Mode Automata and their Compilation into Fault Trees

Antoine Rauzy
Institut de Mathématique de Luminy
163, avenue de Luminy, Case 907
13288 Marseille cedex 9 – FRANCE
arauzy@iml.univ-mrs.fr

Abstract

In this article, we advocate the use of mode automata as a high level representation language for reliability studies. Mode automata are states/transitions based representations with the additional notion of flow. They can be seen as a generalization of both finite capacity Petri nets and block diagrams. They can be assembled into hierarchies by means of composition operations.

The contribution of this article is twofold. First, we introduce mode automata and we discuss their relationship with other formalisms. Second, we propose an algorithm to compile mode automata into Boolean equations (fault trees). Such a compilation is of interest for two reasons. First, assessment tools for Boolean models are much more efficient than those for states/transitions models. Second, the automated generation of fault trees from higher level representations makes easier their maintenance through the life cycle of systems under study.

1 Introduction

Different formalisms are used in the reliability engineering framework in order to design models of the systems under study: Boolean formalisms (e.g. Fault Trees, Block Diagrams) and states/transitions formalisms (e.g. Petri Nets, Markov Graphs). All of them stand at a rather low level which makes their design and their maintenance a difficult task. Higher level formalims are needed to fill the gap between systems and models.

In this article, we advocate the use of mode automata as such a formalism. A mode automaton is a states/transitions system with input and output flows. States are called modes for in each mode a different transfer function determines the values of output flows from the values of input flows. Mode automata can be combined in order to design hierarchical models. They generalize both bounded Petri nets and block diagrams. The term "mode automaton" itself is borrowed from the work by Maraninchi & al. [14, 15] on the

introduction of modes into reactive languages. We use this term here in a different way, although the two concepts are strongly related.

The popularity of above mentioned formalisms (Fault Trees, Petri nets, ...) relies not only on their mathematical soundness, but also on the possibility to represent graphically each mathematical construct. Mode automata have also this property. Moreover, different views of the same automaton can be given in order to emphasize an aspect or another of the model. Two main categories of views are provided: states/transitions views and hierachical views. The latters are close to functional representations (e.g. block diagrams).

The design of a model always results of a tradeoff between the acuracy of the description and the tractability of computations to be performed. For instance, the fault tree method ignores deliberately the sequencing among events in order to decrease the complexity of qualitative and probabilistic assessments. As a generalization of Petri nets, mode automata make it possible to describe complex phenomena, leading to difficult assessment problems. However, a key idea is that they can be compiled efficiently into Boolean formulae (fault trees). Indeed, this compilation process loses some information. This is the price to pay to pass from a functional states/transitions description to a fault tree. However, the clear mathematical semantics of mode automata makes explicit what is lost during this process.

The contribution of this article is thus twofold. First, we introduce mode automata and we discuss their relationship with classical formalisms. Second, we propose an algorithm to compile mode automata into Boolean formulae. This algorithm could be adapted to other high level description formalisms. This work is a part of a larger project, so-called AltaRica, that aims to design a normalized high level language for reliability studies as well as a workbench supporting this language [18, 4].

The remainder of this article is organized as follows. Section 2 presents mode automata. Section 3 explains the different mechanisms to combine them. Section 4 is devoted to their compilation into Boolean formulae. Finally, section 5 examines related formalims.

2 Mode Automata

2.1 Informal presentation

A mode automaton is an input/output automaton. It has a finite number of states, that are called modes. At each instant, it is in one (and only one) mode. It may change of mode when an event occurs. In each mode, a transfer function determines the values of output flows from the values of input flows.

Consider, for instance, the valve pictured Fig. 1 (a). Assume that it can be either open or closed and that it may be stuck, either open or closed. The valve changes from open to closed (resp. from close to open) if it is not stuck and if the event *close* (resp. *open*) occurs. It gets stuck when the event *fail* occurs. If the valve is open, its output flow equals its input flow. Otherwise, its output flow is null.

Fig. 1 (b) shows the mode automaton that describes such a valve. Modes are represented by rectangles with rounded corners. The mode itself and the transfer function are described

respectively above and under the separation line. The initial mode is pointed by an arrow. Transitions are represented by arrows joining modes.

Such normalized graphical notations — and the correspondance between graphical and textual constructs — is actually essential to make the formalism both mathematically sound and user friendly.

2.2 Formal definition

Let $\mathcal{D}$ be a finite set of symbols called constants and let $\mathcal{V}$ be a finite set of symbols called variables ($\mathcal{D} \cap \mathcal{V} = \emptyset$). $\mathcal{D}$ is called a domain. We assume given a mapping dom from $\mathcal{V}$ to $2^{\mathcal{D}}$ (the powerset of $\mathcal{D}$), such that forall v in $\mathcal{V}$, $dom(v) \neq \emptyset$. $dom(v)$ is called the domain of the variable v, i.e. $dom(v)$ is the set of possible values of v.

Let $\mathcal{U} \subseteq \mathcal{V}$. We denote by $dom(\mathcal{U})$ the cartesian product of the domains of the variables of $\mathcal{U}$: $dom(\mathcal{U}) = \Pi_{v \in \mathcal{U}}\ dom(v)$. In other words, $dom(\mathcal{U})$ is the set of all possible valuations of the variables of $\mathcal{U}$. Let $U \in dom(\mathcal{U})$. We denote by $U[v]$ the value of the variable v in the valuation U and by $U[v \leftarrow c]$, $c \in dom(v)$, the valuation that is equal everywhere to U but possibly in v, where it is equal to c.

A mode automaton is a ninetuple $\mathcal{A} = \langle \mathcal{D}, dom, \mathcal{S}, \mathcal{F}^{in}, \mathcal{F}^{out}, \Sigma, \delta, \sigma, \mathcal{I} \rangle$, where

- $\mathcal{D}$ and dom are a domain and domain function defined as above.
- $\mathcal{S}$, $\mathcal{F}^{in}$, $\mathcal{F}^{out}$ are three pairwisely disjoint subsets of $\mathcal{V}$. Variables of $\mathcal{S}$, $\mathcal{F}^{in}$ and $\mathcal{F}^{out}$ are called respectively state variables, input flows and output flows.
- Σ is a finite set of symbols called events.
- δ is a partial function from $dom(\mathcal{S}) \times dom(\mathcal{F}^{in}) \times \Sigma$ to $dom(\mathcal{S})$. δ gives the next values of state variables from their current values, the values of input flows and the event the occurrence of which induces a change of mode.
- σ is a total function from $dom(\mathcal{S}) \times dom(\mathcal{F}^{in})$ to $dom(\mathcal{F}^{out})$. σ gives the values of output flows from the current values of state variables and input flows.

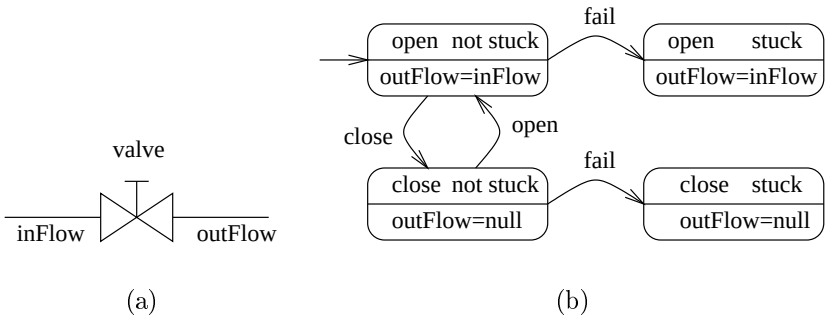


Figure 1: A valve and the mode automaton that describes it.

