



HAL
open science

The Synthesis Toolbox - From modal automata to controller synthesis

Gérald Point

► **To cite this version:**

Gérald Point. The Synthesis Toolbox - From modal automata to controller synthesis. 2005. hal-00349874

HAL Id: hal-00349874

<https://hal.science/hal-00349874>

Preprint submitted on 5 Jan 2009

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THE Synthesis TOOLBOX

FROM MODAL AUTOMATA TO CONTROLLER SYNTHESIS



Research Report 1342-05

Gérald Point
LaBRI - UMR CNRS 5800

January 11, 2005

Contents

1	Introduction	5
2	Starting Synthesis	7
2.1	Getting started	7
2.2	Redirection	8
2.3	Input file formats	8
2.4	Displaying objects	9
2.5	Obtaining help	9
3	Synthesis in practice	11
3.1	A centralized controller	12
3.2	The decentralized case	13
4	Theoretical aspects of Synthesis	17
4.1	Processes, modal automata, games and strategies	18
4.2	Product and quotient operations	19
4.3	From modal automaton to game	20
4.4	From multi-parity to parity	21
4.5	Computation of a winning strategies	22
A	Commands	27
B	Syntaxes	33

Chapter 1

Introduction

Synthesis is a tool used to synthesize controllers for discrete event systems. **Synthesis** operates in a wider framework than the one used Ramadge and Wonham[6]. Here we consider infinite behaviours as well as finite ones; moreover, the user can specify some kind of dynamic controllability.

Synthesis is an implementation of algorithms proposed by several authors[3, 8]. We have implemented operations presented in [2] allowing the synthesis of decentralized controllers; the case of partial observability is not yet handled.

The tool considers the synthesis problem as a workflow which transforms the couple $(system, specification)$ into several objects : a multi-parity game, a parity game, a winning strategy and, finally, the controller. Since **Synthesis** has been developed as an experimental tool, it does not implement the workflow in a one-step operation; on the contrary the workflow must be executed step by step. The user has the opportunity to study generated objects all along the **Synthesis** workflow.

Synthesis is a command-line oriented program. It is developed on a Linux system but works also on MacOS platforms. Future works will tackle other Unix-like platforms.

In this primer, we present formal aspects of the synthesis workflow as well as **Synthesis** in use. Appendices present the commands allowed by the **Synthesis** shell and the syntaxes of files supported by the program.

The first version of this toolbox was implemented by students under the direction of A. Arnold in 2002 and 2003: A. Cailley, S. Canto, H. Desheraud, A.-C. Froment, R. Gobard, S. Labrune, F. Mendes, S. Muhr and F. Zucconi.

The image used for the cover page of this report is an engraving called “Adam and Eve” made in 1504 by Albrecht Dürer.

Remarks and comments might be sent to the architects of **Synthesis**:



André Arnold, Gérald Point
LaBRI – UMR CNRS 5800 – Université Bordeaux I
351, cours de la Libération
33 405 Talence Cedex - France
Andre.Arnold@labri.fr, Gerald.Point@labri.fr

Chapter 2

Starting Synthesis

`Synthesis` runs as a command interpreter offering history and file completion if it has been compiled with the GNU `Readline` library[7]. If the program can be used interactively like a UNIX shell, it can be used also in a batch mode in order to execute existing script files.

2.1 Getting started

When the program is started a `synthesis` prompt appears:

```
point@localhost: ~/tmp
synthesis
synthesis>
```

From this prompt the user can enter commands or leaves the interpreter. All the commands allowed by `Synthesis` are described in Appendix A page 27.

```
synthesis>print "hello world"
hello world
synthesis>exit
point@localhost: ~/tmp
```

To leave the program one can use an EOF character (CTRL-D on most systems) or the `exit` command; in both cases the history of the commands entered while the session are stored into the file `.synthesis_history.syn` in the current directory; this file is reloaded the next time `Synthesis` is started from this directory.

```
point@localhost: ~/tmp
cat .synthesis_history.syn
print "hello world"
exit
point@localhost: ~/tmp
```

The program might be executed in batch mode. This mode consists in calling `Synthesis` with file names; the program loads (or execute for scripts) each file and terminates. If the command-line option `-i` is used then `Synthesis` does not terminate i.e. after the reading of files the interactive mode is started. Example:

```
point@localhost: ~/tmp
cat t1.fam test.syn
name A;
x = nu -> <a>x.<b>y;
y = nu -> <a>y;
<initial={x}>.
```



```

print "this is a test"
point@localhost: ~/tmp
synthesis t1.fam test.syn
this is a test
point@localhost: ~/tmp
synthesis -i t1.fam test.syn
this is a test
synthesis>

```

2.2 Redirection

`Synthesis` uses a similar notation (but less powerful) than standard UNIX shells to redirect output streams into files. The output of a command can be used to create a file (using `>`) or appended at the end of an existing file (using `>>`). If the output file does not exist `>>` has the same effect than `>`. Example:

```

point@localhost: ~/tmp
synthesis -i t1.fam
synthesis>print "// the multi-parity automaton A is translated in a simple" > result.fam
synthesis>print "// parity automaton" >> result.fam
synthesis>parity A >> result.fam
synthesis>exit
point@localhost: ~/tmp
cat result.fam
// the multi-parity automaton A is translated in a simple
// parity automaton
name $$;
T = 2 -> [a, b]T;
y = 2 -> <a>y.[b]T;
x = 2 -> <a>x.<b>y;
<initial={x}>.
point@localhost: ~/tmp

```

In the current version of `Synthesis`, just `>` and `>>` are implemented; there is no pipe mechanism and the error stream can not be redirected.

2.3 Input file formats

`Synthesis` supports several types of files but uses only one command to read them: `load`. The format of a file is identified by its extension:

- `.syn` is associated with the script file format. These files contain lists of `Synthesis` commands. For instance this is the file format of the history file `.synthesis_history.syn` seen above.
- `.mec` is associated with Mec 4[1] files restricted to simple transition systems (i.e. not obtained by a synchronized product).
- `.fam` is associated with files describing modal automata (see section 4.1).
- `.game` is associated with files describing parity games (see also section 4.1).

Transition systems, modal automata and games can not be entered directly from the `Synthesis` prompt.

2.4 Displaying objects

Any object manipulated by **Synthesis** can be displayed using the **show** command. Each object is displayed using its native format; some translators exist (see Appendix A). In order to determine the objects in use by the program, one uses **show** with a type name: **processes**, **automata** or **games**. Example (2.2 continued):

```
synthesis>point@localhost: ~/tmp
syn -i t1.fam
synthesis>show processes
synthesis>show automata
A
synthesis>show games
synthesis>show A
name A;
T = nu -> [a, b]T;
y = nu -> <a>y.[b]T;
x = nu -> <a>x.<b>y;
<initial={x}>.
synthesis>
```

2.5 Obtaining help

The user can use the **help** command in order to obtain some explanation about **Synthesis** commands. **help** without argument lists all commands supported by the program and **help cmd** gives informations about the **cmd** command:

```
synthesis>help help
help usage :
Show help about commands. Type 'help command-name' in order to obtain help
about the command 'command-name'
synthesis>
```


Chapter 3

Synthesis in practice

The purpose of this chapter is to show **Synthesis** in use. All along these pages we focus on a “small” problem: to prevent the collision of two trains. The system considered in this chapter is composed of two trains, T_1 and T_2 , sharing the same railroad. In order to prevent the struck of trains, a deviation track is added along the main road. The trains can choose to continue on the main track or to follow the deviation. This system is depicted on the figure 3.1.

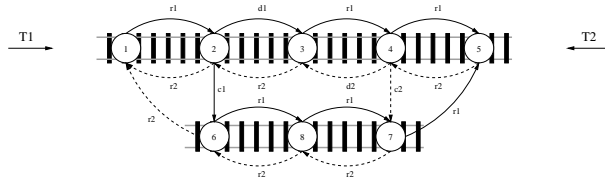


Figure 3.1: T_1 comes from the left (at location 1) and T_2 comes from the right (at location 5). The possible moves for T_1 are drawn with filled arrows and those for T_2 with dashed arrows. Each train can move to the deviation track (from location 2 for T_1 and from 4 for T_2).

Each train is modelled using a transition system. The movements of T_i are modelled with the events r_i , c_i and d_i : c_i and d_i are used in place of r_i to indicate the change of way of the train. We make this replacement for two reasons:

- We use two distinct events because **Synthesis** allows only deterministic processes.
- We distinguish these events from r_i because in the next section, c_i and d_i will be considered as controllable events while r_i remains uncontrollable.

Each train is modelled by a transition system which looks like the system on figure 3.1 except that each state is equipped with loops labelled with the actions of the other train; the figure 3.2 depicts the model of the train T_1 . These loops indicate that a movement of a train does not affect the state of the other.

The model of the whole system has 64 states and 112 transitions. **Synthesis** do not accept composite transition systems so this model has been obtained by hand:

- Load the models of trains T_1 and T_2 into **Synthesis**.
- Synchronize the two transition systems and generate a Mec 4 file containing this synchronized product.
- Edit the file :
 - Identify the 8 states where trains are struck and mark them with the property **danger**.
 - Remove useless transitions originating for **danger** states.

