

Université Paris Sud - XI  
DEA Informatique Distribuée

# Modélisation et vérification des aspects temporisés des langages pour automates programmables industriels

*Rapport de stage de DEA élaboré par :*

**Mohamed El Mongi BEN GAID**

*Encadré par :*

**Antoine PETIT et Philippe SCHNOEBELEN**

Laboratoire Spécification et Vérification

1 Avril – 31 Juillet 2003

# Remerciements

A l'issue de ce stage, je tiens à exprimer ma reconnaissance, ma gratitude et mes vifs remerciements à Béatrice Bérard, Antoine Petit et Philippe Schnoebelen pour m'avoir apporté leur connaissance et un soutien permanent dans mon stage. Leur soutien moral et leur aide précieuse m'ont permis d'effectuer ce stage dans les meilleures conditions.

Je tiens également à remercier tous les membres du LSV pour l'ambiance exceptionnelle qui règne dans le laboratoire.

# Sommaire

<b>INTRODUCTION .....</b>	<b>3</b>
<b>CHAPITRE 1 PROGRAMMATION DES AUTOMATES PROGRAMMABLES INDUSTRIELS DANS LES LANGAGES DE LA NORME IEC 61131-3 .....</b>	<b>5</b>
1.1 Les automates programmables industriels .....	5
1.2 La norme IEC 61131-3 .....	7
<b>CHAPITRE 2 LES ASPECTS TEMPORISES DANS LA PROGRAMMATION DES API .....</b>	<b>10</b>
2.1 Durée du cycle automate .....	10
2.2 Les constructions temporisées définies dans la norme IEC 61131-3 .....	11
2.2.1 Le temporisateur d'enclenchement .....	11
2.2.2 Le temporisateur de déclenchement .....	13
2.2.3 Le temporisateur impulsion .....	13
2.3 Temps de réponse et précision des temporisateurs .....	14
2.3.1 Temps d'immunité des entrées (TIE) .....	14
2.3.2 Temps de commutation des sorties (TCS) .....	14
2.3.3 Temps de réaction de l'API (TR) .....	14
2.3.4 Précision des temporisateurs .....	15
<b>CHAPITRE 3 MODELISATION DES ASPECTS TEMPORISES DES LANGAGES POUR API : ETAT DE L'ART .....</b>	<b>18</b>
3.1 Introduction .....	18
3.2 Modèles à base d'automates temporisés .....	19
3.2.1 Les automates temporisés .....	19
3.2.2 Modèle de Mader-Wupper .....	22
3.2.3 Approche de Zoubek-Roussel- Kwiatkowska .....	28
3.3 Modèles à base de systèmes conditions/événements temporisés .....	31

3.4	Modèles à base de PLC-automates .....	32
3.5	Conclusion .....	34
<b>CHAPITRE 4 VERIFICATION DE MODELES TEMPORISES PAR LA</b>		
<b>TECHNIQUE DU MODEL-CHECKING .....</b>		<b>36</b>
4.1	Le model-checking .....	36
4.2	Les cas d'étude .....	37
4.2.1	Le mélangeur .....	37
4.2.2	L'évaporateur .....	38
4.3	Modélisation du comportement cyclique et des contraintes temporisées.....	44
4.3.1	Modèle de Mader-Wupper .....	45
4.3.2	Modèle basé sur l'exécution instantanée et atomique du programme .....	46
4.4	Étude expérimentale de l'exemple du mélangeur .....	47
4.4.1	Codage du programme dans l'outil UPPAAL suivant le modèle de Mader-Wupper .....	47
4.4.2	Codage dans l'outil UPPAAL suivant le modèle basé sur l'exécution atomique du programme. .....	50
4.4.3	Étude de l'influence des valeurs des paramètres $\varepsilon_1$ , $\varepsilon_2$ et PT sur la complexité de la vérification du programme codé suivant modèle de Mader-Wupper.....	50
4.4.4	Étude comparative du modèle de Mader-Wupper et du modèle basé sur l'exécution atomique instantanée du programme.....	54
4.5	Étude expérimentale de l'exemple de l'évaporateur.....	55
4.5.1	Codage dans l'outil UPPAAL .....	55
4.5.2	Formalisation des propriétés.....	59
4.5.3	Résultats expérimentaux.....	59
4.5.4	Conclusion.....	60
<b>CONCLUSION .....</b>		<b>61</b>
<b>BIBLIOGRAPHIE.....</b>		<b>63</b>

# Introduction

Notre expérience quotidienne avec les ordinateurs nous montre à tel point les erreurs de programmation sont présentes. La complexité des logiciels que nous utilisons fait en sorte que ces fameux ‘bugs’ sont une conséquence indissociable de la programmation. Ces erreurs ont des conséquences beaucoup plus graves lorsqu’elles se produisent dans un système informatique amené à interagir avec un environnement physique mettant en œuvre des quantités d’énergie importantes. Dans ces applications dites critiques, une mauvaise consigne du système de contrôle-commande peut engendrer des défaillances conduisant par exemple à un arrêt de production dans une usine ou dans des cas extrêmes occasionner des blessures ou des décès. L’explosion de la fusée ARIANE 5 en est une triste illustration.

Les *automates programmables industriels (API)*, en anglais *programmable logic controllers (PLC)*, ont été inventés à la fin des années 60. Depuis, leur utilisation s’est largement répandue dans l’industrie, où ils représentent l’outil de base de l’automatisation des systèmes de production. Un API permet de piloter un système de production conformément à un programme placé dans sa mémoire.

Le besoin de garantir la sûreté des programmes pour API se fait ressentir de plus en plus en milieu industriel. Depuis quelques années, de nombreux travaux de recherches visant à améliorer la sûreté de ces applications par les méthodes formelles ont été réalisés [FL00]. L’apport de ces méthodes a été très important. Nous nous intéressons particulièrement à la technique du *model-checking* [SBB+99, CGP99]. Cette technique permet de vérifier si un modèle vérifie une propriété en explorant exhaustivement tous les comportements décrits par le modèle.

De nombreux travaux visant à la vérification des programmes d’API par la technique du model-checking ont été réalisés. Une grande contribution a été accomplie dans le cadre du projet VULCAIN, entrepris par deux laboratoires l’École Normale Supérieure de Cachan : le Laboratoire Spécification et Vérification (LSV) et le Laboratoire Universitaire de Recherche en Production Automatisée (LURPA) d’une part, et Alcatel Research & Innovation d’autre part. Ces travaux [CCL+00, CDP+00, LL00, LRL99, RS00, RSC+00] ont permis de définir

une démarche complète permettant la vérification de programmes d'API écrits dans les langages standardisés définis par la norme IEC 61131-3. Ils se sont focalisés sur des programmes exécutés sur des API mono-tâche et sans prendre en compte explicitement les aspects temporisés présents dans ces langages. La pertinence des choix théoriques de cette démarche a été illustrée sur des exemples réels. Ce stage de DEA s'inscrit dans la continuité de ces travaux et vise à étendre ces travaux par la prise en compte des aspects temporisés de ces langages.

Le premier chapitre illustre le principe de fonctionnement des API et les langages normalisés utilisés pour leur programmation et définis dans la norme IEC 61131-3.

Le deuxième chapitre décrit les différents aspects temporisés qui interviennent lors de l'exécution d'un programme sur un API.

Le troisième chapitre dresse un état de l'art des modèles prenant en compte ces aspects temporisés, et destinés à être vérifiés par model-checking.

Le quatrième chapitre présente certains aspects de modélisation permettant de réduire considérablement la complexité de la vérification des propriétés temporisées. L'apport des modèles décrits est validé expérimentalement sur des exemples non triviaux.

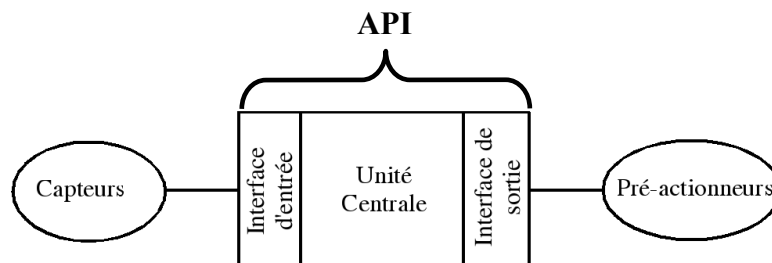
# Chapitre 1

## Programmation des automates programmables industriels dans les langages de la norme IEC 61131-3

### 1.1 Les automates programmables industriels

Un automate programmable industriel (API) est une machine électronique programmable utilisée pour piloter des systèmes automatisés. Sa flexibilité explique son large domaine d'utilisation, qui comporte certaines applications critiques, où des erreurs de programmation sont susceptibles de causer des dommages humains ou matériels.

Un API est généralement placé en ambiance industrielle, où il représente le cœur de la partie commande d'un système automatisé. Il est en relation avec les autres parties du système



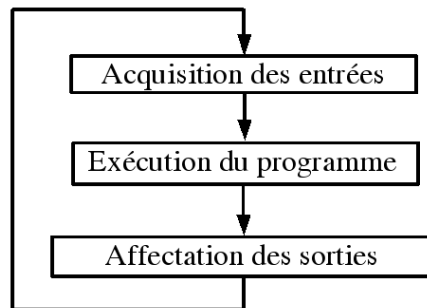
*Figure 1*  
*Relation d'un API avec les autres parties d'un système automatisé*

grâce à son interface d'entrée-sortie (figure 1).

Une grande partie des API du marché possèdent un moniteur d'exécution mono-tâche. Cependant, pour des raisons de performance, de plus en plus de constructeurs proposent des modèles dont le moniteur d'exécution supporte le multi-tâches et les interruptions. Dans la suite, on s'intéressera particulièrement aux API mono-tâche cycliques. Néanmoins, la vérification des API multi-tâches est beaucoup plus difficile que celle des API mono-tâche.

Dans un API cyclique, le programme s'exécute dans une boucle permanente. Dans chaque itération de cette boucle ou cycle, trois types d'actions (l'acquisition des entrées, l'exécution du programme et l'affectation des sorties) sont effectuées. L'ordre et la répartition de ces actions

dans le cycle conditionnent sa structure. Il existe diverses structures possibles pour un cycle [Mic88]. Nous nous intéresserons dans la suite aux API dont le cycle est composé des trois phases suivantes (figure 2):



*Figure 2*  
*Le comportement cyclique d'un API*

#### *Phase d'acquisition des entrées*

Durant cette phase, les signaux appliqués à l'interface d'entrée de l'API sont copiés en mémoire dans des emplacements accessibles au programme et qui correspondent aux variables d'entrée. Les variables d'entrée sont uniquement accessibles en lecture. Leurs valeurs resteront ainsi inchangées lors des deux phases suivantes du cycle. En d'autres termes, au moment de l'acquisition des entrées, l'API "prend une photo" de l'environnement physique.

#### *Phase d'exécution du programme*

La phase d'exécution du programme permet de calculer les nouvelles valeurs des variables de sortie.

#### *Phase d'affectation des sorties*

Les variables de sortie sont affectées à l'interface de sortie pour pouvoir être appliquées aux préactionneurs.

Notons enfin qu'il existe une classe particulière d'API : les API périodiques. Contrairement aux API cycliques où les cycles sont enchaînés sans attente (si le cycle courant se termine, le cycle suivant est immédiatement commencé), dans les API périodiques, les cycles commencent à des intervalles réguliers, fixés par l'utilisateur et dont la durée doit être supérieure à la somme des durées d'exécution du plus lent chemin du programme et de la phase d'entrée-sortie. Dans ces API, les entrées et les sorties sont toujours lues et écrites périodiquement (deux lectures ou écritures successives sont séparées par la période fixée par l'utilisateur).



## 1.2 La norme IEC 61131-3

Un API est programmé à l'aide de langages spécialisés, fournis par son constructeur et utilisables au travers d'une interface (un logiciel sur PC, un pupitre...). Ces langages peuvent être classés en 5 grandes familles. Cependant, deux langages de la même famille et fournis par deux constructeurs différents ne sont pas forcément compatibles, ce qui est de nature à nuire à la portabilité des applications et à limiter la réutilisation du code. C'est pour cette raison que la commission électrotechnique internationale a entrepris un grand effort de normalisation visant à uniformiser les langages utilisés dans le domaine de la programmation des API, ce qui a donné naissance à la norme IEC 61131-3 [IEC93]. Ce standard définit cinq langages correspondant aux familles de langages les plus utilisées pour la programmation des API [Lew98].

Les langages sont :

- **Instruction List (IL)** : un langage textuel de type assembleur.

```
PROGRAM And
VAR_INPUT
I1 : BOOL;
I2 : BOOL;
END_VAR

VAR_OUTPUT
O : BOOL;
END_VAR

LD I1
AND I2
ST O

END_PROGRAM
```

*Figure 3*  
*Un API exécutant ce programme IL*  
*joue le rôle d'une porte ET*

- **Structured Text (ST)** : un langage textuel structuré similaire au Pascal.

```
IF RUN THEN XOUT := XOUT + K * (XIN - XOUT) ;
ELSE XOUT := XIN ;
    K := TIME_TO_REAL(CYCLE) / TIME_TO_REAL(CYCLE + TAU) ;
END_IF ;
```

*Figure 4*  
*Fragment de code ST donné comme*  
*exemple dans [IEC93]*

- **Ladder Diagram (LD)** : un langage graphique, très utilisé en milieu industriel, car il s'inspire des circuits de commande basés sur la logique électrique, les équations combinatoires étant câblées à l'aide de contacts et de relais. Un programme est décrit par un diagramme sous forme d'échelle. Chaque échelon de l'échelle contient un ensemble de symboles graphiques qui peuvent être des contacts ou des bobines. Un contact permet la lecture d'une variable booléenne tandis qu'une bobine permet d'affecter une valeur à une variable booléenne. La figure 5 illustre un exemple de programme Ladder.

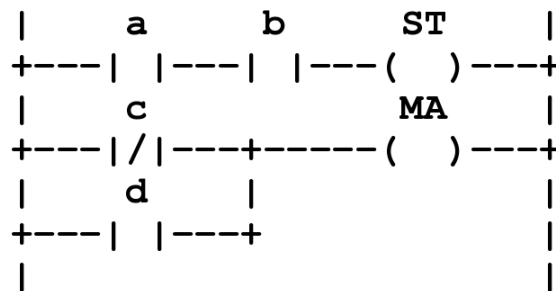


Figure 5  
 Ce programme Ladder réalise les opérations suivantes :  
 $ST := a \text{ and } b$   
 $MA := \text{not}(c) \text{ or } d$

- **Function Block Diagram (FBD)** : un langage graphique permettant d'exprimer le comportement des fonctions, des blocs fonctionnels ou des programmes comme un ensemble de boîtes noires interconnectées (à la manière des portes logiques en électronique).

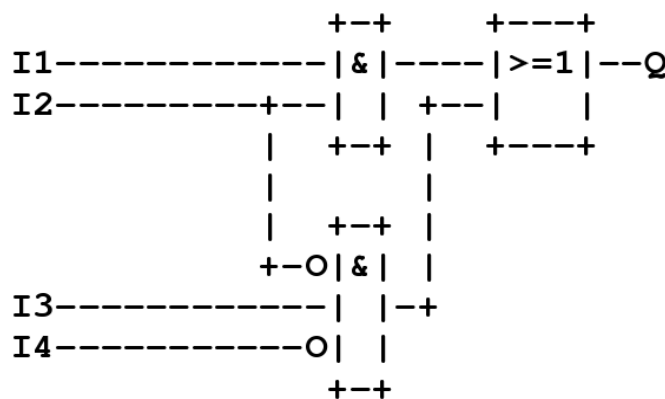


Figure 6  
 Ce programme FBD réalise l'opération suivante:  
 $Q := (I1 \text{ and } I2) \text{ or } (\text{not}(I2) \text{ and } I3 \text{ and } \text{not}(I4))$

- **Sequential Function Charts (SFC)** : un langage graphique permettant de structurer tout comportement séquentiel pouvant être décrit dans l'un des quatre autres langages de la norme.

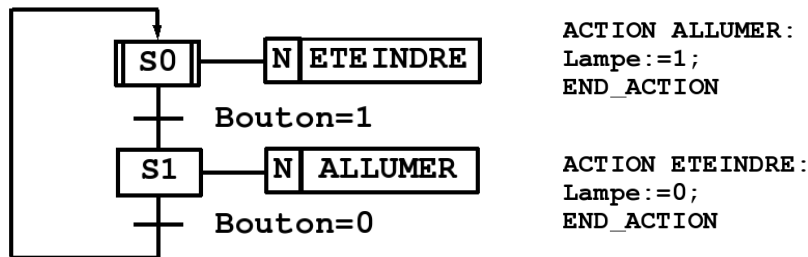


Figure 7

Initialement, l'étape **S0** est activée. Quand la réceptivité **Bouton=1** devient vraie alors l'étape **S0** est immédiatement désactivée et l'étape **S1** immédiatement activée : la transition entre ces deux étapes est franchie. Si l'étape **S1** est activée et la réceptivité **Bouton=0** est vraie, alors l'étape **S1** est désactivée et l'étape **S0** activée. Le qualificateur d'action **N** indique que l'action associée à l'étape est exécutée durant l'activation de l'étape. Les actions **ETEINDRE** ET **ALLUMER** associées respectivement aux étapes **S0** et **S1** sont décrites en ST (de même que les réceptivités).

## Chapitre 2

# Les aspects temporisés dans la programmation des API

Les API sont souvent utilisés comme plates-formes matérielles d'exécution d'applications temps-réel, c'est-à-dire d'applications où des contraintes quantitatives sur les temps de réaction aux entrées doivent être satisfaites. Ces contraintes sont généralement exprimées de façon informelle dans leurs cahiers des charges. Les différents cas d'études rencontrés dans la littérature fournissent de nombreux exemples de telles propriétés :

- La valve V1 a été ouverte durant 5 secondes avant le démarrage de la pompe P1 [ZRK03].
- Dans chaque intervalle de temps de largeur au plus égale à 30 secondes, la durée de fuite du gaz ne dépasse pas 4 secondes [Die00].
- Le signal de sortie n'est engendré que lorsque les deux boutons poussoir sont manœuvrés dans un délai de temps inférieur à 0.5 secondes (Norme européenne EN 574).

La vérification de ces propriétés fait intervenir des caractéristiques temporelles relatives à l'automate programmable et au programme.

### 2.1 Durée du cycle automate

La connaissance des durées possibles d'un cycle est nécessaire dans le cas où des propriétés faisant intervenir des durées du même ordre de grandeur doivent être vérifiées.

La durée d'un cycle peut être exprimée comme la somme de la durée de la phase d'entrées-sorties et de la phase d'exécution du programme.

#### *Durée de la phase d'entrées-sorties*

Elle dépend de la technologie des entrées et des sorties (transistors, relais,...). Sa durée est généralement constante. Elle se situe typiquement entre quelques 10 microsecondes et 100 millisecondes.

#### *Durée de la phase d'exécution du programme*

La durée de cette phase est variable, cette variation est due aux différents chemins susceptibles d'être pris au cours de l'exécution du programme.

Sa valeur se situe entre 1 microseconde et 500 millisecondes (en fonction des caractéristiques de l'unité de traitement et de la taille du programme).