- Finally, $\mathcal{I}$ belongs to $dom(\mathcal{S})$ and is called the initial mode.

Example: Consider again the valve pictured Fig. 1. Assume that its input and output flows range in the set $\{null, low, medium, high\}$. The mode automaton that describes the valve is formally defined as follows.

- $\mathcal{S} = \{state, stuck\}$, with $dom(state) = \{open, close\}$ and $dom(stuck) = \{true, false\}$.
- $\mathcal{F}^{in} = \{inFlow\}$, $\mathcal{F}^{out} = \{outFlow\}$, with $dom(inFlow) = dom(outFlow) = \{null, low, medium, high\}$.
- $\Sigma = \{open, close, fail\}$.
- δ and σ are defined as pictured Fig. 1 (b).
- $\mathcal{I} = \langle open, false \rangle$.

2.3 Reachable modes

A mode automaton describes a set of possible behaviours: from its initial mode, it may evolve and change of mode by executing transitions. Let $\mathcal{A} = \langle \mathcal{D}, dom, \mathcal{S}, \mathcal{F}^{in}, \mathcal{F}^{out}, \Sigma, \delta, \sigma, \mathcal{I} \rangle$ be a mode automaton. The set of reachable modes of $\mathcal{A}$, denoted by $Reach(\mathcal{A})$ is the smallest subset of $dom(\mathcal{S})$ such that:

- $\mathcal{I} \in Reach(\mathcal{A})$.
- If $S \in Reach(\mathcal{A})$ and it exists $I \in dom(\mathcal{F}^{in})$, $e \in \Sigma$ and $T \in dom(\mathcal{S})$ such that $T = \delta(S, I, e)$, then T is in $Reach(\mathcal{A})$.

In the above definition, we assumed finite domains. Therefore, the set of reachable modes is finite as well. All of the classical questions about behavioral properties (reachability, deadlock freeness, liveness, model checking, ...) are thus decidable. Since mode automata are very similar to n-bounded Petri-nets, it is easy to prove that they are of the same complexity in both formalisms (namely PSPACE-hard, see [8] for an survey of this topics). If the definition is modified to allow infinite domains, it is easy to see that mode automata are a superset of Petri nets with inhibitor arcs. Since the latters have the power of Türing machines, almost all of the above questions are undecidable [8].

2.4 Textual and Graphical Constructs

To describe mode automata, one needs some syntactic constructs. For instance, transitions can be represented as triples $\langle G, e, A \rangle$, denoted by $G \xrightarrow{e} A$, where

- G is a Boolean formula involving state variables, input flows and output flows. G is called the guard, or the pre-condition, of the transition.

- $e \in \Sigma$.
- A is a set of assignments in the form $v := exp$, where v is a state variable and exp is an expression involving state variables, input flows and output flows. A is called the action, or the post-condition, of the transition.

For instance, the failure transition of the valve can be written $not\ stuck \xrightarrow{\text{failure}} stuck := true$.

In a similar way, transfer functions can be represented as equations in the form $o = S$, where o is an output flow and S is a formula that involves state variables and input flows.

Mode automata can be also represented graphically in different ways. Fig. 2 shows three different representations of the mode automaton pictured Fig. 1 (b). Fig. 2 (a) puts the emphasis on the transfer function. A generic mode is defined for each transfer function. Transitions are labeled with their guards, events and actions. Fig. 2 (b) presents the valve as a component to be inserted in a more general (functional) description. The component is represented by a box. Its input and output flows are represented by arrows that respectively come in and go out the component. Fig. 2 (c) is a Petri net like representation. It puts the emphasis on state variables.

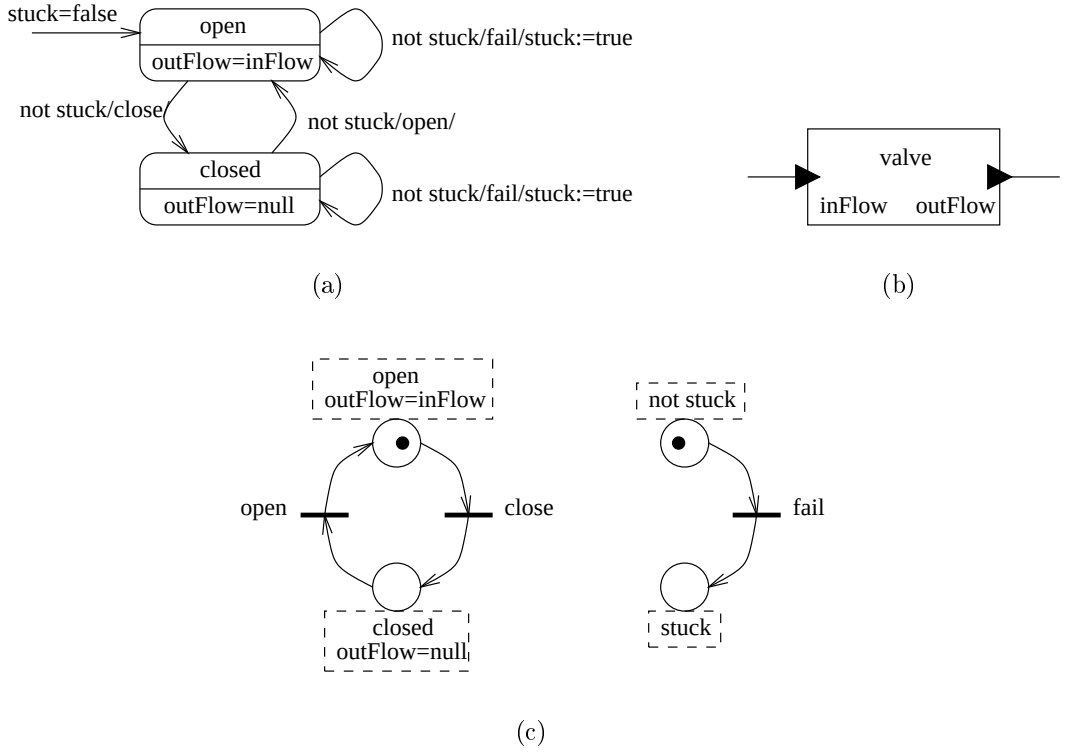


Figure 2: Three alternative representations for a mode automaton.

Fig. 1 and 2 illustrate that different representations may contain a different information. In general, it is not possible to display all of the data associated with a model within a

single figure. Specification languages such as UML [20] advocate that it is often much more convenient to have several views of the same object than to have a single overloaded view. Graphical notations associated with mode automata provide the designer with this ability.

3 Composition of mode automata

A major prerequisite for a high level description language is to be compositional, i.e. to allow the description of systems as hierarchies of (reusable) components. Mode automata can assembled by means of three operations: parallel composition, connection and synchronization. These operations produce mode automata: any hierarchy can be "flattened" into an equivalent mode automaton. Moreover, this process is efficient because it involves only very simple syntactic operations.

3.1 Parallel composition

Let $\mathcal{A}_1,\ldots,\mathcal{A}_n$ be n mode automata ($\mathcal{A}_i = \langle \mathcal{D}, dom, \mathcal{S}_i, \mathcal{F}_i^{in}, \mathcal{F}_i^{out}, \Sigma_i, \delta_i, \sigma_i, \mathcal{I}_i \rangle$). We can assume without a loss of generality that their vocabularies are distinct, i.e. that for any i and j, $1 \leq i < j \leq n$, $(\mathcal{S}_i \cup \mathcal{F}_i^{in} \cup \mathcal{F}_i^{out}) \cap (\mathcal{S}_j \cup \mathcal{F}_j^{in} \cup \mathcal{F}_j^{out}) = \emptyset$ and $\Sigma_i \cap \Sigma_j = \emptyset$.