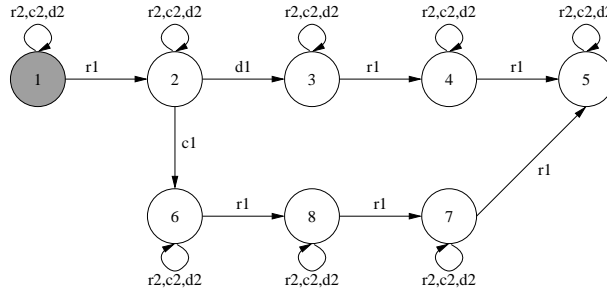


Figure 3.2: The transition system modelling the train T_1 . An action of T_2 does not change the state of T_1 . The initial state is drawn in gray.

- Identify the final state (in this state no more danger exists) and mark it with the property `final`.
- Save the file and go back into `Synthesis`.

Now, our goal is to prevent the trains to strike themselves. In order to achieve this objective we are looking for a controller which leads the trains into the `final` state and prohibits the paths to a `danger` state. Thus, the states of the supervised system (i.e. the system synchronized with the controller) must be a model of the following modal automaton:

```
name sys_spec;
x = mu -> ~danger.~final. (<r1>x. [r2,c1,c2,d1,d2]x
    +<r2>x. [r1,c1,c2,d1,d2]x
    +<c1>x. [r1,r2,c2,d1,d2]x
    +<d1>x. [r1,r2,c1,c2,d2]x
    +<d2>x. [r1,r2,c1,c2,d1]x
    +<c2>x. [r1,r2,c1,d1,d2]x)
    + final. ([r1,r2,c1,c2,d1,d2]T);
<initial={x}>.
```

This automaton can be interpreted as follows:

- Either the system is in the final state (i.e there is no more danger for the trains);
- Either the system is in a state which is neither the final nor a danger state and from this state there must exist a transition leading the system into a state having the same property or being the final state.

The reader should have noted the T in the second member of the equation defining x. This T denotes an implicit state (called *top*) of the automaton. This state is defined by the following equation : $T = \nu [*]T$ where [*] is a shortcut replacing all the events of the automaton.

3.1 A centralized controller

In the previous section we have defined the specification (`sys_spec`) that must be satisfied by the controlled system. In order to obtain the constraint that must be fulfilled by the controller, we compute the quotient of the specification by the system:

```
synthesis>load sys.mec spec.fam
Loading file 'spec.fam'
Loading file 'sys.mec'
synthesis>show processes
sys
synthesis>show automata
```

```

sys_spec
synthesis>controller_spec := quotient sys_spec sys

```

The last line computes a new modal automaton, `controller_spec`, modelling the expected behaviours for the controller. Now we are able to compute a correct controller but, with no more specifications, the result would be a controller that is allowed to control any action of the trains. If the controller is seen as an automatic switch between tracks then it can't act on movement actions of the trains (i.e r_i); thus, the events r_1 and r_2 of the system must be specified uncontrollable:

```

name controller_additional_spec;
x = nu -> <r1,r2>x.[c1,c2,d1,d2]x;
<initial={x}>.

```

This automaton specifies that for each state of the controller there must exist at least two transitions, one labelled with r_1 and the other with r_2 . In order to obtain one controller it remains:

- to compute the product `final_spec` of `controller_spec` with `controller_additional_spec`;
- to generate the parity game `G` computed from `final_spec`;
- to compute a winning strategy `S` in the game `G`;
- to generate the controller `C` from `S`;
- to synchronize the system and the controller to get the supervised system.

All these steps are done as follows:

```

synthesis>final_spec := product controller_spec controller_additional_spec
synthesis>G := game final_spec
synthesis>S := strategy G
synthesis>C := minimize(unmark(control S))
synthesis>
synthesis>Csys := sync sys C
synthesis>dot C > C.dot
synthesis>dot Csys >> Csys.dot

```

The controller is generated using the `control` command. To obtain `C` we apply two other operations to the result of `control`:

`minimize` This command reduces the number of states of the process. This is essentially an algorithm reducing the state space of an automaton; here we consider that all states are accepting but they are distinguished according to the properties labelling them (e.g. `danger` or `final` in our example).

`unmark` This command removes properties which label the states of a process. This is useful to reduce further more the number of states of the controller when calling the `minimize` command.

The last lines computes the controlled system and outputs the controller and the controlled system into `dot` file format[5]. These results are shown on figure 3.3.

3.2 The decentralized case

In the previous section, the system was managed by a global controller which could act on controllable actions of both trains; the controller was considered as a program managing switches located at position 2 and 4.

Now we consider that there is two independant programs controlling switches. The controller `C1` (resp. `C2`) at location 2 (resp. 4) can control only the events c_1 and d_1 (resp. c_2 and d_2). While the constraint for the controlled system remains the same than in the centralized case, the additional specifications for the new controllers indicate that only c_i and d_i are controllables:

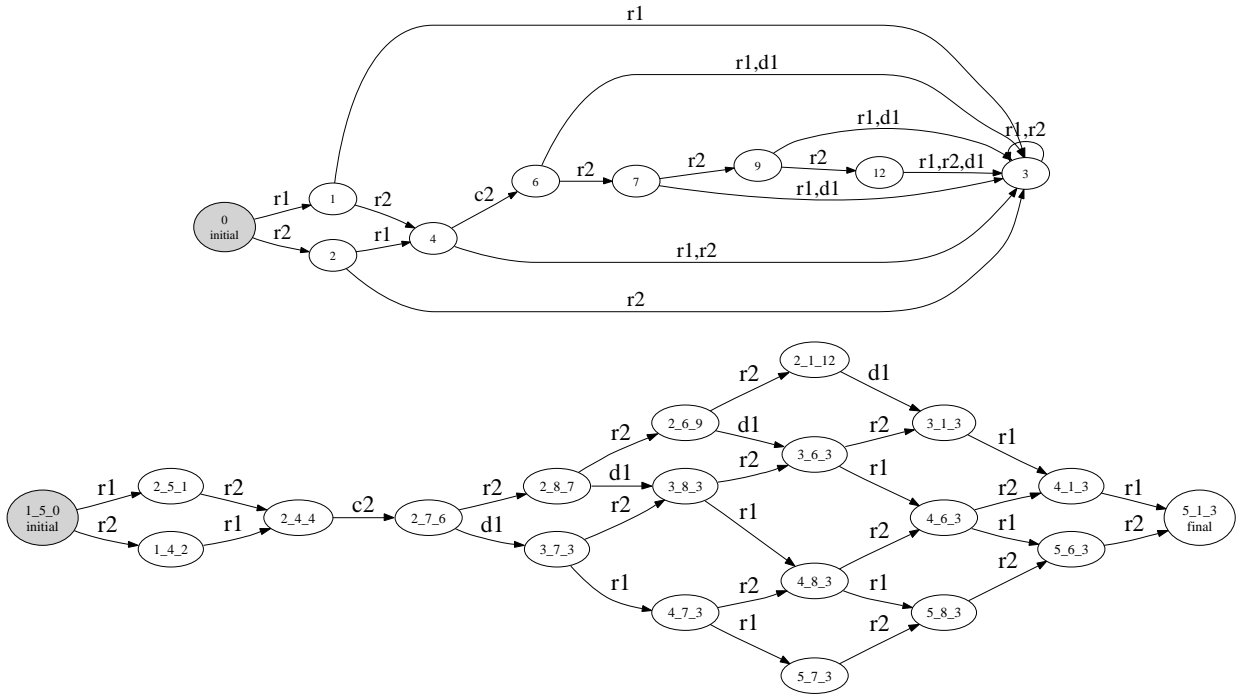


Figure 3.3: *The controller and the supervised system generated by Synthesis.*

```

name controller_at_2_additional_spec;
x = nu -> <r1,r2,c2,d2>x. [c1,d1]x;
<initial={x}>.

name controller_at_4_additional_spec;
x = nu -> <r1,r2,c1,d1>x. [c2,d2]x;
<initial={x}>.

```

In order to obtain a controller $C1$ for the switch at location 2, we have to proceed as explained in Chapter 4: the specifications of the system `sys_spec` have to be divided by those of $C2$ (i.e. `controller_at_4_additional_spec`) and by the process `sys`. The resulting quotient is multiplied by the specifications of $C1$.

```

synthesis>q := quotient sys_spec controller_at_4_additional_spec sys
synthesis>C1_spec := product q controller_at_2_additional_spec
synthesis>C1 := minimize(unmark(control (strategy (game C1_spec))))
synthesis>dot C1 > C1.dot

```

The figure 3.4 depicts $C1$ and the system under its control. One should note that if $C1$ is the only controller in use there remains critical scenarii. For instance $C1$ allows the sequence $(r2, r1, d2, r2)$ which leads both trains in the same place 2.

Finally, the controller $C2$ (drawn on figure 3.5) is computed using the following commands:

```

synthesis>q := quotient sys_spec (sync sys C1)
synthesis>C2_spec := product q controller_at_4_additional_spec
synthesis>C2 := minimize(unmark(control (strategy (game C2_spec))))
synthesis>Csys12 := sync (sync sys C1) C2
synthesis>dot Csys12 > Csys12.dot

```

The last commands compute the whole controlled system. The result is depicted on the figure 3.6. One can verify that the controlled system is safe: there is no **danger** state. Thanks to `Synthesis` we get a system for which the trains can not strike themselves.

