronous clock) but is not well defined with simultaneous execution.

4) An *irrelevant conflict* occurs if conflicting values are assigned to a variable, but this value will never be read after this occurs. It will either never be read at all or will be written without a conflict before being read. This is not so relevant for parallel automata but may be important in wider contexts. As we shall see later that conflict detection is P-space complete. We therefore propose that the programmer indicates an irrelevant conflict by using for example "_" to indicate a “don't care” situation for the value that will never be read subsequently. So for example if "div" is a function, which returns two values, a quotient and a remainder, we can indicate an irrelevant conflict situation as follows.

```
x, _ := div(a,b); _, y := div(c,d);
```

("_" is/are variable(s) which can only be written/assigned. Though conflicting values are assigned to the variable "_", these values are never read.)

1.2 Types of conflict free automata

We consider three kinds of *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 execution method 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.

1.3 Review of extended automata models (also see appendix D)

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][30]. Some view generalized automata as an extension of sequential or parallel algorithms [39-42] having the power of Turing Machines.

Even though there has been considerable work on these extensions (also see appendix D), we did not find that the issue of conflict and its complexity was addressed in the framework of extended and parallel automata, although similar problems as reachability in Petri nets have been studied in the literature [43]. So our study of conflict and complexity, will take place only within the framework of abstract parallel automata.

2. Abstract parallel automata - explanation and formal definitions

Our study of conflict and complexity, will take place within the framework of abstract parallel automata, which must conform to the following requirements.

1) An *abstract parallel automaton (APA)* consists of a finite number of rules, which test the values of variables and make assignments to variables. The number of variables is finite, and the range of values each variable takes is finite.

2) Variables of various types are either compared with a constant on the left hand side of a rule or assigned a constant on the right hand side of a rule. The usual logical connectives may be used on the left hand side of a rule. On the right hand side of any rule, a variable may not be assigned more than once.

3) The types of automata variables are: *internal* variables, *input* variables, *output* variables, and clock variables. Initially, all variables are zero. Unless otherwise indicated, we shall not deal with clocks (timed automata) in this paper. We shall adopt the convention concerning input and output variables, that zero indicates a null input or output. Input variables are usually denoted by "?variable name". Output variables are usually denoted by "!variable name". (Internal variables are called state variables by other authors.)
4) Execution of automata rules takes place in a succession of cycles. In each cycle, all automata rules are executed once only. (Later in our study of conflict, we shall indicate conditions ensuring well-defined automata behavior, with respect to various execution orders.)
5) During each cycle, variables are tested and assigned. However, the new value of a variable becomes available in the following cycle. The new value of an internal variable is determined by the assignments made by rules whose left hand sides are true. The new value of an input variable is determined by the input received, or is zero if no input is received. The new value of an output variable is determined by the assignments made by rules whose left hand sides are true, or is zero if it is not assigned. Recall that zero indicates a null input or output. Also recall that input values are initially null (zero), and that no input is received during the initial cycle. Also note that by this definition, the final value of any input variable is null (zero).
(In an implementation different locations can be used for current and new values of variables to enable parallel or sequential execution of the rules, in any order. Alternatively, all tests can be performed *and then* all assignments made.)

Let us now give some formal definitions.

Definition 2.1 Abstract parallel automaton (APA)

An *abstract parallel automaton (APA)* M = (I, V, O, K, R) consists of the following components.

$I = \{x_1,...,x_r\}$ is a finite set of input variables.

$V = \{y_1,...,y_s\}$ is a finite set of internal variables.

$O = \{z_1,...,z_t\}$ is a finite set of output variables.

K is a non-negative integer used to define the range of values the variables take. All variables take integer values in the range 0,1,...,K. We shall adopt the convention concerning input and output variables, that zero indicates null input or output.

R is a finite set of rules. Each rule has the form $C \rightarrow /A_1//A_2/.../A_m/$ where C is a condition and each A_i is an assignment. On the right hand side of any rule, a variable may not be assigned more than once.

Let c denote an integer in the range $0 \le c \le K$.

Primitive conditions have the form $v=c$, $v \ne c$, $v<c$, $v>c$, $v \le c$, $v \ge c$ where v is an internal or input variable. Primitive conditions can be combined using the usual logical connectives to form compound conditions.

Each assignment A_i has the form v:=c where v is an internal or output variable.

Definition 2.2 Input and output sequence

An *input sequence* of an APA is a sequence of tuples where each tuple consists of r integers in the range 0,1,...,K. The values in the tuples are received by the input variables $x_1,...,x_r$.

An *output sequence* of an APA is a sequence of tuples where each tuple consists of t integers in the range 0,1,..., K. The values in the tuples are produced from the output variables $z_1,...,z_t$.

Definition 2.3 Configuration

A *configuration* of an APA is a function "conf" giving a value to each variable of the APA.
$conf: I \cup V \cup O \rightarrow \{0,1,..., K\}$. For the *initial configuration*, all variables are given the values zero.

Definition 2.4 Active rule

A rule $C \rightarrow /A_1//A_2/.../A_m/$ of an APA is said to be an *active rule* in the configuration "conf", if the condition C is true for the values of the variables in the configuration "conf". (Note that the function "conf" determines the value of each variable of the APA.)

Definition 2.5 Computation and terminated computation

Let S_1, S_2, S_3,.... be an input sequence as defined above.

A *computation* on this input sequence of an APA, is a sequence of configurations $conf_0$, $conf_1$,... where $conf_0$ is the initial configuration and for each p, $conf_{p+1}$ is obtained from $conf_p$ as follows. The values of input variables in $conf_{p+1}$ are determined by the values in the tuple S_{p+1}. The values of internal and output variables are determined by the assignments made by the active rules in $conf_p$. If more than one assignment is made to a variable we have a WW conflict and $conf_{p+1}$ is indeterminate. If no assignments are made to an internal variable, the value of the variable in $conf_p$ carries over to $conf_{p+1}$. If no assignments are made to an output variable its value in $conf_{p+1}$ is zero (null output).

A *terminated computation* is a computation in which all rules of the automaton are not active in the final configuration.

2.6 Some comments

The behavior defined above, prevents RW conflicts, but WW conflicts can occur and will cause indeterminate behavior of the automaton. Ways need to be found for preventing or handling WW conflicts.