The parallel composition of $\mathcal{A}_1,\ldots,\mathcal{A}_n$, denoted by $\Pi_{i=1}^n \mathcal{A}_i$, is a mode automaton $\mathcal{A} = \langle \mathcal{D}, dom, \mathcal{S}, \mathcal{F}^{in}, \mathcal{F}^{out}, \Sigma, \delta, \sigma, \mathcal{I} \rangle$ such that:

- $\mathcal{S} = \bigcup_{i=1}^n \mathcal{S}_i$, $\mathcal{F}^{in} = \bigcup_{i=1}^n \mathcal{F}_i^{in}$, $\mathcal{F}^{out} = \bigcup_{i=1}^n \mathcal{F}_i^{out}$, $\Sigma = \bigcup_{i=1}^n \Sigma_i$.
- δ is obtained by lifting the δ_i's up to $\mathcal{A}$: Let $S_1 \in dom(\mathcal{S}_1)$, ..., $S_n \in dom(\mathcal{S}_n)$. Let $I_1 \in dom(\mathcal{F}_1^{in})$, ..., $I_n \in dom(\mathcal{F}_n^{in})$. Finally, let $T_i \in dom(\mathcal{S}_i)$ and $e \in \Sigma_i$ such that $T_i = \delta_i(S_i, I_i, e)$. Then,

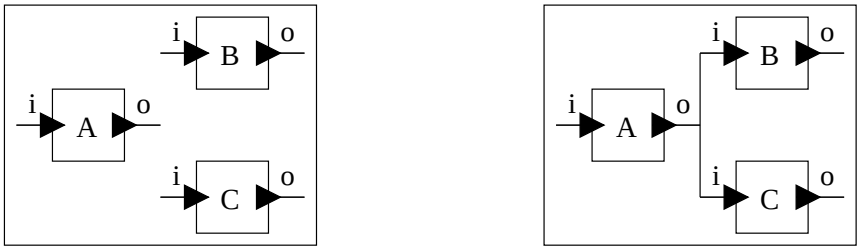
$$\delta(S_1, \ldots, S_n, I_1, \ldots, I_n, e) = \langle S_1, \ldots, S_{i-1}, T_i, S_{i+1}, \ldots, S_n \rangle$$

- Similarly σ is obtained by lifting up the σ_i's:

$$\sigma(S_1, \ldots, S_n, I_1, \ldots, I_n) = \langle \sigma_1(S_1, I_1), \ldots, \sigma_n(S_n, I_n) \rangle$$

- Finally, $\mathcal{I} = \langle \mathcal{I}_1, \ldots, \mathcal{I}_n \rangle$.

In other words, the parallel composition consists in glueing together the $\mathcal{A}_i$'s. This operation is sometimes called a free product [3]. From a graphical point of view, the parallel composition consists in drawing a box to surround the graphical representation of its components, as illustrated Fig. 3 (a) for components A, B and C.



(a) Parallel composition of A, B and C

(b) Connection of B.i and C.i to A.o

Figure 3: Parallel composition and connection of mode automata.

3.2 Connection

The connection consists in compelling one or more input variables to be equal to an output variable. This process is illustrated Fig. 3 (b) where input flows of components B and C are plugged in the output flow of the component A.

To be valid, a connection must introduce no loop. This is the causality problem, that has been extensively studied in the framework of reactive languages [10]. We introduce here the notion of causal dependence, which is specific to mode automata.

Let $\mathcal{A} = \langle \mathcal{D}, dom, \mathcal{S}, \mathcal{F}^{in}, \mathcal{F}^{out}, \Sigma, \delta, \sigma, \mathcal{I} \rangle$, let $i \in \mathcal{F}^{in}$ and let $o \in \mathcal{F}^{out}$. o is said causaly independent from i, if the following property holds.

$$\forall S \in dom(\mathcal{S}),\ \forall I \in dom(\mathcal{F}^{in}),\ \forall c \in dom(i)\ \ \sigma(S, I)[o] = \sigma(S, I[i \leftarrow c])[o]$$

o is said causaly dependent of i otherwise.

This definition of causality is indeed very costly to check. Fortunately, there are very simple syntactic conditions that ensure causal independence. For instance, it suffices that i does not occur in the equations that define the value of o.

Let $\mathcal{A} = \langle \mathcal{D}, dom, \mathcal{S}, \mathcal{F}^{in}, \mathcal{F}^{out}, \Sigma, \delta, \sigma, \mathcal{I} \rangle$ be a mode automaton, let $o \in \mathcal{F}^{out}$ and let $i_1, \ldots, i_k \in \mathcal{F}^{in}$ such that o is causaly independent of i_1, ..., i_k and $dom(o) \subseteq dom(i_1)$, ..., $dom(o) \subseteq dom(i_k)$.

Let $\mathcal{F}^{in}_\star = \mathcal{F}^{in} \setminus \{i_1, \ldots, i_k\}$, let $S \in dom(\mathcal{S})$ and let $I \in dom(\mathcal{F}^{in}_\star)$. We denote by $I_{S, o/i_1, \ldots, o/i_k}$ the valuation of $\mathcal{F}^{in}$ such that:

$$I_{S, o/i_1, \ldots, o/i_k}[v] \ \stackrel{\text{def}}{=}\ \begin{cases} I[v] & \text{if } v \in \mathcal{F}^{in}_\star \\ \sigma(S, I')[o] & \text{otherwise} \end{cases}$$

where I' is any extension of I into a valuation of $\mathcal{F}^{in}$ (by definition, $\sigma(S, I')[o]$ is the same, no matter what this extension is).

The mode automaton $\mathcal{A}$ in which i_1, ..., i_k are connected to o is the mode automaton $\mathcal{A}_{o/i_1, \ldots, o/i_n} = \langle \mathcal{D}, dom, \mathcal{S}, \mathcal{F}^{in}_\star, \mathcal{F}^{out}, \Sigma, \delta', \sigma', \mathcal{I} \rangle$, where δ' and σ' are defined as follows.

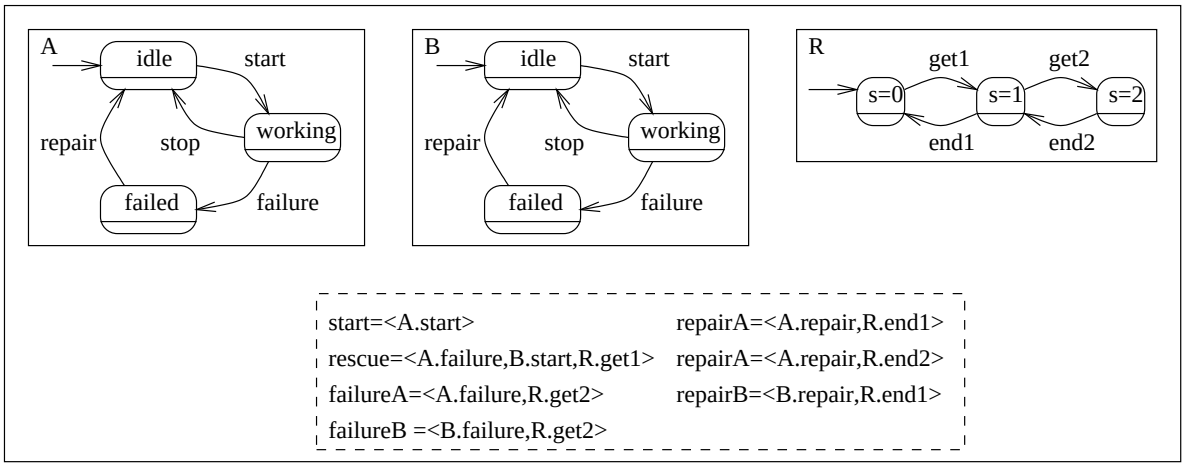


Figure 4: Two identical engines A and B, and a repairer R, with B is in cold redundancy.

