Université de Montréal

Un générateur CP pour la vérification temporelle des contrôleurs d'interfaces

par

Ying Zhang

Département d'Informatique et de Recherche Opérationnelle

Faculté des Arts et des Sciences

Mémoire présenté à la Faculté des études supérieures

en vue de l'obtention du grade de

Maître ès Sciences (M. SC.) en informatique

Juillet 2001

Université de Montréal

Faculté des études supérieures

Ce Mémoire intitulé

Un générateur CP pour la vérification temporelle des contrôleurs d'interfaces

présenté par

Ying Zhang

a été évalué par un jury composé des personnes suivantes :

Rachida Dssouli	présidente-rapporteuse
Eduard Cerny	directeur de recherche
Gilles pesant	membre du jury

Mémoire accepté le : 26, juillet 2001

Sommaire

Ce mémoire présente un outil et une méthodologie permettant pour le contrôleur d'interface d'un système numérique, la saisie rapide de sa spécification temporelle sous forme de chronogramme (TD : Timing Diagram) et son implantation sous forme de machines à états finis (FSM : Finite-State Machine) synchrones dans le but de vérifier si cette implantation est correcte, c'est-à-dire, sous les contraintes imposées par l'environnement (les contraintes du type *assume* du chronogramme), tous les événements produits par l'implantation satisfont les contraintes du type *commit* spécifiées dans le chronogramme TD correspondant.

Les travaux de recherche présentés dans ce mémoire se basent principalement sur ceux de deux anciens chercheurs du LASSO [2] [3]. Dans [2], une étude extensive sur la vérification temporelle des interfaces à l'aide de la programmation logique avec contraintes a été faite. Dans [3], une méthodologie a été développée qui permet de valider pour un contrôleur d'interface son implantation par rapport à sa spécification.

La programmation par contraintes intégrant l'arithmétique relationnelle des intervalles (RIA : Relational Interval Arithmetic) fournit un outil prometteur dans le but de la vérification des propriétés temporelles des interfaces. Le langage de programmation utilisé pour le développement est ILOG Solver, un produit développé en C++ par la compagnie Ilog. ILOG Solver fait coopérer le paradigme de la programmation orientée objet et le paradigme de programmation par contraintes dans le but de faciliter la modélisation et la résolution des problèmes combinatoires à l'aide d'un ensemble de fonctions et classes prédéfinis. Dans cet environnement, on a développé un ensemble de classes et modules permettant de :

1) Saisir un chronogramme à l'aide des classes prédéfinis ;
2) Saisir une FSM qui décrit l'implantation d'un contrôleur d'interfaces ;
3) Calculer pour un chronogramme donné les séparations temporelles des événements par rapport à un événement de référence ;
4) Vérifier la causalité d'un chronogramme en cherchant une bloc-machine causale;
5) Valider l'implantation d'un contrôleur d'interface par rapport à sa spécification.

Les fonctionnalités (3) et (4) ci-dessus constituent un pré-traitement dans l'implantation de la fonctionnalité (5) parce que d'une part si un chronogramme est non causal qui signifie qu'aucune machine à états finis peut implanter correctement ce chronogramme, alors la vérification de leur implantation par la fonctionnalité (5) ne sera pas nécessaire ; d'autre part ces méthodes facilitent l'implantation de la fonctionnalité (5) en fournissant le nombre maximum de cycles nécessaires à découler pour vérifier une contrainte du type *commit* et une bloc-machine causale qui permet d'ordonner partiellement les contraintes à vérifier.

L'implantation d'ensemble de méthodes de vérification temporelle des interfaces avec ILOG Solver permet non seulement d'exprimer facilement la représentation d'un chronogramme et celle d'une machine à états finis, mais aussi de rendre le processus de vérification plus transparent, plus souple pour les utilisateurs. L'exemple d'application présenté dans le chapitre 4 illustre le fonctionnement de l'outil développé.

Mots clés : Vérification temporelle des interfaces, Chronogramme, Machine à états finis, Programmation logique avec contraintes, L'arithmétique relationnelle des intervalles.

Table des matières

Liste des tableaux

Liste des figures

Liste des symboles

ASIC	Application Specific Integrated Circuit
ASM	Abstract State Machine
APSD	All-Pair Shortest Distance
CLP	Constraint Logic Programming
CP	Constraint Programming
CRT(RIA)	Constraint Resolution Techniques based on Relational Interval Arithmetic
CSP	Constraint Satisfaction Problem
FSM	Finite State Machine
HAAD	Hierarchical Annotated Action Diagram
PI	Primary Inputs
PO	Primary Outputs
RTL	Register Transfer Level
TD	Timing Diagram
TDTA	Timing Diagram Transition Automaton
WTA	Waveform Transition Automaton

Chapitre 1 Introduction

1.1 Contexte et problématique

En informatique, le processus de vérification pose un problème délicat. Il faut, d'une part, démontrer la validité des propriétés qualitatives d'un système informatique et, d'autre part, confirmer la cohérence des propriétés quantitatives de ce même système.

En général, lorsque l'on parle de vérification, il s'agit du problème de la vérification qualitative ou vérification fonctionnelle. Dans le passé, ce type de problème a suscité plus d'intérêt que celui de la vérification quantitative (aussi désigné par l'expression analyse temporelle ou vérification temporelle, en anglais timing analysis ou timing verification) et de nombreux algorithmes "satisfaisants" (le problème demeure $\mathcal{NP}$-complet) ont été développés.

La vérification temporelle a connu récemment un regain d'intérêt. Diverses théories ont été élaborées et plusieurs tentatives d'unification ou de ratification sont mises de l'avant. Dans le cadre de nos travaux, nous nous intéressons à une forme restreinte de vérification des propriétés temporelles des interfaces matérielles (interface timing verification) ou, plus simplement, vérification des interfaces.

1.1.1 Vérification temporelle

Le matériel informatique se définit à un certain niveau d'abstraction comme un ensemble de systèmes réactifs. Nous n'aborderons dans ce mémoire que cette classe de systèmes temporels et plus particulièrement, nous nous intéressons aux systèmes numériques avec délais. Des systèmes réactifs répondent aux stimuli de leur environnement "avec vi-

tesse". Cette notion de vitesse les distingue des purs systèmes interactifs propres au domaine de la vérification fonctionnelle et conduit à considérer un nouveau facteur, la notion de temps réel. La vérification temporelle s'intéresse aux systèmes temporisés.

Le temps physique se présente à nous sous une forme ininterrompue. Sa représentation abstraite peut être discrète ou dense. D'une façon ou d'une autre, la représentation adoptée présente des avantages et des inconvénients. Un temps discret est plus facile à manipuler : les opérations se font sur des entiers. D'un autre côté, la nécessité de fixer une unité de base pour mesurer la progression du temps entraîne imprécision : quelques circonstances (c'est-à-dire configurations temporelles) peuvent passer inaperçues. Et une unité plus précise a l'inconvénient d'engendrer un plus grand nombre d'éventualités à considérer. Un temps dense, c'est-à-dire reposant sur l'usage de valeurs réelles ou rationnelles, est plus complexe à manipuler mais aussi plus expressif. Dans notre recherche, nous considérons le temps sous forme dense.

L'introduction de la notion de temps n'affecte en rien les objectifs de la vérification. Qu'il s'agisse de vérification fonctionnelle ou de vérification temporelle, il convient de démontrer qu'une implantation $I(M)$ où M est un système interactif ou réactif, se comporte tel que le prescrit une spécification $S(M)$. Selon les modèles en présence, il peut être avantageux d'aborder ce problème selon l'optique d'une équivalence

$$I(M) \equiv S(M) \tag{1.1}$$

d'une implication logique

$$I(M) \Rightarrow S(M) \tag{1.2}$$

ou d'un contexte sémantique (l'implantation définissant un modèle sémantique qui valide la spécification)

$$I(M) \models S(M) \tag{1.3}$$

Pour que ces propositions soient vraies dans le contexte de la vérification temporelle, les propriétés temporelles spécifiées par le premier modèle doivent être respectées dans le second. Pour la méthodologie de vérification de contrôleur d'interface qui va être

présentée dans le chapitre suivant, la spécification temporelle du contrôleur $S(M)$ est sous forme de chronogramme (voir la définition 1.8) et l'implantation du contrôleur $I(M)$ est représentée par une machine à états finis. (FSM : Finite-State Machine) (voir la définition 1.9.)

La notion de modèle est cruciale. Implantation et spécification sont des termes relatifs et ils décrivent, à des niveaux d'abstraction différents, le comportement d'un même système. L'implantation est le modèle à vérifier et la spécification, le modèle servant de base à cette vérification. Sans entrer outre mesure dans les détails, nous pouvons différencier deux paradigmes de vérification : la simulation et la vérification formelle.

- **Simulation**

La simulation est la méthode de vérification la plus simple et la plus courante. Typiquement, un système réactif répond de diverses façons selon les stimuli auxquels il est soumis. La simulation consiste à étudier tout l'éventail de ces stimuli et observer dans chaque configuration particulière le comportement du système à vérifier.

En pratique, ce paradigme se heurte à la complexité des systèmes à vérifier. Le nombre de configurations distinctes résultant des diverses combinaisons de stimuli est tout simplement trop grand. Même pour des systèmes de taille restreinte, le processus de vérification se révèle long et fastidieux.

- **Vérification formelle**

Pour éviter les obstacles auxquels la simulation se heurte, des méthodes analytiques plus rigoureuses ont été développées. Au lieu de procéder par simple observation, une preuve formelle des propositions est construite. Nous entrons dans le domaine de la vérification formelle.

Il est à noter qu'une simulation exhaustive est une preuve formelle, à strictement parler, de la validité d'un modèle. Pour éviter toute ambiguïté, l'appellation "vérification formelle" est cependant réservée à une classe bien particulière de méthodes analytiques. (model checking, theorem proving, etc.)

1.1.2 La problématique de la vérification des interfaces

- **L'origine du problème**

La conception des systèmes informatiques est un processus modulaire : à partir de divers composants aux fonctions limitées, des entités d'usage plus général sont élaborées. Ces dernières peuvent à leur tour être imbriquées dans des réalisations plus importantes: le processus est ainsi hiérarchique.

Tous ces modules, quelle que soit leur complexité, sont donc conçus pour être utilisés conjointement avec d'autres matériels informatiques. Or, cette conception est généralement un processus isolé. La nature exacte des futurs éléments connexes est inconnue. Des hypothèses doivent être formulées sur les conditions d'usage d'une pièce particulière et la modélisation de celle-ci, énoncée en conséquence. Ces hypothèses définissent ce que l'on appelle l'environnement d'un composant informatique, c'est-à-dire, les autres composants avec lequel il sera éventuellement relié pour réaliser un système plus complexe.

Le concepteur de systèmes informatiques dispose ainsi d'une collection de composants. Sa tâche, vue de façon un peu simpliste, se résume à en faire une sélection et ensuite à les assembler. Tout le problème est de déterminer si des composants aux fonctions désirées et complémentaires peuvent communiquer correctement ensemble si l'on relie les ports appropriés de leurs interfaces. En d'autres mots, déterminer quels composants exhibent des interfaces compatibles. En termes succincts, c'est là tout le problème de la vérification des interfaces.

Cette vérification des interfaces, lorsqu'elle s'applique à des systèmes réactifs, se réduit à une vérification des propriétés temporelles de chacun des composants d'un système. C'est dans ce sens que la vérification des interfaces relève du domaine de la vérification temporelle.

- **Spécification des interfaces**

Traditionnellement, la vérification des interfaces a pour champ d'application les systèmes avec délais : bus, CPU, mémoires, etc. Représenter les interfaces de ces systèmes sous forme de chronogrammes (timing diagrams) est une pratique d'usage courant. Le chronogramme est un outil de spécification essentiellement graphique. Il permet de décrire de manière naturelle les interfaces matérielles. Les concepteurs de systèmes informatiques y ont grandement recours à des fins de référence et de communication. Et pourtant, très peu de standard existe...

Éloge de la facilité d'emploi du chronogramme et de son expressivité, l'absence de standard est fâcheuse pour le domaine de la vérification des interfaces. Sans bases solides, il est malaisé de développer une méthodologie stricte pour la vérification. Et l'emploi de méthodes de vérification impromptues pour suffire aux besoins de chaque nouvelle spécification n'est pas une alternative sérieuse.

Dans la section suivante, nous introduisons certains concepts qui seront exploités dans les chapitres suivants.

1.2 Concepts de base

Dans cette section, nous précisons les composants d'un chronogramme. En général, un chronogramme est composé de :

1) Un ensemble de ports ;
2) Un ensemble d'événements spécifiés sur les ports ;
3) Un ensemble de contraintes temporelles explicites et implicites mettant en relation les événements spécifiés entre eux.

Définition 1.1 : *Les ports.* Les ports sont des canaux utilisés par un système pour communiquer avec son environnement. Trois caractéristiques définissent l'usage d'un port: sa direction, le type des données qu'il transporte et son interprétation.

Direction d'un port. La direction d'un port indique d'où proviennent les événements qu'il transporte. Trois valeurs de direction sont possibles :
Entrée : cette direction signifie que les événements que transmet le port émanent de l'environnement et ils sont destinés à être envoyés au système ;

1) Sortie : les événements sont cette fois produits par le système et ils sont destinés à l'environnement ;
2) Entrée-sortie : le port peut véhiculer des événements en entrée et en sortie. Un port n'est utilisé que dans une direction à la fois et donc ne peut pas être au même instant utilisé en entrée et en sortie.

Type d'un port. Le domaine des valeurs de l'information transmise par un port s'appelle le type du port. Cela inclut : les types les plus courants des langages de programmation en plus des types pouvant représenter les valeurs possibles de signaux numériques et des types abstraits comme des enregistrements. Nous ne considérons que des ports du type booléen actuellement.

Interprétation d'un port. L'interprétation d'un port modifie la définition de ce qu'est un événement sur un port. Il y a deux interprétations possibles :

L'interprétation signal. Un événement a lieu sur un port quand la valeur transmise par celui-ci change. Ainsi, il suffit d'observer un changement de valeur du port pour conclure qu'un événement a lieu.

L'interprétation message. L'interprétation message d'un port est un peu plus subtile. Un événement ne donne pas lieu nécessairement à un changement de valeur du port parce que la nouvelle valeur déposée peut être équivalente à l'ancienne.

Par exemple, la figure 1.1 (extrait de [1]) illustre un chronogramme avec 5 ports dont MCLK, ALE et nRW sont des ports ayant l'interprétation signal et ADDR et DATA sont des ports ayant l'interprétation message.

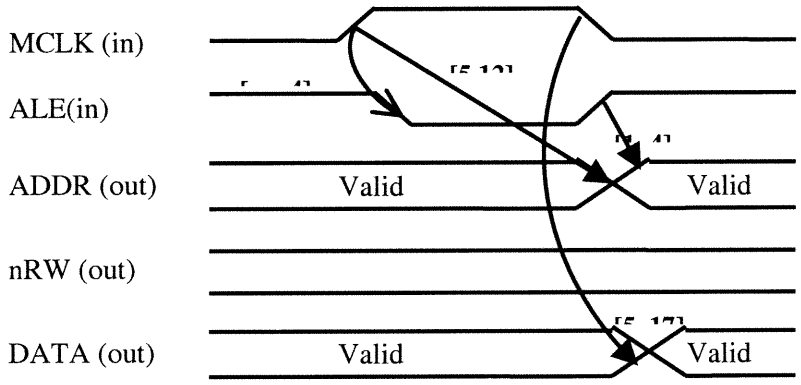


Figure 1.1 Un exemple des ports ayant des interprétations différentes

Définition 1.2 : *Les états d'un port.* À tout instant un port se trouve dans l'un des trois états suivants :

- L'état *constant*. Dans cet état, le port transmet une information constante dont la valeur est connue. La valeur de la constante est dans le domaine du type du port.
- L'état *valide*. Cet état est semblable à l'état constant. La différence entre cet état et l'état constant est que dans le cas de l'état valide, la valeur transportée par le port est inconnue, au moment d'écrire la spécification.
- L'état *indéfini*. Dans cet état, un port peut avoir un comportement quelconque, stable ou changeant.

Définition 1.3 : *Événements spécifiés*[1]. Le passage d'un état à un autre d'un port constitue un événement spécifié. Le genre des événements peut être soit constant, soit valide ou indéfini et est déterminé par l'état que prend le port après cet événement.

- Événement *constant*. Le passage d'un port de n'importe quel état à l'état constant constitue un événement constant.
- Événement *valide*. Le passage d'un port de n'importe quel état à l'état valide constitue un événement valide.
- Événement *indéfini*. Le passage d'un port de n'importe quel état à l'état indéfini constitue un événement indéfini.

Pour des ports ayant l'interprétation signal, il est utile de scinder en deux classes les événements spécifiés d'entrée : les événements spécifiés observables et les non-distinguables.

Définition 1.4 : *Événements observables et non-distinguables*[2] sur des ports ayant l'interprétation signal. Les événements spécifiés *observables* sont ceux qui définissent le passage du port de l'état constant à l'état constant. La définition des événements *non-distinguables* est tout événement définissant le passage d'un port à l'état valide ou indéfini à n'importe quel autre état.

Nous utilisons pour les définitions suivantes $\Re$ pour désigner l'ensemble des nombres réels, Q pour désigner l'ensemble des nombres rationnels finis, Q^+ pour l'ensemble des nombres rationnels finis positifs ou zéro.

---

[1] La méthodologie de vérification développée dans [3] permet de traiter les ports de direction quelconque (un port entrée-sortie est divisé en deux ports de type entrée et sortie), du type booléen actuellement, avec interprétation signal en état constant seulement, c'est-à-dire, tous les événements spécifiés considérés sont des événements constants.

[2] Dans la méthode de vérification de contrôleur développée dans [3], on demande que tous les événements d'entrée doivent être observables sur les ports ayant interprétation signal.

Définition 1.5 : *Intervalle temporel.* (extrait de [1]) Un intervalle temporel π est un ensemble de nombres réels représenté par sa borne inférieure T_{min} et supérieure T_{max}. Nous ne considérons que l'ensemble des intervalles $\mathcal{I}$ dans lesquels $T_{min} \in Q \cup \{-\infty\}$, $T_{max} \in Q^+ \cup \{\infty\}$, et $T_{min} \leq T_{max}$. Un intervalle π de $\mathcal{I}$ est un sous-ensemble de nombres réels tels que, $\forall\, t \in \pi$, t est fini (mais possible non-borné), et

$T_{min} \leq t \leq T_{max}$ si T_{min} et T_{max} sont tous finis (π est dénoté par $[T_{min}, T_{max}]$).

$T_{min} \leq t$ si T_{min} est fini et $T_{max} = \infty$ (π est noté par $[T_{min}, \infty]$).

$t \leq T_{max}$ si $T_{min} = -\infty$ et T_{max} est fini (π est noté par $[-\infty, T_{max}]$).

$\pi = \Re$ si $T_{min} = -\infty$ et $T_{max} = \infty$ (π est noté par $[-\infty, \infty]$).

L'ensemble des intervalles $\mathcal{I}$ est divisé en deux sous-ensembles $\mathcal{I}_{conc}$ pour les intervalles de *concurrence* et $\mathcal{I}_{prec}$ pour les intervalles de *préséance*. Les éléments de $\mathcal{I}_{conc}$ ($\mathcal{I}_{prec}$) sont caractérisés par $T_{min} \leq 0$ ($T_{min} > 0$).

Définition 1.6 : *Relation temporelle* entre deux événements. Une relation temporelle est un triplet $r(e_i, e_j, \pi)$ où e_i et e_j sont les événements *source* et *cible* respectivement tels que $e_i \neq e_j$. π est un intervalle temporel dans $\mathcal{I}$. Une relation temporelle ayant un intervalle de *préséance* (*concurrence*) est une relation temporelle de *préséance*. (*concurrence*)

Définition 1.7 : *Contrainte temporelle*[1] entre les événements. Une contrainte temporelle est composée d'un ensemble de relations temporelles qui partagent un même événement cible.

Contraintes *assume* ou *commit*. Les contraintes *assume* servent à mettre en relation des événements quelconques avec des événements d'entrée de l'interface. Leur rôle est de spécifier les bornes temporelles à l'intérieur desquelles ces événements doivent être observés. Quant aux contraintes *commit*, elles servent à mettre en relation des événe-

---

[1] Une relation temporelle est une contrainte temporelle ayant un seul composant.

ments quelconques avec des événements de sortie et leur rôle est de spécifier les bornes temporelles à l'intérieur desquelles ces événements doivent être générés par le système.

Type de contraintes temporelles. Lorsqu'un événement est la cible de plusieurs relations temporelles, celles-ci se combinent entre elles pour former une contrainte temporelle du type linéaire (*conjunctive*), *max* (*latest*) ou *min* (*earliest*).

Étant donné un ensemble composé de n relations temporelles RT dont la cible est l'événement e, P_i est utilisé pour désigner une relation temporelle de *préséance* et X_i est utilisé pour désigner une relation temporelle de *préséance* ou *concurrence* ($i = 1, \ldots, n$). e_i est la source d'une relation temporelle X_i. $\pi_i = [l_i, u_i]$ est l'intervalle temporel associé à X_i. t_i , t : le temps d'occurrence de e_i et e respectivement. L'intersection de π_i et π_j est définie comme $\cap(\pi_i , \pi_j) = [max(l_i , l_j), min(u_i , u_j)]$.

- Contrainte linéaire composée de RT : Cette contrainte c permet de faire l'intersection des relations temporelles entre elles. Sa sémantique formelle est

$$c(X_1, \ldots, X_n) \text{ est linéaire} \Rightarrow t = \bigcap_i [\, t_i + l_i , t_i + u_i \,],\ 0 \le l_i \le u_i,\ (i = 1, \ldots, n)$$

- Contrainte *max* (*latest*) composée de *RT*. Cette contrainte permet d'exprimer que l'intervalle de temps dans lequel e se produit, est retardé le plus possible. Sa sémantique formelle est

$$c(P_1, \ldots, P_n) \text{ est } max \Rightarrow t = \max_i (t_i + \delta_i),\ 0 \le l_i \le \delta_i \le u_i,\ (i = 1, \ldots, n)$$

- Contrainte *min* (*earliest*) composée de *RT*. Cette contrainte permet d'exprimer que l'intervalle de temps dans lequel e se produit est devancé le plus possible. Sa sémantique formelle est

$$c(P_1, \ldots, P_n) \text{ est } min \Rightarrow t = \min_i (t_i + \delta_i),\ 0 \le l_i \le \delta_i \le u_i,\ (i = 1, \ldots, n)$$

Définition 1.8 : *Chronogramme*[1] (Timing Diagram). Un chronogramme est un 4-tuple $TD = (S, E, o, C)$, où S est l'ensemble de ports, E est l'ensemble des événements spécifiés, o est l'événement original $o \in E$ et C l'ensemble de contraintes temporelles.

Chronogrammes sont très couramment utilisés pour spécifier les relations temporelles entre les événements sur les ports. Le chronogramme est avant tout un diagramme décrivant la progression d'événements dans le temps.

Il existe certaines contraintes *implicites* pour un chronogramme. Tout d'abord, il existe des contraintes de préséance entre l'événement original et le premier événement de chaque port sous forme (e_{Origine}, e_{Premier}, $(0, \infty]$). En plus, il existe pour deux événements consécutifs d'un même port X, une relation de préséance sous forme (e_{Xi}, e_{Xi+1}, $(0, \infty]$). Cette contrainte est due au fait que les événements spécifiés d'un même port ont nécessairement lieu les uns après les autres. La figure 2.5 du chapitre 2 peut être utile pour comprendre ces concepts.

1.3 Revue de littérature

La vérification temporelle des interfaces basée sur le chronogramme constitue un domaine de recherche très large et il y a beaucoup de chercheurs qui s'y sont intéressés.

Borriello [13] a été le premier à identifier la vérification des interfaces comme un problème à part entière. Entre autres choses, il reconnaît l'utilité des chronogrammes et développe un algorithme pour vérifier la validité de réseaux de relations temporelles linéaires. Cependant, les résultats de ses recherches sont surtout axés vers le domaine de la synthèse des contrôleurs.

---

[1] Plus précisément, il s'agit d'un chronogramme feuille en terme HAAD. La définition est extraite de [1].

Gahlinger [14] tout d'abord et Amon [15] ensuite abordent le problème de façon plus rigoureuse. Le problème de vérification est posé selon les termes d'un calcul de séparations minimales et maximales entre les événements comme un ensemble de suppositions (assume constraints) et d'engagement (commit constraints) qu'il convient de vérifier. Les notions de consistance et de "satisfiabilité" sont introduites.

Comme Borriello, Gahlinger s'intéresse surtout aux relations conjonctives. Les relations causales ne sont envisagées que sous l'optique d'énumérations les réduisant à une dimension linéaire. Plus tard, Vanbekbergen [16] et McMillan et Dill [12] reconnaissent la nécessité de développer un algorithme plus efficace pour le traitement des relations causales. Dans [12], les auteurs démontrent que résoudre des systèmes de contraintes mixtes, plus complexes, où les contraintes peuvent être linéaires, *min* et *max*, est un problème *NP*-complet. Ils proposent deux algorithmes que nous nommerons MD1 et MD2, pour des versions restreintes de ce problème. Le premier MD1 permet de calculer les séparations maximales entre les événements de systèmes de contraintes strictement du type *max*, le deuxième MD2, entre les événements de systèmes de contraintes de type linéaire et *max*. Ces algorithmes ne sont valides que dans le contexte de systèmes de contraintes acycliques. Amon et al. [17] ont étendu l'algorithme MD1 au cas des contraintes cycliques. L'algorithme MD2 bien qu'efficace dans la plupart des cas exhibe parfois une complexité exponentielle. Une alternative a été proposée par Yen et al. [18].

Traiter la vérification des interfaces comme un problème de satisfaction de contraintes semble naturel. Older [19] a proposé et analysé à l'aide de la programmation logique avec contraintes un modèle restreint de systèmes temporisés. Son modèle est un réseau d'événements. Les événements de ce modèle sont caractérisés par les délais finis qui les séparent. Le processus de vérification consiste à vérifier sous quelles conditions, le système modélisé n'est pas cohérent. Le problème est formulé de telle sorte qu'une réponse négative (i.e., le modèle n'est pas cohérent) constitue une preuve de la validité du système à vérifier. Les résultats de cette approche sont convaincants : une réponse est généralement obtenue en un temps linéaire relativement à la taille du réseau d'événements.