In a more general framework, there may be other kinds of conflict e.g. contradictory actions such as "motor on", "motor off" being given simultaneously, or simultaneous sends of contradictory values, etc. These can often be modeled by WW conflicts. The "motor on" - "motor off" conflict could be modeled a WW conflict on an output variable such as !motor_on:=0, !motor_on:=1. The simultaneous send of contradictory values can be modeled by a WW conflict on a internal variable such as event:=3, event:=5, etc. Thus the results we shall prove for abstract parallel automata (APA), may also be valid in situations where there are various kinds of actions in addition to assignments.

Example 2.7 Rules of an APA where there may be a conflict

Note that in the following, the left hand side of the rules should be understood as a conjunction. The symbol ?z means that the variable z is an input from outside.

/pc1=0//y=0/ → /pc1:=1//x:=0/
/pc1=1//y=0/ → /pc1:=0//y:=0/
/pc2=0//?z=1/ → /pc2:=1//y:=1/
/pc2=1//?z=1/ → /pc2:=0//x:=1/

In the above pc1, pc2 are variables of the automaton and behave like program counters for running two cooperating threads.

Depending on the sequence of values received by the input variable "?z", there may or may not be a conflict. For the sequence 0, 1, 1, no conflict will occur. For the sequence 0, 0, 1, a conflict occurs on the variable "y" during the final execution cycle. Here are the configurations of the APA for these input sequences.

No conflict on input sequence 0, 1, 1

x	y	pc1	pc2	?z	
0	0	0	0	0	*Initially all variables zero.*
0	0	1	0	0	Input received from here onwards.
0	0	0	0	1	
0	1	1	1	1	Final values. No more input.
1	1	1	0	0	

Conflict on input sequence 0, 0, 1

x	y	pc1	pc2	?z
0	0	0	0	0
0	0	1	0	0
0	0	0	0	0
0	0	1	0	1
0	0??1	0	1	0

The lesson we learn from this small example is that it is hard to see if conflicts may occur during the course of execution. Later we show that the complexity of conflict detection is P-space complete, and propose methods for dealing with this issue.

2.8 Prioritized execution (also see section 5)

In the above definitions, all rules have equal priority. It may be preferable to give priority to rules, which do no input or output, since in practice, access to internal variables is very much faster than performing input or output. This idea is discussed further in section 5.

3. Complexity of Conflict detection and handling conflict

Here we prove various results concerning the complexity of conflict detection, and present techniques for handling conflicts a-priori and a-posteriori.

Lemma 3.1 Conflict detection for parallel automata is P-space hard.

Proof. We shall show that conflict detection is P-space hard, by reducing the problem of determining the truth value of $Q_1x_1...Q_nx_n:\phi(x_1,...x_n)$ to the conflict detection problem. Here $\phi(x_1,...x_n)$ is a propositional logic formula in the boolean variables $x_1,...x_n$ and $Q_1...Q_n$ are boolean quantifiers ($\forall$ or $\exists$).

The formula $\phi(x_1,...,x_n)$ can be computed on the following abstract parallel automaton.

$/pc_{n+1}=1/$ and $\phi^*(x_1,...,x_n) \rightarrow /Q_{n+1}:=1//pc_{n+1}:=0/$

/pc_{n+1}=1/ and ~ϕ*(x_1,...,x_n)→/Q_{n+1}:=0//pc_{n+1}:=0/

where ϕ* is obtained from ϕ by replacing each boolean variable x in ϕ by x=1. This is done because primitive automaton conditions must be in the form of a comparison of a variable with a constant.

The value of ϕ is computed in the automaton variable Q_{n+1}. The variable pc_{n+1} is used for synchronization with rules, which we add later. When the variable pc_{n+1} is "1" computation starts, and the value "0" is used to indicate inactivity.

We shall make additions to these rules and incrementally build an automaton to determine the value of $Q_1x_1...Q_nx_n:\phi(x_1,...x_n)$. Let us now deal with the quantifiers. We shall first deal with the rightmost quantifier only. So we need to construct an automaton for computing $Q_nx_n:\phi(x_1,..., x_{n-1}, x_n)$ from the previous automaton?

1) Add a rule for proceeding to the next step and starting the previous automaton with x_n zero.

/pc_{n+1}=0//pc_n=1/→/pc_{n+1}:=1//pc_n:=2//x_n:=0/

2) There are two possibilities, Q_n may be ∀ or ∃:

(a) If Q_n is ∀, add the rules:

/pc_{n+1}=0//pc_n=2//x_n=0//Q_{n+1}=1/→/pc_{n+1}:=1//x_n:=1/ - repeat with x_n=1, pc_n is unchanged.

/pc_{n+1}=0//pc_n=2//x_n=1//Q_{n+1}=1/→/pc_n:=0//Q_n:=1/

/pc_{n+1}=0//pc_n=2//Q_{n+1}=0/→/pc_n:=0//Q_n:=0/ - performed whether x_n is 0 or 1.

(b) If Q_n is ∃, add the rules:

/pc_{n+1}=0//pc_n=2//x_n=0//Q_{n+1}=0/→/pc_{n+1}:=1//x_n:=1/ - repeat with x_n=1, pc_n is unchanged.

/pc_{n+1}=0//pc_n=2//x_n=1//Q_{n+1}=0/→/pc_n:=0//Q_n:=0/

/pc_{n+1}=0//pc_n=2//Q_{n+1}=1/→/pc_n:=0//Q_n:=1/ - performed whether x_n is 0 or 1.

Now we have an automaton which computes $Q_nx_n:\phi(x_1,..., x_{n-1}, x_n)$ in the automaton variable Q_n.

We can now repeat the above process on the automaton for $Q_nx_n:\phi(x_1,..., x_{n-1}, x_n)$ replacing n with n-1 in the above construction and thereby obtain an automaton for computing $Q_{n-1}x_{n-1}Q_nx_n:\phi(x_1,...,x_{n-1},x_n)$ in the automaton variable Q_{n-1}. We carry on in this way until we obtain an automaton for computing $Q_1x_1...Q_nx_n:\phi(x_1,...x_n)$ in the automaton variable Q_1.