- For all $S \in dom(\mathcal{S})$, $I \in dom(\mathcal{F}_\star^{in})$ and $e \in \Sigma$, if $\delta(S, I_{S,o/i_1 \ldots o/i_n}, e)$ is defined, then $\delta'(S, I, e)$ is defined and

$$\delta'(S, I, e) = \delta(S, I_{S,o/i_1 \ldots o/i_n}, e)$$

- For all $S \in dom(\mathcal{S})$, $I \in dom(\mathcal{F}_\star^{in})$,

$$\sigma'(S, I) = \sigma(S, I_{S,o/i_1 \ldots o/i_n})$$

In practice, to realize a connection, it suffices to substitute o for the i_j's. Therefore, a connection can be seen just as a renaming.

3.3 Synchronization

As in other states/events formalisms such as Petri nets, transitions of mode automata are assumed to be asynchronous: two transitions cannot be fired simultaneously. A synchronization consists in compelling a set of events to occur simultaneoulsy. The set itself, so-called a synchronization vector [3], can be seen as a macro-event. Intuitively, the synchronization of a mode automaton $\mathcal{A}$ with synchronization vectors $\vec{e_1}, \ldots, \vec{e_r}$, consists in two steps. First, events that occur in the $\vec{e_i}$'s are removed together with transitions they label. Second, transitions labeled with each $\vec{e_i}$ are created by gathering (according to the $\vec{e_i}$'s) guards and actions of the removed transitions.

As an illustration, consider the system pictured Fig. 4. It is made of two identical components A and B — say two engines — and a repairer R. B is in cold-redundancy, i.e. it is started only if A fails. The repair of A preempts the repair of B.

The three events "A fails", "B starts" and "R gets a first engine to repair" can be considered as simultaneous, therefore a synchronization vector $rescue = \langle A.failure, B.starts, R.get1 \rangle$

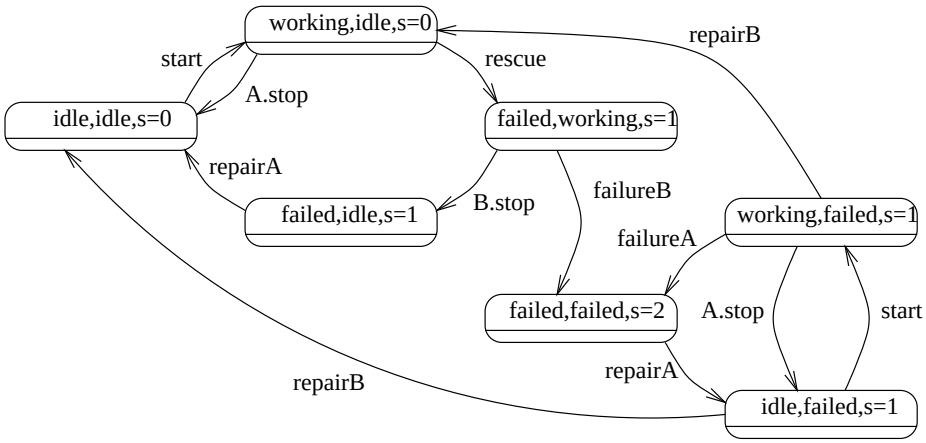


Figure 5: The synchronized system.

is created. The other the synchronization vectors given Fig. 4 describe the synchronization resulting of the repairing policy. Now, the synchronization removes the three transitions:

$$\begin{aligned} A.state = working &\xrightarrow{\text{A.failure}} A.state := failed \\ B.state = idle &\xrightarrow{\text{B.start}} B.state := working \\ R.s = 0 &\xrightarrow{\text{get 1}} R.s := 1 \end{aligned}$$

and creates the new one:

$$\begin{pmatrix} & A.state = working \\ \wedge & B.state = idle \\ \wedge & R.s = 0 \end{pmatrix} \xrightarrow{\text{rescue}} \begin{pmatrix} A.state := failed \\ B.state := working \\ R.s := 1 \end{pmatrix}$$

Fig. 5 shows a fully expanded version of the mode automaton pictured Fig. 4.

In order to define formally the synchronization, we need to introduce the notions of compatibility and composition of valuations.

Let $\mathcal{S}$ be a subset of $\mathcal{V}$ and let S, S' and $S'' \in dom(\mathcal{S})$.

S' and S'' are said incompatible w.r.t. S if it exists $v \in \mathcal{S}$ such that $S[v]$, $S'[v]$ and $S''[v]$ are all distinct. They are said compatible otherwise.

The composition of S' and S'' w.r.t. S, denoted $S' \circ_S S''$, is the valuation defined as follows.

$$\forall v \in \mathcal{S}, \quad S' \circ_S S'' \stackrel{\text{def}}{=} \begin{cases} S'[v] & \text{if } S'[v] \neq S[v] \\ S''[v] & \text{otherwise} \end{cases}$$

The following property holds.

Property 1 (Composition) *Let $\mathcal{S}$ be a subset of $\mathcal{V}$ and let $S \in dom(\mathcal{S})$. Applied to valuations that are pairwisely compatible w.r.t. S, $\circ_S$ is commutative and associative.*

We can now define formally the synchronization.

Let $\mathcal{A} = \langle \mathcal{D}, dom, \mathcal{S}, \mathcal{F}^{in}, \mathcal{F}^{out}, \Sigma, \delta, \sigma, \mathcal{I} \rangle$ be a mode automaton and let $\vec{e_1}, \ldots, \vec{e_r}$ be r synchronization vectors. Let Σ' be the subset of Σ of the events that occur in the $\vec{e_i}'s$. The synchronization of $\mathcal{A}$ by $\vec{e_1}, \ldots, \vec{e_r}$ is a mode automaton $\mathcal{A}|\vec{e_1}, \ldots, \vec{e_r} = \langle \mathcal{D}, dom, \mathcal{S}, \mathcal{F}^{in}, \mathcal{F}^{out}, (\Sigma \setminus \Sigma') \cup \{\vec{e_1}, \ldots, \vec{e_r}\}, \delta', \sigma, \mathcal{I} \rangle$, where δ' is defined as follows.

- For any $e \in \Sigma \setminus \Sigma'$, $S \in dom(\mathcal{S})$ and $I \in dom(\mathcal{F}^{in})$, if $\delta(S, I, e)$ is defined then $\delta'(S, I, e)$ is defined as well and

$$\delta'(S, I, e) \stackrel{\text{def}}{=} \delta(S, I, e)$$

- For any $\vec{e_i} = \langle e_1, \ldots, e_k \rangle$, $S \in dom(\mathcal{S})$ and $I \in dom(\mathcal{F}^{in})$, if $\delta(S, I, e_1)$, $\ldots$, $\delta(S, I, e_k)$ are defined and pairwisely compatible, then $\delta'(S, I, \vec{e_i})$ is defined and

$$\delta'(S, I, \vec{e_i}) \stackrel{\text{def}}{=} \delta(S, I, e_1) \circ_S \ldots \circ_S \delta(S, I, e_k)$$

4 Compilation into Boolean formulae

By compiling mode automata into Boolean formulae, one expects two benefits: a better efficiency in the assessment of the models and a simplification of the design and the maintenance of Boolean models. The price to pay is the loss of the sequencing among events: sequences of events are compiled into conjuncts of events.