Amon et Borriello [20] ont développé une approche fondée sur CLP pour un modèle plus complexe dans le cadre de la vérification des interfaces. Un graphe bipartie dirigé est employé. Deux types de nœuds sont définis : les événements dont la fonction est de fixer un point de référence dans le temps, et les différents opérateurs sont définis qui précisent les relations entre ces événements. Le modèle ainsi spécifié impose la définition d'une grande quantité de règles supplémentaires pour la manipulation des contraintes. Contrairement à Older [19] qui grâce au mécanisme de résolution de CLP(BNR) Prolog [21] utilise directement l'arithmétique relationnelle des contraintes sur intervalles, dans [20] les auteurs ont implanté leur propre mécanisme de résolution.

Dans [1], les auteurs ont fait une synthèse des travaux de recherche effectués par l'auteur et les chercheurs du LASSO en s'appuyant sur la notion de chronogrammes hiérarchiques annotés.

Dans [2], P. Girodias explore la programmation logique avec contraintes pour la vérification des interfaces. L'approche proposée est fondée sur les notions de consistance, de satisfiabilité et de causalité. En réduisant les chronogrammes à une dimension d'abstractions mathématiques, l'auteur constate que leur structure épouse parfaitement celle des problèmes de satisfaction de contraintes qui caractérisent le domaine de la programmation logique avec contraintes et que chronogrammes et CSPs partagent une vue identique de la consistance. À partir de cette observation, l'auteur indique qu'étendre la notion de causalité au contexte des CSPs est chose relativement aisée. P. Girodias montre que des techniques de consistance basées sur l'arithmétique relationnelle des intervalles permettent en fait d'obtenir des réponses exactes. Qui plus est, un modèle mathématique traitant d'intervalles de valeurs convient particulièrement à la vérification des interfaces où les délais de propagation entre les événements sont généralement représentés à l'aide de bornes minimales et maximales plutôt que par des valeurs ponctuelles. En dépit de sa simplicité, ce modèle suffit à illustrer les avantages à retirer de la programmation logique avec contraintes.

Une fois résolus tous les problèmes liés aux notions de consistance, de causalité, il devient nécessaire de considérer un autre genre de problèmes : vérification d'implantation de contrôleur des interfaces. Dans [3], F. Jin passe de la vérification temporelle de spécification des interfaces à la vérification d'implantation de contrôleur d'interfaces. L'auteur modélise l'implantation d'un contrôleur sous forme de machine à états finis. Leur introduction dans le cadre des CSP se fait de façon simple et intuitive.

Certains chercheurs s'intéressent à décrire des systèmes numériques à l'aide des chronogrammes hiérarchiques annotés. Dans [4], l'auteur a développé un outil et une méthodologie permettant la génération de modèles exécutables de système en langage de description de matériel à partir de leur description sous forme de chronogrammes hiérarchiques annotés. Dans [5], l'auteur a rendu le logiciel développé dans [4] utilisable et démontré avec un exemple pratique l'utilisation du logiciel.

1.4 Plan du mémoire

Ce mémoire se compose des quatre chapitres suivants :

Le chapitre 2 présente tout d'abord certains concepts de base qui seront utilisés par la suite. Puis il aborde brièvement l'usage de la programmation logique avec contraintes et de l'arithmétique relationnelle des intervalles pour résoudre des problèmes importants : déterminer la séparation maximum entre les événements ; vérifier la causalité d'un chronogramme et vérifier une implantation d'un contrôleur sous forme de machine à états finis. Il s'agit d'une synthèse des méthodes de vérification temporelle des interfaces développées dans [2] et [3] qui sont considérées comme le point de départ de nos travaux.

Dans le chapitre 3, nous détaillons le développement d'un prototype de vérification temporelle des interfaces sous ILOG Solver ainsi les différentes classes et méthodes

implantées en exploitant les avantages de paradigme orienté objet et ceux de programmation par contraintes fournis par ILOG Solver.

Le chapitre 4 élabore le prototype avec un exemple de vérification sur deux cycles d'écriture consécutifs d'un contrôleur d'interfaces par rapport à sa spécification temporelle sous forme de chronogramme.

Le chapitre 5 conclut cette présentation et indique les travaux futurs possibles.

Chapitre 2
Vérification Temporelle des Interfaces à l'aide de la Programmation Logique avec Contraintes

Dans ce chapitre, nous abordons d'abord les travaux de recherche de P. Girodias présentés dans [2], notamment les méthodes pour déterminer les séparations temporelles d'événements et la causalité d'une spécification sous forme de chronogramme, puis nous présentons la méthode de vérification d'un contrôleur des interfaces développé par F. Jin, présentée dans [3]. Ces méthodes sont déjà mises en œuvre dans un prototype qui sera présenté dans le chapitre suivant.

2.1 Recherche de séparation maximum

Le problème de vérification temporelle des interfaces peut être défini comme un problème de recherche de séparation maximum entre une paire d'événements e_i et e_j dont les temps d'occurrence sont t_i et t_j respectivement

$$d_{ij} = max(t_j - t_i)$$

Les bornes inférieure et supérieure de t_i et t_j sont déterminées par un système de contraintes dont les contraintes linéaires sont de la forme :

$$t_j - t_i \leq \delta_{ij} \tag{2.1}$$

et celles de type non linéaire sont sous la forme :

$$t_j = min_{\ i \in preds(j)} (t_i + \delta_{ij}) \tag{2.2}$$

ou

$$t_j = max_{\ i \in preds(j)} (t_i + \delta_{ij}) \tag{2.3}$$

où : *preds(j)* est l'ensemble des événements affectant le temps d'occurrence de

l'événement e_j et δ_{ij} est une valeur dans l'intervalle temporel associé à la relation temporelle liant les événements e_i et e_j. (voir les définitions 1.5 et 1.6)

Nous abordons brièvement dans cette section la résolution de quatre classes de systèmes de contraintes :

1) les systèmes de contraintes strictement linéaires ;
2) les systèmes de contraintes *max* ou *min*, exclusivement ;
3) les systèmes de contraintes du type *linear* et *max* ou *min* ;
4) les systèmes de contraintes du type *linear*, *max* et *min*.

Nous présentons une approche de résolution de problèmes de séparation temporelle des événements basée sur la résolution des problèmes de satisfaction de contraintes (CSP: Constraint Satisfaction Problem) et les techniques de la programmation logique avec contraintes (CLP: Constraint Logic Programming).

2.1.1 Introduction aux CSP et CLP

Nous présentons dans cette section certaines terminologies utilisées dans les domaines CSP et CLP qui constituent les techniques essentielles exploitées dans notre recherche et développement.

Définition 2.1 : *CSP* . Un *CSP* est un 3-tuple $P(I, D, C)$, où :

- $I = (X_1, X_2, \ldots, X_n)$ un ensemble fini de variables,
- $D = (D_1, D_2, \ldots, D_n)$ où D_i est un domaine fini dans lequel X_i peut prendre ses valeurs,
- $C = (C_1, C_2, \ldots, C_m)$ un ensemble de contraintes. Une contrainte $c(X_1, X_2, \ldots, X_n) \in$ C est un sous-ensemble du produit cartésien $D_1 \times D_2 \times \ldots \times D_n$ qui spécifie quelles valeurs des variables sont compatibles entre elles.

Définition 2.2 : *Consistance d'un CSP*. Un *CSP P*(*I*, *D*, *C*) est *globalement consistant* si et seulement si $\forall X_i \in I$, $\forall a \in D_i$, $X_i = a$ appartient à une solution de *P*. Deux *CSP*s sont *équivalents* si et seulement s'ils possèdent le même ensemble des solutions.

Résoudre un *CSP P* peut être défini comme un processus de recherche d'un *CSP P'* équivalent qui est globalement consistant. Mais la recherche de la consistance globale est un problème *NP*-dur. Donc, les critères moins stricts, c'est-à-dire la consistance partielle sont plus couramment utilisés. Un système de contraintes est *inconsistant* s'il n'admet aucune solution.

Il est clair que la détermination des séparations temporelles entre les événements peut être modélisée comme un *CSP*. En plus, en vérification temporelle, les contraintes entre les événements peuvent être spécifiées comme les intervalles des valeurs. Ainsi les valeurs exactes sont exprimées approximativement par les bornes inférieures et supérieures. Donc, un modèle basé sur l'arithmétique des intervalles semble naturel et ce genre de système de contraintes peut être résolu avec les techniques de résolution de contraints CRT(RIA) (Constraint Resolution Techniques based on Relational Interval Arithmetic) en prenant l'abréviation utilisée dans [2].

2.1.2 CRT(RIA) : Arithmétique relationnelle des intervalles

Un intervalle est un ensemble borné des nombres

$$[a, b] = \{x \mid a \leq x \leq b\}.$$

Les lettres majuscules sont utilisées pour désigner les intervalles. Les deux bornes d'un intervalle X sont notées par x' et x''. Donc, $X = [x', x'']$. L'intervalle $[x, x]$ est un intervalle dégénéré qui est équivalent à la valeur singleton x. Les deux intervalles sont égaux si leurs bornes correspondantes sont égales. Si le nombre x est dans l'intervalle X, on écrit $x \in X$. L'intersection de deux intervalles est définie comme

$$X \cap Y = [max(x', y'), min(x'', y'')].$$

L'addition intervalle est définie comme

$$X + Y = [x', x''] + [y', y''] = [x' + y', x'' + y''].$$

La négation d'un intervalle, duquel les règles pour la soustraction intervalle peuvent être déduites, peut être définie comme

$$-X = -[x', x''] = [-x'', -x'] = \{ -x \mid x \in X \}$$

Le *max* de deux intervalles (le résultat peut être généralisé sur *n* intervalles. Le *min* de deux intervalles est semblable) est défini comme

$$max(X, Y) = [max(x', y'), max(x'', y'')]$$

Approximation et le problème *faux positif*

Quand on utilise CRT(RIA), les contraintes sont exprimées sur les variables sous forme d'intervalles. Les systèmes de contraintes sont résolus par la propagation des contraintes, c'est-à-dire les bornes qui respectent les contraintes sont calculées pour le domaine de chaque variable. Une ou plusieurs variables sont initialisées avec les bornes finies. Les autres variables sont initialisées avec valeur [-∞, ∞]. Un mécanisme dirigé par événement (*event-driven*) sélectionne les contraintes primitives et adapte les valeurs des variables en respectant ces contraintes. Donc, les valeurs finies sont propagées à travers le système de contraintes et remplacent éventuellement les valeurs infinies. L'état stable atteint par ce processus est bien défini et correct dans le sens que toutes les solutions se trouvant dans les intervalles initiaux restent toujours dans les intervalles finaux.

Malheureusement, en général les états stables atteints par ce processus sont peut-être plus larges que nécessaire. Par exemple, considérons le système de contraintes suivant dans lequel X et Y sont des nombres positifs :

$$\begin{cases} X + Y = 2 \\ Y \leq X+1 \\ Y \geq X \end{cases}$$

La résolution de ces inégalités donne $0.5 \leq X \leq 1$ et $1 \leq Y \leq 1.5$. Mais l'arithmétique relationnelle des intervalles donne comme résultat $X = [0, 2]$ et $Y = [0, 2]$. Le mécanisme de propagation sur intervalles ne peut pas garantir pour toutes les valeurs d'un intervalle qu'il existe des valeurs possibles dans les autres intervalles. Donc, il est possible que même les valeurs dans le résultat n'admettent pas une solution, la propagation sur intervalles conduit à une consistance partielle, identifiée comme arc B-consistance.

Définition 2.3 : A *CSP* $P = (I, D, C)$ est arc *B-consistant* si et seulement si $\forall X \in I$, $D_x = [a_x, b_x]$, $\forall\ c(X, X_1, \ldots X_k) \in C$, une contrainte sur X,
$\exists\ (v_1, \ldots, v_k) \in D_1 \times \ldots \times D_k$ tel que $c(a_x, v_1, \ldots, v_k)$ est satisfaite, et
$\exists\ (v_1, \ldots, v_k) \in D_1 \times \ldots \times D_k$ tel que $c(b_x, v_1, \ldots, v_k)$ est satisfaite.

En général, les valeurs retournées par CRT(RIA) sont approximatives. Quand les intervalles de solution sont trouvés, ils peuvent être encore plus larges que nécessaire et le système peut être en effet inconsistant. Ceci est appelé le problème *faux positif.* Seul un échec de retourner un ensemble de solutions satisfaisant, c'est-à-dire une ou plusieurs variables de type intervalle sont égales à un intervalle vide ∅, peut être accepté comme une réponse exacte et définitive : le système de contraintes considéré est inconsistant. C'est la raison pour laquelle quand on veut résoudre un problème avec CRT(RIA), le problème est souvent modélisé par un CSP qui est ensuite prouvé inconsistant.

2.1.3 Problème de séparation des événements en CRT(RIA)

Nous pouvons maintenant exprimer le problème de séparation des événements et leurs solutions en terme de l'arithmétique relationnelle des intervalles et CRT(RIA).

Car les temps d'occurrence des événements sont restreints seulement entre eux et il n'existe pas un temps de référence absolu, donc, il est nécessaire de désigner un événement comme référence pour tous les autres événements et fixer son temps d'occurrence à une constante, par exemple, 0. Les temps d'occurrence des autres événements sont

initialisés comme [-∞, ∞]. Les intervalles de temps d'occurrence calculés par CRT(RIA) sont en effet les séparations dans le temps par rapport l'événement de référence choisi. S'il y a n événements e_i, $1 \leq i \leq n$, la résolution du système de contraintes pour les n points de référence permet de trouver les séparations temporelles entre tous les événements. La méthode de résolution de systèmes de contraintes exploite le concept suivant.

Définition 2.4 : *Projection d'une contrainte.*
Étant donné un système de contraintes défini sur n événements. Soient $T_1, T_2, \ldots, T_n$ les variables de type intervalle représentant les domaines de $t_1, t_2, \ldots, t_n$ respectivement, où t_i est le temps d'occurrence de l'événement e_i avec $1 \leq i \leq n$. Soit $E = T_1 \times T_2 \times \ldots \times T_n$, la projection sur t_i, $(1 \leq i \leq n)$ d'une contrainte $c(t_1, t_2, \ldots, t_n)$ avec les domaines $T_1, T_2, \ldots, T_n$ est l'intervalle $\prod_i(c(t_1, t_2, \ldots, t_n), E) = \{ t_i \in T_i \mid \exists (t_1, t_2, \ldots, t_{i-1}, t_{i+1}, \ldots, t_n) \in T_1 \times T_2 \times \ldots \times T_{i-1} \times T_{i+1} \times \ldots \times T_n$ tels que $c(t_1, t_2, \ldots, t_n)$ est satisfaite$\}$.

Il s'ensuit que la résolution d'un système de contraintes consiste à rétrécir les bornes des domaines T_j $(1 \leq j \leq n)$, donc, le système des équations suivant est satisfait.

$$\forall j, 1 \leq j \leq n, T_j = \cap_{i=1}^{m} \prod_j(c_i(t_1, t_2, \ldots, t_n), E) \qquad (2.5)$$

où m est le nombre de contraintes du système.

Système de contraintes linéaires dans CRT(RIA)

Considérons les temps d'occurrence t_i et t_j des événements e_i et e_j bornés par les contraintes suivantes :

$$t_j - t_i \leq s_{ij} \qquad (2.6)$$

$$t_i - t_j \leq s_{ji} \qquad (2.7)$$

Ces contraintes peuvent être exprimées en CRT(RIA) par une contrainte sur intervalle $c^*(t_i, t_j)$: $t_i - t_j = [-s_{ij}, s_{ji}]$.

Supposons que T_i et T_j sont les domaines de t_i et t_j respectivement, la projection de cette contrainte sur t_i et t_j donne :

$$\prod_i(c^*(t_i, t_j), T_i \times T_j) = T_j + [-s_{ij}, s_{ji}] \cap T_i = \Delta^i_{ij}(T_i, T_j)$$

$$\prod_j(c^*(t_i, t_j), T_i \times T_j) = T_i + [-s_{ji}, s_{ij}] \cap T_j = \Delta^j_{ij}(T_j, T_i)$$

$\Delta^i_{ij}(T_i, T_j)$ est appelé la projection sur t_i de la contrainte définie par les événements e_i et e_j dont les domaines sont T_i et T_j respectivement.

En général, s'il y a n événements, la résolution d'un système de contraintes avec CRT(RIA) consiste à chercher une solution du système des équations dont l'intersection des projections est définie comme suit :

$$(\forall i,\ 1 \le i \le n \quad T_i = \Delta^i_{1i}(T_i, T_1) \cap \Delta^i_{2i}(T_i, T_2) \cap \ldots \cap \Delta^i_{ni}(T_i, T_n)\ ) \qquad (2.8)$$

où $\Delta^i_{ji}(T_i, T_j) = [x'_j - s_{ij}, x''_j + s_{ji}] \cap T_i$ et $\Delta^i_{ii}(T_i, T_i) = Ti$ (car $s_{ii} = 0$) avec $T_i = [x'_i, x''_i]$ et $T_j = [x'_j, x''_j]$.

Le résultat de la fonction $\Delta^i_{ji}(T_i, T_j)$ représente un intervalle le plus large possible pour T_i par rapport à T_j.

Comme ce qu'on a expliqué, un événement e_k est désigné comme référence dont le domaine de temps d'occurrence est restreint dans l'intervalle $T_k = [0, 0]$. Pour tous les autre événements $i \neq k$, leurs domaines initiaux sont $T_i = [-\infty, \infty]$. Le système de contraintes ainsi obtenu doit être consistant (aucun intervalle ne doit être vide après la propagation de ces contraintes) quand n'importe quel événement est choisi comme référence afin d'assurer la consistance du système de contraintes linéaires considéré.

Systèmes de contraintes de type *min* ou *max* exclusif

Considérons la contrainte de type *max* suivante (semblable pour *min*) :

$$T_j = max_{i \in preds(j)}(t_i + \delta_{ij})$$

Où $l_{ij} \leq \delta_{ij} \leq u_{ij}$. l_{ij} et u_{ij} sont les bornes inférieure et supérieure de la relation temporelle entre t_i et t_j. Soit T_i et T_j les domaines de t_i et t_j, les projections de cette contrainte sont

$$\begin{aligned} \prod_j(m(t_1, t_2, \ldots, t_n), E) &= max_{\ i \in preds(j)}\ (T_i + [l_{ij}, u_{ij}]) \cap T_j \\ &= \omega^j_j\ (T_j, T_i, \forall i \in preds(j)\ ) \end{aligned} \tag{2.9}$$

$$\forall i \in \text{ preds}(j),\ \prod_i(m(t_1, t_2, \ldots, t_n), E) = (T_j + [-\infty, -l_{ij}]) \cap T_i = \omega^j_j(T_i, T_j) \tag{2.10}$$

où $m(t_1, t_2, \ldots, t_n)$ est la contrainte du type *max* et $E = T_1 \times T_2 \times \ldots \times T_n$. L'expression $\omega^i_j(T_i, T_j)$ est interprétée comme la projection sur t_i de la contrainte de type *max* définie sur l'événement j avec T_i et T_j comme domaines.

En général, s'il existe n événements, l'intersection des projections définit le système des équations suivant :

$$\forall j,\ 1 \leq j \leq n \quad T_j = \omega^j_j(T_j, T_i, \forall i \in \text{ preds}(j)) \cap_{\ p \in succs(j)} \omega^j_p(T_j, T_p) \tag{2.11}$$

Comme pour les systèmes de contraintes linéaires, le système de contraintes ainsi obtenu doit être consistant pour tous les événements de référence afin d'assurer la consistance du système de contraintes du type *min* ou *max* exclusif.

Systèmes de contraintes de type *linear* et *max*

À partir de deux sous problèmes précédents nous pouvons écrire le système des équations plus général où les contraintes du type linéaire et celles du type *max* sont permises :

$$\begin{aligned} \forall j,\ 1 \leq j \leq n, \quad T_j &= \omega^j_j(T_j, T_i, \forall i \in \text{ preds}(j)) \\ &\cap_{\ p \in succs(j)} \omega^j_p(T_j, T_p) \cap \Delta^j_{qj}(T_j, T_q) \end{aligned} \tag{2.12}$$

où q est l'indice d'un événement quelconque e_q lié à l'événement e_j par une contrainte linéaire, c'est-à-dire $-s_{jq} \leq t_j - t_q \leq s_{qj}$

En général, dans le domaine de la vérification temporelle des interfaces, c'est souvent le cas où les contraintes linéaires et celles de type *min* et *max* se présentent simultanément. Il s'agit d'un problème *NP*-complet. Donc, il n'existe aucun algorithme en temps polynomial pour résoudre ce genre de problème.

Une approche plus simple consiste en une analyse exclusive de cas basée sur les contraintes de type *min* ou *max* (celles moins nombreuses sont choisies) est effectuée. De cette façon, le problème de vérification temporelle des interfaces est donc réduit en un problème de calcul de séparation maximum des événements d'un système de contentes linéaire et *min* ou linéaire et *max*. Le système de contraintes ainsi transformé doit être consistant pour tous les événements de référence pour garantir que le système original est consistant. Nous abordons par la suite la vérification de la consistance et de la causalité d'une spécification temporelle d'interface à l'aide de la résolution d'un problème de séparation d'événements.

2.2 La consistance et la causalité

2.2.1 La consistance d'un chronogramme

Définition 2.5 : *Consistance d'un chronogramme*

Soit t_i le temps d'occurrence d'un événement $e_i \in E$. Une spécification sous forme de chronogramme TD = (S, E, o, C), $| E | = n$ est *faiblement consistant* s'il existe au moins un vecteur $T(t_1, t_2, \ldots, t_n)$ tel que toutes les contraintes temporelles $c_i \in C$ sont satisfaites. Autrement dit, il est possible de calculer les séparations temporelles entre tous les événements $e_i \in E$. Si pour e_i et e_{i+1} deux événements consécutifs d'un même port quelconque, on a toujours $t_{ei+1} - t_{ei} > 0$, on dit le chronogramme TD est *consistant*.

La consistance garantit qu'une spécification ne contient aucune contrainte temporelle contradictoire.

2.2.2 La causalité d'un chronogramme

Les positions des événements dans le temps sont fixées soit par le composant soit par son environnement (les autres composants du système). Si cette décision peut être faite par un composant sans soucis sur les événements futurs déterminés par les autres composants, alors la spécification est causale. La causalité d'une spécification garantie que le composant peut être réalisé indépendamment d'autres composants qui l'interagiront plus tard. Considérons le chronogramme illustré dans la figure 2.1. (extrait de [1])

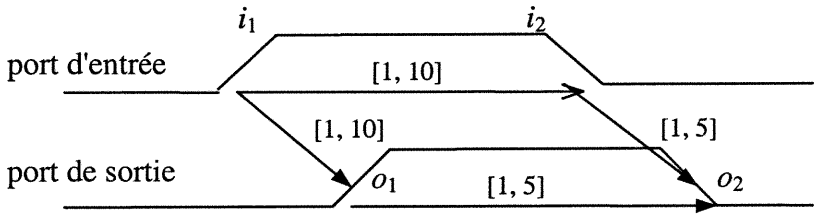


Figure 2.1 Une spécification non-causale

L'ensemble des contraintes est consistant. Quand on implante un composant selon cette spécification, le délai pour l'événement o_1 après l'occurrence de l'événement i_1 doit être choisi dans l'intervalle [1, 10]. Cependant, ce délai dépend du choix de temps d'occurrence de l'événement i_2 qui peut être produit après o_1. Par exemple, si l'on choisit $t_{o1} - t_{i1} = 1$ dans l'implantation, et si i_2 se produit tel que $t_{i2} - t_{i1} \in (5, 10]$, alors il n'existe aucune valeur faisable pour le temps d'occurrence de o_2. L'environnement est obligé de suivre le temps d'occurrence de o_1 et produit i_2 après o_1, mais on n'a aucun contrôle sur l'environnement, d'où le problème.

Symétriquement, il serait possible également pour l'implantation du composant qu'on décide d'attendre i_2 puis produire o_1, par conséquent, on est dans une impasse. Il est clair

que telle spécification n'est pas causale parce que les décisions faites par l'implantation dépendent des événements futurs de l'environnement. En plus, ces décisions ne se trouvent pas dans la spécification elle-même, donc il est impossible d'implanter chaque composant indépendamment et de vérifier la compatibilité de deux composants strictement en se basant sur leur spécifications temporelles sous forme de chronogramme.

Donc, on peut conclure que la consistance seule n'est pas suffisante pour garantir qu'une spécification sera réalisable. On doit prendre en compte la causalité. Dans [1], on propose une description intuitive d'un chronogramme causal :
La décision qu'un événement de sortie (entrée) e_i doit être produit à temps $t(e_i)$ selon les contraintes commit (assume) ne devrait pas dépendre du temps d'occurrence des événements qui seront produits par l'environnement (le composant) en temps $t \geq t(e_i)$.

Ceci indique que dans un chronogramme causal, on peut partitionner l'ensemble des événements dans des blocs de la manière à ce que dans chaque bloc, il soit possible de calculer le temps d'occurrence des événements locaux qui ne dépend que des événements passés se trouvant dans les blocs précédents. Si une telle partition existe, alors le chronogramme a une interprétation causale selon la description intuitive ci-dessus et ce chronogramme est considéré comme réalisable. Un chronogramme et une partition spécifiée de l'ensemble de ses événements définissent une machine qu'on désigne comme bloc-machine.

Définition 2.6 *Bloc-Machine*. Une bloc-machine est un 4-tuples $M = (A, o, B, T)$, où :

1) A est l'ensemble des événements ;
2) est l'événement original, $o \in A$. Soit $A^- = A - \{o\}$;
3) B est l'ensemble de k "blocs" ; Un bloc $B_i = (A_i, C_i) \in B$ où :

   $A_i \subseteq A$, tous les événements de A_i ont la même direction (entrée ou sortie). En plus, $\forall i, j, i \neq j \Rightarrow A_i \cap A_j = \varnothing$;

   $C_i = \{c(x_1, x_2, \ldots, x_m) \in C \mid \exists\, x_j \in B_i, x_j$ est l'événement cible de $c(x_1, x_2, \ldots, x_m)\}$, C est l'ensemble des contraintes temporelles du chronogramme correspondant.