3) However, as all automaton variables are initially zero, the execution will not start. So we add the following rule to start the execution:

/start=0/→/pc_1:=1//start:=1/

We note that in adding a quantifier, a fixed number of rules are added to the automaton. So if this is performed n times, the number of rules in the new automaton will be linear in n. So clearly, the truth value of $Q_1x_1...Q_nx_n:\phi(x_1,...x_n)$ can be determined on a parallel automaton of size polynomial in the number of quantifiers. We also note that the original automaton is free of conflict and the changes made in adding quantifiers, does not add conflicts.

4) Now add rules to force a conflict if the value of Q_1 is "1".

/start=1//pc_1=0//Q_1=1/→/a:=1//b:=1//start:=2/

/a=1/→/c:=0/

/b=1/→/c:=1/

Here a, b, c, are new variables. Clearly, even with this addition, the size of the constructed automata is also polynomial in the number of quantifiers.

Now a conflict will occur during the execution of this automaton, if and only if the formula $Q_1x_1...Q_nx_n:\phi(x_1,...x_n)$ is true. So conflict detection is at least as hard as determining that a fully quantified formula $Q_1...Q_nx_n:\phi(x_1,...x_n)$ is true, and this has been shown to be P-space complete [31]. So conflict detection is P-space hard.

A note concerning liveness, looping and deadlock in parallel automata

The above approach can be used with hardly any change at all to show that "liveness" is P-space hard too. We note that the rules forcing the conflict are active, if and only if the fully quantified formula is true. Therefore checking for liveness is P-space hard. Similarly, by forcing an infinite loop after checking that quantified formula is true, it follows that checking for infinite looping is P-space hard.

Deadlock however, never occurs with synchronous execution of parallel automata, as there are no circular wait situations.

Lemma 3.2 Conflict detection for parallel automata is in P-space.

Proof. Here is a non-deterministic algorithm for conflict detection, which we shall later show to be in NP-space.

Initialize all automaton variables to 0;
conflict:=false; counter:=0;
loop

```
        if there is a conflict on the right hand sides of the active rules
        then conflict:=true;
        else counter:=counter+1;
            update variables using the rules of the parallel automaton;
        endif;
        choose input values arbitrarily     (non determinism here);
until counter < (number of configurations of internal variables);
```

The space needed for the algorithm is the space needed for storing the original automata, the automaton variables, the counter, and the number of configurations of internal variables. The storage needed for the counter equals the storage needed by the variables of the original automaton, and similarly for the number of configurations of internal variables. Therefore conflict detection is in NP-space. By Savitch's theorem NP-space and P-space are equal [31], and therefore conflict detection is in P-space.

Theorem 3.3 Conflict detection for parallel automata is P-space complete.

Proof. By Lemmas 3.1, 3.2, conflict detection is P-space hard and is also in P-space. It therefore follows that conflict detection is P-space complete.

3.4. Potential and actual conflict

Definition 3.4.1 Two rules are involved in a *potential conflict* if the conjunction of their conditions is satisfiable and their right hand sides may cause a write/write conflict.

Definition 3.4.2 Two rules are involved in an *actual conflict* if they are involved in a conflict at run time. (NOTE: Unless otherwise stated, "conflict" shall mean "actual conflict".)

Definition 3.4.3 An APA is a (*potential*) *conflict free* if it has no rules that are involved in a potential or actual conflict, respectively.

Note that concerning potential conflict, the run time behavior is not considered at all, and that there may never be an actual conflict at run time even though there are potential conflicts. It is clear however, that if there are no potential conflicts, there are no actual conflicts.

Example 3.4.3 The following automaton has a potential conflict as the conditions on the left hand side of the first two rules are simultaneously true when x=8 and y=3. However as x and y never receive these values at run time, there will never be an actual conflict at run time.

/x>7//y=3/ → /!z:=2/
/x<9//y<8/ → /!z:=4/
/?w=1/ → /x:=9//y:=3/
/?w=2/ → /x:=1//y:=1/

3.5 The complexity of detecting potential conflict

We shall show that detecting potential conflicts is NP complete, but if the conditions are in disjunctive normal form, potential conflict detection is possible in polynomial time. We also show that if the conditions are in conjunctive normal form, then the complexity of testing potential conflict is NP complete.

As shown earlier, a priori detection of actual conflicts is P-space hard. However 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. In the general case, determining the satisfiability of these conditions is NP complete, which would make checking for potential conflict NP complete.

However, if 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;

(Except for section 3.6 and section 5, in all automata in the paper, the conditions are conjunctions of primitive conditions.)

Furthermore, checking for potential conflicts can be done in polynomial time if the conditions are in disjunctive normal form, as we shall now explain. In this case, the complexity of checking for potential conflict between two rules is equivalent to checking for the satisfiability of the conjunction of two disjunctive normal formulae $(C_1, \vee ..., \vee C_m) \wedge (D_1, \vee ..., \vee D_n)$ where each C_i, D_j is a conjunction of primitive conditions, where i ranges from 1 to m and j ranges from 1 to n. Checking the satisfiability of this formula is equivalent to checking that at least one of the conjunctions $(C_i \wedge D_j)$ is satisfiable, and this can clearly be done in polynomial time. This is because $(C_i \wedge D_j)$ is a conjunction of primitive conditions as each C_i, D_j is a conjunction of primitive conditions.

Regarding conjunctive normal form automata conditions however, the complexity of testing potential conflict is NP complete, as checking for satisfiability of conjunctive normal form formulae is known to be NP complete [28].

3.6 Removing potential conflicts

In order to avoid actual conflicts, one could suggest avoiding even potential conflicts. However this requirement seems, at first sight, to be too much restricted, since there are conflict free automata that have potential conflicts. However, from our next theorem it follows that this requirement does not restrict the generality. We show that any parallel automaton can be converted in polynomial time into a nearly equivalent parallel automaton with no potential conflicts, with size proportional to the size of the original automaton. The new automaton does not execute the original assignments causing conflicts, and **is equivalent** to the original automaton, when the original automaton is free of conflicts.

Theorem 3.6.1 For each APA, M, there exists a **potential conflict free** APA M', of size that is polynomial in the size of M, such that, if M is **conflict free** then M and M' are equivalent.

Proof. Let M be an APA. We convert M into a potential *conflict free* APA by the following stages.
(a) Ensure that there is only one assignment on the right hand side of a rule.
(b) Group rules.
(c) Ensure no conflict.
(d) Combine rules (optional).
We only need to process rules, which are involved in potential conflicts.