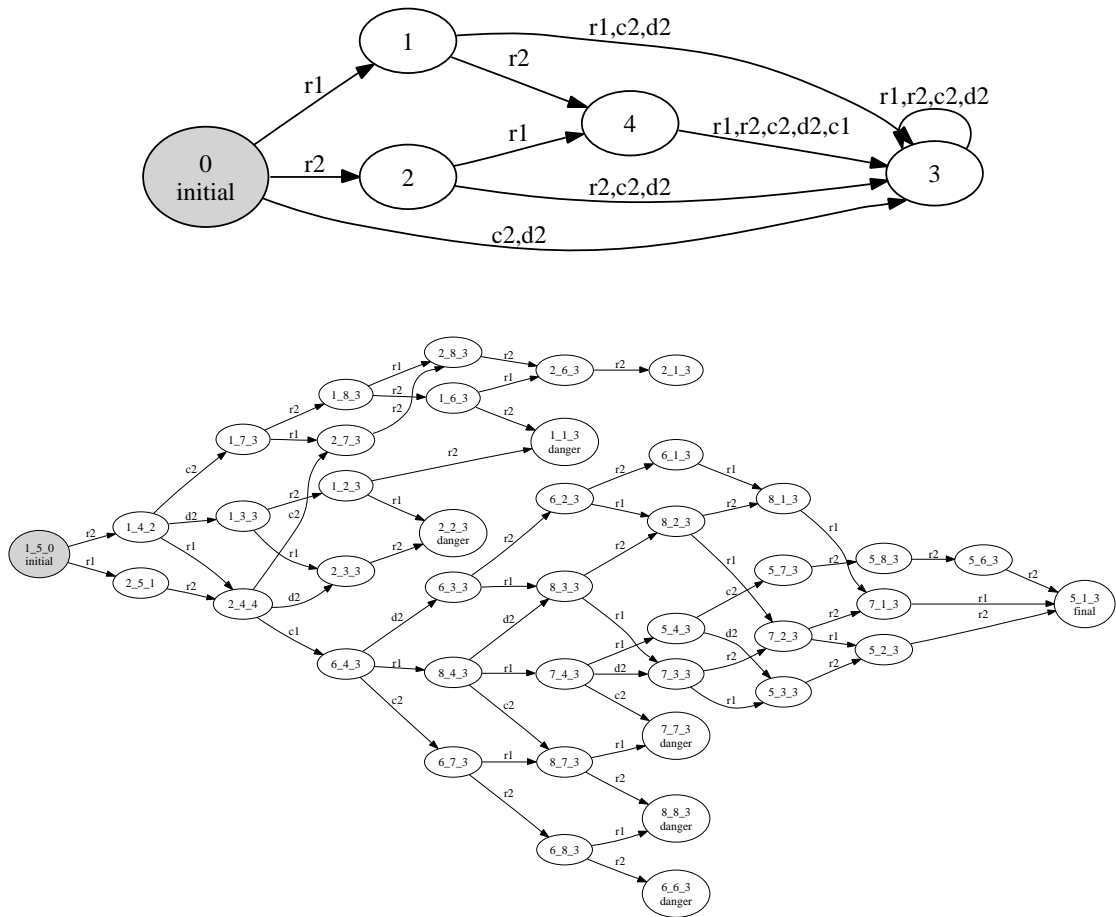


Figure 3.4: The controller $C1$ (upper graph) and the system under its control. One can note that all the states of the controller are complete with respect to uncontrollable events (i.e. r_1 , r_2 , c_2 , d_2).

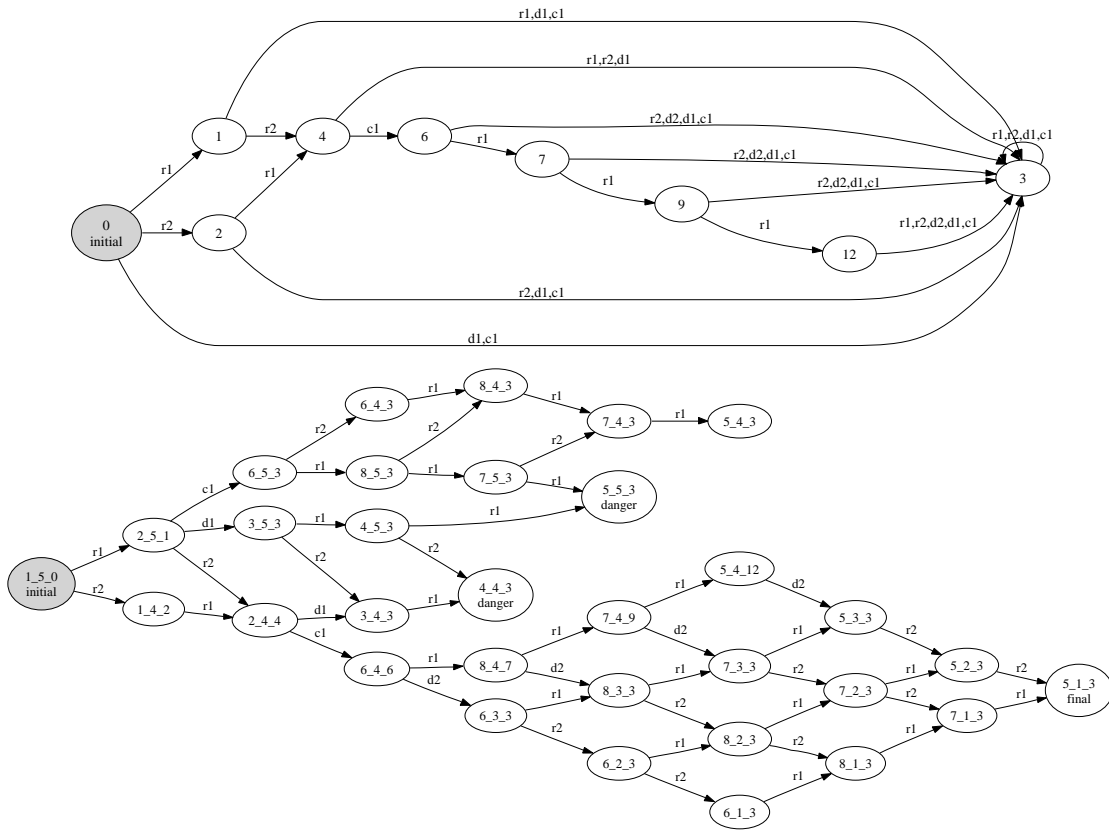


Figure 3.5: The controller $C2$ (upper graph) and the system under its control. States are complete w.r.t r_1 , r_2 , c_1 , d_1 .

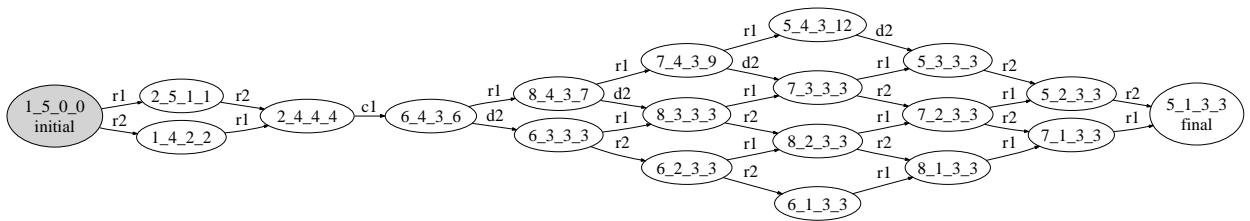


Figure 3.6: The system supervised by the controllers $C1$ and $C2$.

Chapter 4

Theoretical aspects of Synthesis

From now we consider finite and deterministic processes having finite or infinite behaviours. We denote by $P \models \Phi$ the fact that a process P satisfies a property Φ . Two processes P_1 and P_2 over the same set of actions might be synchronized on their common actions; this synchronization is denoted $P_1 \times P_2$.

From this setup, the *controller synthesis problem* can be expressed as follows:

- (*) Given a process P and $n + 1$ properties Φ and Ψ_i for $i = 1 \dots n$, the problem is to find n processes C_i such that $P \times C_1 \times \dots \times C_n \models \Phi$ and, for all $i = 1 \dots n$, $C_i \models \Psi_i$. $P \times C_1 \times \dots \times C_n$ is called the *controlled* or *supervised system*. If $n > 1$ we say that the control is *decentralized*.

In the standard Ramadge and Wonham framework[6] we only require that all the prefixes of controlled behaviours belong in the language defined by Φ ; furthermore, Ramadge and Wonham's framework assumes a predefined set of *controllable* and *uncontrollable* events.

In the context of **Synthesis** we are also interested by infinite behaviours of the supervised system and there is no predefined controllable sets. The requirements, that the controlled system must fulfill, are expressed in terms of μ -calculus formulas. The controllability of events is also expressed by this mean; this allows the user to describe dynamic controllability.

In [8] this problem is addressed and extended for partial observability in [2]. The authors define *division* operations ($/$) such that the controllers C_i in the problem (*) verify the following properties:

$$\begin{aligned} C_1 &\models \Phi / \Psi_n / \dots / \Psi_2 / P \wedge \Psi_1 \\ C_2 &\models \Phi / \Psi_n / \dots / \Psi_3 / (P \times C_1) \wedge \Psi_2 \\ &\vdots \\ C_n &\models \Phi / (P \times C_1 \times \dots \times C_{n-1}) \wedge \Psi_n \end{aligned}$$

Now, in order to synthesize expected controllers, we have to find models of μ -calculus formulas. The problem of finding a model of a μ -calculus formula F is addressed from the game theory point of view: finding a model for F consists in finding a winning strategy in a parity game $G(F)$ built from F .