4) T est une relation de déclenchement (*trigger relation*) est sur $A \times B$. Si une paire (e, B) $\in T$, on dit que "e est un déclencheur de B". Un déclencheur est l'événement source d'une ou plusieurs relations temporelles de *préséance*. L'ensemble des déclencheurs d'un bloc B est désigné comme *trigs*(B).

Une bloc-machine est une partition ordonnée des événements et les contraintes d'un chronogramme dans les blocs. Les auteurs dans [8] ont montré que l'existence d'une partition causale des événements d'une spécification est une preuve suffisante pour que la spécification elle-même soit causale.

Définition 2.7 : *Causalité d'un chronogramme*. Un chronogramme est causal s'il existe une bloc-machine telle que :

- Les déclencheurs d'un bloc doivent toujours devancer les événements du bloc qu'ils déclenchent (*well-defined triggers*) ;
- Les séparations temporelles entre les événements déclencheurs déterminées par les contraintes locales d'un bloc doivent être moins serrées que celles produites par les blocs précédents. (*past-dominated machines*). On dit que l'intervalle $\pi_1 = [T1_{min}, T1_{max}]$ est moins serré que l'intervalle $\pi_2 = [T2_{min}, T2_{max}]$ si $T1_{min} \leq T2_{min}$ et $T1_{max} \geq T2_{max}$.

La figure 2.2 illustre un exemple simple d'une spécification non causale. Une partition possible indiquée dans la figure 2.2 consiste en deux blocs, l'un avec les deux événements entrées (A et B), l'autre avec un événement sortie C. Les événements A et B sont les déclencheurs du bloc *2*. La première condition de la causalité est satisfaite car $t_C - t_A \geq 11$ et $t_C - t_B \geq 11$. La seconde condition de la causalité n'est pas satisfaite car la séparation imposée par bloc 1 est $t_B - t_A \in [-10, 10]$ et celle déterminée par le bloc *2* est $t_B - t_A \in [-9, 9]$, donc, cette partition particulière n'est pas causale. En effet, il n'existe aucune partition valide pour cette spécification : la spécification n'est pas causale.

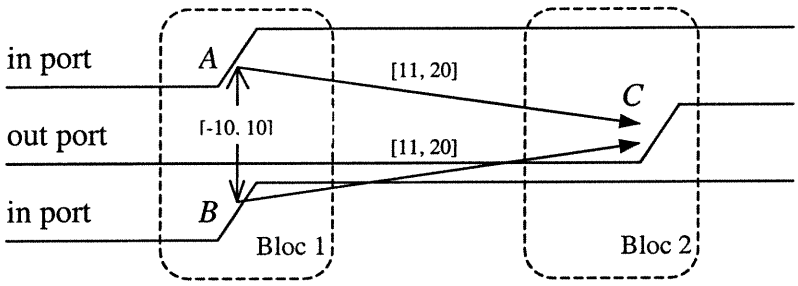


Figure 2.2 Une spécification et une possible partition

2.2.3 Vérification de la causalité en CSP

En effet, une bloc-machine est strictement équivalente à la spécification originale en terme des événements et les relations temporelles. Puisque produire une bloc-machine appropriée dans le but de vérification consiste à simplement générer toutes les partitions possibles des événements jusqu'à ce que la causalité soit prouvée ou il n'y avait plus de partition à produire, on va limiter la discussion à la vérification d'une seule partition de bloc-machine ayant m événements partitionnés dans n blocs.

Vu le raisonnement présenté dans la section 2.1.2, pour prouver que cette partition de bloc-machine est causale, nous considérons l'inverse du problème : on construit un CSP qui sera inconsistant si la bloc-machine est causale.

Les blocs de la bloc-machine sont des sous-ensembles de la spécification initiale sous forme de chronogramme. Soit $B_i = (A_i, C_i)$ une partie de chronogramme correspondant à $i^{ème}$ bloc. Nous supposons aussi, sans perte de généralité, que les blocs sont ordonnés de façon suivante :

$$(a \in A_i, a \in trigs(B_j)) \Leftrightarrow B_i < B_j.$$

où $trigs(B_k)$ est l'ensemble des déclencheurs du bloc B_k et "<" est une relation d'ordre. Cette supposition doit être satisfaite parce que les contraintes de déclenchement

(contraintes ayant les déclencheurs comme les événements source) définissent les relations de *préséance* entre les blocs. La première condition de la causalité (*well-defined triggers*) garantit qu'il n'y a pas de cycles dans la partition des blocs selon la relation d'ordre. Il faut noter que si la bloc-machine est construite correctement, alors, $\forall i$, $c_i \in C$ ne contient qu'un seul type de contraintes (*assume* ou *commit*).

On définit une série de CSPs $P_i = (I_i, D_i, C_i)$, où :

- $t_i \in I_i \Leftrightarrow a_i \in (B_1 \cup B_2 \cup \ldots \cup B_i)$,
- $C_i = C_1 \cup C_2 \cup \ldots \cup C_i$.

Chaque CSP P_i représente le bloc B_i et ses prédécesseurs.

Première condition : déclencheurs bien définis (*well-defned triggers*)

Pour que la première condition de la causalité soit satisfaite, les déclencheurs doivent précéder les événements du bloc qu'ils déclenchent. À cette fin, on construit une série des CSPs qui sont consistant seulement si les événements déclencheurs peuvent actuellement être produits après les événements déclenchés. Chaque paire (x, y) événement déclencheur et événement déclenché doit être prise en compte individuellement. Dans tous les cas, on force le déclencheur x à apparaître après l'événement déclenché y. Si tous les déclencheurs de la bloc-machine sont bien définis, la condition suivante doit être satisfaite :

$$\forall i, \forall x \in \text{trigs}(B_i), \forall y \in E_i, P_i \cup \{t_x \geq t_y\} \models \varnothing.$$

Autrement dit, une inconsistance reflète un fait : un déclencheur ne peut pas apparaître après un événement appartenant au bloc qu'il déclenche. Il faut noter que la définition ci-dessus implique que $P_i \subseteq P_{i+1}$, qu'il reflète le fait que seuls les événements ayant déjà apparu dans le passé puissent être utilisés pour déterminer les positions des événements suivants.

Deuxième condition : "*past-dominated*" blocs

La seconde condition de la causalité, "*past-dominated*" blocs, peut être vérifiée de la manière similaire. On analyse individuellement chaque paire des événements qui déclenchent le même bloc. Les contraintes locales du bloc ne doivent pas restreindre la séparation entre ces déclencheurs. Au niveau d'implantation de la méthode, les paires des déclencheurs (x, y) peuvent être étiquetées de nouveau dans les contraintes du bloc déclenché afin de les distinguer de leurs temps d'occurrence déterminés par les blocs précédents. Puis on construit une série de CSPs dans lesquels on force les séparations $(t_x - t_y)$ et $(t_y - t_x)$ entre les deux déclencheurs x et y déterminées par les blocs précédents à être plus larges que $(t_{x'} - t_{y'})$ et $(t_{y'} - t_{x'})$ déterminées par le bloc déclenché.

Plus précisément, soit x/x' la substitution de x par x', pour toutes les paires de déclencheurs du B_i, $x, y \in \text{trigs}(B_i)$, $x \neq y$, on définit une série de CSPs $P_{ixy}(I_{ixy}, D_{ixy}, C_{ixy})$, où :

- $I_{ixy} = I_i \cup \{t_{x'}, t_{y'}\}$
- $D_{ixy} = D_i \cup \{D_{x'}, D_{y'}\}$, $D_{x'} = D_x$, $D_{y'} = D_y$,
- $C_{ixy} = C_{i-1} \cup \{(t_x - t_y) \geq (t_{x'} - t_{y'}) \mid (t_y - t_x) \geq (t_{y'} - t_{x'})\} \cup$
  $\{c(t_1, t_2, \ldots, t_x/t_{x'}, \ldots, t_m) \mid c(t_1, t_2, \ldots, t_x, \ldots, t_m) \in C_i\} \cup$
  $\{c(t_1, t_2, \ldots, t_y/t_{y'}, \ldots, t_m) \mid c(t_1, t_2, \ldots, t_y, \ldots, t_m) \in C_i\} \cup$
  $C_i \setminus \{ c(t_1, t_2, \ldots, t_k, \ldots, t_m) \in C_i \mid k = x \text{ ou } k = y\}$.

Si les blocs de la bloc-machine sont "*past-dominated*", on peut obtenir le résultat :

$$\forall i, \forall x, y \in \text{trigs}(B_i), P_{ixy} \models \varnothing \qquad (2.13)$$

Autrement dit, si x et y sont les déclencheurs d'un bloc, une inconsistance reflète le fait que la séparation temporelle spécifiée par ce bloc entre ces deux événements soit plus large que celle spécifiée par tous les blocs précédents.

Dans la section suivante, on présente une méthode de vérification de contrôleurs d'interfaces [3] qui exploite les méthodes présentées dans les sections 2.1 et 2.2 pour déterminer les séparations temporelles maximum entre les événements et la causalité d'une spécification sous forme de chronogramme.

2.3 Vérification de contrôleurs d'interfaces

Dans cette session, nous nous intéressons aux problèmes de vérification temporelle des interfaces qui ne se contentent pas seulement de vérifier la spécification d'interfaces, mais veulent mettre en relation la spécification et l'implantation d'un contrôleur d'interfaces. Ce genre de problème de vérification est composé d'une spécification sous forme de chronogramme et d'une implantation correspondant sous forme d'une machine à états finis (FSM : Finite States Machine).

2.3.1 Contrôleurs pseudo-synchrones

Un contrôleur pseudo-synchrone est implanté comme une FSM synchrone (circuit séquentiel synchrone). Il est composé d'un circuit combinatoire et des registres contenant les valeurs des variables d'états. (Voir figure 2.3). Le circuit combinatoire est alimenté par des signaux entrées primaires synchronisée (PI: Primary Inputs) et les sorties des registres d'état. On suppose qu'il y a pour chaque signal sortie du circuit combinatoire, un registre qui filtre les bruits possibles, donc, le FSM peut être considéré comme une machine *Moore*. On suppose aussi que toutes les bascules sont contrôlées par la même horloge et que la fréquence de l'horloge et les délais du circuit assurent le fonctionnement correct de toutes les bascules. On va étudier une FSM du type *Mealy* avec un exemple d'application dans le chapitre 4.

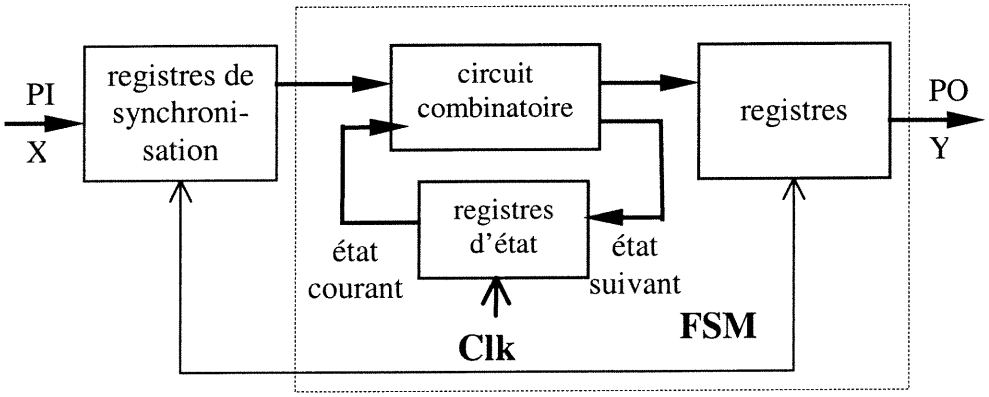


Figure 2.3 Implantation d'un contrôleur pseudo-synchrone

Nous prenons, comme exemple par la suite, la machine Moore ci-dessous.

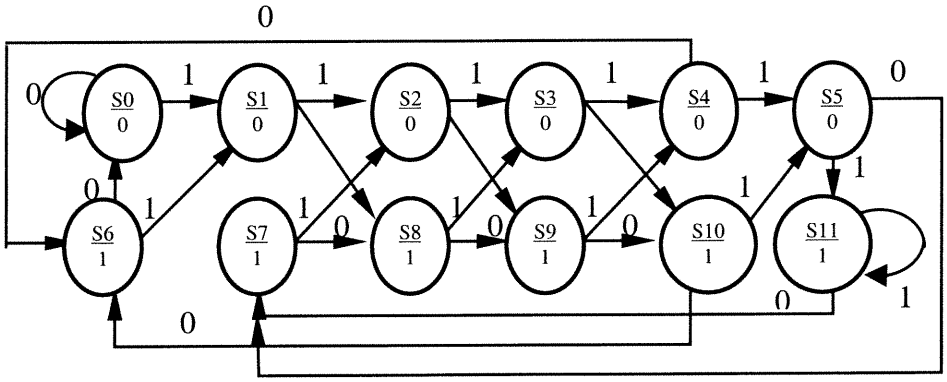


Figure 2.4 Un exemple de FSM Moore

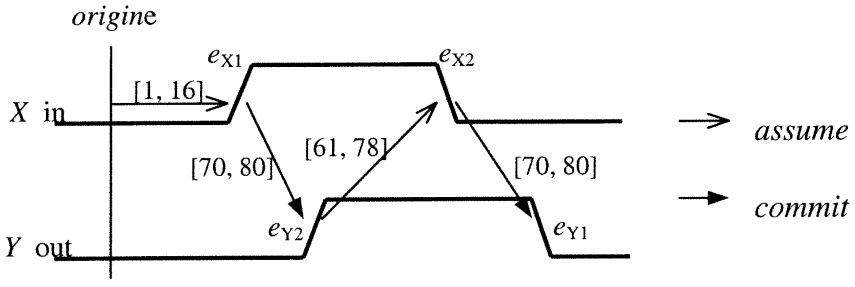


Figure 2.5 Une spécification en forme de chronogramme pour la FSM de la Figure 2.4

La figure 2.5 illustre un chronogramme qui décrit la spécification de la FSM de la figure 2.4. Pour ce chronogramme, l'ensemble des ports est $W = \{X, Y\}$, l'ensemble des événements est $E = \{e_{X1}, e_{X2}, e_{Y1}, e_{Y2}\}$, l'ensemble des contraintes est $C = \{(origine, e_{X1}, [1, 16]), (e_{Y1}, e_{X2}, [61, 78]), (e_{X1}, e_{Y1}, [70, 80]), (e_{X2}, e_{Y2}, [70, 80])\}$. L'origine indique l'état initial. On suppose que $t_{origine} = 0$.

2.3.2 Définition du problème

On constate que la spécification sous forme de chronogramme est basée sur des événements, mais le circuit séquentiel pseudo-synchrone fonctionne au niveau de signaux. Donc, on a besoin d'établir une relation entre ces deux représentations. Pour ce faire, on a besoin d'introduire certaines définitions.

Définition 2.8 : Une *onde* (*waveform*) peut être vue comme un vecteur $WV = (wv_0, wv_1, \ldots, wv_m)$ dont les valeurs $wv_i \in B$ (0 ou 1) telles que le changement de wv_i à wv_{i+1} indique le $i^{ème}$ événement sur cette onde, wv_0 est la valeur initiale à l'origine.

Définition 2.9 : *Automate de transition d'une onde* (*Waveform Transition Automaton*) $WTA = (V, E, G, v_0, v_f)$ est un 5-tuple, où :

- $V = \{v_i\} = \{0, 1, \ldots, m\}$ l'ensemble des états correspondant aux index de WV ;
- $E = \{e_1, e_2, \ldots, e_m\}$ l'ensemble d'événements, $e_i \in E$ apparaît au moment t_i ;
- $G : V \times E \rightarrow V$ fonction de transition définie comme $G(i, e_{i+1}) = i+1$, $0 \leq i \leq m$;
- $v_0 = 0 \in V$ l'état initial ;
- $v_f = m \in V$ l'état final.
-

Quand *WTA* est dans l'état i, la valeur de l'onde est wv_i.

Définition 2.10. *Automate de transition d'un chronogramme* (*Timing Diagram Transition Automaton*) $TDTA = (V, E, G, v_0, v_f)$ est défini comme le produit cartésien des *WTA*s spécifiés dans le chronogramme. Sans perte de généralité, considérons un chrono-

gramme avec deux automates de transition d'onde $WTA_1 = (V_1, E_1, G_1, 0, m_1)$, $WTA_2 = (V_2, E_2, G_2, 0, m_2)$, $TDTA = WTA_1 \times WTA_2$ est défini comme :

- $V = V_1 \times V_2$;
- $E = E_1 \cup E_2$;
- $G : V \times E \rightarrow V$ fonction de transition définie comme

$$G((i,j), e_{1,i+1}) = (i+1, j) \text{ ssi } G_1(i, e_{1,i+1}) = i+1 \text{ dans } WTA_1,$$

$$G((i,j), e_{2,j+1}) = (i, j+1) \text{ ssi } G_2(j, e_{2,j+1}) = j+1 \text{ dans } WTA_2 ;$$

- $v_0 = (0,0) \in V$ est l'état initial ;
- $v_f = (m_1, m_2) \in V$ est l'état final.

L'ensemble d'événements peut être partitionné en E_{IN} et E_{OUT} de manière à ce que tous les événements d'entrée sont dans E_{IN} et tous les événements de sortie sont dans E_{OUT}. Le produit de ces deux *WTA* forme le *TDTA* correspondant. En général, on ne construit jamais explicitement un *TDTA*, par contre on décrit leur comportement par l'exécution concurrente et synchronisée de ses *WTA* constituants en imposant des contraintes temporelles sur le temps d'occurrence des événements.

Définition 2.11. une *trace temporisée (timed) d'un TDTA* $tr_{TD} = ((e_1, t_1), (e_2, t_2), \ldots, (e_n, t_n))$ est une séquence de paires qui groupe chaque événement e_i et son temps d'occurrence t_i, telle que

$$v_0 \xrightarrow{e_1} v_1 \xrightarrow{e_2} v_2 \rightarrow \ldots \xrightarrow{e_n} v_n,\ v_i \in V \text{ et } G(v_i, e_{i+1}) = v_{i+1},\ t_{i+1} \geq t_i,\ 0 \leq i \leq n,\ v_n = v_f \text{ et}$$

$t_1, \ldots, t_n$ satisfont toutes les contraintes de C, l'ensemble des contraintes du chronogramme TD.

On utilise $TR_{TD} = \{tr_{TD}\}$ pour désigner l'ensemble de toutes les traces temporisées d'un *TDTA*. Par exemple une trace temporisée de *TDTA* de l'exemple dans la figure 2.5 peut être la suivante :

$$tr = ((e_{X1}, 10), (e_{Y1}, 80), (e_{X2}, 150), (e_{Y2}, 220)).$$

On va lier les temps d'occurrence des événements t_i, $i = 1, \ldots, n$ à $i^{ème}$ transition sur le *WTA* correspondant afin d'obtenir les valeurs d'onde que le contrôleur a besoin comme signal d'entrée et de délimiter le temps d'occurrence déterminé par les contraintes sur t_i.

Définition 2.12 Un *contrôleur FSM Moore* implantant un protocole spécifié par un chronogramme est défini comme $M = (S, WV_{IN}, N, O, s_0)$, où :

- S : l'ensemble d'états ;
- WV_{IN} : un vecteur des valeurs d'ondes entrées du *TDTA* associé ;
- $N : S \times WV_{IN} \rightarrow S$, fonction de transition d'états ;
- $O : S \rightarrow WV_{OUT}$ fonction de sorties ;
- $s_0 \in S$: l'état initial.

Étant donnée la période P de l'horloge du contrôleur M, un événement d'entrée $e_i \in E_{IN}$ est capturé par M grâce à la transition $s_j \rightarrow s_{j+1}$ si $wv_{j-1} \neq wv_j$. Le temps d'occurrence de e_i observé par M est donc $\tau_i = (j+1)*P+ \delta$. Un événement de sortie $e_o \in E_{OUT}$ est produit dans l'état s_{j+1} si $O(s_j) \neq O(s_{j+1})$. Le moment où e_o est produit est $\tau_o = (j+1)*P+ \delta$. δ est le délais de décalage de phase de l'horloge par rapport à l'origine. Par la suite, on ignore le délai δ.

On dit qu'un contrôleur M accepte des événements d'un *TDTA* si le contrôleur peut détecter tous les événements d'entrée de E_{IN} et observer tous les événements de sortie de E_{OUT} pour le TDTA considéré.

Définition 2.13 Une *trace temporisée* (*timed*) $tr = ((e_1, \xi_1), (e_2, \xi_2), \ldots, (e_n, \xi_n))$ *d'un contrôleur M* qui accepte les événements d'un *TDTA* est une séquence de paires (e_i, ξ_i), $\xi_{i+1} \geq \xi_i$, $i>0$, telle que si e_i est un événement d'entrée, alors $\xi_i = t_i$, le temps d'occurrence de e_i ; si e_i est un événement de sortie, alors $\xi_i = \tau_i$, le temps où e_i est produit par le contrôleur comme la conséquence des événements d'entrée observés.

On désigne $TR_M = \{tr\}$ comme l'ensemble possible des traces temporisées d'un contrôleur *M*. La différence entre TR_M et TR_{TD} est que le temps d'occurrence des événements de sortie dans TR_M sont ceux produits par *M* avec des signaux d'entrée échantillonnés du chronogramme. Maintenant, on peut décrire le problème de vérification de contrôleurs d'interfaces comme suit :

Étant donné une spécification sous forme de chronogramme TD et une implantation sous forme de FSM d'un contrôleur M correspondant qui prend les signaux d'entrée de TD comme signaux d'entrée, est-ce que l'ensemble de traces temporisées produites par M est inclus dans l'ensemble de traces temporisées spécifié par TD? C'est-à-dire $TR_M \subseteq TR_{TD}$ *?*

2.3.3 Méthode de vérification

En utilisant les techniques de programmation par contraintes, TR_M et TR_{TD} peuvent être caractérisés par un ensemble de contraintes dérivé de la fonction de transition du contrôleur, par l'identification des événements quand les *WTA*s changent leur états et par des contraintes temporelles spécifiées dans le *TD*. Les deux ensembles de traces TR_M et TR_{TD} partagent les contraintes *assume* de *TD*. Pour vérifier si $TR_M \subseteq TR_{TD}$, on vérifie si le temps où un événement sortie est produit par *M* se trouve dans l'intervalle spécifié par les contraintes *commit* de *TD*. Autrement dit, on vérifie si le système de contraintes caractérisant TR_M et le complément des contraintes *commit* de *TD* constitue un système inconsistant.

Modélisation de l'ensemble de traces temporisées TR_{TD}

Selon la définition 2.11, TR_{TD} est produit en générant tous les événements de *TDTA* et en imposant les contraintes temporelles sur le temps d'occurrence des événements. Pour décrire la génération des événements, on explique d'abord l'exécution d'un *WTA* et d'une *TDTA*.

Une exécution de *WTA* commence à l'état 0 avec $t_{origine} = 0$. Le temps passe continuellement. Dans le cycle de l'horloge *Clk*, c'est-à-dire, l'intervalle $[Clk \times P, (Clk+1) \times P]$, *WTA* produit une transition de v_{i-1} à v_i si e_i apparaît, c'est-à-dire $t_i \in [Clk \times P, (Clk+1) \times P]$, sinon *WTA* reste à v_{i-1}. Une exécution $\sigma = v_0, v_1, \ldots, v_k$ de *WTA* dans k cycles avec une séquence d'événements $e_0, e_1, \ldots, e_m$, $e_i \in E$, $0 \leq i \leq k$ est acceptée par *WTA* si $v_k = m$.

Une exécution de *TDTA* est acceptée dans k cycles si l'exécution de chaque WTA_i ($0 \leq i \leq w$) est acceptée. Donc, on peut extraire une trace temporisée $tr_{TD} = \{(e_1, t_1), \ldots, (e_n, t_n)\}$ à partir d'une exécution de *TDTA*.

L'exécution d'une *WTA* au cycle *Clk* peut être représentée par les contraintes suivantes :

$$\{(t_i < (Clk+1) \times P) \wedge (t_i \geq Clk \times P) \wedge (vp = i\text{-}1) \rightarrow (vn = i)\} \wedge$$
$$\{\sim((t_i < (Clk+1) \times P) \wedge (t_i \geq Clk \times P)) \wedge (vp = i\text{-}1) \rightarrow (vn = i\text{-}1)\} \qquad (2.14)$$

où *vp*, *vn* sont l'état courant et l'état suivant du *WTA* respectivement et '$\rightarrow$' représente une implication.

Le système de contraintes dont les solutions sont les traces dans TR_{TD} peut être construit de la manière suivante :

1) initialiser $t_{origine} = 0$, $Clk = 0$, $vp_1 = 0, \ldots, vp_w = 0$, où vp_i, $1 \leq i \leq w$, sont les états courants de *WTA* et w est le nombre des ondes dans *TD* ;
2) dans le cycle Clk, si Clk < k, inclure une contrainte de la forme (2.14) pour chaque événement, sinon Clk = k, $vp_1 = m_1, \ldots, vp_w = m_w$, où $m_1, \ldots, m_w$ sont les états finaux, aller à 4) ;
3) Incrémenter Clk = Clk +1, assigner $vp_i = vn_i$, prendre les valeurs des états suivants des *WTA*s obtenues dans le cycle précédent comme celles des états courants, aller à l'étape 2) ;
4) Inclure les contraintes temporelles de la forme : $l_{ij} \leq t_j - t_i \leq u_{ij}$.