[Per99] décrit une méthode expérimentale permettant d'estimer la durée du cycle d'un API exécutant un programme écrit en Grafset en fonction de ses différentes modalités (nombre total de transitions, nombre total d'étapes, complexité des réceptivités, nombre de transitions validées simultanément, nombre d'entrées ayant varié au cours du cycle, nombre de sorties émises... ). Cette méthode se base sur la méthode des plans d'expérience de Taguchi [Pil94]. La plate forme expérimentale ayant servi à la validation de cette méthode y est également décrite. Dans [Lav98], une démarche similaire est appliquée aux programmes Ladder.

## 2.2 Les constructions temporisées définies dans la norme IEC 61131-3

Les langages de programmation des API, et en particulier dans leurs descriptions au sein de la norme IEC 61131-3, offrent un certain nombre de constructions qui font intervenir le temps «physique» de façon explicite. Les constructions les plus utilisées sont les temporisateurs.

La temporisation est une fonction que l'on retrouve dans un grand nombre d'applications. Elle est utilisée principalement afin de différer, d'une durée choisie fixée à l'avance, l'activation ou la désactivation d'une sortie. Elle est implémentée de façon logicielle dans la quasi-totalité des API existant sur le marché. La norme IEC 61131-3 définit trois types de temporisateurs sous forme de blocs fonctionnels, c'est à dire dans une approche boîte noire, définissant les entrées, les sorties et les valeurs susceptibles d'être prises par les sorties en fonction des valeurs actuelles ou passées des entrées.

### 2.2.1 Le temporisateur d'enclenchement

Le temporisateur d'enclenchement, ou *Timer On-delay* (TON) possède deux entrées (figure 8) :

- IN, de type BOOL, qui permet de lancer ou d'annuler la temporisation.
- PT (*Preset Time*), de type TIME, permet de spécifier la durée de temporisation.

et deux sorties :

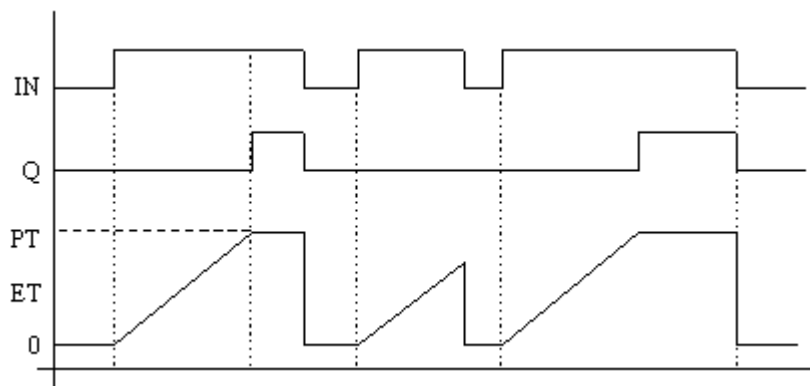
- Q, de type BOOL, qui indique si la durée de temporisation a expiré.

- ET (*Elapsed Time*), de type TIME, indique le temps écoulé depuis le début de la temporisation.



*Figure 8*  
*Le temporisateur TON*

La norme décrit le comportement du temporisateur TON à l'aide du chronogramme de la figure 9.



*Figure 9*  
*Chronogramme décrivant le comportement du temporisateur TON*

Ce chronogramme décrit un comportement possible du temporisateur dans le temps.

Lorsque IN passe de 0 à 1, la temporisation est initiée, la sortie ET, qui indique le temps écoulé depuis le début de la temporisation, croît alors jusqu'à atteindre la valeur de PT. A ce moment, la sortie Q est mise à 1.

Quand IN passe à 0, alors ET et Q sont immédiatement mises à 0.

Si la temporisation est abandonnée (IN passe de 1 à 0), alors ET est immédiatement remise à 0.

Nous verrons, dans le chapitre suivant, les différentes formalisations du temporisateur TON proposées dans la littérature.

Notons enfin que l'effet du changement de valeur de PT, en cours de temporisation, n'a pas été spécifié par la norme (il dépend de l'implémentation).

### 2.2.2 Le temporisateur de déclenchement

Le bloc TOF possède les mêmes entrées et sorties que le bloc TON (Figure 10), cependant



Figure 10  
Le temporisateur TOF

son comportement est différent (Figure 11).

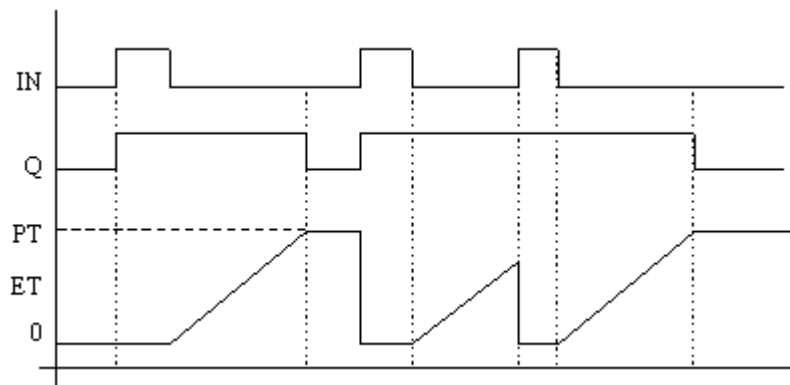


Figure 11  
Chronogramme décrivant le  
comportement du temporisateur TOF

Lorsque l'entrée IN passe de 0 à 1, la sortie Q est immédiatement mise à 1. La temporisation n'est déclenchée que lorsque IN passe de 1 à 0. La sortie ET croît alors jusqu'à atteindre PT. A ce moment, la sortie Q est remise à zéro tandis que ET garde la valeur du seuil PT. La sortie ET est remise à zéro lorsque IN passe de 0 à 1.

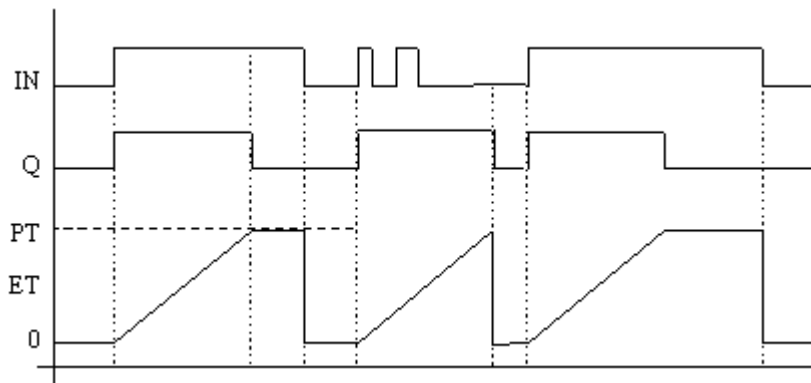
### 2.2.3 Le temporisateur impulsion

Le temporisateur impulsion possède les mêmes entrées et les mêmes sorties que les temporisateurs TON et TOF (Figure 12).



Figure 12  
Le temporisateur TP

Ce temporisateur permet de créer une impulsion de durée PT, en réponse à un front montant du signal d'entrée, et à condition qu'au moment du front montant de l'entrée IN, la sortie Q soit à 0 (Figure 13).



*Figure 13*  
*Chronogramme décrivant le*  
*comportement du temporisateur TP*

## 2.3 Temps de réponse et précision des temporisateurs

### 2.3.1 Temps d'immunité des entrées (TIE)

C'est le délai dû au filtrage que subissent les signaux d'entrée à travers les cartes d'entrées. Ce filtrage vise à éliminer les bruits capteur. Cette durée est de l'ordre de quelques dizaines de millisecondes.

### 2.3.2 Temps de commutation des sorties (TCS)

C'est le délai nécessaire à la commutation des sorties. Il dépend de la technologie des sorties (transistors ou relais,...).

L'ordre de grandeur de ces délais se situe entre quelques centaines de microsecondes (sorties transistor) et quelques dizaines de millisecondes (pour les sorties relais).

### 2.3.3 Temps de réaction de l'API (TR)

C'est le délai séparant le changement de valeur d'un signal appliqué à l'interface d'entrée de l'API et l'établissement du signal de sortie adéquat. Bien entendu, le temps de réponse est défini pour un programme qui doit réagir immédiatement au signal d'entrée. Sa valeur est encadrée selon la formule :

$$TIE + TC + TCS \leq TR \leq TIE + 2TC + TCS$$

où TC est le temps de cycle.



La figure 14 correspond à la borne inférieure du temps de réponse : lorsque la durée d'immunité aux entrées se termine juste avant le début de la phase d'acquisition des entrées.

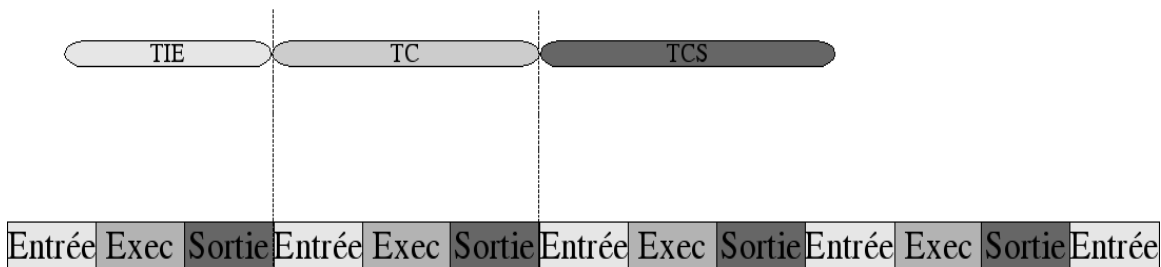


Figure 14  
Borne inférieure du temps de réponse

La figure 15 illustre la borne supérieure du temps de réponse : quand le changement du signal d'entrée ne peut être pris en compte que lors du cycle suivant.

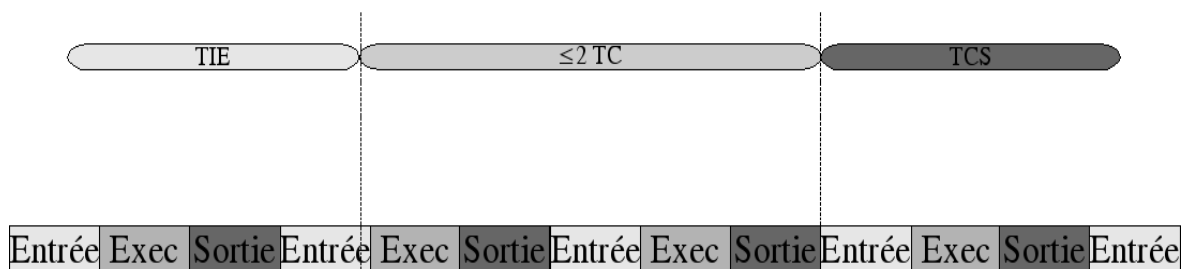


Figure 15  
Borne supérieure du temps de réponse

### 2.3.4 Précision des temporisateurs

L'erreur (logicielle) engendrée par l'utilisation d'un temporisateur logiciel, implémenté sur un API, à la place d'un temporisateur matériel, est la somme de plusieurs erreurs que nous décrivons ci-dessous.

#### a) L'erreur (logicielle) sur l'entrée

C'est la durée séparant l'instant de changement de valeur du signal INPUT appliqué à l'interface d'entrée de la prise en compte de ce changement par l'appel *CAL Timer* (Timer étant une instance du bloc fonctionnel TON).

Dans l'exemple de la figure 16 le signal d'entrée INPUT change de valeur juste après la fin de la phase d'acquisition des entrées. Ce changement ne sera donc pris en compte que lors du cycle suivant. La variable d'entrée I prend alors la nouvelle valeur du signal (I:=TRUE). Le temporisateur ne sera armé que lors de l'exécution de l'instruction *CAL Timer*

(*Timer.IN:=TRUE, Timer.PT*). L'erreur est donc maximale si cette instruction se trouve à la fin de la phase d'exécution.

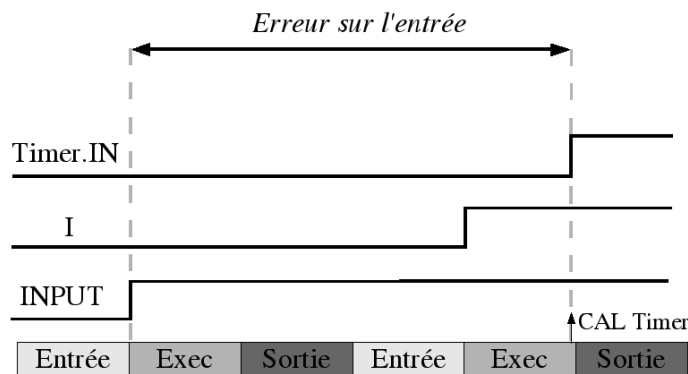


Figure 16  
Erreur logicielle sur l'entrée

Dans ce cas de figure on a :  $Erreur\ sur\ l'entrée = Durée\ du\ cycle + Durée\ de\ la\ phase\ d'exécution$ .

**b) L'erreur (logicielle) sur la sortie**

C'est la durée entre l'expiration de la temporisation et l'ordre de mise à jour du signal de sortie OUTPUT.

Dans l'exemple de la figure 17, la durée de temporisation expire (*TIMEOUT:=TRUE*)

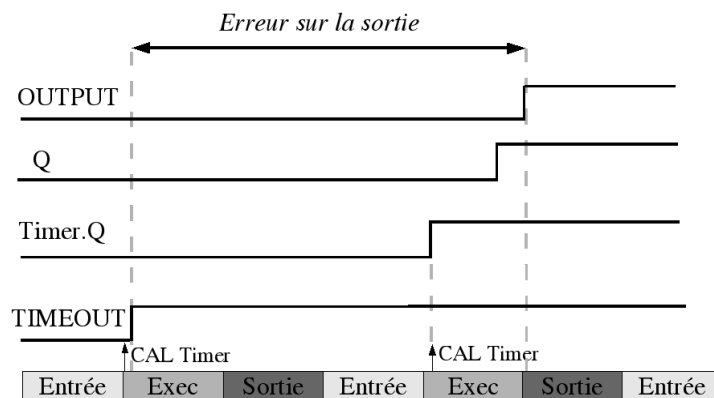


Figure 17  
Erreur logicielle sur la sortie

juste après l'instruction *CAL Timer*. L'expiration de la durée de temporisation ne sera prise en compte que lors du cycle suivant. La sortie *Timer.Q* du bloc fonctionnel *Timer* puis la variable de sortie *Q* sont ainsi mises à jour (*TIMER.Q:=1* et *Q:=1*). L'ordre de mise à jour du signal de sortie *OUTPUT* est mis à un à la fin de ce cycle. Dans ce cas de figure on a :  $Erreur\ sur\ la\ sortie = Durée\ du\ cycle + Durée\ de\ la\ phase\ d'exécution$ .

Dans la plupart des cas, les durées de temporisation utilisées sont beaucoup plus grandes que la durée d'un cycle. Les erreurs évoquées ci-dessous peuvent alors être négligées.

Cependant, certains constructeurs proposent des temporisateurs dont la base de temps est égale à 1ms, ce qui est du même ordre de grandeur que la durée d'un cycle. La précision du temporisateur dans ce cas peut causer la non-satisfaction de propriétés de réponse : La modélisation devrait ainsi en tenir compte.

## Chapitre 3

# Modélisation des aspects temporisés des langages pour API : état de l'art

### 3.1 Introduction

L'aspect critique des applications implémentées dans les API a été l'une des motivations de nombreux travaux de recherche visant à introduire les méthodes formelles dans le domaine de la programmation des API. De nombreux inventaires visant à classer ces travaux ont ainsi été réalisés.

Dans [FL00], une classification des différentes méthodes de vérification et de validation formelle dans le domaine de la programmation des API est donnée. Cette classification se base sur trois critères :

- L'approche, qui peut être soit :
  - Une approche *basée sur un modèle*, c'est à dire une approche où un modèle du processus à contrôler est inclus dans l'analyse.
  - Une approche *non basée sur un modèle* ; l'analyse de la description formelle du programme de contrôle-commande se fait sans prise en compte du processus à contrôler.
  - Une approche *basée sur des contraintes*, et qui peut être classée parmi les approches non basées sur un modèle, mais qui inclut en plus une connaissance très restreinte sur le processus à contrôler.
- Le formalisme utilisé pour décrire les programmes, ou plus précisément, leurs exécutions sur un API (automates, réseaux de Petri, systèmes condition/événement, langages synchrones, systèmes de transitions étiquetés, logiques d'ordre supérieur, équations algébriques...)
- La méthode utilisée pour la vérification et la validation (la simulation, l'analyse d'accessibilité, le model-checking, le theorem proving...)

Dans [Mad00], Mader propose une classification des différents modèles possibles pour un programme qui s'exécute sur un API. Cette classification se base sur trois critères :

- La modélisation du cycle automate, ou en d'autres termes, jusqu'à quel niveau de détail faut-il modéliser le comportement cyclique des API. Quatre modèles ont été proposés :
  - Les modèles qui ne tiennent pas compte du comportement cyclique. Ces modèles sont essentiellement destinés à l'analyse statique des programmes.
  - Les modèles qui ne prennent pas en compte la durée du cycle. Ces modèles conviennent pour des applications où la durée du cycle est négligeable devant la dynamique de l'environnement physique. L'exécution du programme et la phase d'entrée-sortie sont alors supposées instantanées.
  - Les modèles qui représentent de façon implicite la durée du cycle. Dans ces modèles, le comportement cyclique est modélisé, mais la durée du cycle n'est pas considérée explicitement. C'est le cas par exemple des API périodiques et où aucune modélisation de l'environnement n'est considérée.
  - Les modèles qui représentent de façon explicite le comportement cyclique : les bornes inférieure et supérieure de la durée du cycle sont prises en compte dans le modèle. Il s'agit des modèles les plus réalistes, et qui permettent de vérifier de façon naturelle des propriétés temps-réel.
- L'utilisation de temporisateurs.
- Les fragments des langages considérés, par exemple, des fragments qui supportent uniquement les variables booléennes.

On s'intéressera dans la suite plus particulièrement aux démarches permettant la vérification de propriétés temps-réel par la technique du model-checking. On classera ces démarches suivant le formalisme qu'elles utilisent pour modéliser l'exécution d'un programme sur un API.

## 3.2 Modèles à base d'automates temporisés

### 3.2.1 Les automates temporisés

Les automates temporisés ont été proposés en 1990 par Alur et Dill [AD94] afin de permettre de modéliser des systèmes où les délais séparant deux actions doivent être pris en compte de façon explicite. Il existe plusieurs outils permettant de vérifier des systèmes décrits sous forme d'automates temporisés, parmi lesquels on peut citer UPPAAL [LPW97, ABB+01, BDL+01] et KRONOS [OY93, Yov97].

### a) Description

Un automate temporisé est une machine à états finis étendue par des variables à valeurs réelles positives nommées *horloges*. Initialement, toutes les horloges ont la valeur 0. Ensuite, elles avancent toutes à la même vitesse (celle du temps universel) et peuvent être remises à zéro au cours des transitions de l'automate. Une expression booléenne sur les horloges (ou contrainte sur les horloges), appelée *garde*, est associée à chaque transition. Une transition ne peut être prise que si les valeurs actuelles des horloges satisfont les gardes. Une contrainte sur les horloges est également associée à chaque état de contrôle de l'automate. Cette contrainte est appelée *invariant* de l'état de contrôle. Le temps ne peut s'écouler à l'intérieur d'un état de contrôle que si l'invariant est vrai. Les invariants permettent de forcer l'automate à quitter un état de contrôle en franchissant une transition. Si aucune transition ne peut être prise, l'automate se trouve dans une version temporisée du blocage : le *livelock*.