(a) Ensure that there is only one assignment on the right hand side of a rule:
A rule such as "Condition" → /a:=1//b:=2//c:=3/
Would be replaced by the following three equivalent rules.

"Condition" → /a:=1/
"Condition" → /b:=2/
"Condition" → /c:=3/

(b) Group rules: For each variable there is a group consisting of all the rules which assign to it. Between groups there is not even potential conflict. A conflict may only occur in a group.

(c) Ensure no conflict: We shall ensure that there is not even potential conflict in a group. Consider a typical group, which assigns on the variable x.

"Cond. 1" → /x:=1/
"Cond. 2" → /x:=2/
"Cond. 3" → /x:=2/

This would be replaced by:

"Cond. 1" and not "Cond. 2" and not "Cond. 3" → /x:=1/
not "Cond. 1" and "Cond. 2" and not "Cond. 3" → /x:=2/
not "Cond. 1" and not "Cond. 2" and "Cond. 3" → /x:=2/

Note that when the original automata is conflict free only one of the original conditions can be true at run time so the replaced rules will be equivalent to the original ones in this case. If however there is a conflict in the original automata, then the new automaton will make no assignments when a conflict occurs in the original automata since nothing happens in the new automaton if two of the original conditions become true. The new automaton has no potential conflicts, and is equivalent to the original automata when the original automaton is conflict free.

(d) Combine rules (optional): To reduce the size of the new automaton, rules with identical conditions on the left hand side may be combined as follows, for example rules such as

"Cond. 1" and not "Cond. 2" and not "Cond. 3" → /x:=1/
"Cond. 1" and not "Cond. 2" and not "Cond. 3" → /y:=5/

can be replaced by

"Cond. 1" and not "Cond. 2" and not "Cond. 3" → /x:=1//y:=5/

This completes the conversion.

Size of the new automaton: In the worse case there are potential conflicts between all rules and so all rules are processed. After step (a), the number of rules increases by a factor of "m" where "m" is the average number of assignments on the right hand sides of a rule. After step (c), the number of rules increases at most by a factor of "n" where "n" is the average group size or equivalently the average number of assignments to a variable appearing in the rules. After step (d) the number of rules may decrease. So all in all the increase is bounded by a factor of "mn".

In practice, not all rules need to be processed, and the number of rules at step (d) can be reduced. So the increase will hopefully be much smaller.

Run time of the new automaton: This is increased by the need to compute the extra "not"s, and "and"s of the original conditions, for those rules causing an assignment to the same variable. So if M is the maximum number of assignments to the same variable, this increase is bounded by a factor of M.

In view of the above theorem, and the fact that conflict detection is P-space hard, we propose that automata having no potential conflicts should be used, even if it causes some redundancy in that there may be rules, which will never be executed.

NOTE 1: In the above a weak conflict was treated as an error. If we wish to allow weak conflict, the following change needs to be made at steps (c) onwards.

"Cond. 1" → /x:=1/
"Cond. 2" → /x:=2/
"Cond. 3" → /x:=2/

This would be replaced by a conflict-free form:

"Cond. 1" and not ("Cond. 2" or "Cond. 3") → /x:=1/
not "Cond. 1" and ("Cond. 2" or "Cond. 3") → /x:=2/

With this change, simultaneous assignment of the value 2 to x is not treated as a conflict in the original rules, and in the new rules the assignment would take place only once, even when cond.1 is false, and cond.2 and cond.3 are both true.

NOTE 2: In note 1, the resulting form is conflict-free, but perhaps it does not correspond to the execution desired by the designer. A conflict detecting rule such as:

"Cond. 1" and ("Cond. 2" or "Cond. 3") → /!conflict_x:=1/

can be added so that the designer can correct his solution.

Significance of the above result

Superficially working within a framework of avoiding potential conflicts appears restrictive. It is also hard, in that detecting potential conflicts is NP complete. The above result is significant in that these difficulties are bypassed by the above construction.

3.7 Handling conflicts a-priori and a-posteriori

The construction described in 3.6 provides us a way of constructing a-priori, an automaton with no potential and therefore no actual conflicts, or the construction can help detecting conflicts, which can be corrected by the designer.
Another possibility similar to the approach described in 3.6, is that the execution mechanism tests for conflicts a-posteriori at run time, and not perform the assignments involved in conflicts. Alternatively, the bitwise "or" of the values involved in a conflict can be assigned at run time, thereby ensuring well-defined behavior. Bitwise "and" is another option.
Yet another possibility, is to explicitly specify with the parallel automata rules, what value is assigned in the event of a conflict. Conflicts can be checked a-posteriori at run time and these values used when they occur, thereby ensuring well-defined behavior.
Actually the difference between the a-priori and a-posteriori approaches is quite small. The automaton constructed in 3.6 has no conflicts because its rules test for assignments causing conflicts at run time, and so would require more memory for storing the modified automaton rules. The a-posteriori approach requires that the execution mechanism tests for conflicts at run time, which would make the execution mechanism more complex. We see no clear preference between these choices.

4. Simultaneous conflict and Composite Assignments

Languages such as "C" or "Java" allow composite assignments of the form "x f= y", meaning x=(x f y), where f is a binary operator or function of two arguments. (A typical example is x += y.) Can composite assignments be incorporated into conflict free parallel automata? For which kinds of functions and situations is this possible? A basic requirement for parallel automata is that the domain and range of the function is finite. We shall describe situations where sequential execution in any order gives a uniquely defined final value. That is even if there is a simultaneous conflict as defined earlier, if the execution is sequential in any order, the final value is well defined.
Let us consider the following set of assignments, where the current value of x is e_0.

x f= e_1;
...
x f= e_n;

Here is a general condition in which the value of x is uniquely defined after performing all the previous assignments sequentially in any order.
Condition: It is required that ((a f b) f c) = ((a f c) f b). While the first parameter and the value of "f" must be of the same type, no restriction is placed on the second parameter of "f". Examples of functions satisfying this condition are "and", "or", exact arithmetic operations +, -, *, /, **, addition and subtraction with respect to a fixed modulus, etc. (Associative/commutative conditions or the existence of a unit element are not required.)
For now we assume that only one kind of assignment can be used with each variable. Later we will relax this condition.
The final value of x, equals value of the expression (...((((e_0 f e_{i_1}) f e_{i_2}) e_{i_3}) ... e_{i_n}) in an execution according to the above. Here i_1, i_2, ... i_n is a permutation of 1, 2, ... n and are determined by the order in which the composite assignments are performed.
The above condition, allows consecutive e's in the aforesaid expression, except for e_0, to be swapped, without altering the value of the expression. As it is possible to sort by swapping consecutive elements, we can rearrange e_{i_1}, e_{i_2}, ... e_{i_n} to e_1, e_2, ... e_n by sorting on the subscripts of the e's using "bubble sort" for example. So this means that the final value equals the value of (...((((e_0 f e_1) f e_2) f e_3) ... f e_n) and so is uniquely defined.
Furthermore, we shall show that the first condition cannot be relaxed any further. Suppose that there are only two composite assignments and the final result is uniquely defined. If the first is made before the second, then the final result will be
(((e_0 f e_1) f e_2). But if the second is made before the first the final result will be
(((e_0 f e_2) f e_1).