Modélisation de l'ensemble de traces temporisées TR_M

On peut obtenir le temps d'occurrence des événements d'entrée observé par M de la façon suivante :

$$\forall\, e_i \in E_{IN},\ (t_i < Clk \times P) \wedge (t_i \geq (Clk\text{-}1)\times P) \rightarrow (\tau_i = Clk \times P) \tag{2.15}$$

En effet, dans un cycle d'horloge *Clk*, si le temps d'occurrence t_i d'un événement d'entrée e_i satisfait $t_i \cap [(Clk\text{-}1)\times P,\ Clk\times P)] \neq \varnothing$, alors on doit énumérer les deux possibilités d'occurrence pour cet événement dans le cycle *Clk*. Si t_i couvre plusieurs cycles d'horloge, alors on doit énumérer toutes les possibilités d'occurrence pour cet événement dans ces cycles d'horloge afin d'assurer la consistance globale de la solution. A cette fin, au niveau d'implantation de la méthode, on introduit une variable booléenne $C_{i,Clk}$ pour indiquer l'occurrence de $e_i \in E_{IN}$ dans le cycle *Clk* de manière suivante :

$$C_{i,Clk} = (t_i < (Clk \times P)) \wedge (t_i \geq (Clk\text{-}1)\times P) \tag{2.16}$$

Les fonctions de transition et celles de sortie de contrôleur sont modélisées sous forme de contraintes. Par exemple, les contraintes suivantes expriment la fonction de transition de *FSM* de la figure 2.4. *SP* et *SN* sont l'état courant et l'état suivant de *M* respectivement.

$$\begin{aligned}
&\{(SP = 0) \wedge (input_value = 0) \rightarrow (SN = 0)\} \wedge \\
&\{(SP = 0) \wedge (input_value = 1) \rightarrow (SN = 1)\} \wedge \\
&\qquad \ldots \\
&\{(SP = 11) \wedge (input_value = 0) \rightarrow (SN = 7)\} \wedge \\
&\{(SP = 11) \wedge (input_value = 1) \rightarrow (SN = 11)\}.
\end{aligned} \tag{2.17}$$

Les valeurs d'entrée sont liées directement aux états de *WTA* des ondes d'entrée.

$$\{(XP \neq 1) \rightarrow (input\_value = 0)\} \wedge \{(XP = 1) \rightarrow (input\_value = 1)\}. \tag{2.18}$$

Les sorties de *M* sont déterminées par les états courants. Pour l'exemple dans la figure 2.5, les contraintes décrivant la fonction de sortie sont les suivantes :

$$\{((SP \geq 0) \wedge (SP \leq 5)) \rightarrow (output\_value = 0)\} \wedge$$
$$((SP \geq 6) \wedge (SP \leq 11)) \rightarrow (output\_value = 1)\}. \quad (2.19)$$

Un événement de sortie est produit si la valeur de sortie de l'état courant est différente de celle de l'état précédent. Les contraintes suivantes décrivent la production des événements de sortie pour notre exemple :

$$\{((YP = 0) \wedge (YN = 1)) \rightarrow (\tau_1 = (Clk+1)*P)\} \wedge$$
$$\{((YP = 1) \wedge (YN = 2)) \rightarrow (\tau_2 = (Clk+1)*P)\}. \quad (2.20)$$

Les états suivants de *WTA* de sortie *YN* (après les registres de sortie) peuvent être déterminés à partir de *YP* et *output_value* obtenue selon (2.19).

$$\{((YP =0) \wedge (output\_value =0)) \rightarrow (YN = 0)\} \wedge$$
$$\{((YP =0) \wedge (output\_value =1)) \rightarrow (YN = 1)\} \wedge$$
$$\ldots$$
$$\{((YP =2) \wedge (output\_value =2)) \rightarrow (YN = 2)\}. \quad (2.21)$$

En résumé, les contraintes caractérisant TR_M de notre exemple peuvent être générées de la façon suivante :

1) Initialiser $t_{origine} = 0$, $Clk = 0$, $XP = 0$, $YP = 0$, $SP = 0$, $input\_value = 0$;
2) Dans le cycle *Clk*, faire
   - Si $Clk < k$, alors ajoute les contraintes sur les valeurs d'entrée (2.18), les contraintes représentant les états suivants de FSM (2.17), les contraintes qui expriment les fonctions de sortie (2.19) et la détection des événements de sortie (2.20) (2.21) ;
   - Si $Clk = k$, alors tous les *WTAs* arrivent à leurs états finaux, aller à l'étape 4) ;
3) Incrémenter $Clk=Clk+1$, assigner *XP*, *YP*, *SP* avec *XN*, *YN*, *SN*, répéter l'étape 2) ;
4) Inclure les contraintes *assume* sous forme : $l_{ij} \leq t_j - t_i \leq u_{ij}$.

On doit vérifier les traces de TR_M sur un nombre de cycles k suffisamment grand (noté par *Maxcycle*) dans lequel tous les événements de *TD* peuvent être observés (à l'entrée) ou produits (à la sortie) par le contrôleur *M*. *Maxcycle* peut être déterminé en trouvant la

séparation temporelle maximum entre l'origine et le dernier événement dans *TD* selon la méthode présentée dans la section 2.1. L'ensemble des traces TR_M est donc caractérisé par un système de contraintes construit selon les 4 étapes ci-dessus en remplaçant *k* par *Maxcycle*.

2.3.4 Procédure de vérification

On vérifie $TR \subseteq TR_{TD}$ de la façon suivante : pour chaque contrainte *commit* spécifiée dans le chronogramme, on exécute la procédure suivante :

1) Calculer Maxcycle de l'événement cible de cette contrainte ;
2) Inclure le complément de cette contrainte et les contraintes *assume* associées;
3) Inclure les contraintes caractérisant TR_M sur le nombre *Maxcycle* de cycles ;
4) Vérifier si le système obtenu est inconsistant.

Les contraintes *commit* vérifiées peuvent être considérées comme des contraintes redondantes afin d'accélérer la vérification des contraintes suivantes. Si pour toutes les contraintes *commit* du chronogramme, la procédure précédente déduit toujours un système inconsistant, alors, on peut conclure que $TR_M \subseteq TR_{TD}$.

2.4 Conclusions

Dans ce chapitre, nous avons présenté certains travaux de recherche qui ont été développés dans [2] et [3]. Ces travaux s'intéressent à la vérification temporelle des interfaces à l'aide des techniques de programmation logique avec contraintes. On voit bien que ces techniques fournissent les moyens unifiés et efficaces pour résoudre des problèmes importants dans le domaine de la vérification temporelle des interfaces. Dans le chapitre suivant, nous mettons en œuvre ces méthodes pour réaliser un prototype de vérification temporelle des interfaces dans un environnement de développement ILOG Solver qui favorise l'intégration du paradigme programmation par contrainte et du paradigme orienté objet.

Chapitre 3 Un outil de vérification temporelle de contrôleurs d'interfaces basé sur CLP

Dans ce chapitre, nous présentons un outil de vérification temporelle de contrôleurs d'interfaces développé sous ILOG Solver[7], un produit important de la compagnie ILOG fondée en 1987 qui est un leader fournisseur des logiciels avancés.

Rappelons que pour la vérification de contrôleurs d'interfaces, nous avons besoin de deux objets : une spécification temporelle d'un contrôleur d'interfaces sous forme de chronogramme TD et une description de l'implantation du contrôleur sous forme de machine à états finis FSM. Donc, en terme de programmation orienté objet, notre première tâche est d'implanter deux classes : l'une appelée *TimingDiagram* permettant de définir un chronogramme et l'autre appelée *FSM* permettant de décrire l'implantation d'un contrôleur d'interfaces. Étant donnée *td*, une instance de la classe *TimingDiagram*, et *fsm*, une instance de la classe *FSM*, l'outil doit fournir les trois fonctionnalités suivantes :

1) Calculer la séparation temporelle maximum de tous les événements de *td* par rapport à un événement de référence ;
2) Vérifier la causalité de *td* en cherchant une bloc-machine causale ;
3) Vérifier si *fsm* satisfait à toutes les contraintes *commit* spécifiées dans *td*.

Nous expliquons comment l'outil peut remplir ces tâches en utilisant les fonctionnalités fournies par ILOG Solver. Nous donnons d'abord dans la section 3.1 une introduction de ILOG Solver pour montrer comment modéliser les inconnus d'un problème à l'aide de différents types de variables et comment introduire les contraintes sur ces variables et comment chercher une solution du problème à l'aide de classes définies dans le Solver.

Nous présentons les classes et les méthodes définies pour remplir ces trois fonctionnalités ci-dessus dans les sections 3.2, 3.3 et 3.4 respectivement. Dans les sections 3.5 et 3.6, nous expliquons la saisie d'un chronogramme et d'une machine à états finis en donnant la syntaxe des fichiers.

3.1 Introduction à ILOG Solver

ILOG Solver est un système de programmation par contraintes développé en C++. Il s'agit d'une librairie de classes et de fonctions prédéfinies permettant d'appliquer le principe de CLP pour modéliser et résoudre des problèmes combinatoires. Développé et commercialisé depuis plus de 10 ans, ILOG Solver est capable de générer efficacement les solutions fiables (ou une solution optimale par rapport à une fonction objective) qui satisfont toutes les contraintes d'un problème. ILOG Solver possède une architecture unique en trois couches :

1) modélisation puissante et intuitive pour formuler un problème ;
2) contrôle intelligent permettant à l'utilisateur de définir leurs propres stratégies de recherche ;
3) les algorithmes efficaces de recherche de solutions délivrant rapidement les solutions fiables.

L'efficacité de la programmation par contraintes réside dans le fait qu'il détache la représentation du problème des algorithmes utilisés pour le résoudre. Dans ce sens, la programmation par contraintes peut être considérée comme la recherche opérationnelle déclarative. Donc, la résolution d'un problème avec la programmation par contraintes consiste en deux activités relativement distinctes :

La représentation d'un problème : La représentation d'un problème consiste à déclarer les variables et les contraintes du problème. ILOG Solver profite de l'avantage du paradigme orienté objet de C++ en fournissant un ensemble de classes pour faciliter cette activité.

La recherche de solutions : La résolution d'un problème consiste à sélectionner une valeur dans le domaine de chaque variable de manière à satisfaire toutes les contraintes. Pour des problèmes simples, il suffit d'appliquer les algorithmes génériques prédéfinis dans la librairie de ILOG Solver. Pour des problèmes plus compliqués, le Solver permet aussi de définir les nouveaux algorithmes en utilisant la programmation par *but*. De plus, ILOG Solver peut être utilisé pour chercher une solution qui optimise un critère donné.

Par la suite, nous présentons les classes et fonctions principales de ILOG Solver Version 4 (appelé Solver par la suite.) utilisées dans le développement de notre outil.

3.1.1 IlcManager

IlcManager est considéré comme une des classes les plus importantes dans le Solver. Quand une instance de *IlcManager* est crée, elle initialise les données internes pour le Solver et considérée comme un gestionnaire qui se charge de communication de données, de gestion de mémoire et d'autres services généraux pour toutes les variables, les contraintes et les buts (voir les classes suivantes) associés à cette instance. Pour une application donnée, on peut créer et exploiter autant de gestionnaires qu'on veut, mais une variable quelconque appartient à une instance unique de *IlcManager*. Autrement dit, les gestionnaires ne partagent ni les variables ni les contraintes entre eux.

On montre ici une définition simplifiée de cette classe.

```
class IlcManager {
  public:
      IlcManager(IlcEditMode editMode);
      void add(IlcConstraint& ct) const;
      void remove(IlcConstraint& ct) const;
      void add(IlcGoal& goal) const;
      void remove(IlcGoal* goal) const;
      IlcBool nextSolution() const;
      IlcBool solve(IlcGoal g, IlcBool restore = IlcFalse) const;
```

```
        void restart();
        void end();
        void fail(IlcAny label =0) const;
        void setDefaultPrecision(IlcFloat precision) const;
        void setObjMin(IlcIntVar& obj, IlcInt step = 1);
        void printInformation(ostream& stream) const;
}
```

La première étape d'utilisation de Solver est de créer un gestionnaire avec un constructeur de *IlcManager* : *IlcManager m(IlcEdit)* ou *IlcManager m(IlcNoEdit).*

A tout l'instant, un gestionnaire *IlcManager m* se trouve dans un de deux modes : mode d'édition *IlcEdit* et mode de recherche *IlcNoEdit*. Quand *m* est en mode d'édition, aucune contrainte n'est propagée, donc, on peut ajouter (*m.add(...)*) ou retirer (*m.remove(...)*) les contraintes ou buts librement. Le gestionnaire *m* change son mode d'édition en celui de recherche en appelant la méthode *m.nextSolution()* qui déclenche la recherche des solutions. *IlcNoEdit* signifie que le gestionnaire *m* est en mode de recherche et il va toujours rester dans ce mode. Par conséquent, les variables et les contraintes sont propagées dès qu'elles sont ajoutées.

Quand on veut créer une variable, on doit indiquer à quel gestionnaire elle appartient. Par exemple, après avoir créé un gestionnaire avec *IlcManager m(IlcNoEdit),* une variable composé de nombres entiers de *1* à *10* peut être construite de manière suivante :

IlcIntVar varint(m, 1, 10);

On peut terminer une session d'édition et commencer une autre avec un même gestionnaire *m* en appelant *m.restart*(). Cette méthode efface la solution trouvée, mais elle maintient les variables et les contraintes ajoutées au gestionnaire. Donc, on peut modifier le modèle du problème à résoudre avant de lancer une autre recherche de solutions. Autrement dit, on peut réutiliser ce même gestionnaire dans une autre session d'édition.

Il y a deux méthodes dans le Solver qui permettent de contrôler l'exécution des buts (voir la section 3.1.4) : *m.nextSolution()* et *m.solve(...)*. La deuxième *m.solve(...)* est utile quand on peut chercher une solution à l'intérieur d'un but puisque la méthode *m.nextSolution()* ne peut pas être appelée dans un but. La méthode *m.setObjMin(...)* permet d'exprimer une fonction objective pour chercher une solution optimale. La méthode *m.setDefaultPrecision(...)* définit la précision par défaut qui sera utilisée pour toutes les expressions du type flottant sans précision. Cette précision ne peut pas être inférieure à 2.10^{-11}.

Quand on appelle la méthode *m.end*(), elle efface toutes les données internes et l'allocation des mémoires associées au gestionnaire *m*, y compris les variables et les contraintes associées à ce gestionnaire. Avant de quitter une application, on doit appeler *IlcManager::end()* pour tous les gestionnaires utilisés. Après cette suppression, on ne peut plus référer ni aux gestionnaires effacés ni aux variables ou aux contraintes ajoutées.

3.1.2 Les variables

ILOG Solver fournit un ensemble de classes prédéfinies pour modéliser les inconnus d'un problème à résoudre. Il permet de créer les variables de type entier, flottant ou de type tableau (vecteur) ou ensemble ayant des éléments de type quelconque.

1) IlcInt et IlcIntVar

Dans le Solver, on utilise une définition de type suivante pour le type *IlcInt* :

typedef long IlcInt;

IlcInt représente les entiers signés traités par les variables du type entier dans le Solver. Tous les entiers de C++ sont convertis implicitement en *IlcInt*. De cette manière, on peut assurer que les composants d'une application écrits en Solver sont portables sur différentes plates-formes sans aucun changement de programme source.

Une variable du type entier est créée de la façon suivante :

IlcIntVar var(m, IlcInt min, IlcInt max);

La variable *var* est associée au gestionnaire *m* déjà construit. Le domaine de *var* est composé de toutes valeurs du type *IlcInt inclues* dans l'intervalle [*min*, *max*].

Tous les méthodes et opérateurs définis dans la classe *IlcIntVar* qui pourraient modifier les domaines des variables sont réversibles. Donc, le domaine et les contraintes imposés sur une variable du type *IlcIntVar* quelconque seront restaurés quand le Solver fait un retour en arrière.

2) IlcFloat et IlcFloatVar

Comme *IlcInt*, *IlcFloat* est introduit pour la portabilité des logiciels développés sous le Solver. Les valeurs du domaine d'une variable du type *IlcFloatVar* sont de type *IlcFloat*. Une variable du type *IlcFloatVar* peut être obtenue de la manière suivante :

IlcFloatVar varf(m, IlcFloat v1, IlcFloat v2, IlcFloat prec);

Cette variable *varf* est construite et associée au gestionnaire *m*. Le domaine initial de *varf* est composé de tous les nombres flottants x tels que $v1 \leq x \leq v2$. Le paramètre *prec* est optionnel et est la précision associée à la variable *varf*. La sémantique de cette précision est : si $|v2-v1| / max(1, |v1|) \leq prec$, on considère que *varf* est fixée à la valeur $|v2 - v1| / 2$.

Comme les variables du type *IlcIntVar*, la réversibilité des changements due à la propagation des contraintes est garantie pour les variables du type flottant.

3) IlcAny, IlcAnySet, IlcAnySetVar, IlcAnySetVarArray

Un autre type de données de base introduit dans le Solver en vue de la portabilité des logiciels est *IlcAny* qui est défini comme suivant :

typedef void IlcAny;*

Ce type représente les objets traités par des variables énumérables ou ensemblistes du Solver. Il existe également certains types de données prédéfinis dans le Solver pour représenter les variables ensemblistes. Autre que *IlcIntSet*, on peut trouver autres types basés sur *IlcAny* tels que *IlcAnySet*, *IlcAnySetVar* et *IlcAnySetVarArray*.

Une instance de la classe *IlcAnySet* représente un ensemble de valeurs du type *IlcAny*. Une variable ensembliste est une instance de la classe *IlcAnySetVar* ou *IlcIntSetVar*. Le domaine d'une variable ensembliste du type *IlcAnySetVar* est composé de valeurs du type *IlcAnySet*, c'est-à-dire, un ensemble d'ensembles. Dans le Solver, ce genre de domaines est représenté par leur bornes inférieure et supérieure. La borne supérieure est l'union de tous les sous-ensembles. La borne inférieure est l'intersection de tous les sous-ensembles. Quand une variable ensembliste est fixée, leurs bornes inférieure et supérieure sont égaux. Si le domaine est réduit en domaine vide, un échec sera déclenché par la méthode *IlcManager:fail()* qui signifie que les contraintes postulées à ce gestionnaire n'admettent aucune solution. La réversibilité des changements est assurée également pour cette classe.

Une instance de la classe *IlcAnySetVarArray* est un tableau dont les éléments sont de type *IlcAnySetVar*, c'est-à-dire un tableau des ensembles des ensembles.

3.1.3 Les contraintes

Une contrainte est un objet de la classe *IlcConstraint* qui est une sous-classe de la classe *IlcGoal* (voir la section suivante). Une contrainte peut être considérée comme une expression booléenne dont les valeurs possibles sont *IlcTrue* ou *IlcFalse* dépendant de la satisfiabilité de cette contrainte. Ces expressions peuvent être combinées à l'aide d'opérateurs logiques : *or*, *and* ou *not*.

La fonction *void IlcIfThen(IlcConstraint ct1, IlcConstraint ct2)* exprime une implication 'ct1 → ct2'. En effet, cette fonction ajoute la contrainte '!ct1|| ct2' au gestionnaire.

Les opérateurs relationnels possibles dans une contrainte dépendent du type des variables. Les opérateurs tels que <, > , <=, >=, ==, != sont possibles pour des variables du type *IlcIntVar*. Seuls les opérateurs ==, <= et >= sont permis pour les variables du type *IlcFloatVar*. C'est la raison pour laquelle on a introduit une constante globale *IlcFloat delta* dans l'outil afin d'exprimer une contrainte sur des variables du type *IlcFloatVar*. Par exemple, {*varf<10*} devient {*varf <= 10 – delta*}.

Les fonctions spéciales telles que *IlcConstraint IlcMember(IlcAny val, IlcAnySetVar)*, *IlcConstraint IlcNotMember(IlcAny val, IlcAnySetVar)* sont définies pour des variables ensemblistes. Il existe certaines contraintes génériques permettant d'exprimer des contraintes sur un tableau de variables. Par exemple, *IlcAllDiff(IlcIntVarArray vars)* assure que tous les éléments de *vars* sont différents.

3.1.4 Les buts

Les buts sont des blocs composants des algorithmes de recherche dans le Solver. En pratique, les algorithmes de recherche prédéfinis ou définis par l'utilisateur peuvent être exprimés avec le macro suivant :

$$ILCGOALn(name, T_1, P_1, T_2, P_2, \ldots, T_n, P_n)\ \{\ corps\ \}\ ;$$

Ce macro définit une classe de but appelé *name* avec n paramètres ($n \leq 6$). *Ti* et *Pi* sont le type et le nom du ième paramètre.

Les buts peuvent être combinés à l'aide des fonctions *IlcAnd* ou *IlcOr* pour former un but plus complexe. On peut utiliser *IlcAnd* pour composer au maximum 5 sous-buts. Les sous-buts sont exécutés de gauche à droite. On peut créer un point de choix parmi au maximum 5 sous-buts à l'aide de la fonction *IlcOr*. L'exécution d'un point de choix

consiste à essayer successivement les sous-buts jusqu'à ce qu'un sous-but soit réussi, ou tous les sous-buts sont essayés. Dans ce dernier cas, le point de choix est échoué.

Il faut noter qu'un but composite peut être représenté par un arbre de recherche dans lequel chaque nœud constitue un sous-but. La réussite d'un but g consiste à trouver un chemin de la racine de l'arbre (représente le but g) à une feuille de l'arbre dans lequel tous les sous-buts sont réussis. Comme un exemple, la figure 3.1 illustre la représentation d'un but *composite IlcGoal g = IlcOr(IlcAnd(g1,g2), IlcAnd(g3,g4,g5), g6)* par un arbre de recherche.

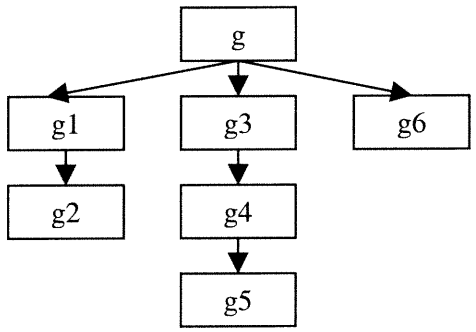


Figure 3.1 La représentation d'un but composite sous forme d'arbre

Les fonctions les plus utilisées pour construire un but sont *IlcInstantiate* et *IlcGenerate*. La fonction *IlcGoal IlcInstantiate(IlcIntVar var)* permet d'affecter successivement les valeurs du domaine de *var*. *IlcGenerate(IlcIntVarArray vars)* retourne un but permettant d'énumérer chaque variable dans *vars*. La fonction *IlcInstantiate(IlcFloatVar var, IlcBool increaseMinFirst = IlcTrue, IlcFloat prec=0)* divise récursivement le domaine de *var* en deux parties et créer un point de choix. La fonction se termine quand la variable *var* est fixée ou quand la précision associée au domaine obtenu est inférieure à *prec*. Si un échec se produit, le domaine est remplacé par l'autre moitié et la fonction est appelée récursivement. Si increaseMinFirst est IlcTrue (IlcFalse), la partie supérieure (inférieure) est testée en premier.

3.2 Le calcul de séparation maximum des événements

Pour calculer la séparation maximum des événements d'un chronogramme, on doit définir une classe *TimingDiagram* qui décrit un chronogramme. Pour définir cette classe, un ensemble de classes doit être implanté pour la description des composants d'un chronogramme (voir la section 1.2). L'implantation de la classe *TimingDiagram* est illustrée dans la figure 3.2. Dans la figure 3.2, pour deux classes liées par une flèche, la classe pointant contient au moins une instance de la classe pointée .

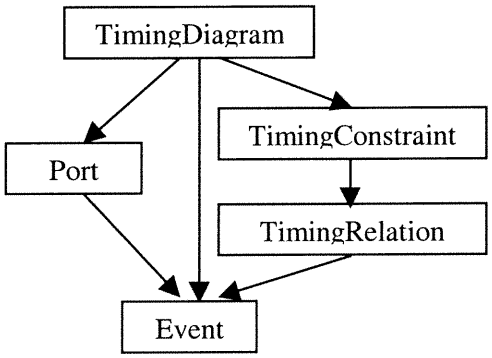


Figure 3.2 L'implantation de la classe TimingDiagram

On décrit ces classes dans les sous-sections suivantes.

3.2.1 Event

La première et la plus élémentaire classe définie est *Event*. Elle contient essentiellement trois données et une méthode d'affichage *display*().

1) *const char* name* : une chaîne de caractères pour étiqueter un événement ;
2) *modeType mode* : indique la direction du port auquel l'événement attache, où *modeType* est un type d'énumération :

   enum modeType {in = 0, out = 1, inout = 2, none = 3}.

   none est utilisé pour l'événement original d'un chronogramme ;
3) *IlcBool initialValue* : une valeur booléenne qui indique la valeur du signal sur ce port avant que cet événement se produise.

Toutes les données sont accessibles à travers les méthodes accessoires. Un événement peut être construit à l'aide de constructeur de cette classe :

Event(IlcManager m, char name, modeType mode, IlcBool initial);*

3.2.2 Port

La définition de la classe *Port* est composée de différents constructeurs, et un ensemble de données pour identifier tous les composants d'un port :

- *const char* name* : identificateur d'un port ;
- *modeType mode* : la direction du port expliquée dans la classe *Event* ;
- *IlcInt nbStates* : nombre d'états. Le nombre d'événements = *nbStates –1* ;
- *IlcIntArray indEventsIn* : un tableau du type entier indique pour un port du type *entrée-sortie* (*inout*), l'index des événements d'entrée ;
- *IlcBool initialValue* : une valeur booléenne initiale qui indique la valeur du signal sur ce port avant que son premier événement se produise ;
- *IlcAnyArray events* : un tableau dont les éléments sont des objets du type *Event* indique une suite d'événements sur ce port.

Cette classe permet de créer un port du type *in*, *out* ou *inout*. Un port du type *in* ou *out* peut être créé en donnant simplement le nom, le mode, le nombre d'états et la valeur initiale du port à l'aide d'un constructeur :

Port(IlcManager m, char name, modeType mode, IlcInt nbStates, IlcBool initial);*

Les objets de la classe *Event* attachés au port sont créés automatiquement. Le nom d'un événement est généré en collant l'index d'événement suivant le nom du port. Par exemple, le nom du premier événement du port *X* est *X0*. Pour créer un port du type *inout*, on doit indiquer de plus les index des événements d'entrée sur le port.