### b) Définitions

*Valuation d'une horloge*

Soit  $X$  un ensemble fini d'horloges.

Une fonction  $v : X \rightarrow \mathbb{R}_+$  est appelée une valuation sur les horloges.

Si  $v \in (\mathbb{R}_+)^X$  est une valuation et  $t \in \mathbb{R}_+$  alors la valuation qui correspond à l'écoulement d'une durée  $t$  et qui est définie par :

$$(v+t)(x) = v(x) + t, \quad \forall x \in X.$$

Soit  $Y \subseteq X$ . La valuation  $v[Y \leftarrow 0]$  est définie par :

$$\begin{cases} v[Y \leftarrow 0](x) = 0 & \text{si } x \in Y. \\ v[Y \leftarrow 0](x) = v(x) & \text{si } x \notin Y. \end{cases}$$

*Garde*

L'ensemble des gardes (ou contraintes)  $C(X)$  sur l'ensemble des horloges  $X$  est défini par la grammaire suivante :

$$\varphi ::= x \sim c \mid x - y \sim c \mid \varphi \wedge \varphi \mid \text{true},$$

avec  $x, y \in X, c \in \mathbb{R}_+, \sim \in \{<, \leq, =, \geq, >\}$

Si  $v$  est une valuation telle que  $(v(x))_{x \in X}$  satisfait la garde  $\varphi \in C(X)$ , on dit que  $v$  vérifie  $\varphi$ .

$x > 3 \wedge y = 5$  est un exemple de garde. Si  $v(x) = 7$  et  $v(y) = 5$ , la valuation  $v$  vérifie cette garde.

### Définition d'un automate temporisé

Un automate temporisé est un 6-uplet  $(L, l_0, X, \Sigma, E, Inv)$  où

- $L$  est un ensemble fini d'états de contrôles (*locations*).
- $l_0$  est l'état de contrôle initial.
- $X$  est un ensemble fini d'horloges.
- $\Sigma$  est un ensemble fini d'actions.
- $E \subseteq L \times \Sigma \times C(X) \times P(X) \times L$  est un ensemble fini de transitions. Une transition  $e$  est définie par un 5-uplet  $(q, a, \varphi, Y, q')$ .  $e$  est la transition reliant l'état de contrôle  $q$  à l'état de contrôle  $q'$ , avec l'action  $a$ , la garde  $\varphi$  (la transition ne peut être prise que si la garde associée est vérifiée pour les valeurs courantes des horloges) et  $Y$  l'ensemble d'horloges à réinitialiser (à mettre à zéro). La transition  $e$  est notée  $q \xrightarrow{a, \varphi, Y} q'$ .
- $Inv \in C(X)$  associe un invariant à chaque état de contrôle.

### Sémantique d'un automate temporisé

La sémantique d'un automate temporisé  $A$  est définie par un système de transitions temporisé  $S_A = (Q, Q_0, \rightarrow)$  où  $Q = L \times (\mathbb{R}_+)^X$ ,  $Q_0 = (l_0, 0)$  est l'état initial et  $\rightarrow$  est définie, pour  $a \in \Sigma$ , et  $t \in \mathbb{R}_+$  par :

- Les transitions discrètes :  $(l, v) \xrightarrow{a} (l', v')$  si et seulement si :

$$\left\{ \begin{array}{l} \exists (l, a, \varphi, Y, l') \in E \text{ tel que :} \\ v \text{ vérifie } \varphi \\ v' = v[Y \leftarrow 0] \\ v' \text{ vérifie } Inv(l') \end{array} \right.$$

- Les transitions correspondant à l'écoulement du temps :  $(l, v) \xrightarrow{t} (l', v')$  si et seulement si :

$$\left\{ \begin{array}{l} v' = v + t \\ \forall t' \in [0, t], (v + t') \text{ vérifie } Inv(l) \end{array} \right.$$

*Exemple :*

L'automate temporisé de la figure 18 décrit une sonnerie. Il comporte deux états (**Inactif** et **Sonne**) et une horloge  $x$ . Si l'utilisateur appuie sur le bouton *push*, alors la sonnerie va sonner durant exactement 5 secondes. L'horloge  $x$  est remise à zéro lors de la transition *push* pour pouvoir mesurer la durée de la sonnerie. L'invariant  $x \leq 5$  et la garde  $x=5$  font en sorte que l'automate quitte l'état **Sonne** après exactement 5 secondes.

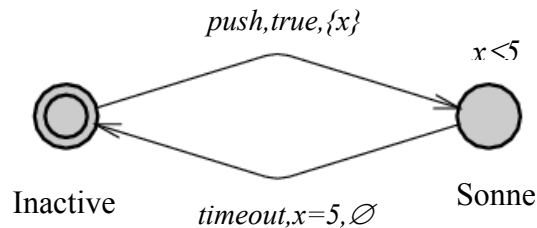


Figure 18  
Automate temporisé d'une sonnerie

### 3.2.2 Modèle de Mader-Wupper

Ce modèle définit une sémantique à base d'automates temporisés pour un fragment du langage IL. Le fragment considéré autorise uniquement l'utilisation de variables booléennes. Ce fragment prend en compte les programmes contenant des temporisateurs TON.

#### a) Sémantique opérationnelle d'un programme ne contenant pas de temporisateurs

Formellement, un programme  $P$  est une fonction  $P : [0 \dots n-1] \rightarrow Inst$ , qui à chaque adresse associe une instruction.

Avec:

- $n$  : Le nombre d'instructions du programme
- $Inst$  : L'ensemble des instructions.

Par exemple,  $P(3)=LD\ a$

On définit :

- $IB=\{false,true\}$ .
- $X$  : L'ensemble des variables booléennes du programme, parmi lesquelles on distingue :
  - $AE$  : qui désigne l'accumulateur.
  - $in$  : un vecteur de variables qui représente la partie de la mémoire réservée aux variables d'entrée.



- $out$  : un vecteur de variables qui représente la partie de la mémoire réservée aux variables de sortie.
- $q_0 : X \rightarrow IB$  : la valuation initiale des variables du programme.

Étant donné un programme  $P$ , l'automate temporisé  $T_P$  définissant la sémantique opérationnelle du programme est défini de la manière suivante :

- Un état de contrôle est un quintuplet  $(io, i, q, IN, OUT)$  dans lequel :
  - $io \in IB$  a la valeur *true* si les données sont transportées entre la mémoire et les ports d'entrées-sorties, la valeur *false* sinon (c'est-à-dire, durant l'exécution du programme).
  - $i \in [0 \dots n]$  est le compteur de programme.
  - $q : X \rightarrow IB$ , est la valuation de toutes les variables (variables d'entrée, variables internes et variables de sortie).
  - $IN \in IB^l$  est le vecteur contenant les valeurs des ports d'entrée.  $l$  représente le nombre d'entrées.
  - $OUT \in IB^m$  est le vecteur contenant les valeurs des ports de sortie.  $m$  représente le nombre de sorties.

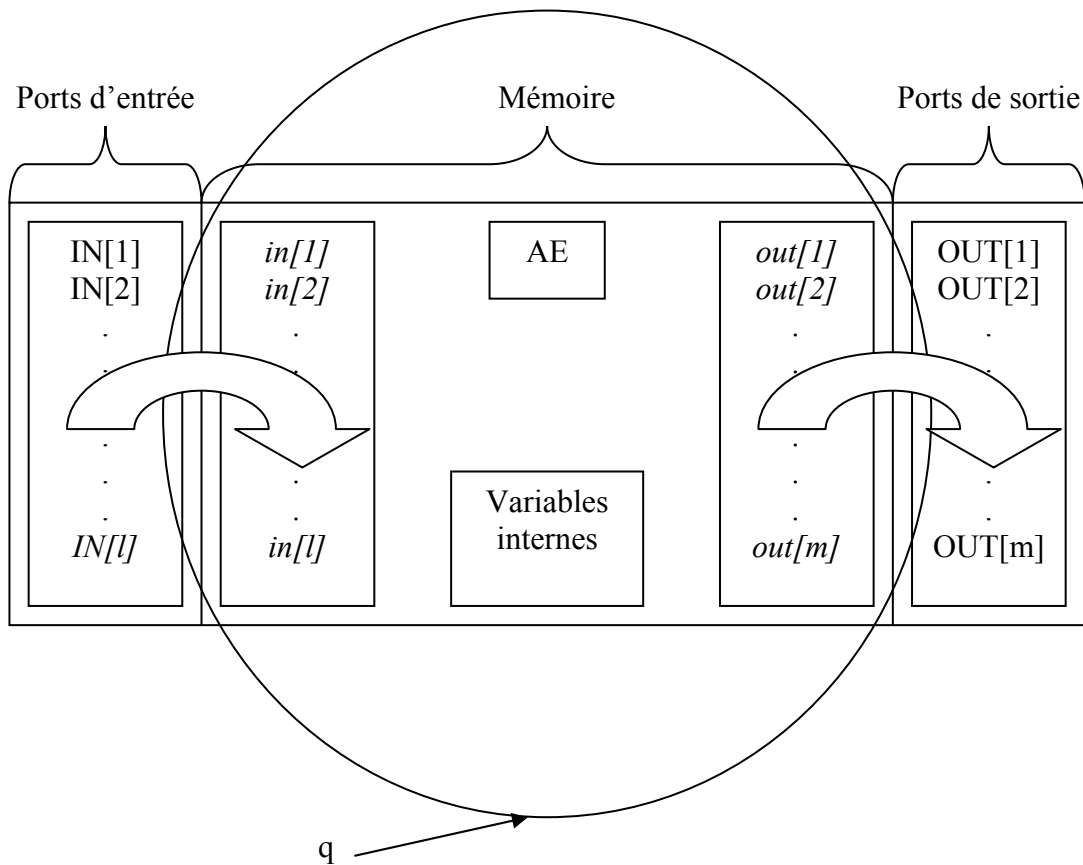


Figure 19

Un API contient une mémoire et des ports d'entrée-sortie  $q$  est la valuation des variables en mémoire, tandis que  $IN$  et  $OUT$  sont des vecteurs contenant les valeurs des ports d'entrée-sortie.  $IN$  est copiée en mémoire en début de cycle tandis que  $out$  est copiée dans  $OUT$  en fin de cycle.

- Les états initiaux sont de la forme  $(true, 0, q_0, IN, OUT)$  où  $q_0$  est la valuation initiale des variables de ce programme.
- Une horloge  $x$  mesure la longueur du cycle. Cette horloge est remise à zéro après chaque exécution du programme.
- Un invariant  $x \leq \varepsilon_1$  est appliqué à tous les états de contrôle où  $io = true$ . La valeur  $\varepsilon_1$  modélise la limite supérieure de la phase d'entrée-sortie.
- Un invariant  $x \leq \varepsilon_2$  (avec  $0 \leq \varepsilon_1 \leq \varepsilon_2$ ) est appliqué à tous les états de contrôle. La valeur  $\varepsilon_2$  modélise la limite supérieure du temps de cycle.
- Les transitions de cet automate sont étiquetées par un nom d'action, une garde et une liste d'horloges qui seront mises à zéro.  $\tau$  représente toute action interne (exécution du programme), *Environnement* correspond à un changement de valeur d'un port d'entrée et *Input/Output* désigne la phase d'entrée-sortie. Ces transitions correspondent :

- Au changement de valeur dans un port d'entrée (Une entrée peut prendre n'importe quelle valeur à n'importe quel instant).

$IN \neq IN'$  ( $IN'$  désigne la nouvelle valeur des ports d'entrée).

$$(io, i, q, IN, OUT) \xrightarrow{\text{Environnement}, true, \emptyset} (io, i, q, IN', OUT)$$

- A la phase d'entrées-sorties

$$(true, i, q, IN, OUT) \xrightarrow{\text{Input / Output}, true, \emptyset} (false, 0, q[in := IN], IN, out)$$

Les valeurs des ports d'entrée ( $IN$ ) sont affectées aux variables d'entrée  $in$ , tandis que les valeurs des variables de sortie  $out$  sont affectées aux ports de sortie.

- A la fin de cycle

$$(false, n, q, IN, OUT) \xrightarrow{\tau, x > \varepsilon_1, \{x\}} (true, 0, q, IN, OUT)$$

La garde  $x > \varepsilon_1$  permet de modéliser la limite inférieure du temps de cycle. L'horloge  $x$  est remise à zéro afin de pouvoir mesurer les durées des phases d'entrée-sortie et du cycle suivant.

- A chaque instruction  $P(i)$  du programme correspond une transition entre deux états de contrôle de l'automate, par exemple :

$P(i) = \text{LD } a$

$$(false, i, q, IN, OUT) \xrightarrow{\tau, true, \emptyset} (false, i + 1, q[AE := a], IN, OUT)$$

Si  $P(i) = \text{ST } a$

$$(false, i, q, IN, OUT) \xrightarrow{\tau, true, \emptyset} (false, i + 1, q[a := AE], IN, OUT)$$

Si  $P(i) = \text{AND } A$

$$(false, i, q, IN, OUT) \xrightarrow{\tau, true, \emptyset} (false, i + 1, q[AE := AE \wedge A], IN, OUT)$$

Si  $P(i) = \text{JMPC } \text{adr} \text{ et } AE = \text{true}$

$$(false, i, q, IN, OUT) \xrightarrow{\tau, true, \emptyset} (false, \text{adr}, q, IN, OUT)$$

Si  $P(i) = \text{JMPC } \text{adr} \text{ et } AE = \text{false}$

$$(false, i, q, IN, OUT) \xrightarrow{\tau, true, \emptyset} (false, i + 1, q, IN, OUT)$$

etc...

### **b) Prise en compte du temporisateur TON**

Dans ce papier, deux façons de prendre en compte le temporisateur TON ont été proposées.

### Premier modèle

Le premier modèle, qui produit un automate  $T'_p$  consiste à remplacer l'appel *CAL Timer* par l'implémentation en IL du bloc fonctionnel TON. Une implémentation possible de ce bloc fonctionnel en IL est également donnée.

Le fragment du langage IL considéré est ainsi élargi pour supporter les instructions nécessaires à l'implémentation de ce bloc fonctionnel. Ces instructions sont :

- La soustraction entière
- La comparaison de deux entiers

Pour modéliser l'appel système *TIME()* (qui fournit l'heure système) dans cette implémentation, une variable entière *time* ainsi qu'une horloge *tic* sont introduites. Initialisée à zéro, *time* est incrémentée à chaque unité de temps, c'est-à-dire chaque fois que *tic* atteint 1.

### Second modèle

Le second modèle représente chaque instance *Timer* du bloc fonctionnel TON par un automate temporisé  $T_{Timer}$  (figure 20) qui fonctionne en parallèle avec l'automate  $T''_p$ , dont la définition est légèrement différente de celle de l'automate  $T_p$  donné précédemment. Les deux

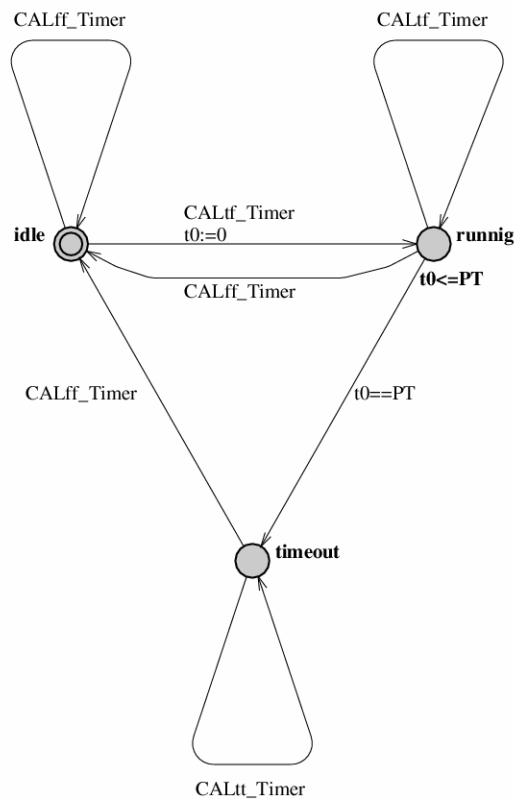


Figure 20

L'automate temporisé  $T_{Timer}$  qui représente une instance du bloc fonctionnel TON

automates sont synchronisés par rapport aux actions appartenant à l'ensemble  $Act := \{CALff\_Timer, CALtf\_Timer, CALtt\_Timer\}$  (la synchronisation considérée ici est une synchronisation binaire. Deux transitions sont synchronisées si elles possèdent la même étiquette d'action).

L'automate  $T_{Timer}$  possède trois états qui indiquent l'état du temporisateur TON. Les actions qui étiquettent ses transitions sont de la forme  $CALxy\_Timer$  où  $x$  désigne la valeur du paramètre  $Timer.IN$  communiqué au bloc fonctionnel lors de l'appel et  $y$  désigne la valeur du paramètre  $TIMER.Q$  mis à jour après l'appel. Deux transitions étiquetées respectivement par  $CALty\_Timer$  et  $CALff\_Timer$  sont issues de chaque état de contrôle du temporisateur. L'horloge  $t0$  permet de mesurer la durée de temporisation. L'automate doit passer de l'état **running** à l'état **timeout** quand  $t0$  atteint  $PT$  ( $PT$ : Durée de temporisation). A partir de chaque état, une transition étiquetée par  $CALff\_Timer$  fait passer l'automate à l'état **idle**. Elle représente l'abandon de la temporisation.

L'automate  $T''_p$  est défini exactement comme  $T_p$  mais avec les extensions suivantes:

- Pour chaque instance  $Timer$  d'un bloc fonctionnel TON, deux variables booléennes sont ajoutées. L'ensemble des variables du programme est donc  $X'' = X \cup \{timer.IN, timer.Q\}$ . Les valuations initiales vérifient  $q_0(timer.IN) = q_0(timer.Q) = false$ .
- L'automate  $T''_p$  supporte de plus les appels à une instance d'un bloc fonctionnel TON. La sémantique opérationnelle définie précédemment est ainsi étendue pour prendre en compte :

- $P(i) = CAL\ Timer(Timer.IN := false, Timer.PT)$

$$(false, i, q, IN, OUT) \xrightarrow{CALff\_Timer, true, \emptyset} (false, i + 1, q[Timer.Q := false], IN, OUT)$$

- $P(i) = CAL\ Timer(Timer.IN := true, Timer.PT)$

$$(false, i, q, IN, OUT) \xrightarrow{CALtf\_Timer, true, \emptyset} (false, i + 1, q[Timer.Q := false], IN, OUT)$$

La synchronisation de l'automate  $T''_p$  avec l'automate  $T_{Timer}$  fait que cette transition ne peut être prise que si l'automate  $T_{Timer}$  se trouve dans l'état **idle** ou dans l'état **running**.

- $P(i) = CAL\ Timer(Timer.IN := true, Timer.PT)$

$$(false, i, q, IN, OUT) \xrightarrow{CALtt\_Timer, true, \emptyset} (false, i + 1, q[Timer.Q := true], IN, OUT)$$

Cette transition ne peut être prise que si l'automate  $T_{Timer}$  se trouve dans l'état **timeout**.