In this section, we propose a compilation algorithm that works well on models that are close to a block diagram representation. Many real-life models are actually conceptually simple, nevertheless hard to assess because of their sizes. Therefore, a good compiler should first of all compile efficiently this kind of models.

4.1 Principle of the compilation

We assume from now that the mode automaton $\mathcal{A} = \langle \mathcal{D}, dom, \mathcal{S}, \mathcal{F}^{in}, \mathcal{F}^{out}, \Sigma, \delta, \sigma, \mathcal{I} \rangle$ under study describes a system that may fail. The initial mode $\mathcal{I}$ represents the nominal state of the system. Events represent failures of its components. Some modes represent failure states. Paths from $\mathcal{I}$ to these modes represent scenarios of failure.

The compilation captures failure scenarios into a set of Boolean equations. It produces a Boolean formula $\phi_{v,c}$ for each pair (v, c), $v \in \mathcal{S} \cup \mathcal{F}^{out}$, $c \in dom(v)$, such that:

- The variables of $\phi_{v,c}$ are the events of Σ.
- The minimal cutsets of $\phi_{v,c}$ one-to-one correspond with sets of events $\{e_1, \ldots, e_k\}$ such that there exists a sequence of modes $M_0, \ldots, M_k$ such that:
  - $M_0 = \mathcal{I}$.
  - $\delta(M_0, I_1, e_1) = M_1$, $\ldots$, $\delta(M_{k-1}, I_k, e_k) = M_k$ for some I_1, $\ldots$, $I_k \in dom(\mathcal{F}^{in})$.

– $\sigma(M_k, I_k)[v] = c$.

The algorithm works from reachability graph of $\mathcal{A}$ (that can be seen as a reliability network [21]): First, the hierarchy is flattened into a single mode automaton according to the semantics described section 3. Second, the reachability graph is computed. Third, one associates with each mode M of this graph the disjunct over the paths π from $\mathcal{I}$ to M of the conjunct of events that label π. Fourth and last, one associates with each pair (v, c) the disjunct of formulae that are associated with the modes M for which there exists $I \in dom(\mathcal{F}^{in})$ such that $\sigma(M, I)[v] = c$.

4.2 Problems raised by the compilation

The above algorithm raises several problems.

The first one stands in the exponential blow up of the number of modes. Consider, for instance, the series-parallel system with m lines of n components pictured in Fig. 6 (a). This automaton has about 2^{nm} modes — a huge number even for small values of n and m. Means should therefore be found to take advantage of independence of subsystems.

The second problem stands in the exponential blow up of the number of paths. Consider, for instance, the graph pictured in Fig. 6 (b). Assume that the initial mode is the left inferior corner and that the failure mode is the right superior corner. Such a grid has only $m \times n$ modes but $\binom{n}{n+m}$ failure paths. A solution to this problem consists in constructing the formula associated with a vertex from the formulae associated with its neighbours. Such a local compilation is however not possible if the graph contains loops [19].

The third problem, which is related to the previous one, stands in the composition of components. Consider, for instance, the mode automaton pictured in Fig. 6 (c). If one would compile the three components A, B and C separately, one would obtain something as the following equations (assuming that all the variables are Boolean).

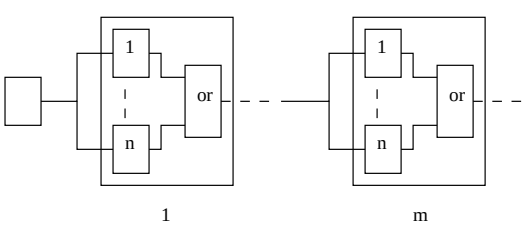
$$A.o1 = A.a \quad A.o2 = A.b \quad B.o = B.a$$
$$C.o = (A.o1 \wedge A.o2) \vee (A.o1 \wedge B.o) \vee (A.o2 \wedge B.o)$$

$C.o$ admits three minimal cutsets $\{A.a, A.b\}$, $\{A.a, B.a\}$ and $\{A.b, B.a\}$. However, by construction of the component A, events $A.a$ and $A.b$ cannot both occur. In order to deal with this kind of conflicts, the compilation algorithm has actually to consider not only events occurs along the paths but also those that do not.

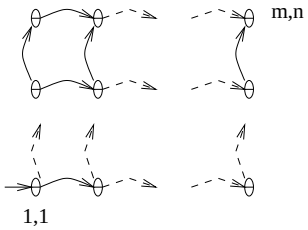
In the next section, we explain how the latter problem can be solved. In section 4.4, we show how to compute the reachability graph by parts.

4.3 A Modified Algorithm

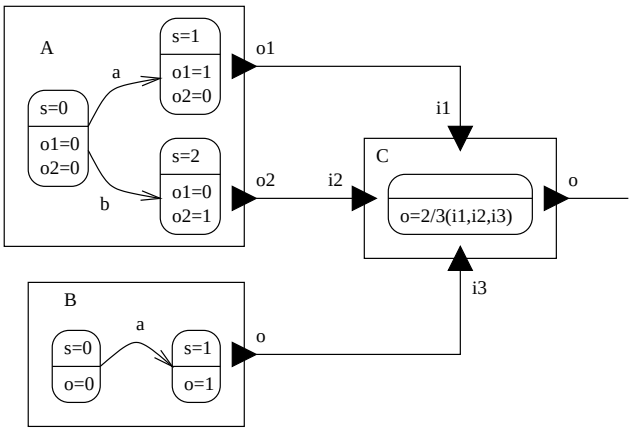
The third step of the algorithm is modified as follows. The paths (starting from the initial mode $\mathcal{I}$) of the reachability graph are computed by means of the algorithm given Fig. 7. This algorithm computes all pairs $\langle M, \vec{e} \rangle$, where M is a mode and $\vec{e}$ is a vector of Boolean



(a) A series-parallel mode automaton



(b) A "grid" mode automaton



(c) A non compositional mode automaton

Figure 6: Three mode automata that illustrate difficulties of the compilation

values (one per event), such that there exists a path from $\mathcal{I}$ to M labelled with the events e_i such that $\vec{e}[e_i] = 1$.