3.2.3 TimingRelation

La classe *TimingRelation* possède les données suivantes :

- *const char* name* : identificateur de l'objet à créer ;
- *Event* source* : pointeur sur l'événement source associé à cet objet ;
- *Event* sink* : pointeur sur l'événement cible associé à cet objet ;
- *IlcFloatVar delay* : une variable représente l'intervalle temporel associé ;

Une relation temporelle est construite à l'aide du constructeur de cette classe :

TimingRelation(IlcManager, char name, Event* source,*
Event cible, IlcFloatVar delay);*

3.2.4 TimingConstraint

La classe *TimingConstraint* permet de créer une contrainte temporelle. Elle possède les données suivantes :

- *const char* name* : identificateur de l'objet ;
- *intentType intent* : indique l'interprétation de la contrainte, où *intentType* est un type d'énumération : *enum intentType {assume=0, commit=1} ;*
- *compositeType composition*: indique le type de composition de la contrainte temporelle, où *compositeType* est un type d'énumération défini comme :

  enum compositeType {linear = 0, min = 1, max = 2};
- *IlcInt nbComposants* : nombre des relations temporelles qui composent cet objet ;
- *IlcAnyArray composants* : un tableau dont les éléments sont de type *TimingRelation.*

Un ensemble de méthodes permet d'accéder toutes les données ci-dessus, d'identifier l'événement cible et de vérifier si les composants sont valides pour construire une contrainte temporelle. Une contrainte temporelle peut être construite à l'aide d'un des constructeurs de cette classe :

TimingConstraint(IlcManager m, char name, compositeType composition,*
IlcInt nbComposants, IlcAnyArray composants);

TimingConstraint(IlcManager m, char name, Event* source,*
Event cible, IlcFloatVar delay);*

Ce dernier constructeur reflète qu'une relation temporelle elle-même est aussi une contrainte temporelle.

3.2.5 TimingDiagram

La classe la plus importante et aussi la plus complexe développée est la classe *Timing-Diagram*. Elle exploite toutes les classes précédentes. Un chronogramme est construit à l'aide de constructeur de la classe *TimingDiagram* :

TimingDiagram(IlcManager m, Event origine, IlcAnyArray ports,*
IlcAnySet constraints);

Cette classe contient un ensemble de données dont les plus importantes sont :

- *IlcAnyArray PortsIn* : représente les ports du type *in* ;
- *IlcAnyArray PortsOut* : représente les ports du type *out* ;
- *IlcAnyArray events* : représente tous les événements spécifiés du chronogramme ;
- *IlcAnySet constraintsAsm* : représente les contraintes *assume* du chronogramme ;
- *IlcAnySet constraintsCmt* : représente les contraintes *commit* du chronogramme ;
- *IlcAnySet addedConstraintsAsm* : les contraintes *assume* supplémentaires dues au port du type *inout* ;
- *IlcAnySet addedConstraintsCmt* : les contraintes *commit* supplémentaires dues au port du type *inout*;

Il faut noter que les ports représentés par le paramètre *Ports* sont partitionnés dans *PorsIn* et *PortsOut* selon la direction de chaque port. Un port du type inout p est divisé en un port d'entrée p_{in} ajouté dans *portsIn* et un port de sortie p_{out} ajouté dans *portsOut*. On utilise *addedConstraintsAsm* ou *addedConstraintsCmt* pour enregistrer les contraintes supplémentaires introduites afin de maintenir l'ordre des événements sur le port original p après cette division. Les événements sur les ports de *portsIn* puis ceux de

portsOut sont rangés dans *events* suivant l'ordre d'occurrence sur chaque port. Chaque événement peut être identifié par son nom qui est construit en collant l'index de l'événement à la fin du nom de port qui l'attache. Par exemple, *X0* représente le premier événement sur le port *X*.

Les méthodes les plus importantes de cette classe sont :

- *IlcAnySet getConstraintsWithSink(Event* event);* cette fonction retourne toutes les contraintes ayant *event* comme événement cible ;
- *IlcFloatVarArray build_constraint_system(IlcFloatVarArray occurs);*
  IlcFloatVarArray build_constraint_system(compositeType cmp, IlcFloatVarArray occurs); Ces fonctions permettent d'obtenir les temps d'occurrence de tous les événements après la propagation de toutes les contraintes spécifiées. La seconde est utilisée dans le cas où deux types de contraintes non linéaires (*earliest* ou *latest*) existeraient simultanément. *cmp* indique le type de contraintes non linéaires (*min* ou *max*) à décomposer.
- *IlcFloatVarArray findDistances(Event* reference);* cette fonction calcule la séparation de tous les événements du chronogramme par rapport à *reference*.
- *IlcInt findEvent(Event* event);* cette fonction permet d'obtenir l'index d'un événement spécifié *event* dans l'ensemble des événements *events*.

3.2.6 Le calcul de séparation maximum

À l'aide de la définition de la classe *TimingDiagram*, le calcul de séparation maximum consiste à simplement un appel à la méthode :

IlcFloatVarArray TimingDiagram::findDistance(Event event);*

Pour un exemple présenté dans [9] ayant des contraintes non linéaires *min* et *max* simultanément, on a obtenu le même résultat en utilisant cette méthode et celle présentée dans [9].

Cette méthode fonctionne de la façon suivante :

1) Construire un vecteur des variables du type flottant *IlcFloatVarArray distances* dans lequel *distances*[*i*] représente le temps d'occurrence de l'événement *events*[*i*]. *events* est une donnée membre de la classe *TimingDiagram* obtenu à l'aide de la méthode *TimingDiagram::getEvents()*. *distances* sera la sortie de cette fonction après la propagation de toutes les contraintes spécifiées dans le chronogramme. *distances[0]* représente le temps d'occurrence de l'événement origine du chronogramme.
2) Fixer la variable *distances[k]* à 0, où *k* est l'indice de l'événement de référence *event* dans l'ensemble des événements du chronogramme, obtenu à l'aide de la méthode *TimingDiagram::getEvent(Event*)*.
3) Ajouter les contraintes implicites indiquées dans le chapitre 1.
4) Appeler la méthode *built_constraint_system* qui se charge d'appliquer au gestionnaire toutes les contraintes spécifiées dans le chronogramme. Pour un chronogramme qui ne contient pas simultanément les contraintes de type *min* et *max*, cette méthode appelle, selon le type de composition de chaque contrainte, à la méthode *applyLinearConstraint* ou *applyNonLinearConstraint*. Appliquer une contrainte linéaire avec la méthode *applyLinearConstraint* consiste à ajouter pour chaque composant, une relation temporelle $e_{sink} = e_{source} + delay$, où *delay* est une variable du type *IlcFloatVar*, au *gestionnaire*. Cette relation temporelle correspond à une contrainte associant à trois variables flottantes :

   m.add(distances[i] == distances[j]+ delay);

   où *i*, *j* sont des indices des événements cible et source respectivement dans l'ensemble des événements du chronogramme. Appliquer une contrainte non linéaire avec *applyNonLinearConstraint* consiste à introduire un tableau de variables du type flottant *IlcFloatVarArray interval* de taille égale au nombre de composants de cette contrainte. Puis pour chaque composant *k*, appliquer la contrainte suivante au gestionnaire :

   m.add(interval[k] == distances[j]+ delay);

où *k* est l'indice du composant et *j* est l'indice de l'événement source de la relation temporelle correspondant au composant *k*. A la fin, selon le type de composition de la contrainte, on applique la contrainte suivante au gestionnaire :

m.add(distances[i] == IlcMin(interval)); ou

m.add(distances[i] == IlcMax(interval));

où *i* est l'indice de l'événement cible de cette contrainte. La première correspond à une contrainte du type *min* et la seconde est pour une contrainte *max*.

Pour un chronogramme contient simultanément les contraintes de type *min* et *max*, si le nombre de contraintes du type *min* est supérieur à celui de contraintes du type *max*, les contraintes du type *max* sont décomposées, sinon les contraintes du type *min* sont décomposées. La décomposition d'une contrainte non linéaire est réalisée par la méthode *TimingDiagram::decomposition* qui énumère la possibilité d'attribution de chaque composant à la détermination du temps d'occurrence de l'événement cible de cette contrainte. (Voir l'annexe C pour le détail de cette méthode.)

3.3 Vérification de la causalité d'un chronogramme

Pour la vérification de la causalité d'un chronogramme, à part les classes définies présentées dans la section 3.2, on a besoin tout d'abord de définir une autre classe appelée *Block* qui permet de décrire une bloc-machine.

3.3.1 La classe Block

La classe *Block* est développée dans le cadre de vérification de causalité d'un chronogramme. Le constructeur de cette classe est le suivant :

Block(IlcManager m, IlcIntVar rk, IlcIntVar dir, IlcIntVar comp,
IlcAnySetVar ev, IlcAnySetVar trig);

Cette classe contient les données suivantes :

- *IlcManager manager* : le gestionnaire associé;
- *IlcIntVar rank* : le rang du bloc pour exprimer un ordre topologique total des blocs de la bloc-machine associant au chronogramme. Si une bloc-machine est composée de *n* blocs, alors le range d'un bloc peut être initialisé par une variable *IlcIntVar rank(m, 0, n-1)*, une bloc-machine est représentée simplement par un vecteur de blocs à l'aide du type *IlcAnyArray*.
- *IlcIntVar dir* : cette variable représente le type des événements possibles dans le bloc. Cette variable peut être initialisée comme *IlcIntVar dir(m, 0, 1)*, où *0* indique que le bloc ne contient que des événement d'entrée et *1* indique que le bloc ne contient que des événements de sortie.
- *IlcIntVar type* : indique le type de bloc qui correspond au type de composition des contraintes temporelles locales dans ce bloc. Cette variable peut être initialisée comme *IlcIntVar type(m, 0, 2)* (*0 = linear*, *1 = min* et *2 = max*).
- *IlcAnySetVar evList*; cette variable représente l'ensemble des événements locaux appartenant à ce bloc.
- *IlcAnySetVar trigList*; cette variable représente l'ensemble des événements déclencheurs du bloc.

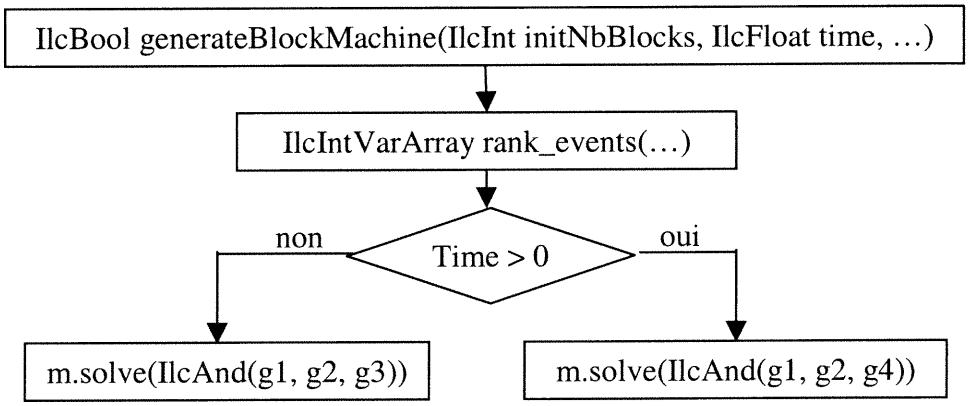
Une méthode *void setConstraints()* est utilisée pour exprimer des contraintes telles que : $evList \cap trigList = \varnothing$; si *type* = *1* ou *2*, alors $|evList| = 1$.

Il faut remarquer que toutes les données de cette classe sont des variables. En effet, une instance de cette classe représente surtout un bloc non fixé. Une bloc-machine peut être représentée par un tableau du type *IlcAnyArray* dont chaque élément est un pointeur sur un bloc. Une bloc-machine causale est générée en fixant tous les blocs après le processus de vérification de la causalité si le chronogramme correspondant est causal.

3.3.2 Les modules pour la génération d'une bloc-machine

La vérification de la causalité d'un chronogramme se réalise par certains modules qui expriment les deux conditions de causalité présentées dans le chapitre 2 telles que "*well-defined trigger*" et "*past-dominated block*". On a testé avec ces modules les exemples présentés dans [8] et on a obtenu les même résultats.

La figure 3.3 illustre un schéma décrivant l'organisation du module principal *generate-BlockMachine*.




```
IlcGoal g1 = IlcInstantiate(nbBlocks);
IlcGoal g2 = create_partition(...);
IlcGoal g3 = generate_blocks(...);
IlcGoal g4 = IlcLimitSearch(g3, IlcTimeLimit(m, time));
```

Figure 3.3 L'organisation du module generateBlockMachine

Le prototype du module *generateBlockMachine* est le suivant :

IlcBool ***generateBlockMachine****(TimingDiagram td, IlcInt direction,*
IlcInt initNbBlocks, IlcFloat time);

Ce module retourne *IlcTrue* si une bloc-machine causale est trouvée sous les conditions spécifiées, sinon il retourne *IlcFalse*. Il comprend 4 paramètres d'entrée :

1) *td* est le chronogramme à vérifier ;
2) *initNbBlocks* le nombre initial de blocs à générer pour une bloc-machine ;
3) *direction* indique la direction de recherche d'une bloc-machine causale. *0* (*1*) signifie qu'on augmente (diminue) le nombre de blocs à générer dans le cas d'échec ; Une variable *nbBlocks* du type *IlcIntVar*, représentant le nombre des blocs à générer, est initialisée de la façon suivante :

   if (direction==1) nbBlocks = IlcIntVar(m, lim, IlcMin(initNbBlocks, evsize));

   else nbBlocks= IlcIntVar(m, IlcMax(lim, initNbBlocks), evsize) ;

   où *evsize* le nombre des événements spécifiés dans le chronogramme. *lim* est déterminé par :

   if(nbEventsIn > 0 && nbEventsOut >0) lim = IlcMax(2, cstMin+cstMax);

   else lim = IlcMax(1, cstMin+cstMax) ;

   où nbEventsIn et *nbEventsOut* sont le nombre des événements d'entrée et de sortie respectivement, et *cstMin* et *cstMax* sont le nombre des contraintes *min* et *max* respectivement.
4) Pour rendre le processus de recherche plus souple, on peut utiliser la stratégie de recherche en profondeur limitée en spécifiant pour chaque nombre de blocs *nblks*, un temps maximum de *time* secondes consacré à la génération d'une bloc-machine de *nblks* blocs. Si *time* <= 0, la stratégie de recherche en profondeur est appliquée (voir la figure 3.3).

Avant d'expliquer les autres modules appelés par la méthode *generateBlockMachine*, il faut rappeler que pour une instance de la classe *TimingDiagram td*, tous les événements de *td* sont ordonnés dans un tableau *IlcAnyArray events* qui peut être obtenu à l'aide d'une méthode accessoire *td.getEvents()*.

La fonction *rank_events* associe une variable du type *IlcIntVar* à chaque événement du tableau *events* (sauf l'événement origine de *td*). Le prototype de cette fonction est le suivant :

IlcIntVarArray ***rank_events****(TimingDiagram td, IlcIntVarArray ranks)*;

Cette fonction tente d'ordonner partiellement les événements du chronogramme *td* en prenant en compte l'enchaînement des événements par les contraintes de *td*. Par exemple : le rang des événements source d'une contrainte ne peut pas être supérieur à celui de l'événement cible de cette contrainte. *ranks* représente le rang de chaque événement qui est initialisé par :

IlcIntVarArray ranks(m, evsize, 0, evsize-1) ;

La variable *ranks* sera fixée par le but *generate_blocks* décrit par le suite. La recherche de solution consiste à appeler un but composé de 3 sous-buts par l'opérateur *IlcAnd* qui sont exécutés un après l'autre (voir la figure 3.3).

- IlcGoal g1 = *IlcInstantiate(nbBlocks)*. IlcInstantiate est une fonction prédéfinie de ILC Solver permettant d'instancier une variable. Le but g1 se charge d'instancier la variable nbBlocks qui représente le nombre de blocs à générer.
- IlcGoal g2 = *create_partition(m, nbBlocks)* est utilisé pour initialisé un tableau de taille *nbBlocks* du type *IlcAnyArray partition* qui représente une bloc-machine. Parce que pour initialiser un tableau, le nombre des éléments dans ce tableau doit être une constante, donc, on est obligé de séparer ce but du but g1.
- IlcGoal g4 = *IlcLimitSearch(g3, IlcTimeLimit(m, time))*; Ce but demande le gestionnaire d'effectuer une recherche en profondeur limitée pour le but g3. La profondeur de recherche pour chaque branche est limitée par le paramètre *time* en secondes.
- IlcGoal g3 = *generate_blocks* se charge d'exprimer les deux conditions de la causalité présentées dans le chapitre 2. Le prototype de ce but est le suivant :

  *ILCGOAL5(**generate_blocks**, TimingDiagram, td, IlcIntVar, nbBlocks, IlcIntVarArray, ranks, IlcIntArray, modes, IlcIntArray, types).*

Où *td* est le chronogramme correspondant. *nbBlocks* est la variable qui représente le nombre des blocs à générer. La variable *nbBlocks* est initialisée dans le module *generateBlockMachine* et instanciée par le but g1 précédent. *ranks* est la sortie de la fonction *rank_events*. *modes* et *types* sont de tableau représente le mode et le type de chaque

événement. Le mode de chaque événement est obtenu directement en appelant la méthode accessoire *getMode()* pour chaque événement. Le type (correspond à *compositeType* de la classe *TimingConstraint*) de chaque événement est déterminé de la façon suivante : par défaut, le type de chaque événement est initialisé par *linear*. Si un événement est l'événement cible d'une contrainte *min* (*max*), son type est défini comme *min* (*max*).

Ce module tente de générer une bloc-machine *partition* (initialisée par le but g2). Tout d'abord, certaines contraintes sont ajoutées dans le gestionnaire, puis un but composite est exécuté. L'organisation de ce module est la suivante :

1) Générer une bloc-machine *partition* ayant *nbBlocks* blocs et appeler la méthode *setConstraints* pour chaque bloc construit, Les deux tableaux du type *IlcAnySetVarArray evList* et *trigList* sont introduits pour représenter l'ensemble des événements locaux et déclencheurs de chaque bloc ; ces deux tableaux sont initialisés de la façon suivante :

   IlcAnySetVarArray evList(m, nbBlocks, events) ;

   IlcAnySetVarArray trigList(m, nbBlocks, events) ;

   Chaque tableau ci-dessus contient *nbBlocks* éléments du type *IlcAnySetVar*, et l'ensemble possible de chaque élément contient tous les événements du chronogramme *events*. En plus, trois tableaux de type *IlcIntVarArray* sont introduits pour représenter le rang, la direction et le type de chaque bloc :

   IlcIntVarArray rk(m, nbBlocks, 0, nbBlocks-1) ;

   IlcIntVarArray dir(m, nbBlocks, 0, 1) ;

   IlcIntVarArray type(m, nbBlocks, 0, 2) ;

2) Pour chaque événement e_j , chercher l'ensemble des événements $e_i \in source$ pour lesquels, il existe une contrainte $c(e_i, e_j, delay)$ dans l'ensemble de contraintes spécifiées dans le chronogramme *td*. Ajouter les contraintes qui expriment que :

   si $|source|>0$ et $e_j \in evList(blk)$, alors, $source \subseteq evList(blk) \cup trigList(blk)$.

   où *evList*(*blk*) et *trigList*(*blk*) sont des ensembles des événements locaux et déclencheurs du bloc *blk* respectivement.

3) Pour chaque événement e_j , chercher l'ensemble des événements $e_i \in$ *cible* pour lesquels il existe une contrainte *c(e_j, e_i, delay)* dans l'ensemble de contraintes spécifiées dans le chronogramme *td*. Ajouter les contraintes qui expriment que :
   - si $|cible| = 0$, alors $e_j \notin \cup trigList(B_i)$, $0 \leq i \leq nbBlocks-1$ (c'est-à-dire e_j ne peut pas être un déclencheur) ;
   - si $|cible| > 0$ et $evList(blk) \cap cible = \varnothing$, alors $e_j \notin trigList(blk)$;

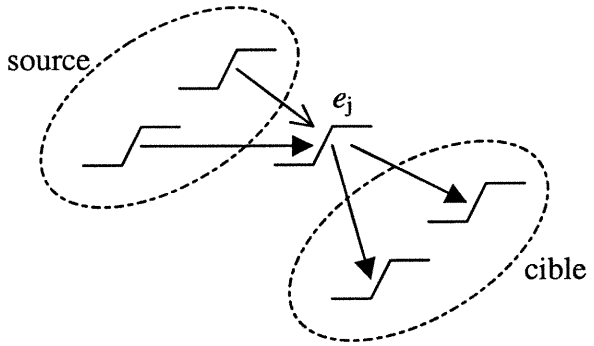


Figure 3.4 Les ensembles de sources et de cibles d'un événements e_j

4) Ajouter les contraintes sur *evList* pour chaque bloc en comparant le mode, la direction et le rang de chaque bloc et ses événements locaux afin d'éliminer certains événements qui ne peuvent pas appartenir à un bloc.

5) Pour obtenir une partition dont les rangs des blocs sont fixés et différents, on applique une contrainte générique *IlcAllDiff(IlcIntArray rks),* où *rks* est un vecteur de variables du type entier représentant le rang de chaque bloc dans la partition.

6) Appliquer une contrainte générique *IlcPartition(IlcAnySetVarArray evList, IlcAnyArray events)* qui assure que chaque élément de *events* doit appartenir à un et seulement un élément de *evList*. Chaque élément de *evList* est une variable ensembliste représentant l'ensemble des événements locaux d'un bloc.

La recherche de solution consiste à ajouter au gestionnaire un but composite *IlcAnd(g1, g2, g3).*

```
IlcGoal g1 = IlcAnd(IlcGenerate(evList, IlcChooseMinSizeAnySet),
                    IlcGenerate(trigList, IlcChooseMinSizeAnySet)) ;
```

IlcGoal g2 = IlcAnd(IlcGenerate(dir, IlcChooseMinSizeInt, SelectMin(m)),
IlcGenerate(type, IlcChooseMinSizeInt, SelectMin(m)),
IlcGenerate(rk, IlcChooseMinSizeInt, SelectMin(m))) ;
IlcGoal g3 = IlcAnd(valid_triggers(m, td, partition)),
valid_blocks(m, td, partition)) ;

Le sous-but g1 composé de deux sous-buts qui se chargent d'instancier l'ensemble des événements locaux et celui des déclencheurs pour chaque bloc. L'ensemble qui contient le moins des éléments possibles est choisi en premier. Le sous-but g2 est utilisé pour fixer les autres variables de chaque bloc. La variable dont le domaine qui contient le moins de valeurs est choisie en premier, et la valeur la plus petite dans le domaine est utilisée d'abord. Jusqu'à ce moment, on obtient une bloc-machine fixée. Le sous-but g3 permet de vérifier la causalité d'une bloc-machine. (voir la figure 3.5.)

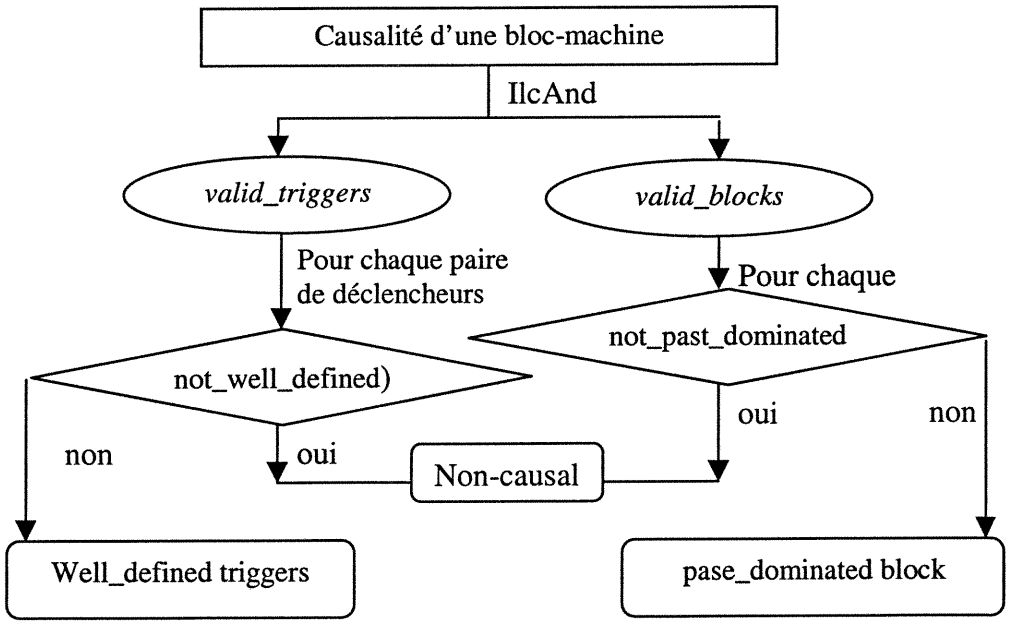


Figure 3.5 La vérification de la causalité dans le but *generate_blocks*

Le but *ILCGOAL2(**valid_triggers**, TimingDiagram, td, IlcAnyArray, partition)* appelle pour chaque bloc, au but ***not_well_defined*** pour toutes les paires d'événements déclencheurs et déclenchés. Ce module vérifie la première condition de la causalité "well-

defined triggers" pour tous les événements déclencheurs. Comme le nom indique, le but *ILCGOAL2 (**not_well_defined**, IlcFloatVar, var1, IlcFloatVar, var2);* exprime la première condition de causalité pour un événement et un de ses déclencheurs. *var1* et *var2* représente le temps d'occurrence d'un déclencheur et celui d'un événement déclenché respectivement.

Le but *ILCGOAL2(**valid_blocks**, TimingDiagram, td, IlcAnyArray, partition)* appelle pour chaque bloc, au but ***not_past_dominated*** pour tous les blocs d'une bloc-machine *partition* pour un chronogramme *td* considéré. Ce module vérifie la seconde condition de la causalité "*past-dominated blocks*" pour tous les blocs d'une partition. Le but *ILCGOAL4(**not_past_dominated**, IlcFloat IlcFloatVar, var1, IlcFloatVar, var2, IlcFloatVar, var3, IlcFloatVar, var4)* exprime la seconde condition de causalité pour une paire de déclencheurs d'un même bloc. var1 et var2 (var3 et var4) sont le temps d'occurrence d'une paire de déclencheurs déterminé par les contraintes locales du bloc (par les blocs précédents).

Les buts *valid_triggers* et *valid_blocks* appellent la fonction *execute_blocks* pour appliquer les contraintes locales d'un bloc au gestionnaire. Le prototype de cette fonction est le suivant :

*IlcFloatVarArray **execute_blocks**(TimingDiagram td, IlcAnyArray part,*
IlcInt rank, IlcFloatVarArray dis, IlcBool continue);

Après avoir généré une bloc-machine *part*, cette fonction retourne *dis* après avoir appliqué toutes les contraintes locales du bloc *part[rank]* si *continue* = *IlcFalse*. Cette fonction est appelée récursivement pour tous les blocs dont leurs rangs sont inférieurs à *rank* si *continue* = *IlcTrue*.