Synthesis implements all operations required to find the controllers C_i 's as expected in the controller synthesis problem (*). The production of a controller can be seen as a workflow transforming a process and μ -calculus formulas (see figure 4.1). In this chapter we formalize the algorithms used to implement this workflow.

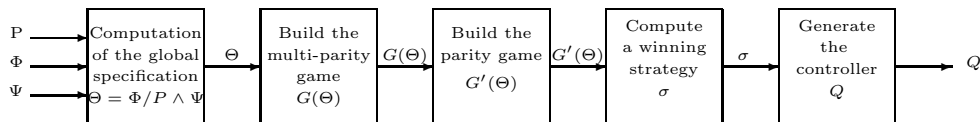


Figure 4.1: The **Synthesis** workflow

4.1 Processes, modal automata, games and strategies

A *process* is a vector $P = \langle A, \Lambda, S, \delta, \lambda, s_0 \rangle$ where A is its set of actions, S is its set of states, s_0 is the initial state and δ is the transition function defined from $S \times A$ to S . Λ is a set of atomic propositions and $\lambda : S \rightarrow 2^\Lambda$ is the labelling function associating to each state a set of propositions supposed true in the state.

Given two processes $P_i = \langle A_i, \Lambda_i, S_i, \delta_i, \lambda_i, s_{0i} \rangle$ for $i = 1, 2$, the *product* of P_1 and P_2 is the process $P_1 \times P_2 = \langle A_1 \cup A_2, \Lambda_1 \cup \Lambda_2, S, \delta, \lambda, (s_{01}, s_{02}) \rangle$ where:

- $S = \{(s_1, s_2) \in S_1 \times S_2 \mid \forall p \in \Lambda_1 \cap \Lambda_2, p \in \lambda_1(s_1) \iff p \in \lambda_2(s_2)\}$
- $\delta((s_1, s_2), a) = (\delta_1(s_1, a), \delta_2(s_2, a))$ if $\delta_1(s_1, a)$ and $\delta_2(s_2, a)$ are defined
- $\lambda((s_1, s_2)) = \lambda_1(s_1) \cup \lambda_2(s_2)$.

Given an alphabet A and a set of atomic propositions Λ , a *modal automaton* is a set of equations of the form:

$$x_i \stackrel{\rho(x)}{=} \bigvee_{j \in J_i} \left(\bigwedge_{p \in K_{i,j} \subseteq \Lambda} p \wedge \bigwedge_{q \in L_{i,j} \subseteq \Lambda} \neg q \wedge \bigwedge_{a \in A} (a)x_{a,i,j} \right)$$

where $x_i, x_{a,i,j}$ are states of the automaton, p and q are atomic propositions and (a) denotes $\langle a \rangle$ or $[a]$. $\rho(x)$ assigns to each equation a vector of natural numbers called *ranks*.

More formally a modal automaton is a vector $\mathcal{A} = \langle A, \Lambda, X, R, \Delta, x_0, \rho \rangle$ where A is an alphabet of actions, X is its state space and x_0 is the initial state, Λ is a set of atomic propositions, and:

- $\Delta : X \rightarrow 2^R$ assigns to each state a set of *rules* taken from the set $R \subseteq 2^\Lambda \times 2^\Lambda \times 2^A \times X^A$. A *rule* $r \in R$ represents one conjunction of the equation system; r is a vector $\langle P, N, E, \sigma \rangle$ such that:
 - $P \subseteq \Lambda$ (resp. $N \subseteq \Lambda$) is a set of positive (resp. negative) atomic propositions and $P \cap N = \emptyset$;
 - E is the set of actions that are existentially quantified;
 - $\sigma : A \rightarrow X$ is a mapping assigning to each action the equation that must be satisfied after the given action.
- $\rho : X \rightarrow \mathbb{N}^n$ is a mapping assigning to each state its *rank*. If $n = 1$ we say that ρ is a *parity* condition otherwise it is a *multi-parity* condition.

Sometimes we use a particular state \top (*top*). For this state we set: $\Delta(\top) = \langle \emptyset, \emptyset, \emptyset, A \times \{\top\} \rangle$ and $\rho(\top) = \langle 0, \dots, 0 \rangle$.

A modal automaton accepts processes. A process P is accepted by a modal automaton \mathcal{A} is denoted by $P \models \mathcal{A}$. This acceptance relation is defined in terms of winning strategies in games built from \mathcal{A} and the accepted processes.

A *multi-parity game* is a vector $G = \langle V, V_0, V_1, v_0, E, \Omega \rangle$ where $V = V_0 \uplus V_1$ is the set of positions and v_0 is the starting position, $E \subseteq V \times V$ denotes allowed moves between positions (if $(v, v') \in E$ then v' is called a *successor* of v) and $\Omega : V \rightarrow \mathbb{N}^n$ is a function assigning to each state a vector of natural numbers called *ranks* (or *priorities*); if $n = 1$ the game is called a *parity* game.

We consider games with two players 0 and 1, pushing a token over positions in V . A *play* is a sequence of positions starting at v_0 . A turn proceeds in the following way: if the token is in a position $v \in V_0$ then the player 0 chooses v' a successor of v , otherwise the player 1 makes the choice. Once the token is moved onto v' a new turn is started. If a player can not move the token (i.e there is no edge outgoing from v in E) he loses the play. Infinite plays are allowed; in this case, for a play $v_0 v_1 \dots$, the player 0 wins if the greatest rank encountered infinitely often in the sequence is even (in the case of multi-parity games, the condition must hold for each component of $\Omega(v_i)$).

A play is *maximal* if it is infinite or terminates in a position without outgoing edge. So, a maximal play is *winning* for player 0 if it satisfies the parity condition or ends in a vertex belonging to V_1 without successor.

For every sequence of vertices \vec{v} ending in a vertex v from V_0 , a *strategy* σ for player 0 is a function such that $\sigma(\vec{v}) \in V$. We require $\sigma(\vec{v})$ to be defined if v has a successor.

A play *respects* a strategy σ if it is a finite or infinite sequence of positions $v_0v_1\dots$ such that $v_{i+1} = \sigma(v_0\dots v_i)$ for every i with $v_i \in V_0$.

A strategy for player 0 is *winning from a vertex* v if and only if every maximal play starting in v and respecting the strategy is winning for player 0. We say that a strategy for player 0 is *winning* if it is winning from every position from which there exists a winning strategy for player 0.

4.2 Product and quotient operations

4.2.1 Product of modal automata

We consider two modal automata $\mathcal{A}_i = \langle A, \Lambda, X_i, R_i, \Delta_i, x_{0_i}, \rho_i : X_i \rightarrow \mathbb{N}^{n_i} \rangle$ for $i = 1, 2$ and we build the automaton $\mathcal{A}_1 \times \mathcal{A}_2$ which verifies: $P \models \mathcal{A}_1 \times \mathcal{A}_2$ if and only if $P \models \mathcal{A}_1$ and $P \models \mathcal{A}_2$.

$\mathcal{A}_1 \times \mathcal{A}_2$ is the vector $\langle A, \Lambda, X, R, \Delta, x_0, \rho \rangle$ defined by:

- $X = X_1 \times X_2$ and $x_0 = (x_{0_1}, x_{0_2})$
- $\Delta((x_1, x_2)) = \{r_1 \times r_2 \mid r_i = \langle P_i, N_i, E_i, \sigma_i \rangle \in \Delta_i(x_i) \text{ for } i = 1, 2\}$ where $r_1 \times r_2$ is the product of rules r_1 and r_2 which is defined if $P_1 \cap N_2 = \emptyset$ and $P_2 \cap N_1 = \emptyset$ by the vector $\langle P, N, E, \sigma \rangle$ with $P = P_1 \cup P_2$, $N = N_1 \cup N_2$, $E = E_1 \cup E_2$ and $\sigma(a) = (\sigma(x_1), \sigma(x_2))$.

The transformation of modalities can be seen as a table:

$r_2 \setminus r_1$	$\langle \cdot \rangle$	$[\cdot]$
$\langle \cdot \rangle$	$\langle \cdot \rangle$	$\langle \cdot \rangle$
$[\cdot]$	$\langle \cdot \rangle$	$[\cdot]$

- ρ is defined from X into $\mathbb{N}^{n_1+n_2}$ by the concatenation of ranks from \mathcal{A}_1 and \mathcal{A}_2 : $\rho((x_1, x_2)) = \rho_1(x_1) \cdot \rho_2(x_2)$.