Then, the (modified) fourth step builds the $\phi_{v,c}$'s as follows. First, a formula $\phi_{M,\vec{e}}$ is created for each pair $\langle M, \vec{e} \rangle$ produced by the above algoritm.

$$\phi_{M,\vec{e}} = \bigwedge_{\vec{e}[e_i]=1} e_i \ \wedge \bigwedge_{\vec{e}[e_i]=0} \neg e_i$$

Second, a formula ϕ_M is associated with each mode M.

$$\phi_M = \bigvee_{\langle M,\vec{e} \rangle} \phi_{M,\vec{e}}$$

Finally, for each output flow v and each $c \in dom(v)$ a formula $\phi_{c,v}$ is created.

$$\phi_{c,v} = \bigvee_{M \in dom(\mathcal{S}),\ \exists I \in dom(\mathcal{F}^{in})\ s.t.\ \sigma(M,I)[v]=c} \phi_M$$

$C \leftarrow \{\langle \mathcal{I}, \vec{0} \rangle\}, D \leftarrow \emptyset$
`while` $C \neq \emptyset$ `do`
  Let $\langle M, \vec{e} \rangle \in C$
  $C \leftarrow C \setminus \{\langle M, \vec{e} \rangle\}$, $D \leftarrow D \cup \{\langle M, \vec{e} \rangle\}$
  `forall` $e_i \in \Sigma$, $I \in dom(\mathcal{F}^{in})$, $M' \in dom(\mathcal{S})$ such that $M' = \delta(M, e_i, I)$ `do`
    $C \leftarrow C \cup \{\langle M', \vec{e}[e_i] \leftarrow 1 \rangle\}$
    `done`
  `done`

Figure 7: The algorithm to compile paths

Consider, for instance, the component A of Fig. 6 (c). The formulae built by the algorithm are as follows.

$$\begin{array}{ll} \phi_{s=0} = \phi_{s=0,\langle 0,0 \rangle} = \neg a \wedge \neg b & \phi_{o_1=0} = \phi_{s=0} \vee \phi_{s=2} = \neg a \\ \phi_{s=1} = \phi_{s=1,\langle 1,0 \rangle} = a \wedge \neg b & \phi_{o_1=1} = \phi_{s=1} = a \wedge \neg b \\ \phi_{s=2} = \phi_{s=2,\langle 0,1 \rangle} = \neg a \wedge b & \phi_{o_1=0} = \phi_{s=0} \vee \phi_{s=1} = \neg b \\ & \phi_{o_2=1} = \phi_{s=1} = \neg a \wedge b \end{array}$$

The formulae ϕ_M's take into account not only the events that label each $\mathcal{I}$-M path, but also those that do not label the path. Therefore, the above algorithm avoids the pitfall of output flow composition. However, it is also clearly subject to the combinatorial explosion of the number of modes and paths. Fortunately, in many practical cases, it is possible to split the mode automaton into several independent parts and to compile it accordingly.

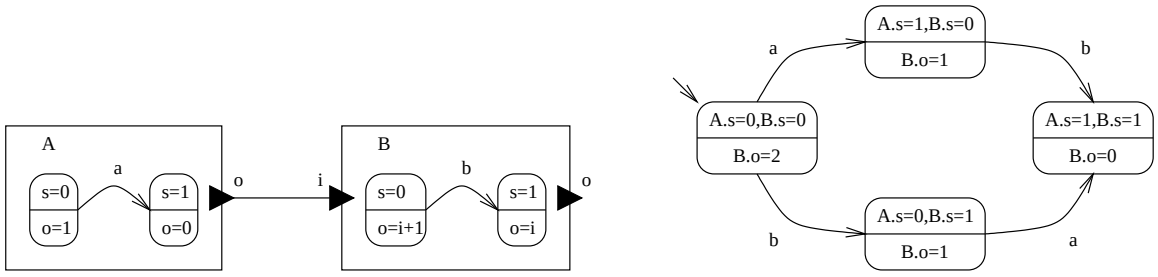
4.4 Application of Partial Order techniques

Consider for instance the series system pictured in Fig. 8. The transition $B.b$ is neither enabled nor disabled by the the transition $A.a$, and vice versa. $A.a$ depends only on $A.s$, $B.b$ depends only on $B.s$, none of them depends on $A.o$ and $B.o$. Therefore, state variables (and input flows) can be splitted two groups: $\{A.s\}$ and $\{B.s\}$. Each of this group can be analyzed independently. The two groups of equations for transitions are as follows.

$A.s$				
$\phi_{A.s=0}$	$=$	$\phi_{0,\langle 0 \rangle}$	$=$	$\neg A.a$
$\phi_{A.s=1}$	$=$	$\phi_{1,\langle 1 \rangle}$	$=$	$A.a$

$B.s$				
$\phi_{B.s=0}$	$=$	$\phi_{0,\langle 0 \rangle}$	$=$	$\neg B.b$
$\phi_{B.s=1}$	$=$	$\phi_{1,\langle 1 \rangle}$	$=$	$B.b$

The third step of the algorithm can be now achieved by considering the textual (or



(a) System (b) Reachability graph

Figure 8: A series system and its reachability graph

graphical) description of the automaton.

$A.o, B.o$				
$\phi_{A.o=1}$	$=$	$\phi_{A.s=0}$	$=$	$\neg A.a$
$\phi_{A.o=0}$	$=$	$\phi_{A.s=1}$	$=$	$A.a$
$\phi_{B.o=2}$	$=$	$\phi_{A.o=1} \wedge \phi_{B.s=0}$	$=$	$\neg A.a \wedge \neg B.b$
$\phi_{B.o=1}$	$=$	$(\phi_{A.o=1} \wedge \phi_{B.s=1}) \vee (\phi_{A.o=0} \wedge \phi_{B.s=0})$	$=$	$(\neg A.a \wedge B.b) \vee (A.a \wedge \neg B.b)$
$\phi_{B.o=0}$	$=$	$\phi_{A.o=0} \wedge \phi_{B.s=1}$	$=$	$A.a \wedge B.b$

The above idea is an application of the so-called Partial-Order methods [9].

Let $\mathcal{A} = \langle \mathcal{D}, dom, \mathcal{S}, \mathcal{F}^{in}, \mathcal{F}^{out}, \Sigma, \delta, \sigma, \mathcal{I} \rangle$ be a mode automaton. R is a valid dependency relation over Σ if for all $e_1, e_2 \in \Sigma$. $(e_1, e_2) \notin R$ (e_1 and e_2 are independent) implies that the two following properties hold for all reachable modes $M \in dom(\mathcal{S})$:

1. If there exists $I_1 \in dom(\mathcal{F}^{in})$ such that $\delta(M, I_1, e_1)$ is defined, then there exists $I_2 \in dom(\mathcal{F}^{in})$ such that $\delta(M, I_2, e_2)$ is defined iff $\delta(M, I_2, e_2)$ is defined (independent transitions can neither disable nor enable each other); and

2. If both $\delta(M, I_1, e_1)$ and $\delta(M, I_2, e_2)$ are defined then the following equality holds: $\delta(\delta(M, I_1, e_1), I_2, e_2) = \delta(\delta(M, I_2, e_2), I_1, e_1)$ (commutativity of enabled independent transitions).

Now, consider a path in the reachability graph. By applying the commutativity rule, it is possible to group together the dependent events without changing the first and last mode. In other words, the path can be decomposed into several independent sequences of dependent events. As a consequence, in our compilation algorithm, equivalence classes for the valid dependency relation R can be considered separately. This is actually what we did on the above example.

In practice, we define the relation R as follows. First, we define the notion of immediate dependence between variables and events. Let $e \in \Sigma$ and let $v \in \mathcal{S}$. We say that e depends

immediately on v if there exists a reachable mode $M \in dom(\mathcal{S} \cup \mathcal{F}^{in})$, $I \in dom(\mathcal{F}^{in})$ and $c \in dom(v)$ such that at least one of the three following conditions is satisfied:

- Either $v \in \mathcal{F}^{in}$, $\delta(M, I, e)$ is defined and either $\delta(M, I[v \leftarrow c], e)$ is not defined or $\delta(M, I[v \leftarrow c], e) \neq \delta(M, I, e)$; or
- $v \in \mathcal{S}$, $\delta(M, I, e)$ is defined and either $\delta(M[v \leftarrow c], I, e)$ is not defined or $\delta(M[v \leftarrow c], I, e) \neq \delta(M, I, e)$; or
- $v \in \mathcal{S}$ and $M[v] \neq \delta(M, I, e)[v]$.