Plus généralement, étant donné un programme P, l'automate temporisé représentant la sémantique opérationnelle de ce programme est définie comme la composition parallèle des automates des automates  $T''_P, T_{Timer-1}, \dots, T_{Timer-m}$  synchronisés par rapport aux actions de l'ensemble  $Act$  :

$T'' = (T''_P \parallel T_{Timer-1} \parallel \dots \parallel T_{Timer-m})$  où  $Timer-1, \dots, Timer-m$  sont des instances du temporisateur TON dans le programme P et  $Act := \{CALff\_timer-i, CALtf\_timer-i, CALtf\_timer-i \text{ avec } 1 \leq i \leq m\}$ .

### c) Codage dans UPPAAL

Dans [Wil99], un codage du modèle de Mader-Wupper dans le langage d'entrée du model-checker UPPAAL est proposé. Cet article décrit en plus un ensemble d'outils permettant de convertir un programme écrit dans un fragment du langage IL en automates temporisés au format UPPAAL. Le fragment considéré correspond aux restrictions suivantes :

- Seulement trois types de base (BOOL, INT et TON) et trois types d'utilisation (INPUT, OUTPUT et NORMAL) sont supportés.
- Les fonctions et les blocs fonctionnels ne sont pas reconnus, excepté le bloc fonctionnel TON.
- Le paramètre PT est fixé une fois pour toute tout au long de la durée de vie d'une instance du bloc fonctionnel TON.
- L'utilisation des constantes *true* et *false* n'est pas permise avec le traducteur.

Cet article part du constat que le plus grand handicap à la vérification de modèles temporisés des programmes de la norme est dû à l'explosion combinatoire du nombre d'états. C'est pour cette raison qu'un ensemble d'outils a été proposé pour permettre de réduire le nombre d'états de contrôle de l'automate temporisé résultant. Cette réduction se base sur la décomposition du modèle en deux parties : une partie temporisée et une partie non temporisée qui est dans la plupart des cas plus volumineuse. L'application des techniques existantes de réduction à la partie non temporisée permet de réduire de façon significative la taille de l'espace d'état. La réduction est réalisée à l'aide de l'outil CAESAR ALDEBARAN DEVELOPPEMENT PACKAGE (CADP).

### 3.2.3 Approche de Zoubek-Roussel- Kwiatkowska

Cette approche [ZRK03] se base sur l'idée qu'il n'est pas nécessaire de traduire l'intégralité du programme pour pouvoir vérifier une propriété. Il suffit simplement d'extraire la partie (ou tranche) du programme concernée par la vérification de la propriété et de traduire

uniquement cette tranche. Elle s'intéresse à la vérification par le model-checker UPPAAL d'un fragment du langage Ladder Diagram.

Le schéma de la figure 21 illustre cette approche :

**a) Algorithme d'abstraction**

Cet algorithme prend en entrée une propriété ainsi que le programme Ladder, il permet de déterminer quelle partie du programme affecte cette propriété. C'est cette partie, ou tranche, qui sera transformée.

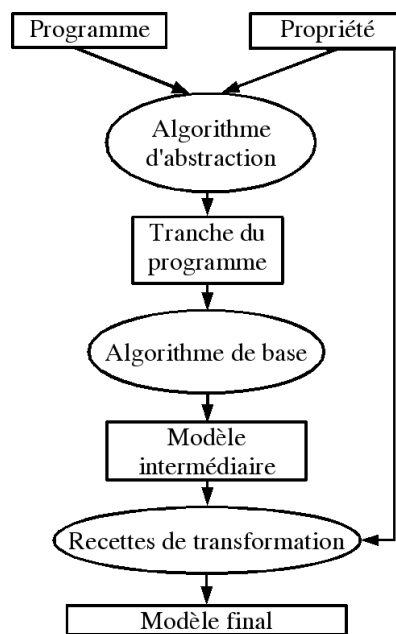


Figure 21  
Les différentes étapes de la démarche

**b) Algorithme de base**

L'algorithme de base transforme la tranche de programme en automates temporisés. La tranche de programme retournée par l'algorithme d'abstraction est ainsi traduite échelon par échelon en modèle UPPAAL. Étant donné la restriction syntaxique sur les programmes Ladder traités (un échelon se compose d'une partie test suivie d'une partie affectation), la traduction est assez intuitive.

La figure 22 donne un exemple d'échelon ainsi que sa traduction en UPPAAL. Cet échelon met la variable c à 1 si a=1 et b=0. Les états de contrôle de l'automate UPPAAL correspondant sont de type *Committed* ce qui signifie que toute exécution passe un temps nul dans l'état. Ainsi, cet échelon est exécuté en temps nul.

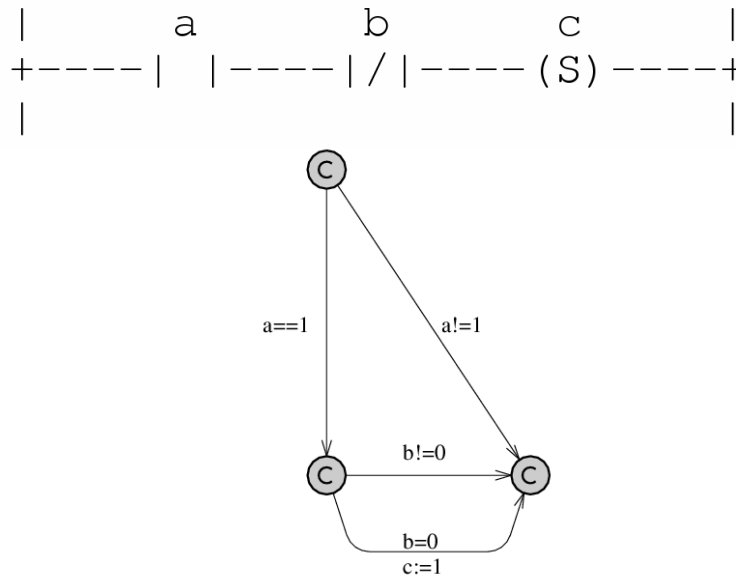


Figure 22  
Échelon suivi par sa traduction en UPPAAL

L'automate interpréteur du programme est décrit par l'automate de la figure 23. Le canal **x** permet d'ordonner l'exécution instantanée du programme Ladder. La synchronisation dans

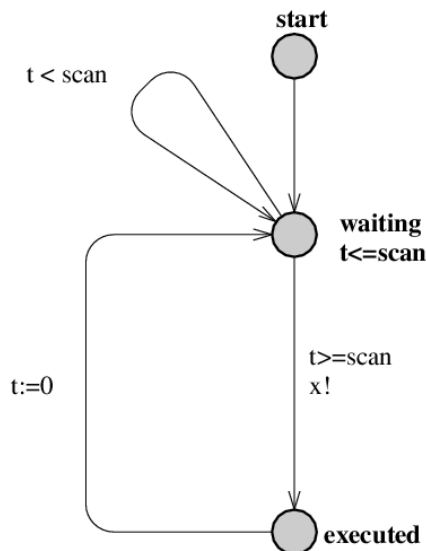


Figure 23  
Automate interpréteur du programme

UPPAAL étant binaire, à l'envoi **x!** dans l'automate interpréteur correspond une réception **x?** dans l'automate du programme.

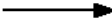



Dans ce modèle, la durée du cycle est constante et égale à *scan*, tandis que l'exécution du programme est instantanée. L'hypothèse de l'exécution instantanée du programme permet de réduire le nombre de variables du programme. En effet, il n'est plus nécessaire de distinguer les variables d'entrée et les variables représentant les ports d'entrée.

### 3.3 Modèles à base de systèmes conditions/événements temporisés

Les systèmes conditions/événements, qui ont été introduits dans [SK91], sont un paradigme de modélisation des systèmes à événements discrets. Leur principal apport consiste à permettre la modélisation de tels systèmes dans une approche « schémas blocs », très couramment utilisée dans les autres branches de l'automatique.

Une présentation détaillée ainsi que les définitions formelles des systèmes condition/événement temporisés, ou Timed condition/event systems (TCES) sont données dans [EHK+97,Huu98].

Dans [TBK00] , la sémantique d'un fragment du langage IL a été définie sous forme de systèmes conditions/événements temporisés. Le modèle proposé (figure 24) prend en compte l'exécution cyclique du programme sur l'API. Il comprend 6 modules (le buffer d'entrée, la mémoire locale, le buffer de sortie, la sortie, le programme et l'interpréteur du programme). Chaque instance d'un bloc fonctionnel de temporisation (TON, TOF ou TP) est représentée par un module supplémentaire. Ces modules communiquent par des signaux qui peuvent être des signaux de condition  ou des signaux d'évènement . Par exemple, le module interpréteur du programme envoie un signal d'évènement forçant le module du programme à exécuter une instruction donnée. Lors de l'exécution d'une instruction, le module du programme a besoin des signaux de condition fournis par les modules buffer d'entrée et mémoire locale pour déterminer son prochain état interne et pour éventuellement envoyer les signaux d'évènement aux modules mémoire locale et buffer de sortie ordonnant le changement de valeur des variables internes ou des variables de sortie.

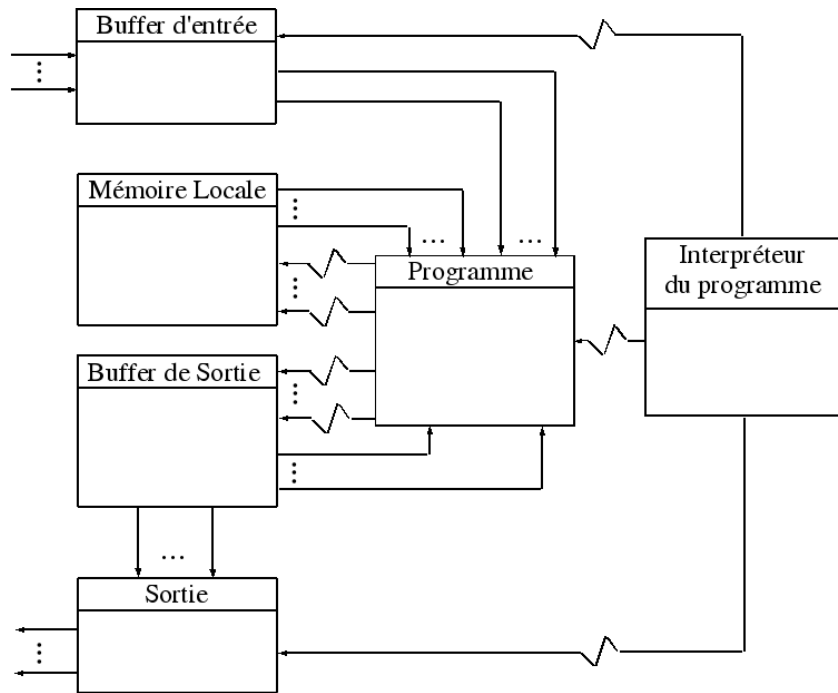


Figure 24  
Organisation des modules et signaux échangés

Notons enfin que cette démarche permet d'analyser un modèle combinant le programme de contrôle-commande et le processus à contrôler. En effet, un module représentant le processus à contrôler peut être simplement ajouté en cascade entre le module de sortie et le module buffer d'entrée.

La traduction des programmes IL en TCES est l'une des fonctionnalités qui ont été implantées dans l'outil VERDICT[KT97], qui permet en plus de convertir les modèles TCES vers du code HYTECH ou SMV.

### 3.4 Modèles à base de PLC-automates

Les PLC-automates sont un formalisme de spécification de programmes de contrôle-commande, qui prend en compte de façon explicite le comportement d'un API (lecture des entrées, affectation des sorties, limite supérieure de la durée d'un cycle) et qui a été proposé par Henning Dierks dans [Die97].

Un tuple  $A = (Q, \Sigma, \delta, q_0, \varepsilon, S_t, S_e, \Omega, \omega)$  est un PLC automate si

- $Q$  est un ensemble non vide d'états
- $\Sigma$  est un ensemble non vide d'entrées
- $\delta: Q \times \Sigma \rightarrow Q$  est une fonction de transition
- $q_0 \in Q$  est l'état initial

- $\varepsilon > 0$  est la limite supérieure du temps de cycle
- $S_t : Q \rightarrow \mathcal{Z}_{\geq 0}$  qui associe à chaque état  $q$  un délai durant lequel les entrées contenues dans  $S_e(q)$  doivent être ignorées.
- $S_e : Q \rightarrow P(\Sigma)$  qui associe à chaque état  $q$  un ensemble d'entrées qui ne causent aucun changement d'état durant les premières  $S_t(q)$  secondes.
- $\Omega$  est un ensemble non vide de sorties
- $\omega : Q \rightarrow \Omega$  est une fonction de sortie.

Exemple tiré de [Die00]

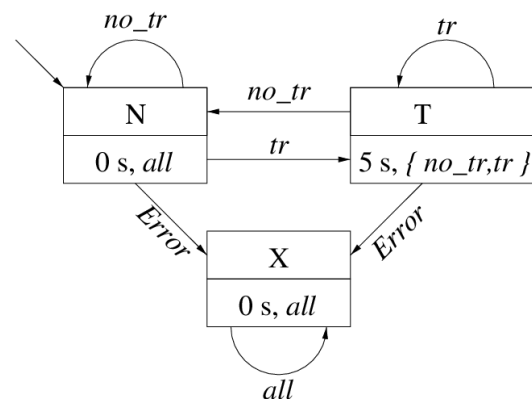


Figure 25  
PLC-automate du filtre

Le PLC-automate de la figure 25 représente un programme permettant de filtrer « l'effet sonnette » engendré par un capteur de trains. Les entrées du PLC-automate sont  $tr$ ,  $no\_tr$  et  $Error$ . Une transition entre deux états étiquetée par  $all$  représente trois transitions entre les mêmes états étiquetées respectivement par  $tr$ ,  $no\_tr$  et  $Error$ . Les sorties sont N, T et X. Initialement dans l'état 'N', le PLC-automate peut réagir immédiatement à l'entrée  $tr$  et passer dans l'état 'T'. Dans l'état 'T', le PLC-automate doit attendre 5 secondes avant de pouvoir réagir aux entrées  $tr$  et  $no\_tr$ . Cependant, il peut réagir immédiatement à l'entrée  $Error$ . C'est la capacité du PLC-automate à ignorer les entrées ( $tr$  et  $no\_tr$ ) durant 5 secondes qui lui permet de filtrer le papillotement du capteur.

La sémantique formelle des PLC-automates a été définie dans la logique Duration Calculus ainsi qu'en termes d'automates temporisés.

Ces résultats ont donné naissance à l'outil MOBY/PLC [TD98], qui offre les fonctionnalités suivantes :

- Un éditeur graphique permettant la saisie du système à vérifier sous forme de PLC-automate.
- Un simulateur qui permet de visualiser des exécutions du PLC-automate.
- Des algorithmes d'analyse basés sur la sémantique Duration-Calculus des PLC-automate permettent de calculer certaines propriétés de l'automate, par exemple, le temps de réaction à une combinaison donnée d'entrées.
- Des compilateurs se basant sur la sémantique en automates temporisés des PLC-automates permettent la génération automatique d'automates temporisés au format d'UPPAAL ou de KRONOS.
- La spécification peut être automatiquement traduite vers du code ST.

### 3.5 Conclusion

Le modèle de Mader-Wupper propose une approche intéressante pour la modélisation des applications sur un API : il reproduit le fonctionnement de la machine, en faisant toutefois abstraction de la durée d'exécution des instructions. Il a été affiné par la suite dans [Wil99] et [ZBK03]. Ces deux derniers articles se sont focalisés particulièrement sur le problème de l'explosion combinatoire due à la prise en compte des aspects temporisés, et ont proposé des méthodes visant à réduire la complexité des modèles.

L'intérêt de l'approche basée sur les systèmes conditions/événements temporisés réside dans sa capacité à intégrer très facilement des modèles des processus à contrôler, très souvent modélisés sous forme de systèmes conditions/événements temporisés (surtout dans le domaine du contrôle des processus chimiques). L'intégration des modèles des processus à contrôler est simplement réalisée par leur mise en cascade avec les modèles représentant les algorithmes de contrôle.

L'objectif principal des PLC-automates était d'offrir un langage de spécification d'applications de contrôle-commande s'exécutant sur des API et assez intuitif pour les programmeurs d'API, qui ne sont pas des informaticiens. Cependant, la vérification de spécifications simples à base de PLC-automates se heurte aux problèmes d'explosion combinatoire. Dans [Die98], l'étude de la vérification d'un exemple simple spécifié sous

forme de PLC-automate à l'aide des outils UPPAAL et KRONOS montre clairement ces limites.

## Chapitre 4

# Vérification de modèles temporisés par la technique du model-checking

### 4.1 Le model-checking

Le model-checking est une technique de vérification formelle permettant d'établir si un modèle vérifie une propriété en explorant exhaustivement tous les comportements décrits par le modèle. Un algorithme de model-checking reçoit en entrée un modèle  $M$  (une structure de Kripke, un automate temporel...), et une formule de logique temporelle  $\varphi$  et répond vrai si  $M$  satisfait  $\varphi$  et faux sinon.

La logique temporelle [SBB+99] est un langage formel permettant de décrire des énoncés et des raisonnements faisant intervenir la notion d'ordonnancement dans le temps. Les logiques temporelles permettent d'exprimer les propriétés que l'on souhaite vérifier à l'aide de

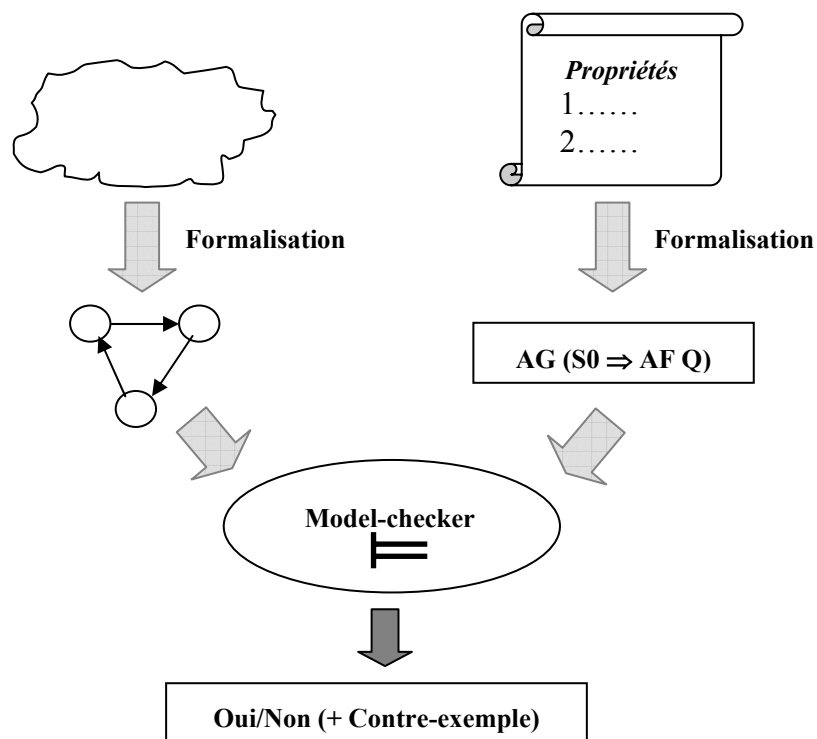


Figure 26  
Le model-checking dans la pratique

formules utilisant des propositions atomiques. Leur intérêt vient du fait qu'elles fournissent

des opérateurs calqués sur des constructions linguistiques comme *toujours*, *jusqu'à* ou *inévitavelmente* permettant une formalisation des raisonnements proche du langage naturel. Différentes logiques temporelles ont été proposées parmi lesquelles on peut citer CTL (*Computation Tree Logic*).