L'outil fournit également un moyen pour vérifier si une partition des événements peut générer une bloc-machine causale. La fonction *IlcBool **testPartition**(TimingDiagram td)* remplit cette tâche. Quand l'utilisateur fournit une partition des événements, ce module permet de vérifier si la bloc-machine construite à partir de cette partition satisfait

les deux conditions de la causalité. Puisque l'exécution de module *generateBlockMachine* peut être très coûteux dans certains cas, ce module fournit un autre moyen pour obtenir rapidement une bloc-machine causale, surtout pour les utilisateurs expérimentés. Dans le cas d'échec, l'utilisateur peut découvrir rapidement le problème selon le message du logiciel qui indique quel déclencheur n'est pas '*well-defined*' et quel bloc n'est pas '*post_dominated*'. Après une modification de la partition, un autre lancement du module peut probablement confirmer que cette partition est causale.

Une partition des événements peut être saisie d'un fichier. Le module *readPartition* se charge de construire une bloc-machine dans laquelle le nombre de blocs, le rang et l'ensemble des événements locaux de chaque bloc sont fixés. Le format du fichier qui définit une partition des événements est le suivant :

Nblks : le nombre de blocs à générer ;

Pour chaque bloc, il faut fournir :

Mode : le mode du bloc (*in* ou *out*) ;

Type : le type du bloc (*linear*, *min* ou *max*) ;

NbEvents : le nombre des événements locaux du bloc ;

EvNames : la liste du nom des événements locaux ;

Le rang de chaque bloc est déterminé selon l'ordre d'apparition des blocs dans le fichier.

3.4 Vérification d'un contrôleur d'interfaces

La vérification d'un contrôleur d'interfaces nécessite tout d'abord de définir la classe FSM qui décrit l'implantation d'un contrôleur d'interfaces. L'implantation de cette classe est basée sur les classes suivantes :

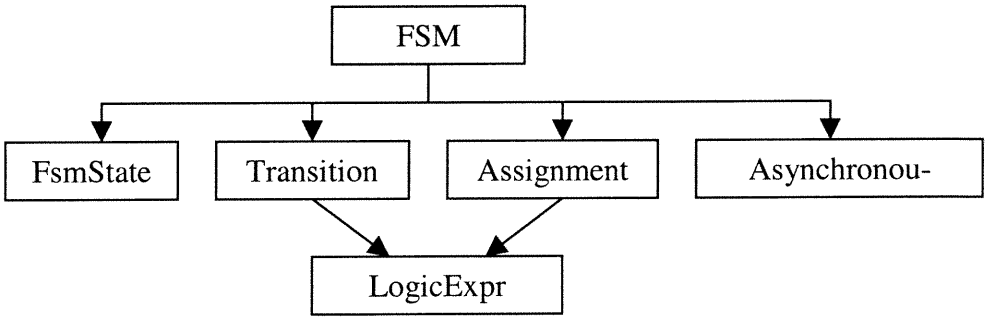


Figure 3.6 L'implantation de la classe FSM

On explique ces classes de base qui permettent de définir la classe FSM dans la section 3.4.1. La classe FSM est décrite dans la section 3.4.2. Dans la section 3.4.3, nous présentons les modules développés dans le but de vérifier un contrôleur d'interfaces.

3.4.1 Les classes de base utilisées dans la classe FSM

La classe *FsmState* est introduite pour créer dynamiquement un type d'énumération pour les états d'une machine à états finis. Vu que dans la spécification d'une FSM, leur états sont souvent désignés par une chaîne de caractères qui indique la signification de l'état (par exemple IDLE, WAIT etc.), et que dans l'implantation d'une FSM, leur états (courant ou suivant) sont représentés par une variable de type *IlcIntVar*, on utilise la classe *FsmState* pour établir le lien entre ces deux représentations d'états. La classe *FsmState* ne contient que deux données *char* label* et *IlcInt no* qui représentent les deux représentations d'un état d'une FSM.

La classe *Transition* est introduite pour modéliser la fonction de transition d'une FSM. Cette classe contient les 3 données suivantes :

1) *char* label* est l'identificateur d'une fonction de transition ;
2) *LogicExpr* condition* exprime la condition de la fonction de transition ;
3) *LogicExpr* conclusion* représente la valeur de la fonction de transition.

Une fonction de transition est interprétée à l'aide de la fonction prédéfinie de ILOG Solver *IlcIfThen* de la façon : *IlcIfThen(ct_condition, ct_conclusion),* où *ct_condition* et *ct_conclusion* sont des contraintes acceptables par Solver, obtenues en interprétant *condition* et *conclusion* respectivement qui sont les pointeurs sur l'instance de la classe *LogicExpr.* (voir la description de la classe *LogicExpr*)

La classe *Assignment* est utilisée pour modéliser l'affectation d'un registre ou celle d'une sortie de contrôleur. Cette classe contient 5 données suivantes :

1) *char* name* est l'identificateur d'une affectation ;
2) *IlcInt nbNames* est la taille du tableau de chaînes de caractères *nameList* suivant ;
3) *char** nameList* est un tableau de chaînes de caractères qui représentent les noms de variables utilisés dans l'expression logique *value* suivante ;
4) *char* label* représente le nom de la variable affectée ;
5) *LogicExpr* value* est une instance de la classe *LogicExpr* qui représente l'expression logique utilisée pour affecter la variable *label.*

Si *Assignment inst* est une affectation d'une variable logique *varlogic,* appliquer *inst* au gestionnaire *m* consiste à ajouter une contrainte '*varlogic* = *ct_value*' au gestionnaire *m,* où *ct_value* est une contrainte acceptable par Solver, obtenue en interprétant *inst.value,* un pointeur sur une instance de la classe *LogicExpr.*

La classe *AsynchronousOutput* est introduite pour modéliser l'affectation du temps d'occurrence de certaines événements sur des ports de sortie asynchrones d'un contrôleur. Cette classe contient 4 données suivantes :

1) *char* output* indique le nom d'un événement sur une sortie asynchrone ;
2) *compositeType indique* l'opération (*min* ou *max*) appliquée ;
3) *IlcInt nbInputs* est le nombre d'événements d'entrée impliqués ;
4) *char** eventList* un tableau de chaînes de caractères qui représente les noms d'événements impliqués.

L'utilisation de cette classe sera expliquée dans le chapitre suivant. Puisque cette classe est utilisée pour exprimer l'affectation du temps d'occurrence de certains événements, elle peut être considérée comme contraintes temporelles de type *min* ou *max* selon le type d'opération, c'est-à-dire, une partie de la spécification sous forme de chrono-gramme. Mais, une sortie d'un contrôleur étant synchrone ou asynchrone est déterminé par l'implantation du contrôleur, donc, on décide que l'introduction de cette classe comme une partie de la classe FSM.

On voit bien dans la figure 3.6 que la classe *LogicExpr* est essentielle pour l'implantation de la classe FSM. Cette classe permet d'exprimer une expression logique composée d'un ensemble de termes connectés par l'opérateur logique "||". Un terme représente un ensemble de prédicats connectés par l'opérateur logique "&&". Chaque prédicat peut être un des types suivants :

1) Prédicat sur un signal exprime une contrainte qui restreint une variable booléenne. Par exemple : $X == 0$, où X est le nom d'un registre ou d'un signal d'entrée de FSM ;

2) Prédicat sur l'état exprime une contrainte qui restreint la variable d'états d'une FSM. Par exemple : *current* == *IDLE*, où *current* est une variable d'état représente l'état courant de FSM, *IDLE* est un état prédéfini de FSM.

3) Prédicat composite : il s'agit d'une instance de la classe *LogicExpr* dans laquelle tous les prédicats sont soit sur un signal soit sur l'état. Pour simplifier l'implantation de cette classe, on impose qu'un prédicat composite ne peut pas contenir à son tour un élément composite. Ce type de prédicat est introduit pour prendre en compte l'affectation d'un signal (la sortie d'un registre de synchronisation ou la sortie d'un contrôleur PO) dans laquelle certains signaux internes sont présents et ils ne sont pas échantillonnés par un registre.

Cette classe contient les données membre suivantes :

- *IlcManager m* : le gestionnaire associé ;
- *IlcInt nbNames* : le nombre de registres ;
- *Char** nameList* : les noms des registres ;

- *IlcInt nbStates* : le nombre des états d'un FSM ;
- *char** *stateList* : le nom des états d'un FSM ;
- *IlcInt nbTerms* : le nombre de termes. Un terme consiste en l'ensemble des prédicats connectés par l'opérateur logique "&&" ;
- *IlcInt nbLogiqueElement* : le nombre de prédicats sur un signal ;
- *IlcInt nbStateElement* : le nombre de prédicats sur l'état ;
- *IlcIntArray memLogical* : un vecteur de nombres entiers de taille *nbLogiqueElement* avec lequel on peut déduire la sémantique du prédicat sur un signal. Si *memLogical*[i] = j, le nom de la variable logique concernée qui se trouve dans *nameList*[j / 2] est X, et si le modulo j/2 = 1 (0), alors ce prédicat est interprété comme X (!X) ;
- *IlcIntArray memState* : similaire à *memLogical*, une vecteur de nombres entiers de taille *nbStateElement* utilisé pour interpréter un prédicat sur l'état ;
- *IlcIntArray stateType* : le prédicat sur l'état courant (0) ou l'état suivant (1) ;
- *IlcIntArray types* : le type de chaque prédicat (0 logique, 1 état , 2 composite) ;
- *IlcAnyArray exprs* : les prédicats composites ;
- *IlcIntArray memExpr* : utilisé pour chercher un prédicat composite correspondant dans *exprs*, la définition de chaque élément de *memExpr* est similaire à celle de *memLogical*.

La transformation d'une expression logique (une instance de la classe *LogicExpr*) en contrainte se réalise par la fonction *translate_logicExpr*. Il s'agit d'une interprétation terme par terme lié par l'opérateur logique "||" et élément par élément lié par l'opérateur logique "&&" pour construire une contrainte représentant la sémantique de cette expression logique. Le prototype de cette fonction est le suivant :

IlcConstraint translate_logicExpr(LogicExpr lexp, IlcBoolVarArray vars,*
IlcIntVar current, IlcIntVar next);

Où *lexp* est une expression logique à transformer ; *vars* est un vecteur de variables booléennes associées à chaque registre de FSM ou un signal d'entrée ; *current* et *next* représentent l'état courant et suivant de FSM respectivement.

On doit clarifier dans une expression logique ayant des prédicats sur l'état, si la variable d'état correspond à l'état courant ou l'état suivant d'une FSM. Une donnée *stateType* est introduite dans la classe *LogicExpr* à cette fin. Toutes les variables d'état correspondent à l'état courant d'une machine à états finis sauf celles utilisées pour représenter la valeur d'une fonction de transition qui correspond à l'état suivant de la machine à états finis.

Un registre R est modélisé par deux variables booléennes B_1 et B_2 dont la première B_1 mémorise la valeur de sortie de R dans le prochain cycle déterminée par une affectation de registre et la deuxième B_2 représente la valeur de sortie du registre pour le cycle courant. Selon la sémantique de chaque expression logique, le nom du registre R peut correspondre à soit B_1 soit B_2.

Au niveau de l'implantation, pour faciliter l'interprétation d'une expression logique (instance de la classe *LogicExpr*) en contrainte, on introduit deux tableaux *LeftSide* et *RightSide* du type *IlcBoolVarArray* qui sont construits en attachant respectivement les entrées et les sorties des registres à la fin de la liste de signaux de sortie et celle de signaux d'entrée. Selon ce que leurs noms indiquent, une variable de *LeftSide* (*RightSide*) remplace toujours un nom de signal ou un nom de registre du côté gauche (droite) d'une affectation (l'instance de la classe *Assignment*). Pour une expression logique décrivant l'affectation d'un signal interne d'un contrôleur ou la fonction de transition, seules les variables de *RightSide* sont utilisées. On donne un exemple concret d'introduction de tableaux *LeftSide* et *RightSide* dans le chapitre 4.

Il faut noter que les affectations des signaux internes d'un contrôleur sont également exprimées à l'aide d'un tableau du type *IlcAnyArray* dont chaque élément est un pointeur sur une instance de la classe *LogicExpr*. Tous les prédicats du type composite d'une instance de la classe *LogicExpr* sont obtenus à partir de ce tableau.

3.4.2 La classe FSM

FSM est une classe assez complexe qui permet de modéliser une machine à états finis de type *Moore* ou *Mealy*. Les classes *FsmState*, *Transition*, *LogicExpr*, *Assignment* et *AsynchronousOutput* sont exploitées pour simplifier la définition de la classe FSM.

Cette classe comprend les données membres suivantes :

- *const char* name* : le nom de FSM ;
- *FsmState* original_state* : l'état initial de FSM ;
- *IlcAnyArray states* : l'ensemble des états de FSM. En effet, chaque élément de *states* est une valeur du type *FsmState* qui associe une chaîne de caractères à un nombre entier. Le premier état *states[0]* contient l'état initial ;
- *IlcInt nbInputs* : il indique le nombre des entrées d'une FSM correspondant ;
- *char** *inputNames* : il indique les noms des signaux d'entrée qui doivent correspondre aux noms des ports du chronogramme associé ;
- *IlcInt nbOutputs*; le nombre des signaux de sortie de FSM ;
- *char** *outputNames*; les noms des signaux sorties qui doivent correspondre aux noms des ports du chronogramme associé ;
- *IlcInt nbRegisters*; le nombre des registres synchrones utilisés dans un FSM ;
- *char** *registerNames*; une liste des noms des registres introduits dans un FSM ;
- *IlcAnyArray transitions*; un vecteur des objets du type *Transition* qui regroupe les deux expressions logiques *c1* et *c2* telles que $c1 \Rightarrow c2$ à l'aide de la fonction *IlcIf-Then* du Solver
- *IlcAnyArray assignments*; un vecteur des objets du type *Assignment* qui permet de modéliser toutes les affectations d'un registre de la manière : $X = expr$, où X est le nom d'un registre ou d'un signal de sortie et *expr* est une expression logique.
- *IlcAnyArray asynchronous*; un vecteur des objets du type *AsynchronousOutput* qui permet à l'utilisateur d'établir la relation entre certains événements de sortie et ceux d'entrée.

Le constructeur de cette classe est le suivant :

FSM (IlcManager m, char nm, IlcAnyArray States, IlcInt Inputs, char** iname,*
*IlcInt Outputs, char** oname, IlcInt Registers, char** rname, IlcAnyArray tr,*
IlcAnyArray as, IlcAnyArray asyn, FsmState* original);

3.4.3 La vérification d'un contrôleur d'interfaces

Il s'agit de l'objectif final de notre développement. La mise en œuvre de la méthode de vérification d'un contrôleur d'interface présentée dans le chapitre 2 consiste en un but et un programme principal de test. On crée trois gestionnaires communiquant dont les deux premiers occupent des pré-traitements et le troisième se charge de la vérification du contrôleur proprement dit.

- Le but *Create_constraint_system*

Le but *Create_constraint_system* se charge de la tâche principale de la vérification d'un contrôleur. Le prototype de ce but est le suivant :

ILCGOAL6(Create_constrait_system, IlcInt, clk, IlcIntVar current,
IlcBoolVarArray, regOutput, IlcIntVarArray, presentStates,
IlcFloatVarArray Occurs, IlcIntVarArray, clist) ;*

Ce module consiste en un appel récursif qui déroule la procédure de vérification de contrôleur (voir section 2.3.4) jusqu'à *Maxcycle* fois. *Maxcycle* est la séparation temporelle entre l'événement origine du chronogramme TD et l'événement cible de la contrainte d'engagement à vérifier. *clk* ($0 \leq clk \leq Maxcycle$) est le compteur de boucle. *current* représente l'état courant de FSM, *regOutput* représente la valeur de sortie de l'ensemble des registres, presentStates indique l'état courant de WTA pour chaque port. *Occurs* représente le temps d'occurrence des événements. *clist* est un tableau de valeur (0 ou 1) de *nbEvents* (le nombre total des événements) lignes et de *Maxcycle* colonnes

dont chaque élément *clist(i,j)* représente la possibilité d'occurrence de l'événement *events[i]* dans le cycle *clk* = *j*, où *events* est un tableau qui représente l'ensemble des événements spécifiés dans un chronogramme. Quand *clk* = *Maxcycle*, un but *generate_clist* est exécuté qui se charge d'appeler le but *IlcGenerate(clist[i])* pour chaque ligne du tableau *clist*.

La communication entre un FSM et un chronogramme TD se réalise par la propagation de l'ensemble de contraintes caractérisant l'ensemble de traces temporisées TR_{TD} et celui de TR_M (voir section 2.3.3). Ces contraintes sont fournies sous forme de chaîne de caractères et leur interprétation en contrainte se réalise à l'aide de la fonction *translate_logicExpr* présentée dans la section 3.4.1.

- Le programme principal

Pour bien distinguer le pré-traitement et la tâche principale, le programme principal crée trois gestionnaires *m1*, *m2* et *m3*. Les deux premiers gestionnaires *m1* et *m2* se chargent de pré-traitement. Le gestionnaire *m1* calcule la séparation temporelle des événements et les valeurs *mincycle* et *maxcycle* pour chaque événement, puis transfère le résultat au gestionnaire *m3* qui se charge de la tâche principale. Le gestionnaire *m2* cherche une bloc-machine pour le chronogramme donné et transfère également le résultat au gestionnaire *m3*. Le gestionnaire *m3* se charge de la vérification de l'implantation d'un contrôleur à l'aide des résultats fournis par les gestionnaires *m1* et *m2*. La figure 3.7 illustre l'algorithme du programme principal.

```
Début
   Créer trois gestionnaires m1, m2 et m3 en mode IlcNoEdit;
   Saisir un chronogramme TD avec m1 ;
      Calculer pour chaque événement les valeurs mincycles et maxcycles pour m3 ;
   Saisir  TD avec m2 et chercher une bloc-machine associée à TD ;
      Transférer le résultat au m ;
   Saisir TD avec m3 et une machine à états finis FSM ;
      Initialiser le temps d'occurrence pour chaque événement ;
     Pour chaque bloc de la bloc-machine blk ordonné selon le rang, faire
         Si blk est un bloc ayant la direction in, alors ajoute toutes les contraintes
            assume du bloc en appelant le module execute_blocks ;
         Sinon ranger les contraintes locales du bloc blk en ordre croissant du
            Maxcycle de l'événement cible de chaque contrainte ;
            Pour chaque contrainte commit c, faire :
               Initialiser IlcBool success = IlcFalse ;
               Déterminer le nombre de boucles à dérouler Maxcycle
               Initialiser un vecteur représentant les valeurs de sortie des registres ;
               Construire un tableau (0 ou 1) clist(nbEvents×Maxcycle) ;
               Ajouter un but generate_constraint_system ;
               Ajouter le complément de la contrainte c ;
               Ajouter un but qui instancie le temps d'occurrence des événements dont la
                 valeur maxcycle satisfait : maxcycle ≤Maxcycle ;
              Si m.nextSolution() = IlcTrue, la contrainte c est violée; break ;
               Sinon la contrainte c est validée, retirer tous les buts, ajouter c comme
                  une contrainte redondante,  assigner Success = IlcTrue ;
               Si success = IlcFalse, break ;
           Fin pour chaque contrainte c ;
        Fin sinon ;
  Fin pour chaque bloc ;
Fermer tous les gestionnaires ;
Fin Début
```

Figure 3.7 L'organisation du programme principal

3.5 Saisie d'un chronogramme

Comme ce que nous avons défini dans le chapitre 1, un chronogramme est composé d'un ensemble de ports et un ensemble de contraintes sur le temps d'occurrence des événements spécifiés attachés à ces ports. Un fichier définissant un chronogramme doit contenir ces informations. La figure 3.8 illustre le format du fichier définissant un chronogramme[1]. Les paramètres sont délimités par des espaces.

```
nbPorts   nbConstraints
name  mode  nbStates  initialValue
                ...                          } répéter nbPorts fois
name  mode  nbStates  initialValue
name  comp  nbComposants                % répéter nbConstraints fois
relname  sourcePortName  sourceInd  sinkPortName  sinkInd  Dmin  Dmax
                              ...        % répéter nbComposants fois
relname  sourcePortName  sourceInd  sinkPortName  sinkInd  Dmin  Dmax
```

Figure 3.8 Le format du fichier définissant un chronogramme

Les paramètres sont expliqués comme suit.

IlcInt NbPorts, nbConstraints : le nombre des ports et le nombre des contraintes;

Il faut fournir les paramètres suivants pour définir un port :

char name, modeType mode, IlcInt nbStates, IlcBool initValue* : les données membres de la classe *Port*.

---

[1] Il s'agit d'un chronogramme feuille spécifique acceptable par l'outil développé dans le but de vérification de contrôleur d'interfaces. Selon ce que nous avons dit dans le chapitre 1, certaines limites sont imposées sur le chronogramme telles que : les ports du type message et les événements non-distinguables ne sont pas acceptables.

Pour définir une contrainte, les paramètres suivants sont nécessaires :
char name, compositeType comp, IlcInt nbComposant* : le nom, le type de contrainte et le nombre des relations temporelles composant cette contrainte.

Pour chaque composant, il faut fournir les paramètres suivants :
- char* *relname* : le nom de la relation temporelle ;
- char* *sourcePortName* : le nom du port sur lequel attache l'événement source;
- IlcInt *sourceInd* : l'index de l'événement source sur le port *sourcePortName* ;
- char* *sinkPortName* : le nom du port sur lequel on trouve l'événement cible ;
- IlcInt *sinkInd* : l'index de l'événement cible sur le port *sinkPortName* ;
- IlcFloat *Dmin, Dmax*: les bornes inférieur et supérieure associé au composant.

Par exemple, le fichier suivant définit le chronogramme présenté dans la figure 2.5.

```
2   4                          % 2 ports et 4 contraintes
X   in   3   0
Y   out   3   0
Asm1  linear 1  rel1  X  -1  X  0  1  16
Cmt1  linear 1  rel2 X  0  Y  0  70   80
Asm2  linear 1 rel3  Y  0  X  1  61  78
Cmt2  linear 1 rel4  X  1  Y  1  70  80
```

Il faut noter qu'un index négatif sur n'importe quel port correspond à l'événement original du chronogramme à construire et que le séparateur des champs est un simple espace ou changement de ligne. Il existe un module *readTimingDiagram* pour saisir un chronogramme d'un fichier :

TimingDiagram readTimingDiagram(IlcManager m, char* filename);*

3.6 Saisie d'une machine à états finis (FSM)

Nous définissons une FSM en introduisant les paramètres divisés en 5 parties :

1) Les paramètres généraux ;
2) Les paramètres définissant l'affectation des signaux internes ;
3) Les paramètres définissant la fonction de transition ;
4) Les paramètres définissant l'affectation de registres ou de la sortie du contrôleur ;
5) Les paramètres définissant les instances de la classe *AsynchronousOutput*.

Le format d'un fichier définissant une machine à états finis est illustré par la figure 3.9.

```
% les paramètres généraux
name Period nbStates nbInputs nbOutputs nbRegistres nbAssignments nbTransi-
tions nbLexpr nbAsynchronousOutputs
stateName1  stateName2 ...  stateNamek  ( k = nbStates)
InputName1  InputName2 ...  InputNamem ( m = nbInputs)
OutputsName1  OutputName2 ...  OutputNamen (n = nbOutputs)
RegistreName1  RegistreName2 ...  RegistreNamel ( l = nbRegistres)
InitRegistre1  InitiRegistre2 ...  InitRegistrei (i = nbRegistres)

% les paramètres pour saisir une instance de la classe LogicExpr
% ici ces instances de LogicExpr représente l'affectation d'un signal interne
nbtermes nbLogicElement nbStateElement
nbElements1  nbElements2 ...  nbElementsj (j = nbTermes)
type1  type2 ...  types (s = ΣnbElementsj et j = nbTermes)      répéter
varname  value (si type = signal)  % pour chaque prédicat        nbLexpr
value  statename (si type = état)                                fois
value  index (si type = composite)

% saisir une instance de la classe Transition (répéter nbTransitions fois)
iden  condition (instance de LogicExpr)  conclusion (instance de LogicExpr)

% saisir une instance de la classe Assignment (répéter nbAssignments fois)
iden  label  value (une instance de la classe LogicExpr)

% saisir une instance de AsynchronousOutput (nbAsynchronousOutputs fois)
outName  op  nbEvents  evName1  evName2 ...  evNamen (n = nbEvents)
```

Figure 3.9 Le format du fichier définissant une FSM

Les paramètres généraux :

- *char* Name* : le nom de FSM à construire ;
- *IlcInt Period* : la période de l'horloge qui synchronise tous les registres de FSM ;
- *IlcInt NbStates* : nombre d'états de FSM ;
- *IlcInt NbInputs, nbOutputs* : nombre d'entrée et de sortie de FSM ;
- *IlcInt NbRegistres* : nombre de registres ;
- *IlcInt NbLexpr* : nombre d'expressions logiques internes[1] ;
- *IlcInt NbTranistions* : nombre d'instances de la classe *Transition* ;
- *IlcInt NbAssignments* : nombre d'instances de la classe *Assignment* ;
- *IlcInt NbAsynchronous* : nombre d'instances de la classe *AsynchronousOutput* ;
- *char** *stateNames* : les noms des états de FSM ; Chaque nom constitue un champ de donnée qui est séparé par un espace.
- *IlcInt InputNames, outputNames* : deux tableaux de chaînes de caractères qui représentent les noms des signaux d'entrée et de sortie de FSM. (ils correspondent aux noms des ports d'entrée et ceux de sortie du chronogramme associé dans le contexte de vérification d'un contrôleur) ;
- *char** *RegistreNames* : un tableau de chaînes de caractères qui représente les noms des registres ; Chaque nom est séparé par un espace.
- *IlcBoolArray InitRegistre* : un tableau de valeurs 0 ou 1 qui représentent la valeur initiale de chaque registre. Chaque valeur est séparée par un espace.

Avant de saisir la fonction de transition et l'affectation des registres ou celle de sortie d'un contrôleur, on doit d'abord introduire les expressions logiques internes qui représentent l'affectation des signaux internes. Car les prédicats composites dans les autres instances de la classe *LogicExpr* permettant d'exprimer la fonction de transition et l'affectation des registres ou celle de sortie d'un contrôleur sont introduits à l'aide de ces expression logiques internes.

---

[1] On appelle 'expression logique interne' une instance de la classe LogicExpr qui exprime l'affectation d'un signal interne d'un contrôleur.