4.2.2 Quotient \mathcal{A}/P

Given a modal automaton $\mathcal{A} = \langle A, \Lambda, X, R, \Delta, x_0, \rho \rangle$ and a process $P = \langle A, \Lambda, S, \delta, \lambda, s_0 \rangle$, we compute a modal automaton, noted \mathcal{A}/P , such that for any process Q : $P \times Q \models \mathcal{A} \iff Q \models \mathcal{A}/P$. \mathcal{A}/P is the vector $\langle A, \Lambda, X_f, R_f, \Delta_f, x_{0_f}, \rho_f \rangle$ such that:

- $X_f = (X \times S) \cup \{\top\}$.
- $\Delta((x, s)) = \{r/s \mid r \in \Delta(x)\}$. If $r = \langle P, N, E, \sigma \rangle$ then r/s is defined if and only if $P \cap (\Lambda \setminus \lambda(s)) = \emptyset$, $N \cap \lambda(s) = \emptyset$ and for all $a \in E$, $\delta(s, a)$ is defined. If r/s exists then it is the vector $r/s = \langle P_f, N_f, E_f, \sigma_f \rangle$ defined by:
 - $P_f = P \cup \lambda(s)$
 - $N_f = N \cup (\Lambda \setminus \lambda(s))$
 - $E_f = E$
 - $\sigma_f(a) = \begin{cases} (\sigma(x), \delta(s, a)) & \text{if } \delta(s, a) \text{ is defined} \\ \top & \text{otherwise} \end{cases}$
- $x_{0_f} = (x_0, s_0)$
- $\rho_f((x, s)) = \rho(x)$

4.2.3 Quotient $\mathcal{A}_1/\mathcal{A}_2$

Given two modal automata $\mathcal{A}_i = \langle A, \Lambda, X_i, R_i, \Delta_i, x_{0_i}, \rho_i : X_i \rightarrow \mathbb{N}^{n_i} \rangle$ we compute a modal automaton, noted $\mathcal{A}_1/\mathcal{A}_2$, such that for any process Q , $Q \models \mathcal{A}_1/\mathcal{A}_2 \iff \exists P, P \models \mathcal{A}_2 \wedge P \times Q \models \mathcal{A}_1$. $\mathcal{A}_1/\mathcal{A}_2$ is the vector $\langle A, \Lambda, X_j, R_j, \Delta_j, x_{0_j}, \rho_j \rangle$ such that:

- $X_j = (X_1 \times X_2) \cup \{\top\}$.
- $\Delta((x_1, x_2)) = \{r_1/r_2 \mid r_1 \in \Delta_1(x_1) \wedge r_2 \in \Delta_2(x_2)\}$. If $r_i = \langle P_i, N_i, E_i, \sigma_i \rangle$ then r_1/r_2 is defined if and only if $P_1 \cap N_2 = \emptyset$ and $N_1 \cap P_2 = \emptyset$. If r_1/r_2 exists then it is the vector $r_1/r_2 = \langle P_j, N_j, E_j, \sigma_j \rangle$ defined by:
 - $P_j = P_1 \cup P_2$
 - $N_j = N_1 \cup N_2$
 - $E_j = E_1$
 - $\sigma_j(a) = \begin{cases} (\sigma_1(x_1), \sigma_2(x_2)) & \text{if } a \in E_1 \cup E_2 \\ \top & \text{otherwise} \end{cases}$

The transformation of modalities can be seen as a table:

$r_2 \backslash r_1$	$\langle \cdot \rangle$	$[\cdot]$
$\langle \cdot \rangle$	$\langle \cdot \rangle$	$[\cdot]$
$[\cdot]$	$\langle \cdot \rangle$	\top

- $x_{0_j} = (x_{0_1}, x_{0_2})$
- $\rho_j : X \rightarrow \mathbb{N}^{n_1+n_2}$ is the concatenation of ranks from \mathcal{A}_1 and \mathcal{A}_2 : $\rho((x_1, x_2)) = \rho_1(x_1) \cdot \rho_2(x_2)$.

4.3 From modal automaton to game

Let $\mathcal{A} = \langle A, \Lambda, X, R, \Delta, x_0, \rho : X \rightarrow \mathbb{N}^n \rangle$ be a modal automaton. In this section we construct a game $G(\mathcal{A})$ such that \mathcal{A} is satisfiable (i.e. there exists a process P such that $P \models \mathcal{A}$) if and only if there exists a winning strategy from the initial position of $G(\mathcal{A})$.

The game $G(\mathcal{A}) = \langle V, V_0, V_1, v_0, E, \Omega : V \rightarrow \mathbb{N}^n \rangle$ is defined as follows:

- $V_0 = X$ and $V_1 = R$
- $v_0 = x_0$
- $E \subseteq V \times V$ is defined by:
 - for all $x \in X$ and for all $r \in \Delta(x)$, $(x, r) \in E$
 - for all $r = \langle p, n, e, \sigma \rangle \in R$ and for all $x \in \sigma(e)$, $(r, x) \in E$
- And the ranks of positions is defined as follows:
 - for all $v \in V_0$, $\Omega(v) = \rho(v)$
 - for all $v \in V_1$, $\Omega(v) = \langle 0, \dots, 0 \rangle$

4.4 From multi-parity to parity

Let $G = \langle V, V_0, V_1, v_0, E, \Omega : V \rightarrow \mathbb{N}^n \rangle$ be a game with a multi-parity acceptance condition and with a starting position v_0 . Let $r = v_0 v_1 \dots$ be an infinite play with its associated sequence of ranks $\Omega(r) = \Omega(v_0)\Omega(v_1)\dots$. This play is winning for player 0 if, for each $i = 1 \dots n$, $\lim_{j \rightarrow \infty} \text{sup}(\Omega(v_j)[i])$ is even (where $\Omega(v_j)[i]$ denotes the i th component of $\Omega(v_j)$).

In this section we present an algorithm transforming the multi-parity game G into an equivalent parity game G' . This algorithm is derived from [4] and is proposed by A. Arnold. Let us define the game G' .

Let $\text{max}_i = \text{max}\{\Omega(v)[i] \mid v \in V\}$ be the maximal rank reached by each component of Ω ; we shall note $L = \Sigma \text{max}_i$. We consider the set of shuffles $M_{\sqcup} \subseteq (\mathbb{N} \times \{1, \dots, n\})^*$:

$$M_{\sqcup} = (\text{max}_1, 1)(\text{max}_1 - 1, 1) \dots (1, 1) \sqcup \dots \sqcup (\text{max}_n, n)(\text{max}_n - 1, n) \dots (1, n)$$

The word $(\text{max}_1, 1) \dots (1, 1) \dots (\text{max}_n, n) \dots (1, n) \in M_{\sqcup}$ is noted M_0 .

For example, if $n = 2$, $\text{max}_1 = 3$ and $\text{max}_2 = 2$ then $M_{\sqcup} = 3^1 2^1 1^1 \sqcup 2^2 1^2 = \{ 2^2 1^2 3^1 2^1 1^1, 2^2 3^1 1^2 2^1 1^1, 2^2 3^1 2^1 1^2 1^1, 2^2 3^1 2^1 1^1 1^2, 3^1 2^2 2^1 2^1 1^1, 3^1 2^2 2^1 1^2 1^1, 3^1 2^2 2^1 1^1 1^2, 3^1 2^1 2^2 1^2 1^1, 3^1 2^1 2^2 1^1 1^2, 3^1 2^1 1^1 2^2 1^2 \}$ (we use superscripts x^y in place of (x, y)). Each element of M_{\sqcup} can be considered as a mapping from $[1, L]$ into $\mathbb{N} \times \{1, \dots, n\}$.

Given a word $w \in M_{\sqcup}$ and a vector of parities $\vec{p} = \langle p_1, \dots, p_n \rangle \in \mathbb{N}^n$ we define:

- $\text{pos}(w, \vec{p}) = \min\{i \in [1, L] \mid \exists j, w(i) = (p_j, j)\}$ is the index of the first p_j occurring in w .
- $\rho(w, \vec{p}) = w(\text{pos}(w, \vec{p}))$
- $\text{succ}(w, \vec{p}) = w'$ is the shuffle such that if $\rho(w, \vec{p}) = (p, k)$ then:
 - $w' = w$ if p is odd;
 - w' starts like w without letters (p', k) such that $p' \leq p$, and terminates with $(p, k)(p-1, k) \dots (1, k)$ if p is even.
- $l(w, \vec{p}) = L - \text{pos}(w, \vec{p}) + 1$ is the length of the suffix of w starting at position $\text{pos}(w, \vec{p})$

For example, if $w = 3^1 3^2 2^2 2^1 1^2 1^1$ and $\vec{p} = \langle 2, 2 \rangle$ then $\text{pos}(w, \vec{p}) = 3$, $\rho(w, \vec{p}) = (2, 2)$ and $\text{succ}(w, \vec{p}) = 3^1 3^2 2^1 1^1 2^2 1^2$.

The parity game $G' = \langle V', V'_0, V'_1, E', \Omega' : V' \rightarrow \mathbb{N}, v'_0 \rangle$ is built from G as follows. Each position from V' is a triple from $V \times M_{\sqcup} \times \mathbb{N}$. Starting from the initial position $v'_0 = (v_0, \text{succ}(M_0, \Omega(v_0)), l(M_0, \Omega(v_0)))$, we add an edge from (v, m, l) to (v', m', l') if $(v, v') \in E$, $m' = \text{succ}(m, \Omega(v'))$ and $l' = l(m, \Omega(v'))$. Ω' is defined by:

$$\Omega'((v, \text{succ}(m, \Omega(v)), l)) = 2l + \begin{cases} 2 & \text{if } \rho(m, \Omega(v)) = (p, k) \text{ and } p \text{ is even} \\ 1 & \text{if } p \text{ is odd} \end{cases}$$

Theorem 1 *Let $r = v_0 \dots$ be a maximal play of G . r is winning for player 0 in G iff its corresponding play in G' is winning for player 0.*

Proof The finite case is straightforward. Let's consider the infinite case.