Simple syntactic conditions ensure immediate dependence.

Second, we consider the relation $\sim$ over Σ defined as follows. $e_1 \sim e_2$ if there exists a variable v such that v depends immediately on both e_1 and e_2. Finally, we consider the transitive closure of $\sim$, which we denote by abuse $\sim$ as well. It is easy to verify that $\sim$ is a valid dependency relation over Σ. The relation $\sim$ splits Σ into one or more groups of events and associates a disjoint set of variables with each group. Steps third and fourth of then algorithm can be applied separately on each group. Step fourth is modified in order to take into account that an output flow may depend on state variables and input flows that belong to different groups.

5 Related Formalisms

Mode automata are related to the formalisms used in both formal methods and reliability engineering frameworks. This section discusses their relationships with these formalisms.

5.1 Block diagrams

Mode automata are clearly a generalization of block diagrams [2]. The translation from the latters to the formers is straightforward, as illustrated Fig. 9. Each block is translated into the small automaton pictured Fig. 9 (b). Then, the diagram is translated element by element. All flows are assumed to be Boolean.

Mode automata generalize block diagrams into several directions:

- Blocks may have several input and output flows.
- Input and output flows may be multivalued (not restricted to Boolean values).
- Blocks may be in more than two modes (working or failed).
- There may be other events than the failures of components.
- Synchronizations can be used to model complex behaviors, such as cold redundancies (as illustrated section 3.3).

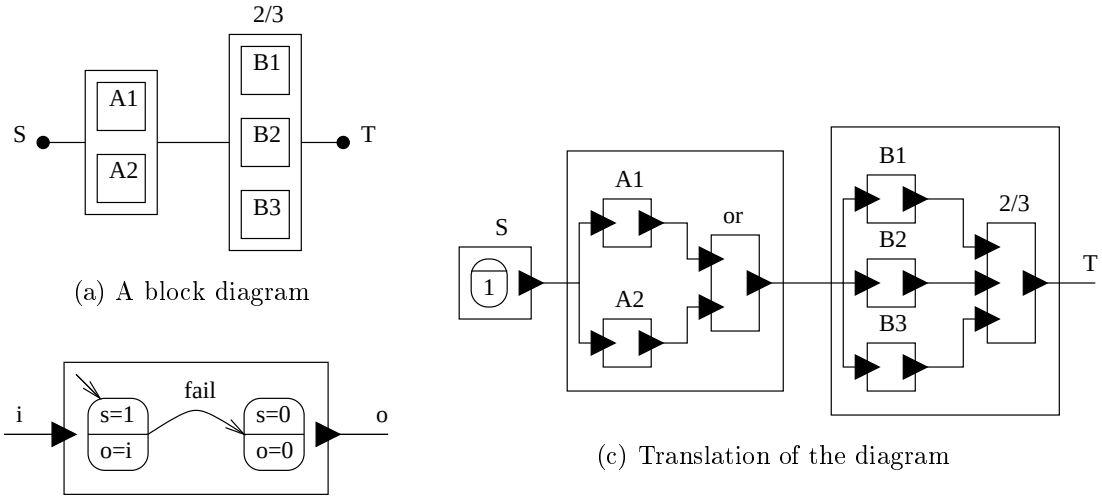


Figure 9: A block diagram and the corresponding mode automaton

5.2 Finite state machines

Mode automata generalize Mealy machines. Mealy machines are also input/output automata, with Boolean inputs and outputs (see [12] for an introduction). They are widely used in the circuit verification framework. The SMV model checker [16], for instance, is based on this formalism (in SMV however, transitions are anonymous).

The notion of synchronization vector comes from the seminal work by Arnold and Nivat on the semantics of concurrent processes [3]. Arnold and Nivat restricted their attention to finite state machines without input and output. Mode automata can be seen as generalized Mealy machines with Arnold-Nivat composition mechanisms.

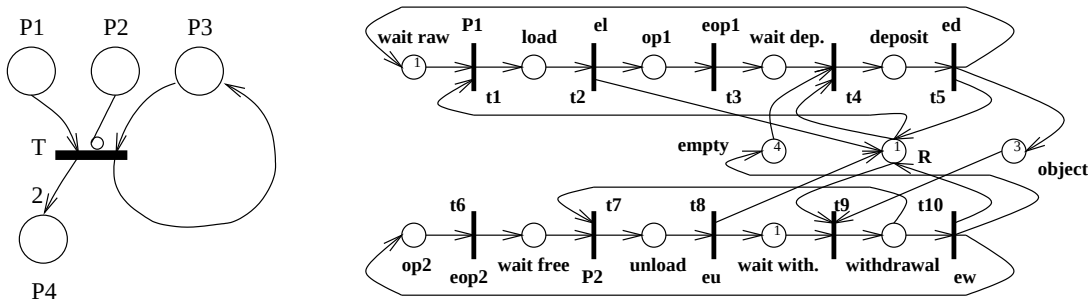
5.3 Petri nets

Petri nets [17] deserve a special mention because for they are often considered as the main alternative to Boolean models (fault trees and block diagrams) for reliability studies.

The translation from finite capacity Petri nets to mode automata is straightforward: it suffices to define an integral state variable for each place and a transition for each transition. For instance, the Petri net transition pictured Fig. 10 (a) is translated as follows.

$$P1 > 0 \ \wedge \ P2 = 0 \ \wedge \ P3 > 0 \quad \xrightarrow{\mathrm{T}} \quad P1 := P1 - 1, \ P4 := P4 + 2$$

Named values give a mean to simplify descriptions. Consider for instance the Petri net pictured Fig. 10 (b), which is taken from [7] (Fig. 1.9, page 16). The upper row of places describe the successive steps of a production process. In order to translate this Petri net, we could apply the method sketched above. However, a much simpler



(a) A transition of a Petri net

(b) A Petri net for production cell

Figure 10: Petri nets

solution consists in creating only one state variable per component of the production cell: the state of machine is described by means of a variable $M1$ ranging in the domain $\{waitRaw, load, op1, waitDeposit, deposit\}$. Transitions can still be translated in a straightforward way. For instance, the transition $t1$ is translated as follows.

$$M1 = waitRaw \ \wedge \ Robot = idle \quad \xrightarrow{\text{P1}} \quad M1 := load, \ Robot := busy$$

Not only the description is simpler, but it contains less variables and can therefore be assessed more efficiently.

In the same vein, synchronization vectors avoid many gadgets, i.e. additional places and arcs that are required to capture simultaneous (or quasi-simultaneous) behaviors with Petri nets (see [17] for a discussion on these gadgets).

Petri nets can be seen as mode automata with several restrictions:

- They have only state variables. Indeed, flows can be replaced by state variables. However, the formers are often much more convenient to model the propagation of an information.
- They put strong requirements on guards and actions of transitions. Namely, guards can be only conjuncts of inequalities of the form $place > constant$ (possibly $place = 0$ if inhibitor arcs are allowed) and assignments can be only of the form $place := place \pm constant$. Several authors suggested to overcome this restriction by introducing extended guards (see for instance [6, 5]).
- They provide no natural synchronization mechanism.

Our feeling is that all of these restrictions are justified more by historical reasons than by technical issues (graphical considerations play a role as well): linear algebra methods cannot work without these restrictions. However, linear algebra methods are seldom used in practice. This is especially true in reliability analysis framework where basically two kinds of assessment methods are used (for a comprehensive survey see [1]): Markovian

assessment method and Monte-Carlo simulations. In both cases, there is no mathematical or algorithmic need for the Petri net restrictions.