## 4.2 Les cas d'étude

Dans la suite, on se basera dans notre étude sur deux exemples de systèmes automatisés, contrôlés par des API et qui doivent vérifier un certain nombre de propriétés. Le premier cas d'étude est un mélangeur, le second est un évaporateur.

### 4.2.1 Le mélangeur

Le premier exemple a été donné dans [RS00]. Il s'agit d'une partie du programme MIX\_2\_BRIX fourni dans les annexes de [IEC93].

#### a) Description

Il s'agit d'un mélangeur déchargeur (figure 27) qui mélange dans un bol les produits qui lui sont fournis par l'étape précédente du processus de fabrication puis déverse le contenu du bol pour être traité par l'étape suivante.

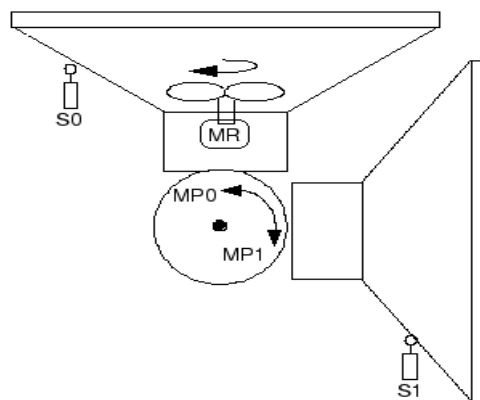


Figure 27  
Le mélangeur en position droite (capteur S0 actionné) et  
en position inclinée (capteur S1 actionné)

Le mélangeur est commandé par un API, qui reçoit en entrée les signaux binaires ST (Bouton Start), STOP (Bouton Stop), S0 (Capteur S0) et S1 (Capteur S1). L'API élabore les signaux (binaires) de sortie MR (mélanger), MP0 (déverser) et MP1 (revenir en position de repos).

Initialement, le mélangeur est en position de repos. Le capteur S0 est à 1, le capteur S1 est à 0 et tous les moteurs sont au repos ( $MR=MP0=MP1=0$ ). L'action sur le bouton Start (ST passe de 0 à 1) met en marche le moteur du mélangeur (MR passe de 0 à 1). Après

l'écoulement d'une durée T, le bol entame une rotation permettant de déverser son contenu (MP1 passe de 0 à 1). Quand la position S1 est atteinte (S1 passe de 0 à 1), les moteurs MR et MP1 sont arrêtés (MR et MP1 passent de 1 à 0) et le bol se met en rotation dans le sens inverse (MP0 passe de 0 à 1). Quand la position MP0 est atteinte (c'est-à-dire quand S0 passe de 0 à 1), le moteur MP0 est arrêté (MP0 passe de 1 à 0).

Enfin, le front montant du signal STOP a pour effet d'arrêter tous les pré-actionneurs (MR, MP0 et MP1 sont mis à 0). Quand le signal STOP passe à 0, le système regagne son état initial.

### b) Programme

Le système est commandé par le programme IL suivant :

<pre> PROGRAM MIX_2_BRIX VAR_INPUT     ST : BOOL;     STOP : BOOL;     S0 : BOOL;     S1 : BOOL; END_VAR  VAR     ST_Previous :     BOOL;     STOP_Previous :     BOOL;     ESTOP : BOOL;     INITOK : BOOL;     MST : BOOL;     TIMER : TON ; END_VAR  VAR_OUTPUT     MR : BOOL;     MP0 : BOOL;     MP1 : BOOL; END_VAR </pre>	<pre>         JMPC <b>INIT</b>         LDN ST_Previous         AND ST         ANDN MP1         ANDN MP0         JMPC <b>MRSet</b>         JMP <b>Next1</b> <b>MRSet</b> LD 1         ST MR <b>Next1</b> LDN ST_Previous         AND ST         OR MST         ANDN MP1         ANDN MP0         ANDN S1         ST MST         LD 100         ST TIMER.PT         LD MST         ST TIMER.IN         CAL TIMER         LD TIMER.Q         JMPC <b>MP1Set</b>         JMP <b>Next2</b> <b>MP1Set</b> LD 1         ST MP1 <b>Next2</b> LD S1         AND MP1         JMPC <b>Mod1</b>         JMP <b>Next3</b> <b>Mod1</b> LD 0 </pre>	<pre>         ST MP1         ST LD 1         ST MP0         LD 0         ST MR         LD S0         JMPC <b>MP0Reset</b>         JMP <b>Next4</b> <b>MP0Reset</b> LD 0         ST MP0 <b>Next4</b> JMP <b>Begin</b> <b>ESTOP</b> LD 0         ST MR         ST MP0         ST MP1         ST INITOK         ST MST <b>INIT</b> LDN S0         JMPC <b>Mod2</b>         JMP <b>Next5</b> <b>Mod2</b> LD 1         ST MP0 <b>Next5</b> LD S0         JMPC <b>Mod3</b>         JMP <b>Begin</b> <b>Mod3</b> LD 1         ST INITOK         LD 0         ST MP0 END_PROGRAM </pre>
--	--	---

## 4.2.2 L'évaporateur

Le second exemple considéré est une version modifiée de l'exemple donné dans [Kow98] et [Huu99]. Il s'agit d'un évaporateur visant à obtenir une solution ayant une concentration déterminée.



### a) Description de l'installation

L'installation considérée se compose de deux réservoirs (réservoir 1 et réservoir 2) et d'un condenseur reliés par des conduites dont certaines sont munies de valves unidirectionnelles (figure 28).

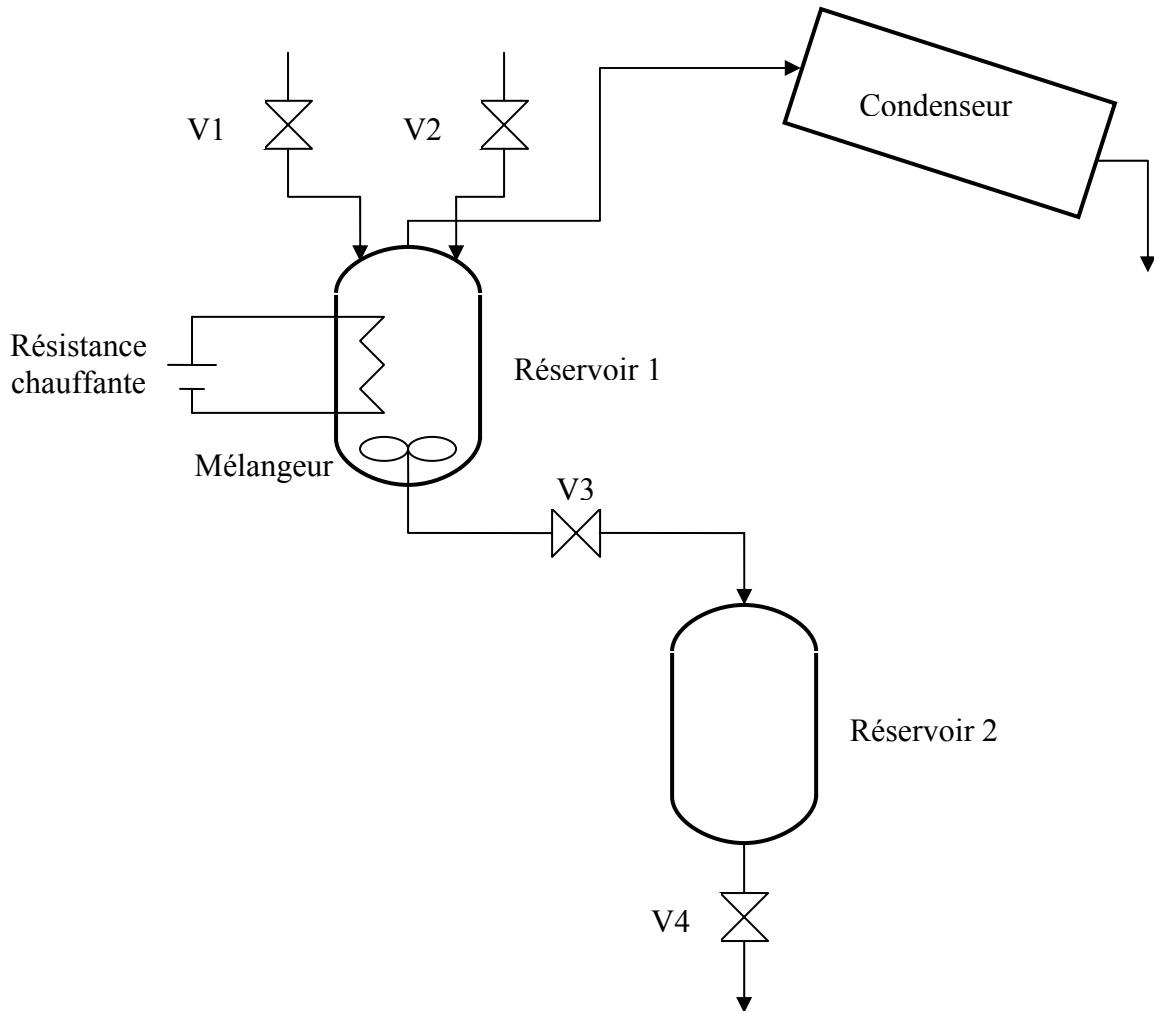


Figure 28  
Schéma de l'installation

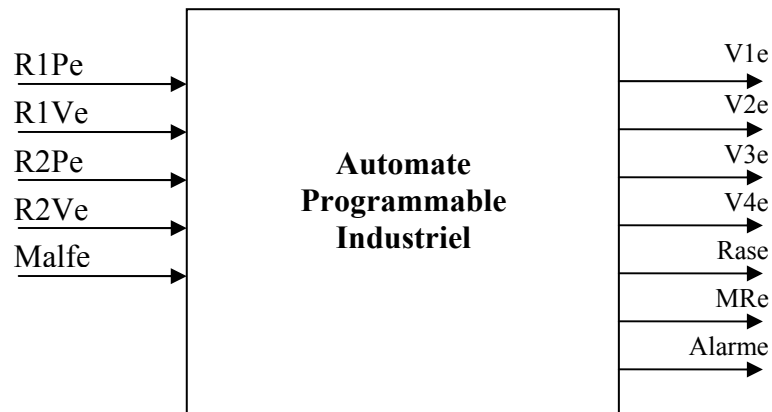
Les valves V1 et V2 assurent le remplissage du réservoir 1. La valve V3 permet de transférer le contenu du réservoir 1 dans le réservoir 2 tandis que la valve V4 permet de vider le réservoir 2. Le réservoir 1 est muni d'une résistance chauffante et d'un mélangeur. La résistance chauffante est utilisée afin de chauffer la solution présente dans le réservoir 1 jusqu'à l'ébullition. Le mélangeur homogénéise les produits versés dans le réservoir 1 par l'intermédiaire des valves V1 et V2.

Le condenseur permet de condenser la vapeur dégagée lors de l'ébullition de la solution présente dans le réservoir 1. L'eau ainsi produite est évacuée du condenseur.

Les réservoirs 1 et 2 sont munis de capteurs qui assurent la détection du niveau haut (réservoir plein) et du niveau bas (réservoir vide).

**b) Les entrées/sorties de l'API**

La figure 29 illustre les entrées-sorties de l'API. Le suffixe *e* (comme environnement) est ajouté à tous ces signaux pour indiquer qu'ils sont relatifs aux ports de l'API (signaux d'entrée reçus à partir des capteurs et signaux de sortie appliqués aux pré-actionneurs).



*Figure 29  
Les entrées-sorties de l'API qui contrôle  
cette installation*

L'évaporateur est contrôlé par un automate programmable industriel, qui reçoit en entrée les signaux binaires :

<b><i>Signal</i></b>	<b><i>Signification</i></b>
R1Pe	Le réservoir 1 est plein
R1Ve	Le réservoir 1 est vide
R2Pe	Le réservoir 2 est plein
R2Ve	Le réservoir 2 est vide
Malfe	Un dysfonctionnement dans le condenseur s'est produit

Les signaux appliqués en sortie de l'API sont les suivants :

<b><i>Signal</i></b>	<b><i>Signification</i></b>
V1e	Ouvrir la valve 1

V2e	Ouvrir la valve 2
V3e	Ouvrir la valve 3
V4e	Ouvrir la valve 4
Rese	Mettre en marche la résistance chauffante
MRe	Mettre en marche le mélangeur
Alarme	Actionner l'alarme

***c) Fonctionnement normal***

Le réservoir 1 est d'abord rempli par l'intermédiaire des valves V1 et V2 par deux solutions de faible concentration. Quand le niveau haut du réservoir 1 est atteint, alors les valves V1 et V2 sont fermées et le mélangeur MR est actionné. Après 2 unités de temps, la résistance chauffante est mise en marche. Au bout d'un temps égal à 20 unités, la concentration désirée est atteinte. La résistance chauffante est arrêtée et la valve V3 est ouverte afin de transférer le contenu du réservoir 1 dans le réservoir 2. Quand le réservoir 1 est vidé, le moteur MR est arrêté. Le mélange subit un traitement dans le réservoir 2. Ce traitement dure 32 unités. Au bout de cette période, la valve V4 est ouverte afin de transférer le contenu du réservoir 2 à l'étape postérieure du processus de fabrication.

***d) Traitement d'un dysfonctionnement du condenseur***

Durant l'évaporation de la solution dans le réservoir 1, un dysfonctionnement du condenseur peut survenir. La vapeur ne pouvant plus être refroidie, la pression à l'intérieur du condenseur commence à augmenter. Il est ainsi nécessaire d'arrêter toute production de vapeur afin de ne pas dépasser une certaine pression limite qui causerait une explosion du condenseur.

D'autre part, l'arrêt de la production de vapeur engendre un autre problème : si la température dans le réservoir 1 descend en dessous d'une certaine valeur (suite à l'arrêt de la production de vapeur), alors la solution présente dans le réservoir 1 se solidifie et ne peut plus s'écouler dans le réservoir 2. Il est donc nécessaire d'ouvrir la valve 2 suffisamment tôt afin d'écouler la solution du réservoir 1 au réservoir 2 où le traitement subi l'empêche de se solidifier. Cependant, avant d'ouvrir la valve V3, il faut s'assurer que le réservoir 2 est vide. Sinon, le réservoir 2 risque de déborder. La valve V4 doit donc être ouverte en premier pour vider ce réservoir.

**e) Propriétés à vérifier**

Afin de garantir la sécurité de l'installation, le programme de contrôle-commande doit respecter les propriétés suivantes :

**(P1)** Afin d'éviter l'endommagement du réservoir 1 par la chaleur dégagée par la résistance chauffante, la propriété suivante doit être vérifiée :

⇒ Chaque fois que la résistance chauffante est en marche, le réservoir 1 doit être plein.

**(P2)** Durant l'étape de vaporisation, la vapeur doit uniquement se dégager vers le condenseur :

⇒ Chaque fois que la résistance chauffante est en marche, les valves V1, V2 et V3 doivent être fermées.

**(P3)** Afin d'éviter d'avoir un flux de liquides incontrôlé, les valves d'entrée et de sortie d'un réservoir ne doivent jamais être ouvertes simultanément

Dans le cas d'un dysfonctionnement du condenseur, des propriétés faisant intervenir des grandeurs quantitatives sur les temps de réponse du programme doivent être satisfaites. Ces propriétés sont déduites à partir des caractéristiques physiques suivantes de l'installation :

- Dans le cas où un dysfonctionnement du condenseur se produit, alors le condenseur risque d'exploser si la production de vapeur se poursuit pendant une durée supérieure à 22 unités de temps depuis l'occurrence du dysfonctionnement.
- Si la résistance chauffante est mise à l'arrêt, alors la production de vapeur cesse après exactement 12 unités de temps.
- Si la production de vapeur s'arrête dans le réservoir 1, alors la solution risque de devenir solide après 19 unités de temps.
- Vider le réservoir 2 prend entre 0 et 26 unités.
- Le vidage du réservoir 1 est très rapide devant les durées mises en jeu, il est ainsi supposé instantané.
- Remplir le réservoir 1 prend au plus 6 unités de temps.

Ainsi, le programme de contrôle-commande doit éviter que :

**(P4)** Le condenseur explose

**(P5)** La solution se solidifie dans le réservoir 1

**(P6)** Le réservoir 2 déborde. En effet, si les réservoirs 1 et 2 sont remplis et la valve V3 ouverte, alors le réservoir 2 va déborder.

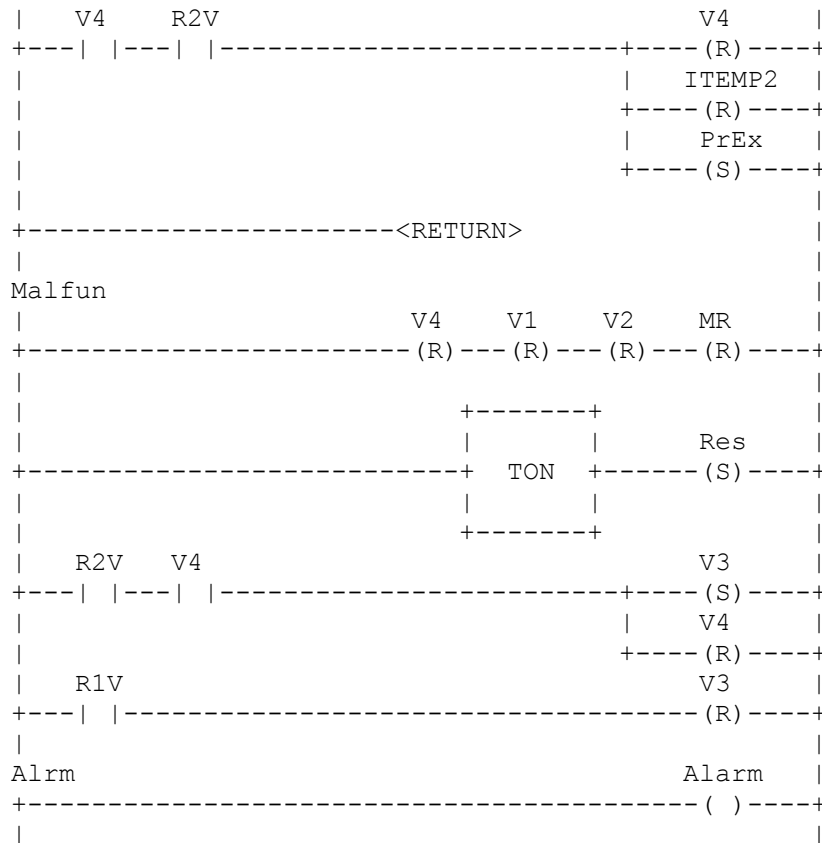
### f) Programme

Pour piloter le système, nous proposons le programme suivant :

```

|  V1  V2  V3  V4  R1V  R2V  INIT_OK  |
+---+---+---+---+---+---+---+---( )---+
|  PrEx INIT_OK   Alrm              |
+---+---+---+--->>                  |
|  malf              Malfun          |
+---+---+--->>                      |
|  malf PrEx INIT_OK                V1  |
+---+---+---+---+---(S)---+         |
|                                     | V2  |
|                                     +---(S)---+
|                                     | PrEx |
|                                     +---(R)---+
|  V1  V2  R1P                V1      |
+---+---+---+---+---(R)---+         |
|                                     | V2  |
|                                     +---(R)---+
|                                     | ITEMP1 |
|                                     +---(S)---+
|                                     | MR      |
|                                     +---(S)---+
|
|                                     +-----+
|  ITEMP1                            | Res  |
+---+---+---+---+---(S)---+         |
|                                     | TON   |
|                                     +-----+
|
|                                     +-----+
|  ITEMP1                            | Res  |
+---+---+---+---+---(R)---+         |
|                                     | TON   |
|                                     +---(R)---+
|                                     | ITEMP1 |
|                                     +---(R)---+
|                                     | FinTR1 |
|                                     +---(S)---+
|  FinTR1                          V3    |
+---+---+---+---+---(S)---+         |
|                                     | MR      |
|                                     +---(R)---+
|  FinTR1 V3  R1V                FinTR1 |
+---+---+---+---+---(R)---+         |
|                                     | V3    |
|                                     +---(R)---+
|                                     | ITEMP2 |
|                                     +---(S)---+
|
|                                     +-----+
|  ITEMP2                            | V4   |
+---+---+---+---+---(S)---+         |
|                                     | TON   |
|                                     +-----+
|
|

```



*Remarque* : La variable interne *PrEx* doit être initialisée à 1. Les autres variables internes à zéro.