As the final result is uniquely defined, it follows that $(((e_0\ f\ e_1)\ f\ e_2) = (((e_0\ f\ e_2)\ f\ e_1)$. Apart from the names used, this is the condition listed above.

4.1 Handling composite assignments with several functions

Consider a more general case:

$x\ f_1 = e_1;$
...
$x\ f_n = e_n;$

where $f_1,...f_n$ are functions, which may or may not be different. (They may even all be the same, which is the case discussed above.) The previous condition needs to be modified as follows:

New condition: It is required that $((a\ f_i\ b)\ f_j\ c) = ((a\ f_j\ c)\ f_i\ b)$ for $i \neq j$.

A similar argument to the above shows that the final value of x equals the value of $(...(((e_0\ f_1\ e_1)\ f_2\ e_2)\ f_3\ e_3)\ ...\ f_n\ e_n)$ and is uniquely defined. (In the sorting on the subscripts described above, the only change is to swap consecutive function value pairs in the sort process and not just consecutive values.)

4.2 A note on synchronous systems

The technique of using composite assignments is similar to the technique of using compositions in synchronous systems [33] for ensuring well-defined behavior. However the condition we gave above for ensuring well-defined behavior is weaker than the commutative/associative condition required of compositions in synchronous systems. Composite assignments make the composition explicit, and more than one kind of composite assignment can be used, as explained above.

4.3 A note concerning subsection 3.6

In 3.6 we presented a technique for conversion in polynomial time of a conflict free automaton into a form without potential conflicts. This technique is valid with composite assignments if weak conflicts are treated as errors. The technique presented in 3.6 prevents simultaneous assignments or composite assignments to a variable, and is different to the approach described in this section.

5. Prioritized execution (expansion of 2.8)

In the description of automata behaviour in section 2, all rules have equal priority. It may be preferable to give priority to rules which do no input or output, since in practice, access to internal variables is very much faster than input or output.

It seems to us that all the previous results hold when prioritized execution is used - little or no change is needed in the proofs and explanations. It also seems to us that prioritized execution is preferable for overall system design where the execution mechanism ensures there is no loss of input. However, the original execution method seems preferable for designing the low level details of input and output as it forces the designer to consider issues concerning loss of input.

The following additional results hold when prioritized execution is used.

Theorem 5.1 Any untimed parallel automaton can be converted in polynomial time into a form where all conditions are conjunctions of primitive conditions.

Let us rewrite all conditions as fully bracketed formulae using the connectives "not" ,"and" only. This can clearly be done in polynomial time.

Let us give level numbers to the brackets in these formulae, such that the outermost brackets have level two (not zero as is usual). Let "maxlevel" denote the maximum level of all these fully bracketed formulae.

1) Here are the rules for updating "level"

(level=0) → /level:=maxlevel/ (This rule is active only initially.)

(level=1) → /level:=maxlevel//t_i:=0/ -- for every new temporary variable t_i introduced later.

(level=maxlevel) → /level:=maxlevel-1/

(level=maxlevel-1) → /level:=maxlevel-2/

...

(level=2) → /level:=1/

2) We will now show how a rule using the connectives "not" ,"and" only, is processed. Consider a typical rule such as:

((not (a=1 and b=2)) and (not c=3)) → "Assignments"

According to the numbering system for levels, the level of the innermost brackets is four and the level of the outermost brackets is two. As the level changes from "maxlevel" to two, the bracketed formulae at that level are computed using new temporary variables t_i, which are unique for each formula, as follows:

(level=4) and (a=1) and (b=2) → /t1:=1/

(level=3) and (t1=0) → /t2:=1/

(level=3) and (c≠3) → /t3:=1/

(level=2) and (t2=1) and (t3=1) → /t4:=1/

3) When the level becomes one, the following rule which uses the value of t_4, will make the assignments required.

(level=1) and (t4=1) → "Assignments"

The above construction clearly requires polynomial time and the conditions on the left hand side of all the rules are conjunctions.

Regarding the run time of the constructed parallel automaton, this is linear in the maximum level of the fully bracketed formulae multiplied by the original run time.

Regarding the size of the constructed parallel automaton, this is linear in the maximum level of the fully bracketed formulae multiplied by the original size.

Corollary 5.1.1 Even if the only primitive condition allowed is "=", the above conversion may be done in polynomial time.

Primitive conditions of the form v≠c, v<c, v>c, v≤c, v≥c can be viewed as disjunctions of equalities. So let K be the largest value an automaton variable can take (K is the constant in the definition of the automaton). So v≠c, would be viewed as (v=0 or v=1 or ...v=c-1 or v=c+1 ...or v=K), and similarly for v<c, v>c, v≤c, v≥c.

Using this viewpoint, such primitive conditions in a rule can be handled by adding extra levels and more temporary variables for computing the value of these primitive conditions.

e.g. to handle a rule such as (level=3) and (x<3) and (y<4) → /t3:=1/ the following changes would be made:

The rule

(level=3) → /level:=2/

is replaced by

(level=3) → /level:=maxlevel+1/ -- use of an extra level

(level= maxlevel+1) → /level:=2/

The following rules are added:

(level=3) and (x=0) → /t3_1:=1/ -- 3 rules which compute (x<3) in t3_1 when level=3

(level=3) and (x=1) → /t3_1:=1/

(level=3) and (x=2) → /t3_1:=1/

(level=3) and (y=0) → /t3_2:=1/ -- 4 rules which compute (y<4) in t3_2 when level=3

(level=3) and (y=1) → /t3_2:=1/

(level=3) and (y=2) → /t3_2:=1/

(level=3) and (y=3) → /t3_2:=1/

(level= maxlevel+1) and (t3_1=1) and (t3_2=1) → /t3:=1/ -- computation of t3 when level= maxlevel+1

Here t3_1, t3_2 are new temporary variables, which also need to be initialized to zero like the other temporary variables. Of course if there are more rules using the primitive conditions v≠c, v<c, v>c, v≤c, v≥c, then levels maxlevel+2, ... would be used.