Finally, it is worth noticing that the graphical formalism of Petri nets can be applied to mode automata (as illustrated Fig. 2 (c)), up to small changes in the interpretation of arcs entering and going out transitions.

5.4 Reactive Languages

Mode automata are stronlgy related to visual specification languages such as StateCharts [11] or Argos [13]. They are also close to the underlying model of reactive languages such as Lustre [10]. The differences between mode automata and these formalisms stand mainly in the way they are used. Lustre, for instance, is a high level language to program applications that interact continuously with their environment. As a consequence, it puts the emphasis on modes and transfer functions (that describe actually the interaction of the program with its environment). On the converse, mode automata are used to design and to study models. The emphasis is put on (sequences of) events: failure scenarios, probability of failure, importance factors of components ....

The term "mode automata" is borrowed from references [14, 15]. These articles propose to extend Lustre with a construct that supports the description of running modes. Since a dataflow program is basically a Mealy machine, our notion of mode automata is very close to the notion of Maraninchi & al. However, the composition operations proposed in [14] are different from the ones we proposed here (the formers are inherited from Argos parallel and hierarchical compositions).

5.5 AltaRica

AltaRica is a high level description language. Its syntax and semantics have been described in references [18, 4]. The full AltaRica language embeds several features, such as bidirectional flows or broadcasting, that make its compilation into Boolean formulae difficult. For this reason, a restriction of AltaRica has been defined that drops out these features. The resulting language, so-called AltaRica data flow, is much easier to compile. Mode automata are the underlying mathematical model of this language.

Fig. 11 shows two typical AltaRica data flow declarations. Fig. 11(a) shows the declaration of the mode automaton that describes the valve of our introductory example (see Fig. 1). Fig. 11(b) shows the declaration of a system that contains three subsystems: a valve, a pipe and an operator. The system has an input flow connected to the input flow of the valve. The output flow of the valve is connected to the input flow of the pipe. Finally, the output flow of the pipe is connected to the output flow of the system. The opening and closing of the valve are synchronized with the corresponding actions of the operator.

```
node Valve;
  state open:bool, stuck:bool;
  flow i:bool:in, o:bool:out;
  event open, close, fail;
  trans
   open and not stuck |- close -> open:=false;
   not open and not stuck |- open -> open:=true;
   not stuck |- fail -> stuck:=true;
  assert not open => i=o, open => not o;
  init open=true, stuck=false;
edon
```

(a) component declaration

```
node System;
  state open:bool, stuck:bool;
  flow  I:bool:in, O:bool:out;
  sub   V:Valve, P:Pipe, O:operator;
  assertI=V.i, P.i=V.o, O=V.o;
  sync  close = <V.close,O.closeValve>,
        open = <V.open,O.openValve>;
edon
```

(b) node declaration

Figure 11: Two AltaRica data flow declarations

6 Conclusion

In this article, we introduced mode automata. We advocated their use as a high level language to design reliability models. Mode automata embed a number of interesting features that are borrowed from different formalisms.

- They are a states/events based formalism, like finite state machines and Petri nets.
- They are a data flow language, like block diagrams and reactive languages.
- They are both mathematically well defined and graphically easy to represent (with the ability to have different views of the same object as in UML).
- They are a hierarchical language with different composition operations: parallel composition, connection and synchronization.
- They can be interpreted stochastically to perform probabilistic analyses, in the same way Petri nets are.

These features make it possible to describe within a single formalism both functional and dysfonctional aspects of the systems under study.

Since mode automata are a high level language with a great expressiveness, the assessment of models is in general a difficult task. In this article, we proposed an algorithm to compile mode automata into fault trees in order to overcome this problem. This algorithm works well in practice. The idea is that simple models (those that are close to a block diagram) can be compiled simply. More complex models should be assessed by other means. For instance, it would be possible to compile complex models into Petri nets. Such a compilation could be interesting to be able to use the many available Petri net tools.

References

[1] M. AjmoneMarsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. *Modelling with Generalized Stochastic Petri Nets.* Wiley Series in Parallel Computing. John Wiley and Sons, 1994.

[2] J.D. Andrews and T.R. Moss. *Reliability and Risk Assessment.* John Wiley & Sons, 1993. ISBN 0-582-09615-4.

[3] A. Arnold. *Finite Transition Systems.* C.A.R Hoare. Prentice Hall, 1994. ISBN 0-13-092990-5.

[4] A. Arnold, A. Griffault, G. Point, and A. Rauzy. The altarica language and its semantics. *Fundamenta Informaticae*, 34:109–124, 2000.

[5] G. Ciardo, A. Blakemore, P.F. Chimento, J. Muppala, and K. S. Trivedi. Automated Generation and Analysis of Markov Reward Models using Stochastic Petri Nets. In C. Meyer and R.J. Plemmons, editors, *Linear Algebra, Markov Chains and Queueing Models*, volume 48 of *IMA Volumes in Mathematics and Applications*, pages 145–191. Springer Verlag, 1993.

[6] G. Ciardo, J. Muppala, and K. S. Trivedi. SPNP: Stochastic Petri Net Package. In *Proceedings of Third Int. Workshop on Petri Nets and Performance Models, (PNPM89)*, pages 142–152, 1989.

[7] F. DiCesare, G. Harhalakis, J.M. Proth, M. Silva, and F.B. Vernadat. *Practice of Petri Nets in Manufacturing.* Chapman and Hall, 1993.

[8] J. Esperza. Decidability and complexity of petri nets problems – an introduction. In W. Reisig and G. Rozenberg, editors, *Lectures on Petri Nets I: Basic Models*, volume 1491 of *LNCS*, pages 374–428. Springer, 1998. ISBN 3-540-65306-6.

[9] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems*, volume 1032. LNCS, 1996. ISBN 3-540-60761-1.

[10] N. Halbwachs. *Synchronous Programming of Reactive Systems.* Kluwer Academic Publisher, 1993. ISBN 0-7923-9311-2.

[11] D. Harel. Statecharts: A visual approach to complex systems. *Science of Computer Programming*, 8(3), 1987.

[12] J.E. Hopcroft and J.D. Ullman. *Introduction to automata theory languages and computation.* Addison Wesley, 1979. ISBN 0-201-1988-X.

[13] F. Maraninchi. Argonaute: graphical description, semantics and verification of reactive systems by using a process algebra. In J. Sifakis, editor, *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, volume 407 of *LNCS*. Springer Verlag, June 1989.

[14] F. Maraninchi and Y. Rémond. Mode-automata: About modes and states for reactive systems. In *European Symposium On Programming*, Lisbon (Portugal), March 1998. Springer verlag.

[15] F. Maraninchi, Y. Rémond, and Y. Raoul. Matou : An implementation of mode-automata into dc. In *Compiler Construction*, Berlin (Germany), March 2000. Springer verlag.

[16] K. McMillan. *Symbolic Model Checking.* Kluwer Academic Publisher, 1993. ISBN 0-7923-9380-5.

[17] T. Murata. Petri Nets: Properties, Analysis and Applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.

[18] G. Point and A. Rauzy. AltaRica – constraint automata as a description language. *Journal Européen des Systèmes Automatisés*, 33(8–9):1033–1052, 1999.

[19] A. Rauzy. A new methodology to handle boolean models with loops. *IEEE Transactions on Reliability*, 2001. To appear.

[20] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language. Reference Manual.* Addison Wesley, 1999. ISBN 0-201-30998-X.

[21] D.R. Shier. *Network Reliability and Algebraic Structures.* Oxford Science Publications, 1991.