### 4.3 Modélisation du comportement cyclique et des contraintes temporisées

Il existe plusieurs manières de modéliser le comportement cyclique de l'automate programmable et les contraintes temporisées telles que la durée du cycle. Bien que la vérification de certaines propriétés de sûreté peut se passer de toute prise en compte de la durée du cycle dans le modèle, d'autres propriétés temps-réel nécessitent la prise en compte de cette durée.

On présentera deux manières de modéliser ces aspects : le modèle de Mader-Wupper et le modèle basé sur l'exécution instantanée du programme. Ces deux modèles seront étudiés expérimentalement et comparés dans le paragraphe 4.4.4. Cette comparaison nous permettra de déterminer les aspects permettant d'améliorer sensiblement la complexité de la vérification.

### 4.3.1 Modèle de Mader-Wupper

Dans ce modèle (figure 30), le programme est considéré dans sa globalité : on ne regarde pas le temps d'exécution d'une instruction élémentaire mais plutôt le temps du cycle automate. Ce temps est borné par deux constantes :  $\varepsilon_1$  et  $\varepsilon_2$ .

Chaque instruction peut ainsi s'exécuter n'importe quand dans le cycle, pourvu que l'instant de son exécution soit compris entre  $\varepsilon_1$  et  $\varepsilon_2$ .

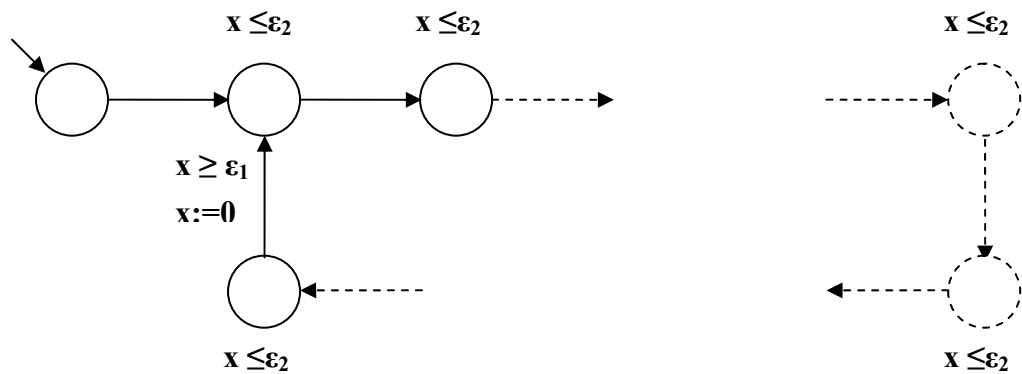


Figure 30  
Comportement cyclique et prise en compte des bornes inférieure et supérieure de la durée du cycle dans le modèle de Mader-Wupper

La vérification de certaines propriétés de sûreté peut se passer de toute cette « lourdeur » du modèle : utiliser ce modèle sans aucune contrainte temporisée pourrait être suffisant (figure 31).

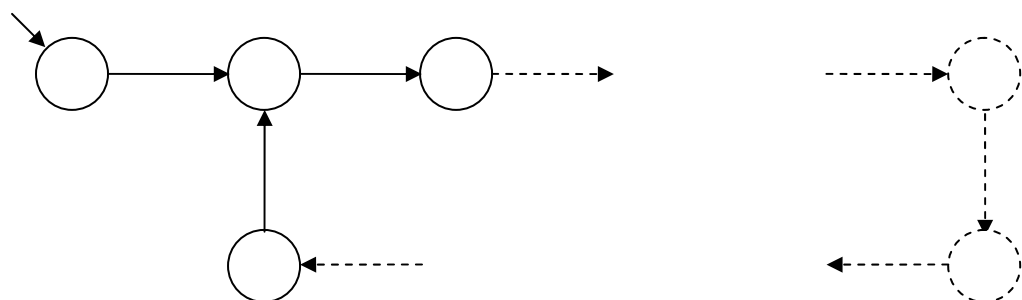


Figure 31  
Modèle faisant abstraction de la durée du cycle

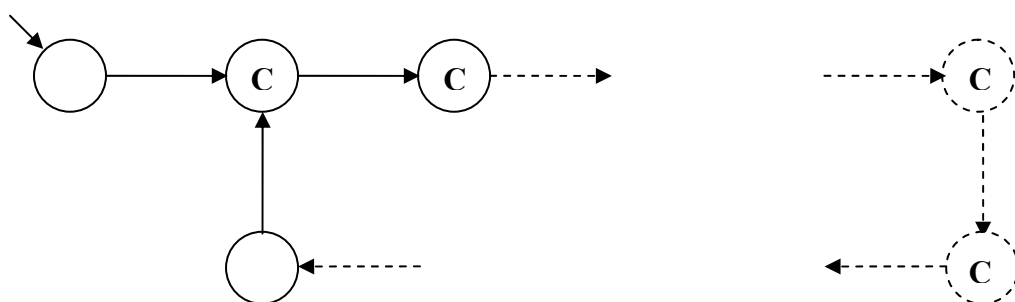
### 4.3.2 Modèle basé sur l'exécution instantanée et atomique du programme

Dans ce modèle, l'acquisition des entrées et l'exécution du programme sont instantanées et atomiques. L'état juste après l'exécution du programme permet de représenter le temps pris par le cycle automate.

La présence d'au moins un état où le temps peut s'écouler est nécessaire pour n'importe quel modèle d'API. En effet, sans un tel état, le programme s'exécuterait une infinité de fois sans progression du temps : le modèle serait alors zénon.

Dans ce modèle, la réaction du programme aux entrées est supposée instantanée (l'affectation des sorties se fait au même instant que l'acquisition des entrées!). De même, les différents délais qui pourraient séparer l'enclenchement ou le déclenchement des différents temporisateurs lors de la phase d'exécution du programme sont négligés. Ces différentes abstractions, aboutissant à des simplifications considérables du modèle, sont entièrement justifiées pour une certaine classe de programmes, d'API et de propriétés. En effet, si les durées des temporisateurs manipulés par le programme ainsi que les contraintes quantitatives spécifiées dans les propriétés sont beaucoup plus grandes que le temps de cycle, le temps de cycle peut apparaître à l'échelle du temps des propriétés et des durées de temporisation assez petit pour être vu comme nul. Le temps de réponse de l'API est vu (de cet angle) comme nul. Le fait de supposer qu'à notre échelle le temps d'un cycle est nul nous permettra simplement de superposer les divers événements qui se produisent dans le cycle (acquisition, déclenchement de temporisateurs...). En réalité, le modèle considère que chaque cycle prend une certaine durée.

Ce modèle (figure 32) convient pour la vérification des propriétés de sûreté.



*Figure 32*  
*Modèle basé sur l'exécution atomique et instantanée*  
*du programme et faisant abstraction de la valeur de*  
*la durée du cycle*

La vérification de propriétés temps-réel nécessite de prendre en compte explicitement la borne supérieure du temps de cycle. En effet, certaines propriétés exigent que le temps de



réponse de la partie commande (l'API) à des sollicitations de l'environnement contrôlé soit inférieur à un seuil fixé. De telles propriétés ne pourraient pas être vérifiées si on autorise le cycle automate à prendre une durée arbitraire (le modèle autoriserait des comportements où l'API réagit après le seuil). Il est ainsi nécessaire, pour pouvoir vérifier ces propriétés, d'utiliser un modèle où le temps de cycle prend au plus une durée  $\epsilon_2$ . La prise en compte de cette contrainte par le modèle s'effectue en ajoutant simplement une horloge  $x$  et un invariant  $x \leq \epsilon_2$  à l'état matérialisant la progression du temps.

La vérification de certaines propriétés peut nécessiter la prise en compte explicite de la borne inférieure du temps de cycle  $\epsilon_1$ . Cette contrainte est ajoutée au modèle en ajoutant une garde  $x \geq \epsilon_1$  à la transition issue de l'état modélisant la progression du temps (figure 33).

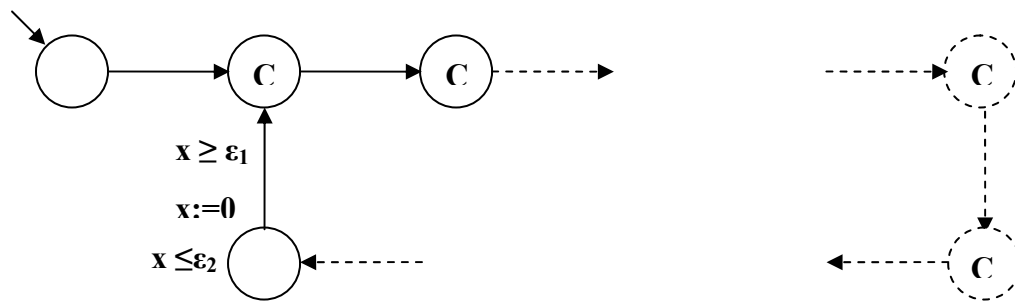


Figure 33  
Modèle basé sur l'exécution atomique et instantanée  
du programme et prenant en compte les bornes  
inférieure et supérieure de la durée du cycle

#### 4.4 Étude expérimentale de l'exemple du mélangeur

L'exemple du mélangeur possède deux caractéristiques principales :

- Il contient un unique temporisateur de type TON. On essaiera ainsi dans la suite de voir l'influence des valeurs des temporisateurs et des bornes inférieure et supérieure de la durée du cycle sur la complexité de la vérification.
- Aucun modèle du processus contrôlé n'est utilisé. On suppose ainsi que chaque variable d'entrée peut prendre n'importe quelle valeur à n'importe quel instant.

##### 4.4.1 Codage du programme dans l'outil UPPAAL suivant le modèle de Mader-Wupper

Pour effectuer le codage du programme Mix\_2\_Brix dans UPPAAL, nous nous sommes basés sur la méthode décrite dans [Wil99]. Le résultat de la traduction est un automate temporisé UPPAAL résultat de la mise en parallèle de trois automates temporisés :

### a) Représentation de l'environnement

La représentation de l'environnement (figure 34) est un automate ayant un unique état et autorisant tous les changements de valeurs possibles dans les ports d'entrée.

Pour toute variable d'entrée du programme, une variable d'environnement est créée. Si  $ST$  est une variable d'entrée, alors la variable d'environnement associée sera désignée par  $ST\_env$ . Les variables d'environnement sont codées dans UPPAAL sous forme de variables partagées. Elles sont modifiées par la représentation de l'environnement et lues par la représentation du programme.

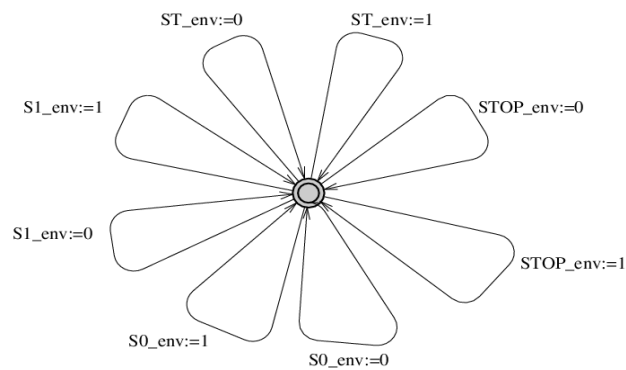


Figure 34  
Représentation de l'environnement

### b) Représentation du programme

- Chaque variable du programme est traduite en une variable partagée d'UPPAAL.
- Chaque instruction IL du programme est traduite vers une ou plusieurs transitions entre deux états : l'état précédent et l'état juste après l'exécution de l'instruction. Par exemple, l'appel CAL TIMER est modélisé par une transition entre deux états comme indiqué à la figure 35.
- Une transition entre l'état initial et l'état représentant le début du cycle est créée. Elle modélise la mise sous tension de l'automate programmable.
- Une horloge **PLCsysteme** est déclarée, cette horloge permet de mesurer le temps de cycle.
- Un invariant **PLCsysteme**  $\leq \epsilon_2$  est ajouté à tous les états de l'automate, sauf à l'état initial. Cet invariant modélise la limite supérieure du temps d'exécution d'un programme.

- La phase d'acquisition des entrées est représentée par une transition entre l'état représentant le début du cycle et l'état juste avant l'exécution de la première instruction. Cette transition permet d'affecter les valeurs des variables d'environnement aux variables d'entrée du programme.

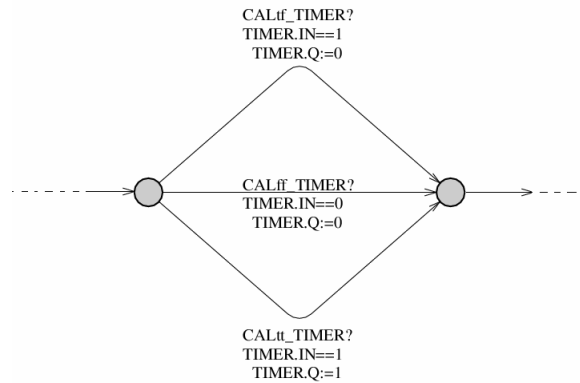


Figure 35  
Traduction dans UPPAAL de l'instruction  
CAL TIMER

- La phase d'affectation des sorties est représentée par une transition entre l'état juste après l'exécution de la dernière instruction et l'état représentant le début du cycle. Cette transition permet la remise à zéro de l'horloge **PLCsysteme**. Elle peut également contenir une garde qui peut être par exemple **PLCsysteme  $\geq \epsilon 1$**  (elle modélise la limite inférieure du temps d'exécution du programme).

c) *Représentation d'une instance du bloc fonctionnel TON*

Elle est décrite par l'automate temporisé de la figure 36. Elle se synchronise avec la

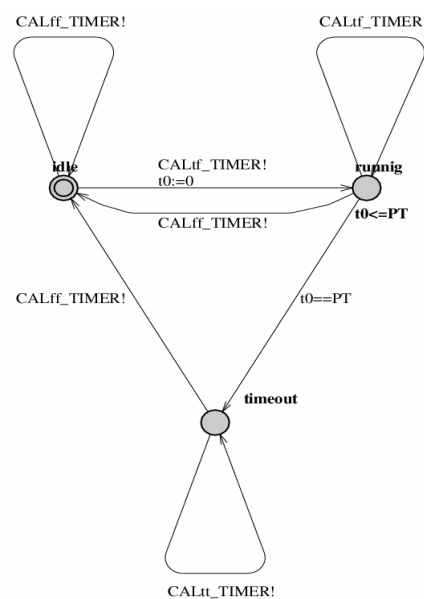


Figure 36  
Représentation d'une instance du bloc  
fonctionnel TON

représentation du programme grâce au mécanisme de synchronisation binaire d'UPPAAL. Le mécanisme de synchronisation est identique à celui décrit au paragraphe 3.2.1.b) à un détail près : la synchronisation est effectuée entre deux transitions portant des étiquettes d'action conjuguées (par exemple  $a!$  et  $a?$ ).

#### 4.4.2 Codage dans l'outil UPPAAL suivant le modèle basé sur l'exécution atomique du programme

Il est similaire au codage décrit précédemment. Le « squelette » du modèle est donné par la figure 33. Le seul changement concerne l'automate représentant le programme. Au lieu d'appliquer l'invariant  $\text{PLCsystem} \leq \varepsilon_2$  à tous les états de contrôle, cet invariant est uniquement appliqué au dernier état du cycle. Les autres états, excepté le dernier état du cycle et l'état modélisant l'initialisation sont de type committed.

#### 4.4.3 Étude de l'influence des valeurs des paramètres $\varepsilon_1$ , $\varepsilon_2$ et $PT$ sur la complexité de la vérification du programme codé suivant modèle de Mader-Wupper

##### a) Notion de zone

Une zone est un ensemble de valuations définies par une contrainte sur les horloges de la forme :  $\varphi ::= x \sim c \mid x - y \sim c \mid \varphi \wedge \varphi$  avec  $\sim \in \{<, >, =, \leq, \geq\}$ . La figure 37 illustre un exemple de zone.

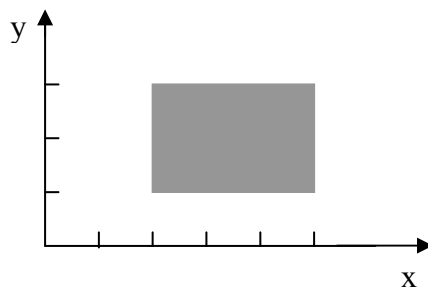


Figure 37

Zone définie par les contraintes :  $x \leq 5 \wedge x \geq 2 \wedge y \leq 3 \wedge y \geq 1$ .

Les zones sont à la base de nombreux algorithmes d'analyse d'accessibilité pour les automates temporisés. Ces algorithmes réalisent trois opérations fondamentales : l'intersection de deux zones, la remise à zéro d'horloges et l'écoulement du temps. Les résultats de l'application de ces opérations sur des zones donne toujours une zone.

##### b) Étude expérimentale de la complexité de la vérification

On se propose d'évaluer expérimentalement la complexité de la vérification d'une propriété de sûreté sur ce modèle en fonction des valeurs des paramètres  $\varepsilon_1$ ,  $\varepsilon_2$  et  $PT$ . La propriété à vérifier concerne la cohérence de la commande du moteur : le moteur ne doit jamais être actionné dans les deux sens à la fois. Autrement dit,  $MP_0$  et  $MP_1$  ne doivent pas

être en même temps égales à 1. Cette propriété s'exprime par la formule CTL : **AG not (MP0 and MP1)**.