Let $r' = (v_0, s_0, l_0)(v_1, s_1, l_1) \dots$ be the play in G' corresponding to r . $s_0 = \text{succ}(M_0, \Omega(v_0))$ and for all $i > 0$, s_i denotes the shuffle word $\text{succ}(s_{i-1}, \Omega(v_i))$. For $i = 1 \dots n$, we note $m_i = \lim_{j \rightarrow \infty} \text{sup}(\Omega(v_j)[i])$ the maximal rank of the i th component of Ω for the positions appearing infinitely often along r (i.e. r is winning iff all w_i 's are even).

r' is infinite so, there exists a position, say (v_k, s_k, l_k) , such that the corresponding suffix of r' , $(v_k, s_k, l_k)(v_{k+1}, s_{k+1}, l_{k+1}) \dots$, contains only the positions repeated infinitely often.

\Rightarrow Here we assume that r is winning.

One can remark that, due to the definition of succ and to the shuffle structure, all the couples (p, i) such that $p \leq m_i$ are "accumulated" at the end of the memory words when m_i is even. Thus, there

exists $k' \geq k$ such that for all $j \geq k'$, $s_j = s.w_j$ where s is a constant word (s contains all the letters (p, i) with $p > m_i$) and w_j is a shuffle of the n words $(m_i, i)(m_i - 1, i) \dots (1, i)$ and, for all (v_j, s_j, l_j) , we have $l_j \leq \Sigma m_i$. Due to the shuffle structure, each w_j starts with some m_l which is repeated infinitely often; so there exists a position where (m_l, l) is selected (infinitely often) and for which l_j takes its maximal value Σm_i . m_l is even so the rank associated to the position is the maximal value $2\Sigma m_i + 2$; we can conclude that r' is winning.

\Leftarrow Now we assume that r is not winning i.e. there exists i_0 such that m_{i_0} is odd. Every m_i is repeated infinitely often and odd m_i 's do not modify memory words. Thus, for some odd parity m_k , there exists a position from which the memory words have the form $s_j = s.(m_k, k).w_j$ where s contains no m_i and remains constant along r' and w_j is modified by even m_i 's. Since $s.(m_k, k)$ is never modified and m_k occurs infinitely often, there exists an edge between two positions $(v_{c-1}, s.(m_k, k).w_{c-1}, l_{c-1})$ and $(v_c, s.(m_k, k).w_{c-1}, l_c)$ such that $\Omega(v_c) = \langle \dots, m_k, \dots \rangle$.

The rank associated to the position $(v_c, s.(m_k, k).w_{c-1}, l_c)$ is equal to $2l_c + 1$ which is odd and repeated infinitely often. This rank is maximal since l_c is maximal thus r' is not winning for player 0.

Encoding M_{\sqcup} : The multi-parity game is unfolded using elements of M_{\sqcup} . The length of each shuffle generated by the algorithm is $L = \Sigma \max_i$ the sum of maximal ranks of the game. This length can be reduced to $L/2$ by replacing each parity p by $\lceil \frac{p}{2} \rceil$ in all the algorithm.

One can remark that all words generated from M_0 using the *succ* operation have the following property: if $w = w_1(2p, i)(2p - 1, i)w_2 \in M_{\sqcup}$ then for all $\vec{p} \in \mathbb{N}^n$, $\text{succ}(w, \vec{p}) = w'_1(2p, i)(2p - 1, i)w'_2$. In other words, *succ* preserves the consecutivity of $(2p, i)$ and $(2p - 1, i)$. Since we start with the word M_0 (which has the property), any word w generated by the algorithm is such that, if $1 \leq 2p \leq \max_i$ then $(2p, i)(2p - 1, i)$ is a factor of w . We call $M'_{\sqcup} \subseteq M_{\sqcup}$ the set of words generated from M_0 .

Each word of M'_{\sqcup} can be encoded using half of its letters by replacing each couple $(2p, i)(2p - 1, i)$ by (p, i) . In fact, instead of starting with M_0 we start with $\lceil \frac{M_0}{2} \rceil = (\lceil \frac{\max_1}{2} \rceil, 1) \dots (1, 1) \dots (\lceil \frac{\max_n}{2} \rceil, n) \dots (1, n)$ and each multi-parity $\vec{p} = \langle p_1, \dots, p_n \rangle$ of G is translated into $\lceil \frac{\vec{p}}{2} \rceil = \langle \lceil \frac{p_1}{2} \rceil, \dots, \lceil \frac{p_n}{2} \rceil \rangle$. Then, instead of looking for some (p_j, j) into a word of M'_{\sqcup} , we look for $(\lceil \frac{p_j}{2} \rceil, j)$ into a successor of $\lceil \frac{M_0}{2} \rceil$. In order to compute the successors of $\lceil \frac{M_0}{2} \rceil$ we just have to ensure that moves of $(\lceil \frac{p_i}{2} \rceil, j)$ are the same than those of $(p_j, j)(p_j - 1, j)$ and that the good parity is assigned to each position. In order to ensure this condition we just have to test the parity of p in place of $\lceil \frac{p}{2} \rceil$ in the definitions of *succ* and Ω' :

- $\text{succ}(w, \vec{p}) = w'$ is the shuffle such that if $\rho(w, \vec{p}) = (\lceil \frac{p}{2} \rceil, k)$ then:
 - $w' = w$ if p is odd;
 - w' starts like w without letters (p', k) such that $p' \leq \lceil \frac{p}{2} \rceil$, and terminates with $(\lceil \frac{p}{2} \rceil, k)(\lceil \frac{p}{2} \rceil - 1, k) \dots (1, k)$ if p is even.
- $\Omega'((v, \text{succ}(m, \Omega(v)), l)) = 2l + \begin{cases} 2 & \text{if } \rho(m, \Omega(v)) = (\lceil \frac{p}{2} \rceil, k) \text{ and } p \text{ is even} \\ 1 & \text{if } p \text{ is odd} \end{cases}$

4.5 Computation of a winning strategies

Given a parity game $G(\mathcal{A})$ we compute winning strategies in G using algorithms similar to those presented in [3]. The algorithms are not exactly the same because, in the framework of **Synthesis**, acceptance conditions for infinite plays are defined using the maximal parity occurring infinitely often. In this section we present the main ideas of the (adapted) algorithms taken from [3].

4.5.1 Memories over odd priorities

Given a parity game $G = \langle V, V_0, V_1, v_0, E, \Omega \rangle$ we denote by d its maximal priority that we assume to be odd (if d is even we consider $d + 1$). For each odd priority $p \in \{1, \dots, d\}$ we denote n_p the size of the set

$\{v \in V \mid \Omega(v) = p\}$. Now we define the set of memories:

$$M(G) = \{\perp, \top\} \cup \prod_{1 \leq p \leq d, p \text{ odd}} \{0, \dots, n_p\}$$

In the sequel we will consider the vectors $\vec{0} = (0, \dots, 0)$ and $\vec{n} = (n_1, \dots, n_d)$. In [3] the game G is said to be \vec{n} -bounded.

$M(G)$ is ordered by the relation \prec defined for all $\vec{m} = (m_1, \dots, m_d), \vec{m}' = (m'_1, \dots, m'_d) \in M(G) \setminus \{\perp, \top\}$ by:

- $\perp \prec \vec{m} \prec \top$
- $\vec{m} \prec \vec{m}' \iff \exists i \in \{1, \dots, d\}, m_i < m'_i \wedge \forall j > i, m_j = m'_j$

Now we define some functions defining transformations of $\vec{m} = (m_1, \dots, m_p) \in M(G)$ for a given priority p :

- $up(\vec{m}, p) = \begin{cases} \top & \text{if } \vec{m} = \top \\ inc(\vec{m}, p) & \text{if } p \text{ is odd} \\ zero(\vec{m}, p) & \text{if } p \text{ is even} \end{cases}$

where:

- $zero(\vec{m}, p) = (0, \dots, 0, m_{p+1}, \dots, m_d)$
- $inc(\vec{m}, p) = \begin{cases} (0, \dots, 0, m_p + 1, m_{p+2}, \dots, m_d) & \text{if } p \leq d \text{ and } m_p < n_p \\ inc(\vec{m}, p + 2) & \text{if } m_p = n_p \text{ and } p < d \\ \top & \text{otherwise} \end{cases}$

- $down(\vec{m}, p) = \begin{cases} \perp & \text{if } \vec{m} = \perp \\ dec(\vec{m}, p) & \text{if } p \text{ is odd} \\ max(\vec{m}, p) & \text{if } p \text{ is even} \end{cases}$

where:

- $max(\vec{m}, p) = (n_1, \dots, n_{p-1}, m_{p+1}, \dots, m_d)$
- $dec(\vec{m}, p) = \begin{cases} (n_1, \dots, n_{p-2}, m_p - 1, m_{p+2}, \dots, m_d) & \text{if } p \leq d \text{ and } 0 < m_p \\ dec(\vec{m}, p + 2) & \text{if } m_p = 0 \text{ and } p < d \\ \perp & \text{otherwise} \end{cases}$

4.5.2 Winning strategies and controller synthesis

In order to get winning strategies in a parity game $G = \langle V, V_0, V_1, v_0, E, \Omega \rangle$ we need to compute a mapping $Mmax : V \rightarrow M(G)$. We obtain $Mmax$ using the following algorithm:

1. For each $v \in V$, set $Mmax(v)$ to \vec{n}
2. Find a position v such that $down(\vec{m}', \Omega(v)) \prec Mmax(v)$ where

$$\vec{m}' = \begin{cases} \max\{Mmax(v') \mid (v, v') \in E\} & \text{if } v \in V_0 \\ \min\{Mmax(v') \mid (v, v') \in E\} & \text{if } v \in V_1 \end{cases}$$

3. If v does not exist then the algorithm returns.
4. If v exists then we set $Mmax(v)$ to $down(\vec{m}', \Omega(v))$
5. Repeat from step 2.