Note that all conditions are conjunctions of primitive conditions of the form v=c. Also note that the use of extra levels prevents enumerating all possible combinations of the variables c, d making (c<3) and (d<4) true. More generally, if all in all there are n primitive conditions of the form v≠c, v<c, v>c, v≤c, v≥c, an order of nK rules will be added. An exponential explosion is avoided and the conversion is done in polynomial time.

6. Conclusions

1) We identified several forms of conflict, and proposed several ways of dealing with this problem. We demonstrated the complexity and hardness of conflict detection. In view of these difficulties, an a-priori conflict prevention approach based on potential conflicts or an a-posteriori run time approach, seem to be useful practical possibilities for dealing with these problems. It is significant, 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.

2) We found it surprising that only if we restrict the form of the conditions, does potential conflict detection become possible in polynomial time. Perhaps the development of parallel systems would be more efficient if such

restrictions are enforced. For teaching, we used conjunctions of primitive conditions only, and this was found satisfactory for student use.

3) We found it surprising that conversion to a form with no potential conflicts is possible in polynomial time, even though that conflict detection is P-space complete. This means that working within the framework of no potential conflicts does not reduce the generality of these automata.

4) 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.

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

7. Further work

1) Here, we did not investigate RW conflicts and very strict conflict free automata - we simply assumed that the execution mechanism prevents these conflicts. Perhaps rules can be similarly found for ensuring that an abstract parallel automaton is free from conflicts in the very strict sense, even if RW conflicts are not prevented by the execution mechanism.

2) While freedom from conflict ensures well-defined behavior of an automaton, it does not ensure that null input and output values do not affect the overall input/output behavior of the automaton. Further investigation is needed to identify rules to ensure well-defined behavior in the presence of null values.

3) We did not deal with clocks in this paper. Suitable definitions should be added and this matter investigated further.

Acknowledgment

Sincere thanks to G. Vidal-Naquet (SupElec - France) for his valuable comments and suggestions regarding this paper and related topics.

APPENDIX A: Representing parallel automata and conflict in the predicate calculus

The approach we take is similar to that used by Floyd, Manna, Cooper for representing a program schema as a logical formula of the predicate calculus [32]. In their approach, all inputs are received initially. We have adapted their approach to handle inputs received incrementally.

We shall describe in detail how to make this representation when prioritized execution is used. (Later we indicate the changes needed when prioritized execution is not used.)

When prioritized execution is used, we may assume without loss of generality that all conditions are conjunctions of primitive conditions having the form "variable=constant" (see corollary 5.1.1).

We shall represent an abstract parallel automaton by a formula in the predicate calculus, which contains no functors (function symbols). In the following, we assume that input variables of the automaton are represented by a vector x, internal variables will be represented by a vector y and output variables by a vector z. Initially all variables are zero, and regarding input and output variables, zero is a null value representing no input or no output.

Each rule of an abstract parallel automaton will be represented by a logical formula of the form $\forall...\forall...:[p(...)\rightarrow p(...)]$ representing its behavior, where "p" is a predicate. The parameters of "p" are x, y, z and "s". The purpose of the added variable "s", is to serialize the application of the rules in the logical formulation. There is an additional logical formula (later) corresponding to s=0, represents receiving inputs, setting all outputs to zero (null value), and a check for freedom from conflict. A value "s" in the range $1 \le s \le r$ represents execution of rule number s. If there is no conflict the rules can be applied in any order, and we use "s" in the logical formulation, to ensure sequential application of the formulae. So s cycles from 0 to r, where "r" is the number of rules. In the logical formulation, the cycle is repeated until a conflict arises.

So for example, if there are two input variables, two internal variables, and one output variables, a typical rule say the 8th rule:

$(x1=7) \wedge (x2=6) \wedge (y2=4) \rightarrow$ /y2:=5//z1:=3/

would be represented by the formula:

$\forall y_1 \forall z_1$:[$p(7, 6, y_1, 4, z_1, 8) \rightarrow p(7, 6, y_1, 5, 3, S')$].

where S' is 9 if this is not the last rule or 0 if this is the last rule. Note that in forming the formula, tests of a variable are made by using a constant parameter on the left side, and assignments to a variable are made by using a constant on the right side. An unchanged variable when tested is represented by the same constant on both sides, and when untested, by the same variable on both sides. Input variables do not change in the logical formulation of a rule as all inputs are dealt with when s=0.

From now on, each rule of the automaton will be denoted by $A_i \rightarrow B_i$. Here is the extra continuation formula corresponding to s=0.

$\forall \underline{x} \forall \underline{x}' \forall \underline{y} \forall \underline{z}$:[[$p(\underline{x}, \underline{y}, \underline{z}, 0) \wedge v_1(x_1') \wedge ... \wedge v_m(x_m') \wedge \sim$(Disjunction of $A_j \wedge A_{j'}$)] $\rightarrow p(\underline{x}', \underline{y}, \underline{0}, 1)$]

Here m is the number of input variables and j, j' range over pairs of rule numbers having a potential conflict with j<j'. So in the logical formulation, "execution" continues at the first instruction if there is no conflict. Similarly, a conflict "stops" the automaton in the logical formulation.

Note the use of $\underline{x}'$ and the predicates $v_i(...)$ for "receiving" new input values. Also note the use of the zero vector $\underline{0}$ for setting all output values to zero (null output). The predicates $v_i(...)$ are used for defining the legal values of input variables, and are defined explicitly by input assertions. For example, if there are only two input variables and x_1 only take the values 0, 6, and x_2 only take the values 0, 7, then assert $v_1(0) \wedge v_1(6) \wedge v_2(0) \wedge v_2(7)$.