Pour réaliser ces mesures, l'utilitaire **memtime** a été utilisé (**memtime** est disponible à l'adresse <http://www.update.uu.se/~johanb/memtime>). Cet utilitaire a permis de mesurer le temps d'exécution de la commande **verifyta** (commande permettant de lancer la vérification sous UPPAAL d'un ensemble de propriétés pour le modèle) et la taille de l'espace mémoire utilisé par le processus créé en termes de mémoire virtuelle (*VSize* : Virtual Size) et résidente (*RSS* : Resident Set Size). Les mesures ont été réalisées sur un PENTIUM 3 à 1 GHz avec une RAM de 514 Mo. La version d'UPPAAL utilisée est celle d'Avril 2003. La propriété a été vérifiée par le modèle pour toutes les mesures. A partir de différentes mesures réalisées, nous avons obtenu les résultats suivants :

- La multiplication par une même constante des paramètres  $\varepsilon_1$ ,  $\varepsilon_2$  et PT n'a aucune influence sur le temps de calcul et sur l'espace mémoire.
- Dans le cas où  $\varepsilon_1 \neq 0$ , le temps de calcul et l'espace mémoire nécessaires ne dépendent que du rapport  $PT/\varepsilon_2$ , plus ce rapport est grand, plus le temps et l'espace mémoire requis sont importants. Les mesures sont récapitulées dans le tableau ci-dessous :

$PT/\varepsilon_2$	<i>Temps (s)</i>	<i>Max VSize (Mo)</i>	<i>Max RSS (Mo)</i>
1	29	41	35
4	45	51	46
5	52	57	53
10	83	89	85
25	188	186	180

- Dans le cas où  $\varepsilon_1 = 0$ , l'espace mémoire requis est indépendant des paramètres  $\varepsilon_2$  et PT. Cependant, le temps de calcul augmente lorsque le rapport  $PT/\varepsilon_2$  croît. Les mesures sont données dans le tableau ci-dessous :

$PT/\varepsilon_2$	<i>Temps (s)</i>	<i>Max VSize (Mo)</i>	<i>Max RSS (Mo)</i>
4	44	41	35
5	50	41	35

10	79	41	35
25	165	41	35
500	2918	41	35

### c) *Interprétation des résultats*

L'explosion de la taille mémoire rencontrée dans le cas où  $\varepsilon_l \neq 0$  est due à l'espace mémoire occupé par les états construits lors de l'exploration du chemin (du système de transitions temporisé mixte) correspondant à l'exécution séquentielle du programme lorsque l'automate du temporisateur est dans l'état **Running**.

La figure 38 illustre les 5 premières zones construites par l'algorithme d'analyse

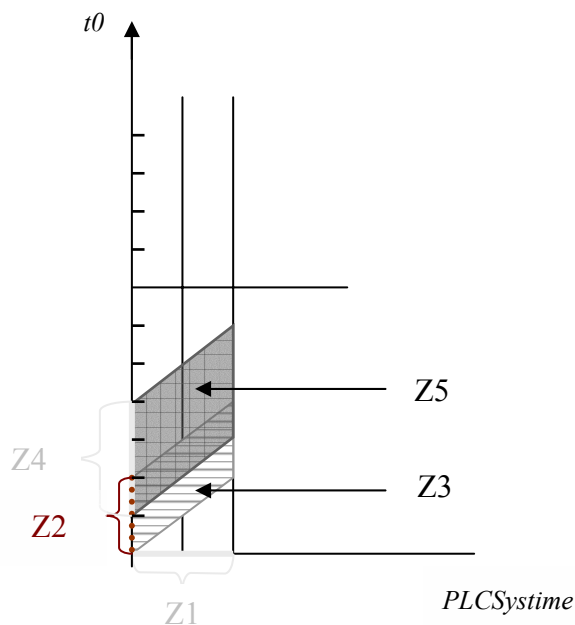


Figure 38  
Les 5 premières zones construites par l'algorithme d'UPPAAL dans le cas où  $\varepsilon_l \neq 0$

d'accessibilité lors du parcours de ce chemin (dans ce cas de figure,  $\varepsilon_l = 1$ ,  $\varepsilon_2 = 2$  et  $PT = 7$ ). Comme  $Z_3 \not\subset Z_5 \not\subset \dots$ , l'algorithme d'analyse d'accessibilité sauvegardera les états correspondant aux zones  $Z_1, Z_3, Z_5 \dots$  dans une structure de donnée particulière, nommée la **passed list**, nécessaire à la terminaison de l'algorithme, ce qui engendre l'explosion mémoire.

Dans le cas où  $\varepsilon_l = 0$ , une optimisation récente d'UPPAAL considère qu'il est inutile de garder dans la **passed list** un état  $(q_v, D)$  ( $q_v$ : vecteur contenant l'état de contrôle courant du produit synchronisé,  $D$ : représentation des contraintes (Zones, valeur des variables...) si un autre état  $(q_v, D')$  vérifiant  $D' \supset D$  y est également présent.

Dans ce cas (figure 39), il n'est nécessaire de garder en mémoire que les états  $Z_1$  et  $Z_5$  car  $Z_5 \supset Z_4$ ,  $Z_5 \supset Z_3$  et  $Z_5 \supset Z_2$ .

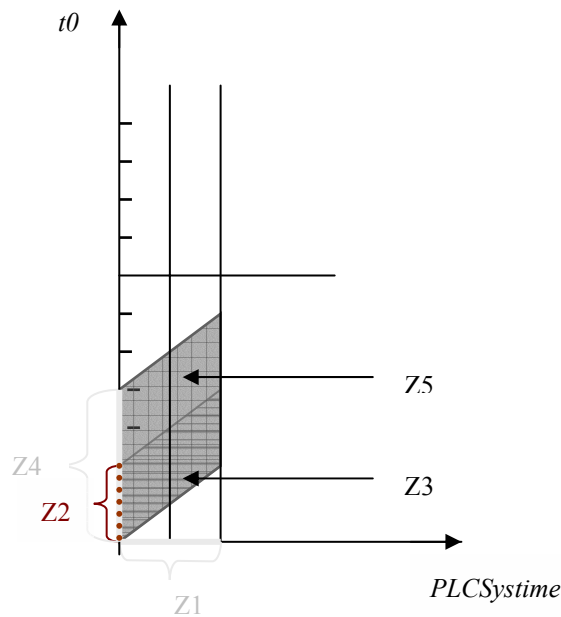


Figure 39

#### d) Conclusion

L'exemple traité est un exemple plus simple (en termes de taille du programme, nombre de temporisateurs, nombre d'entrées-sorties) que d'autres cas réels que l'on rencontre en milieu industriel.

Pour  $\varepsilon_1 \neq 0$ , la vérification de l'invariant pour un rapport  $PT/\varepsilon_2 = 25$  a nécessité 180 Mo de RAM. Sachant qu'en réalité, le rapport  $PT/\varepsilon_2$  se situe entre 1000 et 100000, on voit bien que l'utilisation d'un modèle prenant en compte de façon explicite les bornes inférieure et supérieure de la durée du cycle ne permettrait pas la vérification de programmes plus réalistes. Pour cette raison, il est nécessaire de faire certaines abstractions sur le modèle.

Le fait de considérer que  $\varepsilon_1 = 0$  constitue un exemple d'abstraction. Cette hypothèse permettra de maintenir constante la taille de l'espace mémoire nécessité par la vérification de la propriété, indépendamment du rapport  $PT/\varepsilon_2$ . Cependant, il est nécessaire de s'intéresser à la légitimité d'une telle hypothèse. Dans notre cas, la propriété que l'on cherche à vérifier est une propriété de sûreté. On sait que les propriétés de sûreté vérifient une propriété très intéressante et qui s'exprime par :

*Si une propriété de sûreté  $P$  est satisfaite pour un comportement  $C$ , alors  $P$  est forcément satisfaite pour tout comportement réduit  $C' \subseteq C$  [SBB+99]*

Ainsi, si on arrive à vérifier la propriété de sûreté en autorisant plus de comportements, c'est que la propriété sera également vérifiée pour des comportements plus restreints, correspondants au modèle où  $\varepsilon_1 \neq 0$ .

#### 4.4.4 Étude comparative du modèle de Mader-Wupper et du modèle basé sur l'exécution atomique instantanée du programme

Afin de pouvoir comparer ces modèles, nous avons mesuré expérimentalement le temps de calcul et l'espace mémoire nécessité par la vérification de la propriété de sûreté **AG not (MP0 and MP1)** sur l'exemple du mélangeur. La propriété a été vérifiée pour tous les modèles. Les meilleurs résultats ont été obtenus en utilisant une stratégie de recherche en profondeur d'abord (la plus adaptée à la forme du modèle). Les résultats expérimentaux sont donnés dans le tableau :

<i>Modèle</i>	$\varepsilon_1$	$E_2$	<i>PT</i>	<i>TIME</i> (s)	<i>Max VSize</i> (Mo)	<i>Max RSS</i> (Mo)
Modèle 1 (Mader- Wupper)	1	2	500	$\infty$	-	-
	0	2	500	7.44	30	24
	x	x	500	5.40	25	20
Modèle 2 (Exécution atomique)	1	2	500	18.29	18	15
	0	2	500	0.38	3.7	2.9
	x	x	500	0.27	3.6	2.7

D'après ces résultats, on remarque que :

L'hypothèse de l'exécution instantanée et atomique du programme diminue considérablement la complexité en temps et en espace (cas du modèle 2 par rapport au modèle 1). En effet, l'évolution d'un état *committed* vers un autre état *committed* ne nécessite aucun calcul sur les grandeurs temporisées (horloges) du modèle contrairement aux états de contrôle autorisant la progression du temps. De plus, l'atomicité de l'exécution des états *committed* limite les comportements possibles du modèle et limite ainsi l'explosion combinatoire due à la mise en parallèle de l'automate du programme avec les automates d'environnement et de temporisation.

Pour les valeurs de *PT* et  $\varepsilon_2$  choisies ( $PT/\varepsilon_2=250$ ) la différence entre les modèles prenant en compte explicitement la borne supérieure du temps de cycle et des modèles faisant abstraction des contraintes temporisées n'est pas très marquée. Pour des valeurs beaucoup plus grandes du rapport  $PT/\varepsilon_2$ , cette différence est nette. Elle montre l'importance des modèles faisant abstraction des aspects temporisés lorsqu'ils sont appliqués à la vérification des propriétés de sûreté. Le tableau illustre le cas où  $PT/\varepsilon_2=100000$ .



Modèle	$\epsilon_1$	$\epsilon_2$	PT	TIME (s)	Max VSize (Mo)	Max RSS (Mo)
Modèle 3	0	1	100000	41.91	76.3	74.3
	x	x	100000	0.27	3.6	2.7

## 4.5 Étude expérimentale de l'exemple de l'évaporateur

L'étude précédente de l'exemple du mélangeur nous a montré que les hypothèses consistant à prendre  $\epsilon_1=0$  et se basant sur l'exécution atomique du programme pouvaient améliorer sensiblement la complexité de la vérification. Nous nous proposons dans la suite de tester le modèle basé sur ces deux hypothèses sur un exemple « plus compliqué » : l'exemple de l'évaporateur. Comparé à l'exemple du mélangeur, cet exemple possède trois différences principales :

- Il contient plusieurs temporisateurs de type TON (quatre temporisateurs).
- Un modèle temporisé du processus contrôlé est analysé en même temps que le programme. Ce modèle décrit les évolutions possibles de l'environnement physique, en particulier celles qui peuvent amener à des situations dangereuses. Le programme de contrôle devrait ainsi éviter ces situations.
- Des propriétés temps-réel doivent être vérifiées par le programme.

### 4.5.1 Codage dans l'outil UPPAAL

Le modèle analysé résulte de la mise en parallèle de trois types d'automates temporisés :

#### a) La représentation de l'environnement

*Le réservoir 1*

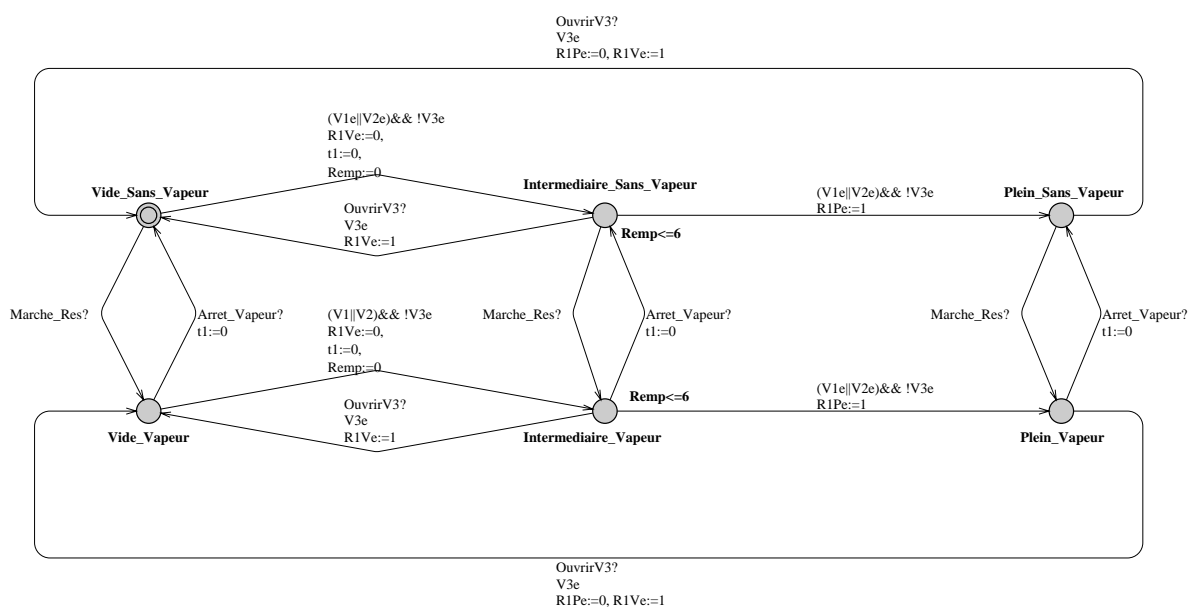


Figure 40  
Modèle UPPAAL du réservoir 1

Le réservoir 1 est modélisé par un automate temporisé à 6 états. Chaque état représente :

- Le niveau d'eau dans le réservoir : **Vide**, **Plein** ou **Intermediaire** (c'est-à-dire ni vide ni plein).
- La présence ou non de vapeur dans le réservoir (Vapeur, Sans\_Vapeur).

Une horloge **Remp** est utilisée afin de pouvoir mesurer la durée de remplissage du réservoir. L'invariant **Remp** ≤ 6 appliqué aux états **Intermediaire\_Vapeur** et **Intermediaire\_Sans\_Vapeur** modélise le fait que le temps de remplissage ne peut dépasser 6 unités de temps.

Le vidage du réservoir est instantané. Il est exécuté instantanément lors de la réception de l'évènement **OuvrirV3!** (Cet évènement étant envoyé par l'automate du programme).

La synchronisation binaire entre l'automate modélisant le réservoir 1, l'automate modélisant la résistance et l'automate du programme au moyen des canaux de communication **Arret\_Vapeur** et **Marche\_Res** permet le passage d'un état où la vapeur est présente à un état où il n'y a pas de vapeur.

L'horloge **t1** est utilisée afin de mesurer la durée passée dans un état où le réservoir contient du liquide et où la vapeur est arrêtée (états **Intermediaire\_Sans\_Vapeur** et **Plein\_Sans\_Vapeur**). L'automate ne doit en aucun cas passer plus de 18 unités dans l'un de ces deux états, sinon le liquide risque de se solidifier.

L'automate permet enfin d'affecter les valeurs adéquates aux variables d'environnement R1Pe et R1Ve.

### Le réservoir 2

Le réservoir 2 est modélisé par un automate temporisé à 5 états : **Vide**, **Plein**,

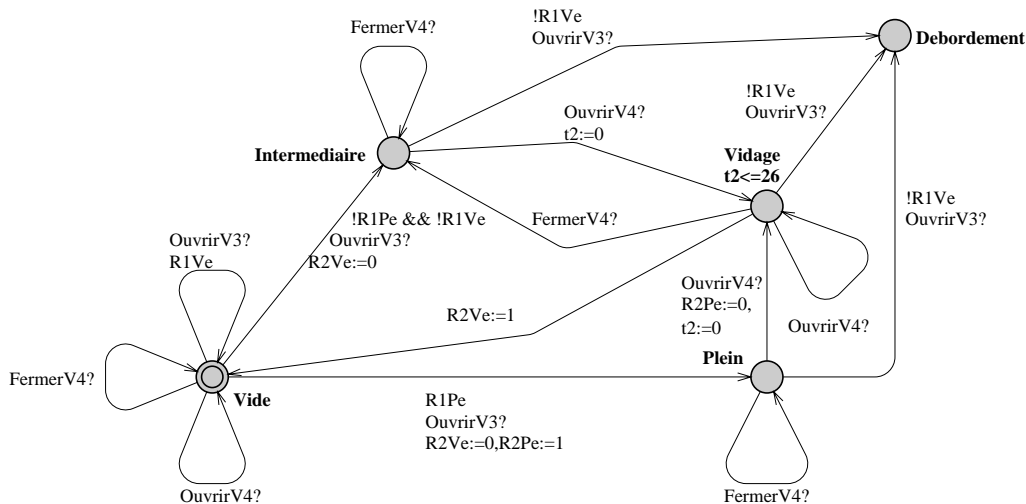


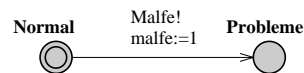
Figure 41  
Modèle UPPAAL du réservoir 2

**Intermédiaire, Vidage et Debordement.** Le remplissage du réservoir 2 est instantané. Il est réalisé lors de la réception de l'évènement **OuvrirV3!** envoyé par l'automate du programme.

Initialement, le réservoir 2 est dans l'état **Vide**. L'ouverture de la valve V3 le fait passer à l'état **Plein** si le réservoir 1 est plein et à l'état **Intermédiaire** si le réservoir 1 n'est ni vide, ni plein.

Contrairement au remplissage du réservoir 2 qui est instantané, le vidage de ce réservoir prend un temps au plus égal à 26 unités. L'ouverture de la valve V4 dans l'état **Plein** ou dans l'état **Intermédiaire** fait passer instantanément l'automate à l'état **Vidage**. La fermeture de la valve V4 dans l'état **Vidage** fait passer instantanément l'automate à l'état **Intermédiaire**. Si le réservoir 2 n'est pas vide, le réservoir 1 n'est pas vide et la valve V3 est ouverte, alors l'automate passe immédiatement à l'état **Debordement**.

#### *Le condenseur*



*Figure 42*  
*Modèle UPPAAL du condenseur*

Un modèle très simple a été retenu pour le condenseur.

Le condenseur est initialement dans l'état **Normal**. Si un défaut se produit, alors il passe instantanément à l'état **Probleme** en envoyant le signal **Malfe!** et en mettant à 1 la variable d'environnement **malfe**.