Saisie des expressions logiques internes

Pour chaque expression logique interne qui représente l'affectation d'un signal interne, on doit fournir :

- *IlcInt Nbterms, nbLogicElement, nbStateElement* : le nombre de termes, le nombre des prédicats sur un signal et le nombre des prédicats sur l'état ;
- *IlcIntArray NbElements* : un vecteur de nombres entiers qui représentent le nombre des prédicats dans chaque terme ;
- *IlcIntArray Types* : un vecteur de nombres entiers qui indiquent le type de chaque prédicat (0 signal, 1 état, 2 composite).

Pour chaque prédicat :

- S'il s'agit d'un prédicat sur un signal (type = 0) , il faut fournir le nom du signal et une valeur 0 ou 1 représentant la valeur du prédicat, par exemple "!X" est traduit comme "X 0" ;
- S'il s'agit d'un prédicat sur l'état (type = 1), il faut fournir une valeur 0 ou 1 représentant la valeur du prédicat et le nom d'un état de FSM, par exemple "current = IDLE" est représenté par "1 IDLE".

Par exemple, l'expression *((current = IDLE) && X) || !Y* peut être introduite par :

2 2 1 2 1 1 0 0 1 IDLE X 1 Y 0

Ces paramètres indique que l'expression contient *2* termes dans lesquels il y a *2* prédicats sur signal et *1* prédicat sur l'état. Le premier terme contient *2* prédicats et le deuxième terme contient *1* prédicat. Le type de ces trois prédicats est *1 0 0* respectivement. Les trois prédicats sont (*current = IDLE*), *X* et *!Y* respectivement.

Un tableau de type *IlcAnyArray lexpr* est ainsi construit qui enregistre toutes les expressions logique internes.

Saisie de fonctions de transition

Pour chaque fonction de transition (instance de la classe *Transition*) qui est composé d'un nom et deux expressions logiques *condition* et *conclusion*, on doit fournir :

- *char* Name*: le nom de la fonction de transition ;
- Les paramètres pour construire *condition* sont comme ceux pour les expressions logiques internes. Ici, *condition* peut contenir des prédicats composites, c'est-à-dire *type* = 2. Dans ce cas, on doit fournir :
  1) *val* : une valeur 0 ou 1 indiquant la valeur de ce prédicat composite, et
  2) *index* : l'index de l'expression logique dans le tableau des expressions logiques internes *lexpr* construit précédemment.
- Les paramètres pour construire *conclusion* sont similaires à ceux de condition.

Saisie des affectations

Une affectation (instance de la classe *Assignment*) est composée d'un nom de variable booléenne *label* et d'une expression logique *expr*. Donc, pour chaque affectation, il faut fournir :

- *char* Name* : le nom de l'affectation ;
- *char* Label* : le nom de la variable booléenne affectée ;
- Les paramètres pour construire une expression logique *expr* sont comme ceux pour la condition de fonction de transition (les prédicats du type composite sont permis).

Saisie des instances de la classe AsynchronousOutput

Une instance de la classe *AsynchronousOutput* est composée d'un nom d'événement de sortie sur un port de sortie asynchrone, l'opérateur (*min* ou *max*) appliqué et une liste des noms des événements d'entrée intervenant. Donc, les paramètres demandés sont les suivants pour chaque instance :

- *char* OutName* : le nom d'un événement de sortie ;
- *compositeType Op* : l'opérateur utilisé (*min* ou *max*) ;

- *IlcInt NbEvents* : le nombre d'événements d'entrée impliqués ;
- *char** EvNames* : un tableau de chaînes de caractères donne les noms des événements d'entrée intervenant.

Il existe un module *readFSM* pour saisir une machine à états finis d'un fichier :

FSM readFSM(IlcManager m, char* filename);*

Comme un exemple, le fichier présenté dans l'annexe A.1 définit la machine à états finis présentée dans le chapitre 2 (voir figure 2.4). L'annexe B.2 définit l'implantation d'un contrôleur décrite dans le chapitre 4.

3.7 Conclusions

Dans ce chapitre, nous avons présenté les composants importants de notre outil ainsi leur structure. Ce développement profite bien du paradigme orienté objet qui rend la modélisation des connaissances d'un domaine d'application plus simple et compréhensible. La librairie de ILOG Solver fournit un ensemble de classes et fonctions prédéfini qui facilite beaucoup le développement d'un outil de résolution de problèmes à l'aide de technique de programmation par contraintes. Mais l'efficacité d'un programme développé basée sur ILOG Solver demande non seulement l'expérience du domaine d'application, mais aussi une étude approfondie sur les techniques de programmation par contraintes, et la sémantique de toutes les classes dans la libraire qui contient plus de 300 classes, fonctions et concepts spéciaux.

Le saisie d'un chronogramme et celui d'une machine à états finis sont implantés actuellement de façon simplifiée car aucun traitement de texte n'est pris en compte dans le but d'améliorer la lisibilité de fichiers.

Dans le chapitre suivant, nous présentons un exemple d'application de la vérification d'un contrôleur d'interface en exploitant notre outil.

Chapitre 4 La vérification d'un contrôleur d'interface de mémoire

Dans ce chapitre, nous prenons un exemple d'application extrait de [3] en appliquant l'outil présenté dans le chapitre précédent. Dans cette expérience, le contrôleur d'interfaces considéré est plus compliqué que celui illustré dans les chapitres précédents car les sorties synchrones et asynchrones du contrôleur existent simultanément. Heureusement, cette complexité est déjà prise en compte dans l'implantation de la classe FSM.

Dans la section 4.1, nous présentons d'abord la spécification du contrôleur à vérifier en donnant un chronogramme sur deux cycles d'écriture consécutifs, puis nous expliquons comment formuler un fichier qui définit ce chronogramme. Dans la section 4.2, nous décrivons d'abord l'implantation du contrôleur sous forme de FSM, puis expliquons la formulation du fichier qui correspond à cette implantation. Dans la section 4.3, nous détaillons la procédure de vérification en expliquant les trois aspects importants : le calcul de séparation temporelle entre les événements, la vérification de causalité d'un chronogramme et la vérification de l'implantation d'un contrôleur en donnant les résultats expérimentaux et une analyse de ces résultats. La section 4.4 conclut cette expérience en indiquant certaines différences entre notre résultat et celui présenté dans [3].

4.1 La spécification d'un contrôleur d'interfaces

Le contrôleur d'interface considéré dans ce chapitre réalise la communication entre un microprocesseur et l'intérieur d'un ASIC (Application Specific Integrated Circuit) qui possède des registres et/ou mémoires accessibles. La figure 4.1 illustre un bloc-diagramme simplifié du contrôleur.

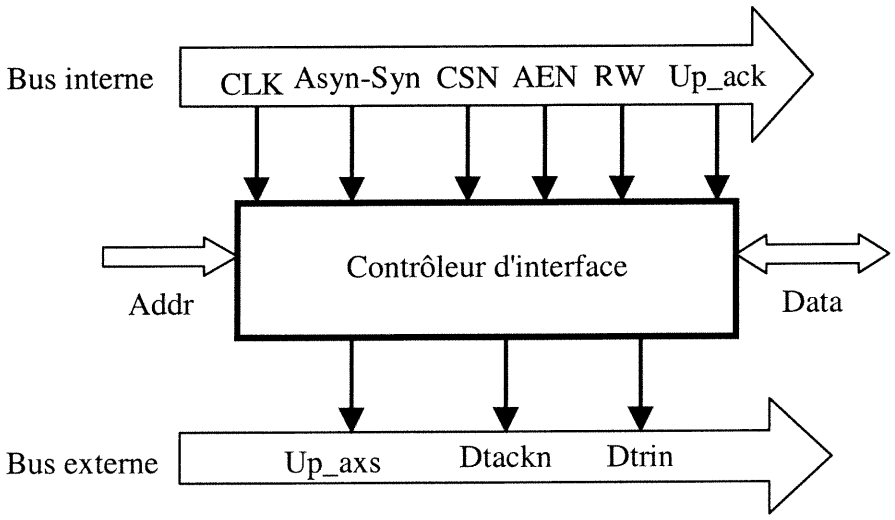


Figure 4.1 Le bloc-diagramme du contrôleur d'interface

4.1.1 Le fonctionnement du contrôleur

Le contrôleur peut fonctionner en mode synchrone (asynchrone) en mettant la broche *Asyn-Syn* à 0 (1). Dans cette expérience, le contrôleur est vérifié en mode asynchrone (*Asyn-Syn* = 1). La broche *CLK* est destinée à recevoir le signal de l'horloge système. Le contrôleur est mis en marche quand la broche *CSN* (Chip Select piN) est à 0.

Sur le bus interne, une transmission de données est initialisée quand la broche *AEN* (*the read/write access enable pin*) est à 0 qui signifie que l'adresse (*Addr*) déposée sur le bus d'adresse est valide et qu'une opération de lecture ou d'écriture peut avoir lieu. La broche *RW* à 1 (0) indique qu'un accès de lecture (d'écriture) est demandé.

Dès que la broche *AEN* est à 0, le contrôleur met la broche *Dtrin* à 1 et mémorise l'adresse sur le bus d'adresse. Pendant un accès d'écriture, le contrôleur met la broche *Dtackn* à 0 pour indiquer que la transmission de données demandée peut procéder. Puis la broche *AEN* est à 1 pendant l'accès d'écriture pour indiquer que la donnée sur le bus de données [7:0] (*Data*) est valide. La broche *Dtackn* est ignorée si aucune transmission de données est en cours.

Sur le bus externe, la lecture ou l'écriture de données est réalisée par un bus intérieur en chaîne (*daisy-chained*). En mode asynchrone, le contrôleur envoie aux blocs intérieurs un pulse d'un cycle de l'horloge sur le port *Up_axs* pour chaque accès de lecture ou d'écriture en espérant recevoir un signal de confirmation (*acknowledge*) (*Up_ack* = 1) qui signifie l'achèvement de l'accès.

4.1.2 La spécification sous forme de chronogramme

On obtient un chronogramme décrivant deux cycles d'écriture consécutifs du contrôleur à partir de la spécification générale du contrôleur. La figure 4.2 montre le chronogramme dérivé qui possède 9 signaux : *Addr, Data, CSN, AEN, RW, Up_axs, Up_ack, Dtackn* et *Dtrin*.

Il faut noter que les ports *Addr* et *Data* ne sont pas pris en compte dans la discussion suivante parce qu'ils possèdent une interprétation message et que notre méthode de vérification de contrôleurs d'interfaces ne traite que des ports ayant de l'interprétation signal (voir définition 1.1).

4.1.3 Le fichier correspondant au chronogramme

Le chronogramme présenté dans la figure 4.2 possède 7 portes (4 entrées : *CSN*, *AEN*, *RW* et *Up_ack*; 3 sorties : *Up_axs*, *Dtackn* et *Dtrin*) et 22 contraintes linéaires (12 *assume* et 10 *commit*). Le fichier définissant ce chronogramme est présenté dans l'annexe B.1. Une relation temporelle est introduite qui fixe la séparation temporelle entre le premier événement du port *AEN* et l'événement original du chronogramme à 19. Cette relation temporelle est nécessaire pour restreindre la séparation temporelle des autres événements par rapport à l'événement original.

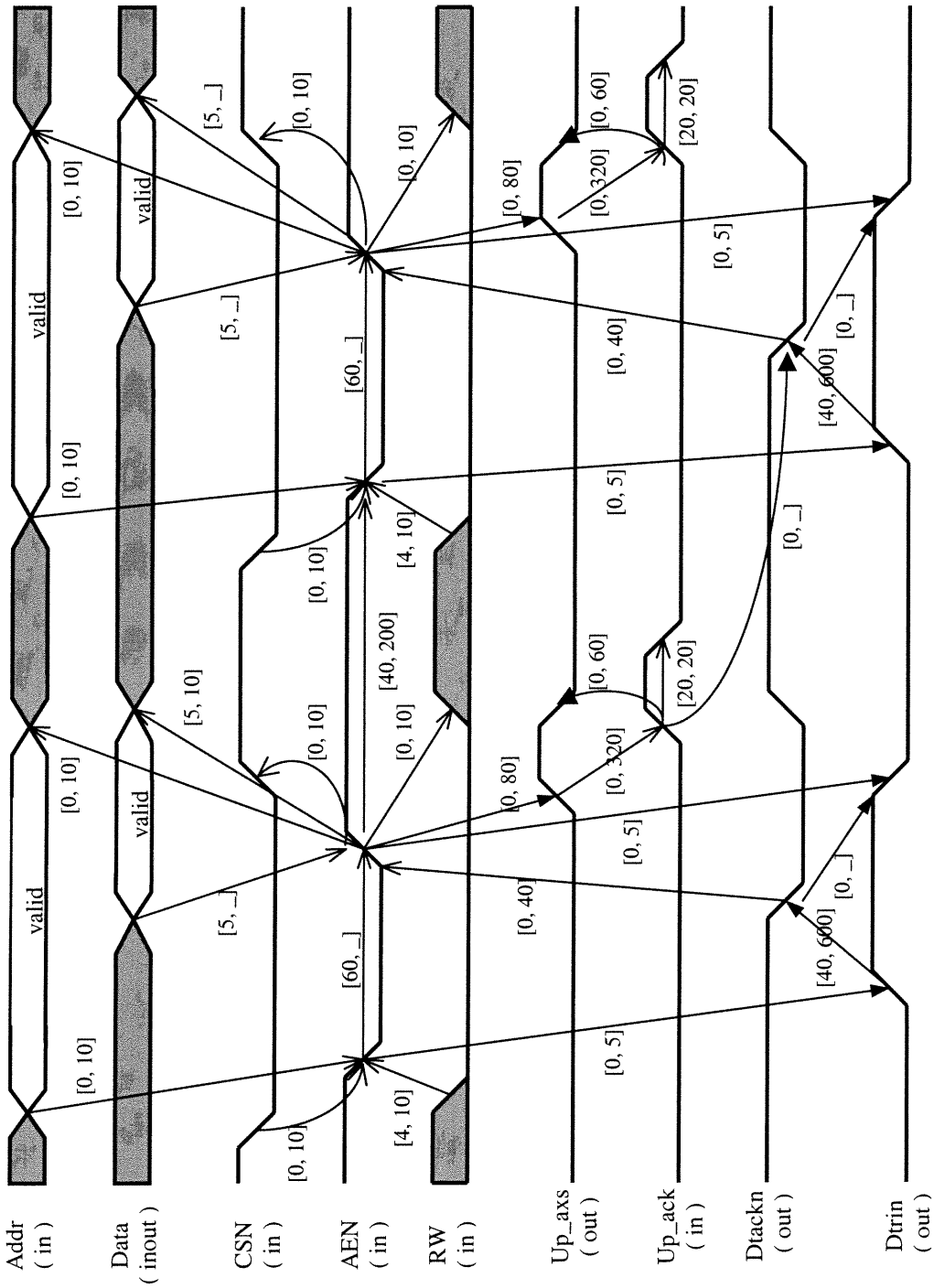


Figure 4.2 Chronogramme du contrôleur sur deux cycles d'écriture consécutifs

4.2 L'implantation du contrôleur

Pour définir l'implantation d'un contrôleur avec un fichier, on doit préciser les entrées, les sorties du contrôleur et les registres utilisés. On doit identifier les signaux internes et trouver leur affectations comme instance de la classe *LogicExpr*. On doit également formuler la fonction de transition comme instance de la classe *Transition* et les affectations de registres et des sorties du contrôleur comme instance de la classe *Assignment* et définir éventuellement l'affectation de temps d'occurrence des événements attachés sur les ports asynchrones comme instance de la classe *AsynchronousOutput*. L'implantation du contrôleur est illustrée dans la figure 4.3.

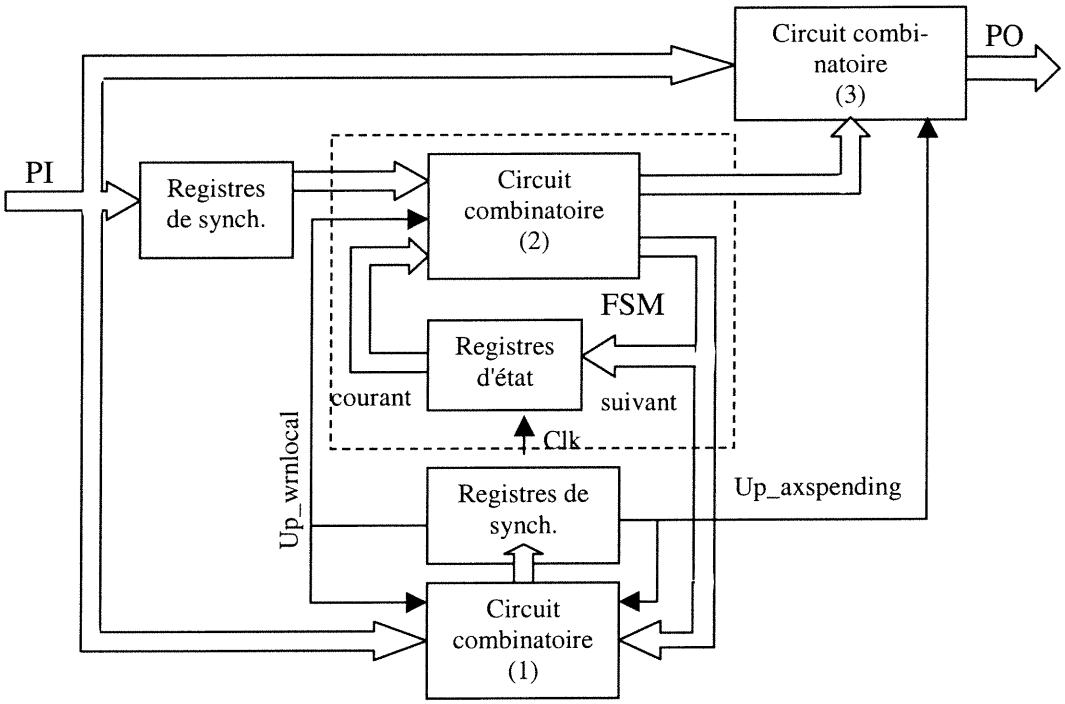


Figure 4.3 L'implantation du contrôleur d'interface

Les entrées primaires (PI: *Primary Inputs*) sont échantillonnées par des registres de synchronisation dont les sorties constituent une partie de signaux d'entrée de FSM. La FSM elle-même est composée du circuit combinatoire (2) et des registres d'état. Une combinaison des sorties de FSM avec PIs déterminée par le circuit combinatoire (1) sont

échantillonnées par deux registres appelés *Up_wrnlocal* et *Up_axspending*. Le circuit combinatoire (2) de FSM est alimenté par la sortie des registres qui échantillonnent PIs, la sortie du registre *Up_wrnlocal* et les sorties des registres d'état. Les sorties primaires (PO: *Primary Outputs*) du contrôleur sont générées à partir de PI, la sortie du registre *Up_axespending* et la sortie de FSM à travers le circuit combinatoire (3).

Dans cette section, nous détaillons chaque composant de l'implantation du contrôleur qui doit être modélisé et introduit par un fichier indiqué au début de cette section.

4.2.1 Les registres synchrones

Ils existent 8 registres qui sont étiquetés respectivement par *CSN_R1*, *CSN_R2*, *AEN_R1*, *AEN_R2*, *RW_R*, *Up_ack_R*, *Up_wrnlocal*, *Up_axspending*. Comme ce qu'on a déjà indiqué dans le chapitre 3, on introduit deux vecteurs de variables booléennes *RightSide* et *LeftSide* pour exprimer les affectations des registres, celles de la sortie de FSM et celles du contrôleur d'interfaces.

Le tableau 4.1 donne les affectations de *LeftSide* et celles de *RightSide*. Il faut remarquer que les deux éléments ayant le même index dans *LeftSide* et *RightSide* ne correspondent pas forcement à l'entrée et à la sortie d'un même registre parce que le nombre des signaux d'entrée PIs et celui des signaux de sortie POs d'un contrôleur peuvent être différents.

Selon ce que leurs noms indiquent, une variable de *LeftSide* (*RightSide*) remplace toujours par la correspondance indiquée dans le tableau 4.1, un nom de signal ou un nom de registre à côté gauche (droite) d'une affectation.

LeftSide	Représentation	RightSide	Représentation
LeftSide[0]	Up_axs (PO)	RightSide[0]	CSN (PI)
LeftSide[1]	Dtackn (PO)	RightSide[1]	AEN (PI)
LeftSide[2]	Dtrin (PO)	RightSide[2]	RW (PI)
LeftSide[3]	Entrée de CSN_R1	RightSide[3]	Up_ack (PI)
LeftSide[4]	Entrée de CSN_R2	RightSide[4]	Sortie de CSN_R1
LeftSide[5]	Entrée de AEN_R1	RightSide[5]	Sortie de CSN_R2
LeftSide[6]	Entrée de AEN_R2	RightSide[6]	Sortie de AEN_R1
LeftSide[7]	Entrée de RW_R	RightSide[7]	Sortie de AEN_R2
LeftSide[8]	Entrée de Up_ack_R	RightSide[8]	Sortie de RW_R
LeftSide[9]	Entrée de Up_wrnlocal	RightSide[9]	Sortie de Up_ack_R
LeftSide[10]	Entrée de Up_axspending	RightSide[10]	Sortie de Up_wrnlocal
		RightSide[11]	Sortie de Up_axspending

Tableau 4.1 Introduction de LeftSide et RightSide

Par exemple, l'affectation suivante signifie que l'entrée de *CSN_R2* est la sortie de *CSN_R1* (voir figure 4.4).

$$\text{CSN\_R2} \Leftarrow \text{CSN\_R1} \tag{4.1}$$

L'expression (4.1) est remplacée à l'intérieur par

$$\textit{LeftSide}[4] = \textit{RightSide}[4] \tag{4.2}$$

Où : *LeftSide*[4] et *RightSide*[4] dans le tableau 4.1 représentent l'entrée de *CSN_R2* et la sortie de *CSN_R1* respectivement.

Le signal d'entrée *CSN* est échantillonné à l'aide d'une cascade de deux registres *CSN_R1* et *CSN_R2* illustrée dans la figure 4.4. Une autre cascade de registres composée de *AEN_R1* et *AEN_R2*, similaire à celle ci-dessus se charge d'échantillonner le signal *AEN*. Les registres *RW_R* et *Up_ack_R* se chargent d'échantillonner les signaux d'entrée du contrôleur *RW* et *Up_ack* respectivement.

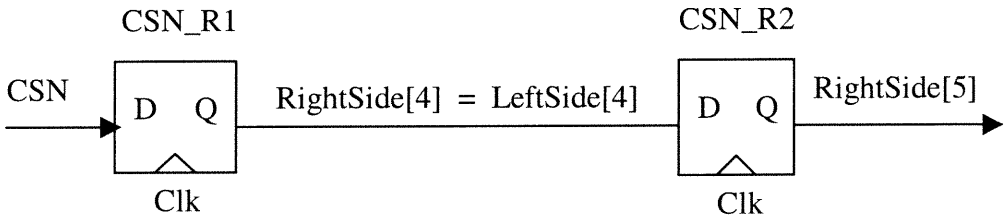


Figure 4.4 Une cascade de registres pour échantillonner le signal CSN

Les registres *Up_wrnlocal* et *Up_axspending* sont introduits pour mémoriser les valeurs passées des signaux internes. Leurs entrées sont déterminées par le circuit combinatoire (1) de la figure 4.3 décrit par les deux expressions suivantes :

$$Up\_wrnlocal = !Up\_wrnlocal\_rst \;\&\&\; (Up\_arwlatch \;\&\&\; !RW \;||\; !Up\_arwlatch \;\&\&\; Up\_wrnlocal) \quad (4.3)$$

$$Up\_axspending = (Up\_naxspls \;||\; Up\_axspending) \;\&\&\; !Up\_ack \quad (4.4)$$

Où *Up_wrnlocal_rst*, *Up_arwlatch* et *Up_naxspls* sont des sorties de FSM qui seront représentées par trois expressions logiques internes ; *RW*, *Up_ack* sont des entrées primaires (PI) du contrôleur.

Il faut noter que les deux *Up_wrnlocal* se trouvant à deux côtés de l'expression (4.3) correspondent aux variables différentes de *LeftSide* et de *RightSide*. *Up_wrnlocal* à la gauche correspond à *LeftSide*[9] et celui à la droite correspond à *RightSide*[10]. Pareil, on peut déduire que *Up_axspending* à la gauche de l'expression (4.4) correspond à *LeftSide*[10] et celui à la droite correspond à *RightSide*[11].

Pour contourner les limites imposées[1] dans l'implantation de la classe *LogicExpr*, les expressions (4.3) et (4.4) sont réécrites en (4.5) et (4.6) respectivement.

$$Up\_wrnlocal = (!Up\_wrnlocal\_rst \;\&\&\; Up\_arwlatch \;\&\&\; !RW) \;||\; (!Up\_wrnlocal\_rst \;\&\&\; !Up\_arwlatch \;\&\&\; Up\_wrnlocal) \quad (4.5)$$

$$Up\_axspending = (Up\_naxspls \;\&\&\; !Up\_ack) \;||\; (Up\_axspending \;\&\&\; !Up\_ack) \quad (4.6)$$

En résumé, il y a 6 registres qui permettent de saisir les signaux d'entrée (PI) du contrôleur dont l'affectation est déterminée en trouvant la correspondance entre l'entrée et la sortie de chaque registre comme ce qu'on explique avec les expressions (4.1) et (4.2). L'affectation du registre *Up_wrnlocal* et celle du registre *Up_axspending* est déterminée par les expressions (4.5) et (4.6) respectivement.

Après de définir les registres, selon l'ordre de définition d'une FSM dans un fichier, on doit identifier la fonction de transition et celle de sortie de la FSM. Les sorties de la FSM sont des signaux internes (qui ne sont pas échantillonnés par des registres) et leur affectations seront exprimées par des instances de la classe LogicExpr.

4.2.2 La description de la FSM

La FSM à l'intérieur du contrôleur est dérivée originalement du code écrit en Verilog dans [3]. Il s'agit d'une machine à états finis du type *Mealy* ayant 4 états, 5 entrées et 4 sorties (voir tableau 4.2).