Lemma A.1 There is an execution sequence with no intermediate conflict, starting (0,0...,0,0) and ending $(\underline{x}, \underline{y}, \underline{z})$ if and only if there is a derivation of $p(\underline{x}, \underline{y}, \underline{z}, 0)$ from:

[p(0,0...,0,0) $\wedge$ (Conjunction of formulae for all rules) $\wedge$ (Continuation formula) $\wedge$ (Conjunction of input assertions)]

Proof. The reason we require no intermediate conflict, is that a conflict terminates the derivation or "execution" in the logical formulation. If no transition rules are used at all or similarly the proof does not use modus ponens at all then $(\underline{x}, \underline{y}, \underline{z})=(0,0...,0,0)$ and $p(\underline{x}, \underline{y}, \underline{z}, 0)$ is derivable since this is just p(0,0...,0,0) which is an assumption. So in the trivial case the lemma is valid. Henceforth suppose that at least one rule of the automaton is used and that modus ponens is used at least once in the derivation.

The way the logical formulae were constructed, every rule corresponds to a logical formula, which expresses the changes in the variables when the rule is executed. So for any execution sequence there will be a corresponding derivation. So for an execution sequence starting (0,0,...,0,0) and ending $(\underline{x}, \underline{y}, \underline{z})$ there is a derivation starting p(0,0...,0,0) and ending $p(\underline{x}, \underline{y}, \underline{z}, 0)$. Since p(0,0...,0,0) is an assertion, and input assertions can be used ate each step, this will makes $p(x_1', x_2', ...x_n', s')$ derivable.

Regarding the converse, consider any derivation of $p(\underline{x}, \underline{y}, \underline{z}, 0)$ using modus ponens. We shall extract a suitably ordered sub-derivation, which corresponds to an execution sequence. The last step of the derivation must be $p(\underline{x}, \underline{y}, \underline{z}, 0)$, and somewhere use must be made of p(0,0...,0,0). Start from the last step and put before it p(...) which was on the left hand side of the formula which was used to derive $p(\underline{x}, \underline{y}, \underline{z}, 0)$. Keep working backwards from p(...) in this way until p(0,0...,0,0) is reached. The derivation sequence so constructed with "p" and the last parameter representing "s" removed, is an execution sequence of the automaton. *This concludes the proof of the lemma, we now continue with the main result.*

We now describe how conflict detection is expressed in terms p(0,0...,0,0) representing initial values, some of the A_j's and the previous formula. Examine all pairs of rules, which have a potential conflict on their right hand sides.

The automaton has a conflict if and only when we start the automaton with the initial values then when s=0, the disjunction of $A_j \wedge A_{j'}$ is true. (Recall that "s" is the last parameter of "p".) In view of the previous lemma, this occurs if and only if the following is derivable/valid:

(1) [p(0,0...,0,0) $\wedge$ (Conjunction of formulae for all rules) $\wedge$ (Continuation formula) $\wedge$ (Conjunction of input assertions)] $\rightarrow \exists \underline{x}' \exists \underline{y}' \exists \underline{z}'$:[$p(\underline{x}', \underline{y}', \underline{z}', 0) \wedge$ (Disjunction of $A'_j \wedge A'_{j'}$)]

This is equivalent to determining unsatisfiability of the formula:

(2) [p(0,0...,0,0) $\wedge$ (Conjunction of formulae for all rules) $\wedge$ (Continuation formula) $\wedge$ (Conjunction of input assertions)] $\wedge \forall \underline{x}' \forall \underline{y}' \forall \underline{z}'$: ~[$p(\underline{x}', \underline{y}', \underline{z}', 0) \wedge$ (Disjunction of $A'_j \wedge A'_{j'}$)]

Here A'_j is like A_j but with $\underline{x}', \underline{y}', \underline{z}'$ written in place of $\underline{x}, \underline{y}, \underline{z}$, and similarly regarding $A'_{j'}$ and $A_{j'}$.

This completes the explanation for prioritized execution.

For non prioritized execution, we can not rely on corollary 5.1.1 (and theorem 5.1) as in the above. However we can carry out the constructions in corollary 5.1.1 and theorem 5.1 in a predicate logic formulation of such an automaton to overcome this difficulty.

So a typical automaton rule $A_i \rightarrow B_i$ would be first written as a logical formula of the form $\forall ... \forall ...:[(A_i \wedge p(...)) \rightarrow p(...)]$. Here A_i is the condition on the left hand side of the rule.

Then the techniques described in theorem 5.1 and corollary 5.1.1 are used on these logical formulae so that the new conditions N_i are in the form of conjunctions of primitive conditions of the type variable=constant. Then these

primitive conditions are removed by substituting the constant for the variable leaving us as before with formulae of the form ∀...∀...:[(p(...))→p(...)]. The proof then continues as before.

We do not present further details of this technique for non prioritized execution.

A.2 Notes

1) The above construction essentially gives an algorithm for conflict detection by testing formula (2) for unsatisfiability. Since formula (2) has no functors (function symbols) and is easily converted to clausal form, this is decidable as the Herbrand universe is finite. This can be tested using a predicate logic theorem prover or perhaps programmed in the PROLOG language. While this approach may be possible for small automata, we are pessimistic of using such an approach for large automata. This is why we proposed testing potential conflict.

2) If PROLOG is used to build a simulator or prover for parallel automata on the basis of the technique described here, inputs and outputs would be handled once in PROLOG clause(s) corresponding to the continuation formula.

APPENDIX B: Certain extensions to abstract parallel automata

So far we have only allowed comparisons of a variable with a constant and assignments of constants to variables. What would happen if we allowed comparisons or assignments be made between variables? Would the complexity results still hold? Further work is needed to clarify these issues.

However, it seems to us that a similar argument to B.1 can be used to convert any APA with these extensions in polynomial time into an APA without these extensions. It also seems to us that the technique presented in 3.6 is valid for these extensions if weak conflicts are treated as errors, and with further work perhaps weak conflicts can be allowed.

APPENDIX C: Various notes and unproved comments

C.1 Sub-automata components with no internal conflicts

It seems to us that if we wish to use a component sub-automaton with no internal conflicts, and the interface to this component is via input variables and output variables only, then the construction described in 3.6 may be used with no knowledge of the internal transitions of the component.

We assume as before that the execution method prevents RW conflicts, and of course the above construction prevents WW conflicts. Also, as there are no in/out variables for the interface, no RW conflict can be caused in assignments and references to the interface variables. We also need to assume that rules of the component are never activated by null (zero) input.