#### *La résistance chauffante*

La résistance est modélisée par 6 états. Chaque état représente :

- L'état de la résistance chauffante et la présence de vapeur dans le réservoir 1. Les états **On** et **On\_Malf** représentent la résistance chauffante en état de marche, les états **Attente** et **Attente\_Malf** représentent la résistance chauffante en état d'arrêt, la vapeur continuant encore à se dégager. Enfin, les états **Off** et **Off\_Malf** correspondent à un état d'arrêt de la résistance chauffante, avec une absence de vapeur dans le réservoir 1.
- Les états **On**, **Attente** et **Off** représentent l'état de la résistance chauffante lorsque le condenseur fonctionne normalement tandis que les états suffixés par **\_Malf** représentent l'état de la résistance chauffante lorsqu'un dysfonctionnement se produit dans le condenseur.

Les ordres de mise en marche et d'arrêt sont reçus grâce aux canaux de communication **Marche\_Res** et **Arret\_Res**. Ces ordres sont envoyés par l'automate du programme.

Une horloge  $p$  est utilisée afin de mesurer la durée passée dans un état où le vapeur se dégage encore et où le condenseur est dans un état de dysfonctionnement (états **On\_Malf** et **Attente\_Malf**)

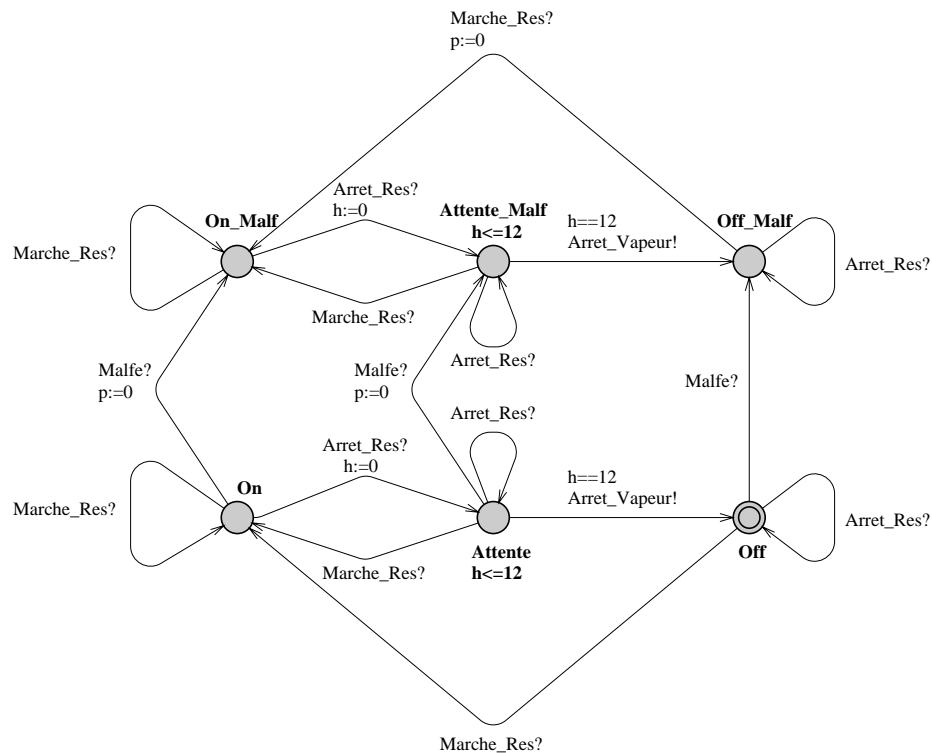


Figure 43  
Modèle UPPAAL représentant la résistance chauffante

### b) La représentation du programme

Le programme est traduit échelon par échelon. La traduction de chaque échelon est assez intuitive. Les principes du codage des appels aux temporisateurs sont très similaires à ceux employés au paragraphe 4.4.1.b).

Cependant, contrairement à l'exemple précédent, la phase d'affectation des sorties ainsi que les variables de sortie sont explicitement modélisés. Après l'exécution du programme, les variables de sorties (suffixées par  $e$ ) sont affectées. Suivant les valeurs de certaines variables, des canaux de communication (**OuvrirV3**, **OuvrirV4**, **FermerV4**, **Marche\_Res** et **Arret\_Res**) sont utilisés afin de forcer l'environnement à réagir aussitôt que les variables d'environnement sont changées.

La structure générale de l'automate représentant le programme est celle décrite à la figure 33.

**c) La représentation d'une instance du temporisateur TON**

Elle est identique à celle décrite au paragraphe 4.4.1.c).

**4.5.2 Formalisation des propriétés**

Les propriétés à vérifier sont celles qui ont été énoncées informellement dans le paragraphe 4.2.2.e). La formalisation de chaque propriété (dans la syntaxe d'UPPAAL) est illustrée par le tableau ci-dessous :

Propriété	Formalisation
P1	A[] not ((V1e or V2e) and V3e) or (V3e and V4e)
P2	A[] (Rese imply (!V1e and !V2e and !V3e))
P3	A[] (Rese imply R1Pe)
P4	A[] not (Resistance.On_Malf or Resistance.Attente_Malf) and p>21
P5	A[] not Reservoir2.Debordement
P6	A[] not (Reservoir1.Plein_Sans_Vapeur or Reservoir1.Intermediaire_Sans_Vapeur) and t1>18

**4.5.3 Résultats expérimentaux**

Les mesures ont été réalisées en utilisant la version 3.3.36 de juin 2003. Les meilleurs résultats expérimentaux ont été obtenus pour une recherche en largeur d'abord en utilisant une représentation en DBM. Ils sont résumés dans le tableau suivant :

Propriété	Résultat	Time (s)	Max VSize (Mo)	Max RSS (Mo)
P1	OUI	4.56	4.02	3.00
P2	OUI	4.58	4.02	3.00
P3	OUI	4.56	4.02	3.00
P4	OUI	16.37	4.71	3.61
P5	OUI	4.48	4.02	3.00
P6	OUI	13.16	4.71	3.58

#### 4.5.4 Conclusion

L'exemple traité n'est pas trivial. Le tableau ci-dessous résume ses caractéristiques essentielles :

Nombre de variables booléennes	37
Nombre de canaux de communication	19
Nombre d'horloges	10
Nombre de processus en parallèle	9

Les performances obtenues sont dues à deux principaux facteurs :

- Les constantes intervenant dans les contraintes sur les horloges n'ont pas des valeurs trop élevées.
- L'hypothèse de l'exécution atomique du programme diminue très fortement l'explosion combinatoire due à la mise en parallèle de plusieurs processus.

# Conclusion

La modélisation de l'exécution d'un programme sur un API fait intervenir différentes grandeurs temporelles relatives au matériel et au programme (la durée d'un cycle, les durées des temporisations...). La vérification de propriétés temps-réel sur de tels modèles fait intervenir d'autres grandeurs : les contraintes temporelles qui figurent dans les formules de logique décrivant ces propriétés. Ces aspects temporisés, qui interagissent à plusieurs niveaux, n'ont pas le même facteur d'échelle, et leurs ordres de grandeur varient sensiblement suivant les applications.

Pour pouvoir vérifier des programmes d'API par model-checking, il est nécessaire d'obtenir des modèles formels de leur exécution. De nombreuses propositions existent dans la littérature. Cependant, ces modèles ne sont pas applicables à toutes les classes de propriétés et se heurtent dans la plupart des cas au problème de l'explosion combinatoire en temps et en espace, inhérent à la vérification par model-checking. La résolution de ce problème nécessite de modéliser l'exécution d'un programme sur un API sous une forme permettant de contourner les limites des outils utilisés pour la vérification. Nous avons montré sur un cas d'étude comment des abstractions permettaient de réduire considérablement la complexité du model-checking, tout en assurant la vérification d'une large classe de propriétés.

L'étude expérimentale réalisée sur les exemples du mélangeur et de l'évaporateur nous a permis de déterminer certains aspects de modélisation (borne inférieure du temps de cycle nulle et exécution atomique du programme) permettant d'améliorer sensiblement la complexité de la vérification. La vérification de certaines propriétés de sûreté peut être réalisée à l'aide de modèles faisant abstraction de la durée du cycle (tels que le modèle de la figure 32). L'utilisation de tels modèles peut avoir un impact considérable sur la complexité de la vérification.

# BIBLIOGRAPHIE

[ABB+01] T. Amnell, G. Behrmann, J. Bengtsson, P. R. D'Argenio, A. David, A. Fehnker, T. Hune, B. Jeannot, K. G. Larsen, M. Oliver Möller, P. Pettersson, C. Weise, and W. Yi. *'Uppaal - Now, Next, and Future'*. In Proceedings of Modelling and Verification of Parallel Processes (MOVEP'2k), Nantes, France, June 19 to 23, 2000. LNCS Tutorial 2067, pages 100-125, F. Cassez, C. Jard, B. Rozoy, and M. Ryan (Eds.), 2001.

[AD94] R. Alur, D. L. Dill. *'A theory of timed automata'*. Theoretical Computer Science, 126(2):183-235, 1994.

[BDL+01] G. Behrmann, A. David, K. G. Larsen, O. Möller, P. Pettersson, and W. Yi. *'Uppaal - Present and Future'*. In Proceedings of the 40th IEEE Conference on Decision and Control (CDC'2001). Orlando, Florida, USA, December 4 to 7, 2001.

[CCL+00] G. CANET, S. COUFFIN, J.-J. LESAGE, A. PETIT, P. SCHNOEBELEN. *'Toward the automatic verification of PLC programs written in instruction list'*. IEEE International Conference on Systems Man and Cybernetics, SMC'2000, Nashville (USA), pp. 2449-2454, Oct. 2000.

[CDP+00] G. CANET, B. DENIS, A. PETIT, O. ROSSI, P. SCHNOEBELEN. *'Un cadre pour la vérification automatique de programmes IL'*. 1ère Conférence Internationale Francophone d'Automatique, IEEE-CNRS-GRAISyHM, CIFA'2000, Lille (France), pp. 693-698, Juil. 2000.

[CGP99] E. M. Clarke Jr., O. Grumberg, D. A. Peled. *'Model Checking'*. The MIT Press. 1999.

[DBL+03] A. David, G. Behrmann, K. G. Larsen, and W. Yi. *'A Tool Architecture for the Next Generation of UPPAAL'*. Technical report Uppsala University 2003.

[Die97] H. Dierks. *'PLC-Automata: A New Class of Implementable Real-Time Automata'*. Transformation-Based Reactive Systems Development (ARTS'97), M. Bertran and T. Rus, editors, Springer LNCS 1231, pp. 111-125, 1997. Available at <http://csd.informatik.uni-oldenburg.de/~dierks/Berichte/ARTS.ps.gz>

[Die98] H. Dierks. *'Comparing Model-Checking and Logical Reasoning for Real-Time Systems'*. In ESSILI' 98, pages 13-22, 1998.



[Die00] H. Dierks. '*PLC-Automata: A New Class of Implementable Real-Time Automata*'. Theoretical Computer Science, 253(1):61-93, December 2000. full version of [Die97]. Available at <http://csd.informatik.uni-oldenburg.de/~dierks/Berichte/TCS.ps.gz>

[EHK+97] S. Engell, R. Huuck, S. Kowalewski, Y. Lakhnech, J. Preußig, L. Urbina. '*Comparing Timed Condition/Event Systems and Timed Automata*'. Proceedings of HART'97, Grenoble, LNCS 1201:81-86, Springer-Verlag, 1997. Available at [http://www.informatic.uni-kiel.de/~kondisk/hart\\_97.ps.gz](http://www.informatic.uni-kiel.de/~kondisk/hart_97.ps.gz)

[FL00] G. Frey and L. Litz. '*Formal Methods in PLC-Programming*'. Proceedings of the IEEE SMC 2000, Nashville, October 08-11, 2000, pp. 2431-2436. Available at : <http://www.eit.uni-kl.de/litz/members/frey/PDF/V132.pdf>

[Huu98] R. Huuck. '*Transformation of Timed Condition/Event Systems into Timed Automata: An Approach to Atomic Verification*'. Master's Thesis, Chair of Software Technology, Christian-Albrecht-University of Kiel, Germany, 1998. Available at : <http://www.informatik.uni-kiel.de/~rhu/thesis.ps.gz>

[Huu99] R. Huuck. '*Verifying timing aspects of VHS case study I*'. Technical Report, Christian-Albrechts-Universität zu Kiel, 1999. Available at : <http://www-verimag.imag.fr/VHS/year1/cs11j.ps>

[Kow98] S. Kowalewski. '*Description of VHS case study I "Experimental Batch Plant"*'. Draft. University of Dortmund, Germany, July 1998. Available at : <http://www-verimag.imag.fr/VHS/year1/cs11b1.ps>

[KT97] S. Kowalewski, H. Treseler. '*VERDICT - A Tool for Model-Based Verification of Real-Time Logic Process Controllers*'. 5th Int. Workshop on Parallel and Distributed Real-Time Systems (WPDRTS'97), Geneva, Switzerland, April 1997.

[Lam99] S. LAMPERIERE-COUFFIN. '*Formal verification of PLC programs described in IEC 61131-3 languages*'. Talk in a lecture series, Department of Electrical Engineering and Information Technology - Institute of Process Automation, Kaiserslautern University (Germany), Nov. 1999.

[Lav98] M. Lavandier. '*Méthode de détermination expérimentale du comportement réactif d'un équipement industriel de commande*'. Mémoire de recherche, DEA de Production Automatisée. 1998.

[Lew98] R. W. Lewis. '*Programming industrial control systems using the IEC 1131-3 (Revised edition)*'. The Institution of Electrical Engineers, 1998.

[LL00] S. LAMPERIERE-COUFFIN, J.-J. LESAGE. '*Formal verification of the sequential part of PLC programs*'. 5th Workshop on Discrete Event Systems, WODES'2000, Ghent (Belgium), pp. 247-254, August 2000.

[LPW97] K. G. Larsen, P. Pettersson and W. Yi. '*Uppaal in a Nutshell*'. In Springer International Journal of Software Tools for Technology Transfer 1(1+2), 1997.

[LRL99] S. LAMPERIERE-COUFFIN, O. ROSSI, J.-J. LESAGE, J.-M. ROUSSEL. '*Formal Validation of PLC programs : a survey*'. European Control Conference 1999, ECC'99, Karlsruhe (Germany), CD-ROM paper n°741, 6 pages, Sept. 1999.

[IEC93] IEC (International Electrotechnical Commission). '*IEC Standard 61131-3 : Programmable Controllers – Part 3*'. 1993.

[MW99] A. Mader, H. Wupper. '*Timed Automaton Models for Simple Programmable Logic Controllers*'. 11th Euromicro Conf. on Real-Time Systems, published by IEEE Computer Society Press, Los Alamitos, California, held in York, UK, Jun. , 1999, pp. 114-122. Available at <http://wwwhome.cs.utwente.nl/~mader/PAPERS/euromicro.ps.Z>

[Mad00a] A. Mader. '*A Classification of PLC Models and Applications*'. 5th Int. Workshop on Discrete Event Systems (WODES'2000) -- Discrete Event Systems, Analysis and Control, R. Boel and G. Stremersch (eds.) , published by Kluwer Academic Publishers, Massachusetts, held in Ghent, Belgium, Aug. , 2000, pp. 239-247. Available at <http://www.cs.utwente.nl/~mader/PAPERS/modelbox.ps>

[Mad00b] A. Mader. '*Precise timing analysis for PLC applications - two small examples*'. Unpublished manuscript. November 2000. Available at <http://wwwhome.cs.utwente.nl/~mader/PAPERS/analysis.ps.gz>

[Mic88] G. Michel. '*Les A.P.I. : Architectures et applications des automates programmables industriels*'. Éditions Dunod. 1988.

[MW00] A. Mader. '*What is the method in applying formal methods to PLC applications?*'. 4th Int. Conf. Automation of Mixed Processes: Hybrid Dynamic Systems (ADPM), S. Engel and S. Kowalewski and J. Zaytoon (eds.) , published by Shaker Verlag, Aachen, Germany, held in Dortmund, Germany, 2000, pp. 165-171. Available at <http://www.cs.utwente.nl/~mader/PAPERS/method.ps.gz>

[OY93] A. Olivero, S. Yovine. '*KRONOS : A tool for Verifying Real-time systems. User's guide et Refrence Manual*'. Verimag, Grenoble, France. 1993.

[Per99] N. Perpère. *'Identification et modélisation du comportement dynamique d'un Automate Programmable Industriel'*. Mémoire de recherche, DEA de Production Automatisée. 1999.

[Pil94] Pillet. *'Introduction aux plans d'expériences par la méthode Taguchi'*. Editions d'organisation universitaires. 1994.

[RS00] O. ROSSI, P. SCHNOEBELEN. *'Formal modeling of timed function blocks for the automatic verification of ladder diagram programs'*. 4th International Conference on Automation of Mixed Processes: Hybrid Dynamic Systems, ADPM'2000, Dortmund (Germany), pp. 177-182, Sept. 2000.

[RSC+00] O. ROSSI, O. DE SMET, S. COUFFIN, J.-J. LESAGE, H. PAPINI, H. GUENNEC. *'Formal verification: a tool to improve the safety of control systems'*. 4th IFAC Symposium on Fault Detection, Supervision and Safety for Technical Processes, SAFEPROCESS'2000, Budapest (Hungary), pp. 885-890, June 2000.

[SBB+99] Ph. Schnoebelen, B. Bérard, M. Bidoit, F. Laroussine, A. Petit and OTHERS. *'Vérification de logiciels : techniques et outils du model-checking'*. Vuibert, 1999.

[SK91] R. S. Sreenivas, B. H. Krogh. *'On Condition/Event Systems with Discrete State Realizations'*. In *Discrete Event Dynamics Systems : Theory and Application 1*, pp 209-236. Kluwer Academic, Boston, 1991.

[SCR00] O. DE SMET, S. COUFFIN, O. ROSSI, G. CANET, J.-J. LESAGE, P. SCHNOEBELEN, H. PAPINI. *'Safe programming of PLC using formal verification methods'*. 4th International PLCopen conference on Industrial Control Programming, ICP'2000, Utrecht (The Netherlands), pp. 73-78, October 2000.

[TBK00] H. Treseler, N. Bauer and S. Kowalewski. *'Verification of IL Programs with a Detailed Model of their PLC Execution'*. In proceedings of the 5th Workshop on Discrete Event Systems (WODES 2000), 21-23 August, 2000.

[TD98] J. Tapken, H. Dierks. *'Moby/PLC - Graphical Development of PLC-Automata'*. In A.P. Ravn and H. Rischel, editors, *Proceedings of FTRTFT'98*, volume 1486 of LNCS, pages 311-314. Springer Verlag, 1998.

Available at <http://csd.informatik.uni-oldenburg.de/~dierks/Berichte/FTRTFT98tool.ps.gz>

[Wil99] H. X. Willems. *'Compact timed automata for PLC Programs'*. Technical Report, University of Nijmegen. November 1999.

Available at <http://www.cs.kun.nl/~mader/rik/plc.ps>

[Yov97] S. Yovine. '*Kronos: A verification tool for real-time systems*'. International Journal of Software Tools for Technology Transfer, Vol. 1, Issue 1/2, pages 123-133, Springer-Verlag. October 1997.

[ZRK03] B. Zoubek, J.-M. Roussel, M. Kwiatkowska. '*Towards automatic verification of ladder logic programs*'. Proceedings of IMACS-IEEE Computational Engineering in Systems Applications (CESA'03), Lille France, 9-12 July 2003 (to be published). Available at <http://www.cs.bham.ac.uk/~bxz/download/cesa03.pdf>