For every $v \in V$ we define $W(v) = \{v' \mid (v, v') \in E \wedge up(Mmax(v), \Omega(v)) \preceq Mmax(v')\}$.

The mapping W allows us to build memoryless strategies which are winning from v_0 : given a position $v \in V_0$ a winning strategy consists to select a successor into $W(v)$. Of course, if no such successor exists for v_0 then no winning strategy has been found and we can not generate a valid controller.

Every memoryless strategy $\tau : V \rightarrow V$ such that $\tau(v) \in W(v)$ allows us to generate one controller C_τ . In particular one might be interested by two classes of controllers corresponding to two classes of strategies. These strategies select the positions which, respectively, minimize or maximize the mapping $Mmax$ in the set $W(v)$:

- $\tau_{\min}(v) \in \{v' \in W(v) \mid \forall v'' \in W(v), Mmax(v') \preceq Mmax(v'')\}$
- $\tau_{\max}(v) \in \{v' \in W(v) \mid \forall v'' \in W(v), Mmax(v'') \preceq Mmax(v')\}$

Controllers generated with a τ_{\max} strategy are the most permissive ones while those generated with τ_{\min} are the most despotic.

Controller synthesis Now we are able to build a model for a modal automaton $\mathcal{A} = \langle A, \Lambda, X, R, \Delta, x_0, \rho \rangle$. As specified in Section 4.3, \mathcal{A} is transformed into a parity game $G(\mathcal{A}) = \langle V, V_0, V_1, v_0, E, \Omega \rangle$ where $V_0 = X$ correspond to the states of the automaton, the initial position is the initial state $v_0 = x_0$ and $V_1 = R$ is the set of its rules. A memoryless strategy for player 0 in $G(\mathcal{A})$ is thus a mapping from X into R .

For each memoryless strategy $\tau : X \rightarrow R$ one can create a valid controller (without labelling) $C_\tau = \langle A, \emptyset, X, \delta, \emptyset, x_0 \rangle$ where the transition function $\delta : X \times A \rightarrow X$ is defined as follows: for all $x \in X$, $\delta(x, a) = \sigma(a)$ where σ is the successor function of the rule $\tau(x) = \langle P, N, E, \sigma : A \rightarrow X \rangle \in R$.

Bibliography

- [1] A. Arnold, D. Bégay, and P. Crubillé. *Construction and analysis of transition systems with MEC*. World Scientific, 1994.
- [2] A. Arnold, A. Vincent, and I. Walukiewicz. Games for synthesis of controllers with partial observation. *Theoretical Computer Science*, 303(1):7–34, June 2003.
- [3] J. Bernet, D. Janin, and I. Walukiewicz. Permissive strategies: from parity games to safety games. *Informatique Thorique et Applications (RAIRO)*, 36:251–275, 2002.
- [4] S. Dziembowski, M. Jurdziński, and I. Walukiewicz. How much memory is needed to win infinite games? In *Proceedings, Twelfth Annual IEEE Symposium on Logic in Computer Science*, pages 99–110, Warsaw, Poland, 29 –2 1997.
- [5] E.R. Gansner and S.C. North. An open graph visualization system and its applications to software engineering. *Software — Practice and Experience*, 30(11):1203–1233, 2000.
- [6] P.J.G. Ramadge and W.M. Wonham. The control of discrete event systems. In *Proceedings of the IEEE*, volume 77, pages 81–98, January 1989.
- [7] C. Ramey. The GNU Readline. <http://cnswww.cns.cwru.edu/php/chet/readline/rltop.html>, 2005.
- [8] A. Vincent. Synthèse de contrôleurs et stratégies gagnantes dans les jeux de parité. In *MSR: Congrès Modélisation des Systèmes Réactifs*, pages 87–98. Hermès Science, October 2001.
- [9] A. Vincent. *Conception et réalisation d'un vérificateur de modèles AltaRica*. PhD thesis, LaBRI, Université Bordeaux 1, December 2003.

Appendix A

Commands

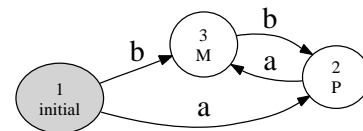
This appendix lists the current commands implemented into **Synthesis**. From the shell prompt the user might execute the **help** command in order to get some informative message about allowed commands. The reader can find the BNF syntax of the **Synthesis** script at the page 33.

I/O commands

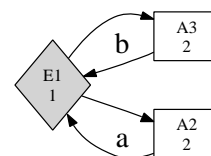
In general the commands of **Synthesis** print some message to the user. By default a command prints its output on the standard output of the program (i.e. the console). Using the same notation than UNIX shells, the outputs of **Synthesis** commands might be redirected (**>**) into a file or appended at the end (**>>**). Indeed, this redirection mechanism is the only way proposed by **Synthesis** to save objects.

dot *object₁...object_n*: This command displays its arguments into the **dot[5]** file format. Its arguments must be graph-based objects i.e. processes and games. Each type of object has its own layout. The figure below depicts the result generated by **dot** applied to the following process and game.

```
transition_system P;
1 |- a -> 2, b -> 3;
2 |- a -> 3;
3 |- b -> 2;
<initial={1};P={2};M={3}>.
```



```
game GP;
E1@1 -> A2, -> A3;
A2@2 |- a -> E1;
A3@2 |- b -> E1;
<initial={E1}>.
```



load *filename₁...filename_n*: With this command **Synthesis** reads the files specified by *filename_i* (no wildcard substitution is implemented in the current version); existing objects are replaced by those readed from files. For each *filename_i* the file format is identified using the filename extension. The extension recognized by **Synthesis** are:

- .syn for synthesis scripts;
- .mec or .mec4 for Mec 4 transition systems;

`.fam` for automaton specifications;

`.game` for parity games.

`mecv object1 ... objectn`: This command translates processes and modal automata into Mec V[9] file format. Processes are translated into AltaRica nodes. Automata translation is not yet implemented but they will be translated into the equation systems supported by Mec V.

`print arg1 ... argn`: This command is usefull only when `Synthesis` is used in batch mode. It prints its arguments as strings of characters; when printed, the arguments are separated by newlines. Example:

```
synthesis>print hello world
hello
world
synthesis>print "hello world"
hello world
synthesis>
```

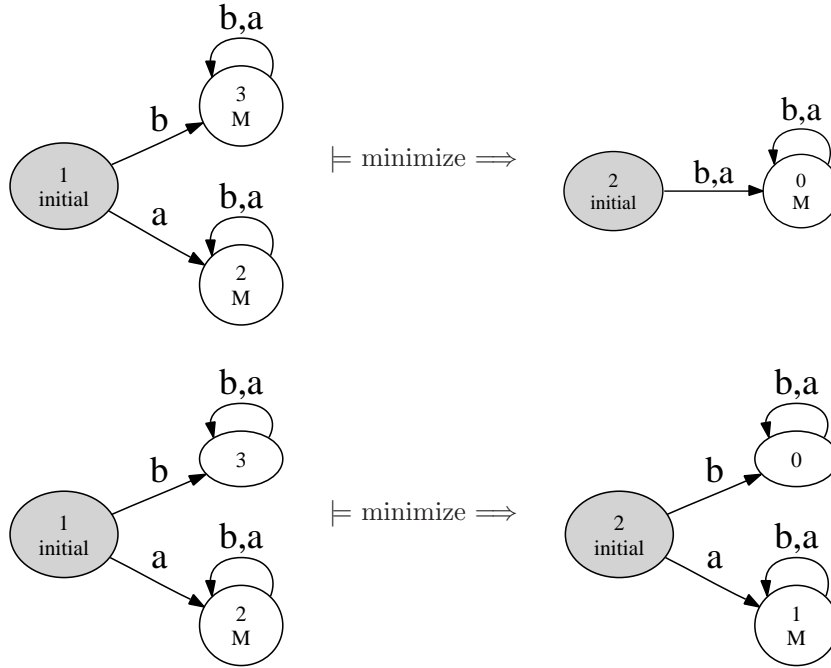
`show object1 ... objectn`: This command prints its arguments in their native format (see the `load` command). Of course, if objects with different types are `showed` into a same file, the user will not be able to reload it later.

Synthesis pipeline

`control g`: This command generates a process (a transition system) from the game *g*. We assume that the game argument is the result of a computation made with the command `strategy` i.e. the computation of winning strategies in the game. The resulting process is computed with the algorithm presented in Section 4.5.2.

`game a`: It computes a parity game from the automaton *a* (see Section 4.1). The acceptance condition used by *a* might be a parity or multi-parity condition; in both cases the acceptance condition is translated into a parity condition.