C.2 Conflict, non determinism, statecharts

It seems to us that a non deterministic finite state automaton or statechart [34] may never have non deterministic behavior, if for example the rules causing the non determinism never become active, or never become active simultaneously. So the usual definition of *non-determinism* corresponds to potential conflict, and one can define *actual non-determinism* if this is realized at run time. Of course, *actual non-determinism* corresponds to our notion of actual conflict.

We expect that the results of this paper would also hold for statecharts when expressed in the terminology of *non-determinism* and *actual non-determinism*. For example to adapt the result in 5.6 for statecharts, the same construction would be used separately on "ε rules" and "non ε rules" of the statecharts. It also seems to us that statecharts are equivalent to parallel automata when prioritized execution is used - see section 5.

C.3 Timed automata

Though we have not formally defined timed automata, here are some unproved comments.

1) We expect that conflict detection for timed parallel automata is P-space hard - 3.1 same proof.

2) We do not know if conflict detection for timed parallel automata is in P-space and therefore P-space complete.

3) The use of potential conflicts 3.4, 3.5, 3.6 seems to be equally relevant for timed parallel automata. In particular, the technique in 3.6 should produce a timed parallel automaton with a well-defined behavior. This may provide an alternative to the zone or region construction [15, 19, 23], without conversion to sequential form.

4) Theorem A.1 concerning untimed automata, may in practice and under certain situations "hold" for timed automata. For example, if the computation cycle is very much faster than the clock cycle (discrete time assumed), the delay introduced by the construction will be less than one clock cycle and the behavior of timing rules will not be affected.

APPENDIX D: Review of extended automata models (expansion of 1.3)

In the following, we review various models of extended automata and compare various proposals of extensions to sequential automata, in which parallelism, synchronization and timing features were introduced. Let us briefly describe some of them.

D.1 Additions 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, states, actions, and boolean conditions. The transition function of this extended automaton is checks conditions, performs actions and makes changes to state and variables.

In other extensions e.g. [5-8] the addition to the state is in the form of an n-tuple.

D.2 Parallel graphs

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. 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 'Parallel Automaton' has been used [10-11] in multi-Web applications and Hyperdocuments.

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 representation.

D.3 Timed automata

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. 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.

D.4 Extended automata with 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 that can read, each time, characters coming from several tapes in parallel. The machine has one global state, and a memory. He used such a machine to model the human language, in which you have to take into account both the grammar and the phonology of a sentence. The Turing machine approach is unsuitable for our work in view of the many undecidable problems associated with Turing machines.

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 formalization of distributed algorithms, than an executable automata specification.

David Harel [34] developed a structured formalism for handling parallelism of events and actions using his statechart approach, and Harel, Drusinsky, Hirst, Globerman [35, 36, 37, 38] has made comparisons of the sizes of various kinds of extended automata and statecharts incorporating various kinds of parallelism. The importance of this structuring is that it simplifies the software development process.

D.5 Abstract state machines:

This model has been proposed by Gurevich and Blass [41, 42], who use an elaborate formalism which includes multisets, proclets, etc. Their model has the power of Turing machines and in [42] they write that any parallel synchronous algorithm can be modeled in their formalism.

D.6 Summary of Review

In the above literature review, we discussed 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], and generalization of states [39-42].

References

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

[25] " 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
[26] R.B. Yehezkael, "SIMPLE FLEXIBLE LANGUAGE - SFL", Presented to the Compiler Group Meeting, IBM Research Laboratories, Haifa, 3rd May 2000. Most recent version revised 2003 available on internet at http://sukka.jct.ac.il/~rafi
[27] R.B. Yehezkael "First Steps in Computer Science: Sequential or Parallel?", Proceedings of the ACIS 1st International Conference on Software Engineering Applied to Networking & Parallel/ Distributed Computing, SNPD '00, pp. 77-82, University of Reims, France, May 2000
[28] Stephen Cook, "The complexity if theorem-proving procedures", Proc. 3rd Ann. ACM Symp.on Theory of Computing, Association for Computing Machinery, New York, 151-158, 1971.
[29] A.H. Teitelbaum, "A unified methodology for the formal design and execution of Real-Time applications", JCT research Seminar, 5th, Febr. 2002
[30] Amir Pnueli, Natarajan Shankar, Eli Singerman, "Fair Synchronous Transition Systems and their Liveness Proofs", Technical Report SRI-CSL-98-02, Computer Science Laboratory, SRI International, Menlo Park CA, USA, 2002.
[31] Michael Sipser, "Introduction to the Theory of Computation", PWS publishing co., 1997
[32] Z. Manna, "Mathematical Theory of Computation", McGraw Hill, 1974.
[33] 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.
[34] D. Harel, "Statecharts: A visual formalism for complex systems", Sci. Comput. Prog. 8, 231-274, 1987
[35] D. Drusinsky and D. Harel, "On the power of bounded concurrency I: Finite automata", JACM 41(3), 517-539, 1994
[36] T. Hirst and D. Harel, "On the power of bounded concurrency II: Pushdown automata", JACM 41(3), 540-554, 1994
[37] N. Globerman and D. Harel, "Complexity results for two way and multi-pebble automata and their logics", Theoretical computer Science 169(2), 161-184, 1996
[38] T. Hirst and M. Lowenstein, "Alternation and bounded concurrency are reverse equivalent", Information and Computation 152, 173-187, 1999
[39] Y. Gurevich. Sequential ASM thesis. *Bulletin of EATCS,* 71(67):93–124, February 1999.
[40] Y. Gurevich, P.W. Kutter, M. Odersky, and L. Thiele, editors. *Abstract State Machines: Theory and Applications.* Number 1912 in LNCS. Springer, 2000.
[41] Yuri Gurevich, "Sequential Abstract State Machines Capture Sequential Algorithms", ACM Transactions on Computational Logic, vol. 1, no. 1, July 2000, 77-111
[42] A. Blass, Y. Gurevich, "Abstract State Machines Capture Parallel Algorithms", ACM Transactions on Computational Logic, 4(4) pp 578-651, October 2003
[43] Javier Esparza "Decidability and Complexity of Petri Net Problems - An Introduction", Lectures on Petri Nets I: Basic Models, Springer Lecture Notes in Computer Science, Vol 1491 (1998), pp 374-428