`minimize p`: This command implements an algorithm for minimizing the number of states in a word-automaton. The initial partition of the state space is the intersection of atomic properties of the process *p*; we consider that all states are accepting. The following figures show the result of `minimize` on a process with different sets of properties; in the first case both states 2 and 3 have the property *m* while in the second case, only 3 has it:



parity a : This command translates the model-automaton a dotted with a multi-parity condition into an automaton with parity condition. This operation is implicitly done by the command **game**. This command implements the algorithm presented in Section 4.4.

product $a_1 a_2$: It computes the intersection of the modal automata a_1 and a_2 . The acceptance condition of the resulting automaton is a multi-parity condition. The algorithm is presented in Section 4.2.1.

quotient $a|p$: This command implements the quotient of a modal automaton by a process or by another automaton. The acceptance condition of the resulting automaton is a multi-parity condition. The algorithms are presented in Sections 4.2.2 and 4.2.3.

strategy g : This command computes winning strategies for the game g . We apply the algorithm of Bernet, Janin and Walukiewicz presented in Section 4.5. The result of this command is a game (which is a subgraph of g) representing the strategies computed by the algorithm. With this result the user can generate a process with the command **control**.

sync $p_1 p_2$: This function computes the synchronized product of two processes p_1 and p_2 . The function checks the consistency of state sets (*marks*) of processes. It might emit a warning message if it fails to synchronize two states because of inconsistency in their sets of marks. This is shown on the following example:

```

synthesis>show P
// # states = 3
// # transitions = 4
transition_system P;
3 |- b -> 2;
2 |- a -> 3;
1 |- a -> 2,
    b -> 3;
<initial = {1}; P = {2}; M = {3}>.
synthesis>show PP
// # states = 3

```

```

// # transitions = 2
transition_system PP;
1 |- a -> 3;
3 |- a -> 2;
<initial={1};P={3};M={}>.
synthesis>
synthesis>sync P PP
warning : states '3' and '2' are \
inconsistent w.r.t their properties:
A: -P M
B: -P -M

```

```

// # states = 2
// # transitions = 1
transition_system $$;
1_1 |- a -> 2_3;

```

```

2_3 |- ;
<P = {2_3}; M = {}; initial = {1_1}>.
synthesis>

```

unmark $p|a$: This command removes propositional constants from its argument. The following lines show the unmarking of a process:

```

synthesis>show P
// # states = 3
// # transitions = 4
transition_system P;
3 |- b -> 2;
2 |- a -> 3;
1 |- a -> 2,
    b -> 3;
<initial = {1}; P = {2}; M = {3}>.
synthesis>unmark P

```

```

// # states = 3
// # transitions = 4
transition_system $$;
0 |- b -> 1;
1 |- a -> 0;
2 |- a -> 1,
    b -> 0;
<initial = {2}>.
synthesis>

```

Memory management

collect: Some type of objects used by **Synthesis** are allocated by pages. This command is used to decrease the memory consumption; it tells **Synthesis** to free each page containing only unused objects. The user might evaluate the memory usage using the **pool** command.

pools: This command displays some statistics about the memory usage. This command is useful when lots of objects have been freed using the **remove** command; in this case the user can **collect** the memory not actually used by **Synthesis**.

Some type of objects are allocated by pages. The **pools** command displays informations about this pages. The output of **pools** is something like the following table:

```

synthesis>pools
compute pools

```

	# P	# 0/P	b/0	free	mem
Strategy PREDs :	1	1024	8	1024	8.00kb
Game AE Edges :	3	690	12	1354	24.26kb
Game EA Edges :	1	1024	8	847	8.00kb
Game Positions :	3	349	24	737	24.54kb
Automaton Edges :	8	690	12	600	64.69kb
Automaton Rules :	2	690	12	560	16.17kb
Automaton States:	2	421	20	330	16.45kb
SID Dico :	1	1024	8	807	8.00kb

table of symbols size = 217

Each line of the previous table concerns a type of objects; it indicates: the number of currently allocated pages (**#P**), the number of objects by page (**#0/P**), the number of bytes used by one object (**b/0**), the number of currently unused objects (**free**) and the memory required to store all this objects (**mem**). Of course, when **free** \gg **#P** \times **#0/P**, a call to **collect** will have a significant effect on the memory consumption of **Synthesis**.

remove $objid_1 \dots objid_n$: An object remains in the memory space of **Synthesis** until its replacement (any identifier might be assigned a new object). The user has the choice to explicitly **remove** objects from memory using this command. The reader should note that the memory used by an object is not freed; the object is just marked as reusable. In order to actually liberate the objects the user must call the **collect** command.

Leaving the program

exit: This command might be used in place of EOF (i.e. CTRL-D on many systems) in order to terminate **Synthesis**. When the program terminates it saves the history of commands (if the program has been compiled with the GNU **readline** library) into the file **.synthesis_history.syn** in the current directory; this history is reloaded when **Synthesis** is restarted (from the same directory).

If this command is used into a script then the **Synthesis** interpreter don't execute commands after the **exit**. Example:

```
point@raoul: ~/tmp
cat script.syn
print "the first 'print' command is executed"
exit
print "the second 'print' command is not executed"
point@raoul: ~/tmp
synthesis script.syn
the first 'print' command is executed
point@raoul: ~/tmp
```


Appendix B

Syntaxes

Synthesis scripts

```
script-file ::= list-of-commands  
list-of-commands ::= redirected-command-or-assignment  
                    ::= redirected-command-or-assignment ; command-list  
redirected-command-or-assignment ::= command-or-assignment > string  
                                       ::= command-or-assignment >> string  
                                       ::= command-or-assignment  
command-or-assignment ::= command  
                           ::= assignment  
command ::= identifier parameter-list  
assignment ::= identifier := command  
parameter-list ::= parameter parameter-list  
                  ::=  
parameter ::= identifier  
               ::= integer  
               ::= string  
               ::= ( command )  
integer ::= [0-9] [0-9]*  
identifier ::= [a-zA-Z_] [a-zA-Z0-9_]*  
string ::= "[^"]*" \
```

MEC4 File Format

```
mec4-file ::= list-of-transition-systems  
list-of-transition-systems ::= transition-system list-of-transition-systems  
                               ::=  
transition-system ::= transition_system ident attributes ; transition-system-body  
transition-system-body ::= list-of-states sets-of-states  
list-of-states ::= list-of-states ; state  
                  ::= state
```

```

state ::= state-name attributes |- transitions
transitions ::= list-of-transitions
           ::=
list-of-transitions ::= transition , list-of-transitions
                   ::= transition
transition ::= event-name -> state-name attributes
event-name ::= name
sets-of-states ::= ; < list-of-set-of-states > .
              ::= .
list-of-set-of-states ::= set-of-states
                    ::= set-of-states ; list-of-set-of-states
attributes ::= < list-of-attributes >
           ::=
list-of-attributes ::= attributes ; list-of-attributes
                  ::= attributes
attribute ::= ident = id-int
          ::= ident
set-of-states ::= ident = { list-of-state-name }
              ::= ident = { }
list-of-state-name ::= state-name , list-of-state-name
                  ::= state-name
state-name ::= name
name ::= ident
      ::= integer
id-int ::= ident
       ::= integer
ident ::= [0-9_]*[A-Za-z_\^[A-Za-z0-9_\^]*
integer ::= [0-9]+

```

FAM File Format

```

fam-file ::= list-of-automata
list-of-automata ::= automaton list-of-automata
                ::=
automaton ::= name ident automaton-width ; states properties
automaton-width ::= < width = integer >
               ::=
states ::= state-list
state-list ::= state
           ::= state state-list
state ::= kind-of-state = ranks -> rules
       ::= kind-of-state = ranks -> ;
ranks ::= ranks1
       ::= ranks2

```

ranks1 ::= **mu**
 ::= **nu**
ranks2 ::= < *integer-list* >
integer-list ::= *integer* , *integer-list*
 ::= *integer*
rules ::= *rule-list* ;
rule-list ::= *simple-rule* + *rule-list*
 ::= *simple-rule*
simple-rule ::= *labels* (*members*)
 ::= *labels conjunction*
 ::= *conjunction*
labels ::= *label-list*
label-list ::= *label* . *label-list*
 ::= *label* .
label ::= *ident*
 ::= ~ *ident*
members ::= *member*
member ::= *conjunction* + *member*
 ::= *conjunction*
conjunction ::= *transitions*
transitions ::= *transition* . *transitions*
 ::= *transition*
transition ::= < *ident-list* > *any-kind-of-state*
 ::= [*ident-list*] *any-kind-of-state*
ident-list ::= *id-int*
 ::= *id-int* , *ident-list*
properties ::= < *property-list* > .
property-list ::= *property* ; *property-list*
 ::= *property*
property ::= *ident* = { *list-of-states* }
list-of-states ::= *any-kind-of-state* , *list-of-states*
 ::= *any-kind-of-state*
any-kind-of-state ::= *sort-of-state*
 ::= **T**
kind-of-state ::= *id-int*
id-int ::= *integer*
 ::= *ident*
ident ::= [A-Za-z_][A-Za-z0-9_]*
integer ::= [0-9]+

GAME File Format

game-file ::= *list-of-games*

```

list-of-games ::= game list-of-games
              ::=
              game ::= game ident ; game-edges initial-position
game-edges   ::= game-edge ;
              ::= game-edge ; game-edges
game-edge    ::= eve-edges
              ::= adam-edges
eve-edges    ::= ident @ number eve-target-list
              ::= ident @ number
eve-target-list ::= -> ident
                ::= -> ident , eve-target-list
adam-edges    ::= ident @ number |- adam-target-list
              ::= ident @ number |-
adam-target-list ::= ident -> ident
                 ::= ident -> ident , adam-target-list
initial-position ::= < initial = { ident } > .
ident           ::= [A-Za-z\_][A-Za-z0-9\_]*
number         ::= [0-9]*

```