---

[1] Dans le chapitre 3, pour simplifier l'implantation de la classe LogicExpr, les limites sont imposées : une expression logique est composée d'un ensemble de termes liés par l'opérateur || et un terme ne contient qu'un ensemble de prédicats liés par l'opérateur &&. Un prédicat peut être de type signal, de type d'état ou de type composite. Un prédicat composite est dérivé d'une expression logique interne qui décrit l'affectation d'un signal interne qui n'est pas échantillonné par un registre.

États	Entrées	Sorties
IDLE : idle WAIT : write wait ACKW : acknowledge REND : read end	RightSide[5] : sortie de CSN_R2 RightSide[7] : sortie de AEN_R2 RightSide[8] : sortie de RW_R RightSide[9] : sortie de Up_ack_R RightSide[10] : sortie de Up_wrnlocal	Up_wrnlocal_rst Up_arwlatch Dtackn_rst Up_naxspls

Tableau 4.2 La description de FSM à l'intérieure du contrôleur

Le fonctionnement de FSM en mode asynchrone est illustré par la figure 4.5.

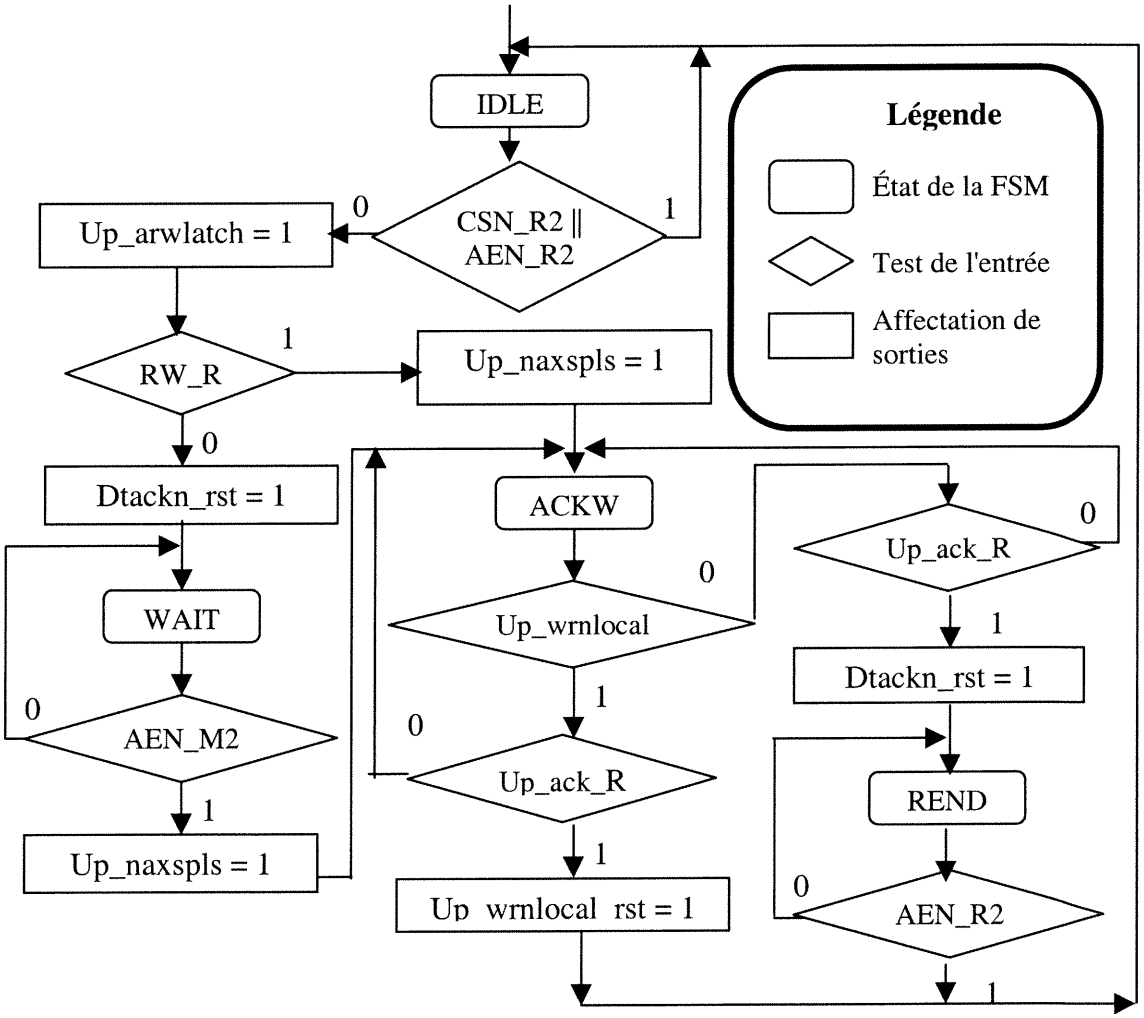


Figure 4.5 Le fonctionnement de FSM en mode asynchrone

Formellement, une FSM est modélisée en décrivant la fonction de transition et celle de sortie de la FSM. La fonction de transition de cette FSM est exprimée par les instances de la classe *Transition* illustrées dans le tableau 4.3 suivants, dans lequel *SP*, *SN* sont l'état courant et l'état suivant de la FSM respectivement :

No	Condition	Conclusion
1	if ((SP = IDLE) && (CSN_R2 || AEN_R2) || (SP = REND) && AEN_R2 || (SP = ACKW) && Up_wrnlocal && Up_ack_R)	then (SN = IDLE)
2	if ((SP = WAIT && !AEN_R2) || (SP = IDLE) && !(CSN_R2 || AEN_R2) && !RW_R)	then (SN = WAIT)
3	if ((SP = IDLE) && !(CSN_R2 || AEN_R2) && RW_R || (SP = WAIT) && !Up_ack_R || (SP = WAIT) && AEN_R2)	then (SN = ACKW)
4	if ((SP = ACKW) && !Up_wrnlocal && Up_ack_R || (SP = REND) && !AEN_R2)	then (SN = REND)

Tableau 4.3 La fonction de transition de la FSM

Rappelons que dans le tableau 4.3, les noms des registres *CSN_R2*, *AEN_R2*, *RW_R*, *Up_ack_R* et *Up_wrnlocal* correspondant à la sortie de ces registres seront remplacés par les variables booléennes de *RightSide*. (voir tableau 4.1)

Le tableau 4.4 montre les 4 instances de la classe *LogicExpr* qui affectent les 4 sorties de la FSM dans le tableau 4.2. Puisque les sorties de FSM sont des signaux internes pour le contrôleur, ces instances doivent être introduites avant la fonction de transition selon la syntaxe de définition d'une instance de la classe FSM présentée dans le chapitre 3. Après la définition de ces expressions logiques internes, un prédicat du type composite dans une expression logique est défini en indiquant la valeur du prédicat et l'index de la variable affectée dans le tableau 4.4. Par exemple, le prédicat *!Up_wrnlocal_rst* de l'expression (4.3) est représenté par "0 0" et le prédicat *Up_naxspls* de l'expression (4.4) est exprimé par "1 3" simplement.

Index	Variable affectée	Expression logique
0	Up_wrnlocal_rst	(SP = ACKW) && Up_wrnlocal && Up_ack_R
1	Up_arwlatch	(SP = IDLE) && !CSN_R2 && !AEN_R2
2	Dtackn_rst	(SP = IDLE) && !CSN_R2 && !AEN_R2 && !RW _R || (AP = ACKW) && !Up_wrnlocal && Up_ack_R
3	Up_naxspls	(SP = IDLE) && !CSN_R2 && !AEN_R2 && RW_R || (SP = WAIT) && AEN_R2

Tableau 4.4 Les fonctions de sortie de la FSM

La fonction de transition et celle de sortie de la FSM sont réalisées par le circuit combinatoire (2) de la Figure 4.3. Après l'introduction de la fonction de transition et celle de sortie, on doit formuler les affectations de registres et celles de la sortie du contrôleur. À part l'affectation de 8 registres qu'on a présentée dans la section 4.2.1, les sorties du contrôleur constituent les 3 autres affectations qu'on doit préciser.

4.2.3 Les sorties du contrôleur

Le contrôleur produit trois signaux de sorties *Up_axs*, *Dtackn* et *Dtrin* correspondant aux trois ports de sortie illustrés dans le chronogramme de la figure 4.2. Ces sorties sont exprimées par les trois instances de la classe *Assignment* présentées dans le tableau 4.5.

On constate que les sorties du contrôleur sont déterminées par les entrées primaires (PIs) du contrôleur (*CSN*, *AEN*), la sortie du registre *Up_axspending* et les sorties de la FSM *Up_naxspls* et *Dtackn_rst*. Les expressions logiques présentées dans le tableau 4.5 sont réalisées par le circuit combinatoire (3) de la figure 4.3.

No	Variable affectée	Expression logique
1	Up_axs	Up_naxspls || Up_axspending
2	Dtackn	CSN || AEN || !Dtackn_rst
3	Dtrin	!CSN && !AEN

Tableau 4.5 Les affectations de sorties du contrôleur

Après la présentation des affectations qui doivent être modélisées, selon la syntaxe du fichier qui définit une FSM, on doit vérifier s'il y a des événements attachés sur les ports de sortie asynchrones afin de modéliser l'affectation de temps d'occurrence de ces événements comme instance de la classe *AsynchronousOutput*.

4.2.4 La prise en compte des sorties asynchrones

Dans le chapitre 2, nous avons montré que si une sortie (représentée par un port de sortie) du contrôleur est synchrone, c'est-à-dire, sa valeur est déterminée par les sorties des registres synchrones, alors, l'occurrence d'un événement de sortie sur ce port peut être détectée simplement en comparant les états courant et suivant de l'onde automate correspondant. Si pour cet automate, l'état suivant est différent de l'état courant dans le cycle *Clk*, alors, un événement de sortie a lieu à *(Clk+1)*Period*, où *Period* est la période de l'horloge.

Si une sortie du contrôleur est asynchrone, il est plus difficile à déterminer le temps d'occurrence d'un événement sur ce port. Par exemple, selon l'expression logique dans le tableau 4.5 correspondant à la sortie du contrôleur *Dtrin*, on sait que les fronts montants sur le port *Dtrin* (correspondant aux événements *Dtrin0* et *Dtrin2*) sont produits par les fronts descendants des ports *CSN* et *AEN* (correspondant aux événements *CSN0*, *AEN0* et *CSN2*, *AEN2*). Le temps d'occurrence de l'événement *Dtrin0* (*Dtrin2*) est déterminé

par le plus récent des événements *CSN0* et *AEN0* (*CSN2* et *AEN2*) car *Dtrin* = !*CSN* && !*AEN*. Autrement dit, on peut déterminer le temps d'occurrence de *Dtrin0* et celui de *Dtrin2* par l'expression (4.7) et (4.8).

$$T_{Dtrin0} = max(T_{CSN0}, T_{AEN0}) \quad (4.7)$$

$$T_{Dtrin2} = max(T_{CSN2}, T_{AEN2}) \quad (4.8)$$

De la même façon, on peut déduire que le temps d'occurrence des événements correspondant aux fronts descendants du port *Dtrin* sont :

$$T_{Dtrin1} = min(T_{CSN1}, T_{AEN1}) \quad (4.9)$$

$$T_{Dtrin3} = min(T_{CSN3}, T_{AEN3}) \quad (4.10)$$

La détermination du temps d'occurrence des événements sur le port *Dtackn* est plus compliquée. Selon le tableau 4.5, *Dtackn* est déterminée soit par les entrées primaires *CSN* ou *AEN* soit par la sortie de FSM *Dtackn_rst* qui est synchronisée. En principe, on ne connaît pas la vraie cause du changement sur le signal *Dtackn*, mais le résultat de simulation sur *Dtackn_rst* indique que les fronts montants de *Dtackn* sont toujours générés par les fronts montants de *CSN* et *AEN*, et les fronts descendants de *Dtackn* sont toujours générés par les fronts montants de *Dtackn_rst*. Donc, pour l'onde automate correspondant au port Dtackn, si son état suivant est différent de son état courant dans le cycle *Clk*, on obtient :

$$T_{Dtackn1} = min(T_{CSN1}, T_{AEN1}) \quad (4.11)$$

$$T_{Dtackn3} = min(T_{CSN3}, T_{AEN3}) \quad (4.12)$$

$$T_{Dtackn0} = (Clk+1) * Period \quad (4.13)$$

$$T_{Dtackn2} = (Clk+1) * period \quad (4.14)$$

Les événements de sortie sur le port *Up_axs*, selon l'expression indiquée dans le tableau 4.5, sont déterminés par les signaux synchronisés, donc le temps d'occurrence des événements sur *Up_axs* peut être affecté à *(Clk+1)*period* si la valeur du signal Up_axs change dans le cycle Clk. Les expressions (4.7)-(4.12) correspondent aux 6 instances de la classe *AsynchronousOutput.*

4.2.5 Le fichier de définition de l'implantation du contrôleur

Le fichier définissant ce contrôleur est présenté dans l'annexe B.2. En résumé, cette implantation est composée de (selon l'ordre d'introduction dans le fichier) :

- 4 états FSM : *IDLE*, *WAIT*, *ACKW* et *REND* (instance de la classe *FsmState*). *IDLE* est considéré comme l'état initial de FSM ;
- *Period* = 20 (ns) : la période de l'horloge ;
- 4 entrées primaires (PI) : *CSN*, *AEN*, *RW* et *Up_ack* ;
- 3 sorties primaires (PO) : *Up_axs*, *Dtackn* et *Dtrin* ;
- 8 registres synchrones présentés dans le tableau 4.1 ;
- les valeurs initiales des registres : 1 1 1 1 1 0 0 0 ;
- 4 expressions logiques internes du tableau 4.4 ;
- 4 instances de la classe *Transition* correspondant au tableau 4.3 ;
- 11 affectations qui correspondent aux affectations de 8 registres synchrones et celles de 3 sorties du contrôleur présentées dans le tableau 4.5 ;
- 6 instances de la classe *AsynchronousOutput* sont introduites pour prendre en compte certaines sorties asynchrones du contrôleur.

Dans cette section, nous avons décrit les différents composants du contrôleur qui permettent de définir une instance de la classe FSM. Dans la section suivante, nous vérifions si cette implantation de contrôleur satisfait toutes les contraintes spécifiées dans le chronogramme décrit dans la section 4.1 en exploitant notre outil.

4.3 La procédure de vérification

Pour vérifier un contrôleur d'interfaces, une spécification sous forme de chronogramme et une implantation sous forme de FSM doivent être fournies sous forme de fichier comme ce qu'on a présenté dans les sections 4.1 et 4.2. Avec ces fichiers, nous obtenons une instance de la classe *TimingDiagram td* et une instance de la classe *FSM fsm*.

Dans les trois sous-sections suivantes, on présente la procédure de vérification en exploitant les trois aspects importants : le calcul de séparation entre les événements pour *td*, la vérification de causalité du chronogramme *td* et la vérification de l'implantation du contrôleur *fsm* par rapport au chronogramme *td*. Ces trois tâches sont réalisées par les trois gestionnaires qui communiquent les résultats entre eux.

4.3.1 Le calcul de séparation des événements

En appliquant la méthode *td.findDistances*, on peut obtenir la séparation temporelle de tous les événements (représenté par le tableau *events* obtenu avec *td.getEvents*()) par rapport à l'événement original de *td*. A partir de ces intervalles temporels, on peut déduire les deux vecteurs de nombres entiers *minCycle* et *maxCycle* tels que :

$\forall\ i$, l'événement *events*[*i*] pourrait se produire dans un cycle *Clk* si et seulement si

$$minCycle[i] \leq Clk \leq maxCycle[i] \tag{4.15}$$

On calcule non seulement *maxCycle*, mais aussi *minCycle* selon les expressions (4.16) et (4.17) dans lesquelles *tmin* et *tmax* sont les bornes inférieure et supérieure de la séparation correspondante.

$$minCycle = max(0, floor(tmin/Period)) \tag{4.16}$$

$$maxCycle = ceil(tmax / Period) + 1 \tag{4.17}$$

où *Period* est la période de l'horloge du système. *Period* = 20 (ns). En effet, le nombre de cycles à dérouler est *maxCycle*, le dernier cycle est pour l'énumération de la possibilité d'occurrence des événements d'entrée sur chaque cycle.

L'introduction de *minCycle* permet de simplifier le système de contraintes généré dans la vérification de l'implantation du contrôleur, car on n'a pas besoin de vérifier si un événement d'entrée *events[i]* se produit dans un cycle $Clk < minCycle$[i].

Index	Label	Séparation	MinCycle	maxCycle
0	Original	[0..0]	0	1
1	CSN0	[9..19]	0	1
2	CSN1	[79..674]	3	34
3	CSN2	[109..864]	5	44
4	CSN3	[179..1519]	8	76
5	AEN0	[19..19]	0	1
6	AEN1	[79..664]	3	34
7	AEN2	[119..864]	5	44
8	AEN3	[179..1509]	8	76
9	RW0	[9..15]	0	1
10	RW1	[79..674]	3	34
11	RW2	[109..860]	5	44
12	RW3	[179..1519]	8	76
13	Up_ack0	[79..1069]	3	54
14	Up_ack1	[99..1084]	4	55
15	Up_ack2	[179..1909]	8	96
16	Up_Ack3	[199..1929]	9	97
17	Up_axs0	[79..744]	3	38
18	Up_axs1	[79..1124]	3	57
19	Up_axs2	[179..1589]	8	80
20	Up_axs3	[179..1969]	8	99
21	Dtackn0	[59..624]	2	32
22	Dtackn1	[59..1469]	2	74
23	Dtackn2	[159..1469]	7	74
24	Dtackn3	[159..1e+100]	7	1.e+100 (1)
25	Dtrin0	[19..24]	0	2
26	Dtrin1	[79..669]	3	34
27	Dtrin2	[119..869]	5	44
28	Dtrin3	[179..1514]	8	76

Tableau 4.6 Les séparations temporelles, minCycle et maxCycle obtenus

[1] Il faut noter qu'on utilise dans l'outil un très grand nombre *1.e+100* pour représenter un nombre réel infini, et une constante 100000000 pour représenter un nombre entier infini.

On constate que le *maxCycle* de l'événement *Dtackn3* est infini car il n'existe aucune contrainte qui limite la borne supérieure du temps d'occurrence de *Dtackn3*. La borne inférieure du temps d'occurrence de *Dtackn3* est limitée par une contrainte implicite de préséance entre *Dtackn2* et *Dtackn3*. Par contre on a obtenu un *maxCycle* fini pour l'événement *Dtackn1* grâce à une contrainte implicite de préséance entre les deux événements consécutifs *Dtackn1* et *Dtackn2* parce que la borne supérieure du temps d'occurrence de *Dtackn2* est finie.

4.3.2 La vérification de la causalité

En appliquant le module *generateBlockMachine* présenté dans le chapitre précédent, on peut obtenir une bloc-machine causale ayant 21 blocs présentée dans le tableau 4.7.

Rang	Événements locaux	Rang	Événements locaux
Bloc0	CSN0, AEN0, RW0	Bloc11	Up_axs1
Bloc1	Dtrin0	Bloc12	Up_axs2
Bloc2	Dtackn0	Bloc13	Up_ack1
Bloc3	CSN1, CSN2, AEN1, AEN2, RW1, RW2	Bloc14	Up_ack2
Bloc4	Up_axs0	Bloc15	Up_axs3
Bloc5	Up_ack0	Bloc16	CSN3
Bloc6	Dtackn1	Bloc17	RW3
Bloc7	Dtrin1	Bloc18	Up_ack3
Bloc8	Dtrin2	Bloc19	Dtackn3
Bloc9	Dtackn2	Bloc20	Dtrin3
Bloc10	AEN3		

Tableau 4.7 Une causale bloc-machine du chronogramme de la Figure 4.2

Cette bloc-machine causale est obtenue avec un processus de recherche limitée dans lequel on impose que la recherche commence avec un nombre de blocs initial 20. Pour chaque nombre de bloc, le temps maximum de recherche est 2 secondes. Dans le cas d'échec, on augmente le nombre de blocs. Le gestionnaire utilisé pour cette tâche peut être initialisé par

IlcManager m(IlcNoEdit) ;

Les informations statistiques sur la recherche de cette bloc-machine obtenues à l'aide de la méthode *m.printInformation()* sont les suivantes :

```
Number of fails                 : 58
Number of choice points         : 66
Number of variables             : 3577
Number of constraints           : 4311
Reversible stack (bytes)        : 506544
Solver heap (bytes)             : 1865304
And stack (bytes)               : 4044
Or stack (bytes)                : 4044
Constraint queue (bytes)        : 16144
Total memory used (bytes)       : 2424292
Running time CPU (seconds)      : 0.57
```

L'explication de ces résultats est la suivante :

- le nombre des échecs rencontrés par le gestionnaire *m* est 58 ;
- le nombre des points de choix générés par le gestionnaire *m* est 66 ;
- le nombre des variables associées au gestionnaire *m* est 3577 ;
- le nombre des contraintes ajoutées au gestionnaire *m* est 4311 ;
- la taille de la pile de restauration (≈ 506 KB) qui est proportionnelle au nombre maximum des affectations réversibles ;
- la taille maximum du monceau d'allocation du Solver ≈ 1865KB ;
- la taille de la pile des buts (= 4044 Bytes) qui est proportionnelle au nombre d'appels à la fonction IlcAnd;
- la taille de la pile utilisée par la fonction IlcOr (= 4044 Bytes) qui est proportionnelle au nombre d'appels de la fonction IlcOr ;

- la taille de la queue de propagation de contraintes (=16144 Bytes) qui est proportionnelle au nombre des contraintes activées ;
- la taille de la mémoire utilisée par Solver pour le gestionnaire $m \approx 2424$ KB ;
- le temps d'exécution depuis la création du gestionnaire m est 0.57 seconde.

4.3.3 La vérification de l'implantation du contrôleur

On ajuste le rang des blocs pour que les contraintes puissent être vérifiées suivant le même ordre que celui utilisé dans [3]. Les blocs de la bloc-machine sont vérifiés un à un suivant l'ordre de leurs rangs. Dans cet exemple, on doit vérifier 10 contraintes *commit*. Puisque les délais peuvent être exprimés en nombre réel ou en nombre entier, pour comparer la performance de l'outil dans ces deux cas, on teste cet exemple avec deux versions du programme : une pour les délais en nombre réel (P1), l'autre pour les délais en nombre entier (P2). Notons que le programme utilisé dans [3] (noté P3) exprime les délais en nombre entier. Les programmes P1 et P2 ont été lancés sur une machine Sun Ultra-1 et aucune contrainte n'est violée.

Les résultats de la vérification sont les suivants :

No	Contrainte	Nb de cycles déroulés			Temps CPU (seconde)			Mémoire (KB)		
		P1	P2	P3	P1	P2	P3	P1	P2	P3
1	(AEN0, Dtrin0, [0, 5])	2	2	2	0.06	0.02	5.00	382	201	191
2	(Dtrin0, Dtackn0, [40, 600])	32	32	32	0.32	0.12	6.00	3252	3116	256
3	(Dtackn0, Dtrin1, [0, _]) (AEN1, Dtrin1, [0, 5])	34	34	35	3.00	0.96	56.0	3586	3445	3051
4	(AEN1, Up_axs0, [0, 80]) *[0, 60]*	38	38	38	7.62	2.55	10.0	3988	3843	656
5	(Up_ack0, Up_axs1, [0, 60]) *[0, 40]*	57	57	70	265	93	1350	5861	5705	6243
6	(AEN2, Dtrin2, [0, 5])	44	44	68	312	78	19687	5861	5705	6094
7	(Up_ack0, Dtackn2, [0, _]) (Dtrin2, Dtackn2, [40, 600])	74	74	88	280	101	12241	7465	7385	7988
8	(AEN3, Dtrin3, [0, 5]) (Dtackn2, Dtrin3, [0, _])	76	76	90	20720	3149	12452	7678	7518	8198
9	(AEN3, Up_axs2, [0, 80]) *[0, 60]*	80	80	90	9289	3292	6497	8048	7900	8084
10	(Up_ack2, Up_axs3, [0, 60]) *[0, 40]*	99	99	90	97493	35099	107699	9813	9668	8210

Tableau 4.8 Les résultats de vérification de l'implantation du contrôleur

L'ordre de vérification des contraintes *commit* (indiqué dans la première colonne du tableau 4.8) est obtenu à partir de la bloc-machine (voir tableau 4.7) de la façon sui-

vante : tout d'abord, les contraintes sont ordonnées partiellement suivant l'ordre croissant des rangs de blocs, puis les contraintes locales d'un bloc sont rangées selon l'ordre croissant des Maxcycles de l'événement cible de chaque contrainte.

Dans le tableau 4.8, chaque composant d'une contrainte dans la deuxième colonne est exprimé sous forme (source, cible, [intervalle]) dans lequel les événements sont étiquetés avec le nom du port suivi par l'index d'occurrence de l'événement sur ce port. Un délai en italique sous la contrainte indique le délai associé à cette contrainte utilisé dans [3].[1] Il faut noter que dans [3], une contrainte correspond à une relation temporelle. Donc, une contrainte ayant deux composants est considérée comme deux contraintes dans [3]. La troisième colonne correspond au nombre de cycles à dérouler pour vérifier une contrainte. Les deux dernières colonnes correspondent au temps d'exécution et à la mémoire utilisée pour vérifier la contrainte correspondante. Pour ces trois dernières colonnes, chaque case a trois colonnes qui représentent les résultats obtenus en utilisant les trois programmes P1, P2 et P3 respectivement.

Puisque dans [3], chaque relation temporelle est vérifiée individuellement et dans notre expérience, les relations temporelles ayant le même événement cible sont considérées comme les composants d'une seule contrainte, le temps CPU correspondant au résultat de [3] est la somme du temps d'exécution pour vérifier tous les composants, et la mémoire est le maximum des mémoires utilisées.

Maintenant, on peut analyser les résultats présentés dans le tableau 4.8. Tout d'abord, on précise certains paramètres qui sont susceptibles d'influencer la performance de l'outil. La valeur qui représente l'infini est 1.e100 pour P1 (réel) et 1e8 pour P2 (entier) et 2147483647 pour P3, le programme utilisé dans [3]. La précision par défaut des expressions du type flottant pour P1 est 1.e-4.

---

[1] Avec ce délai, la contrainte correspondante est violée et un contre-exemple est fourni par notre outil. C'est la raison pour laquelle on a modifié le délai de la contrainte correspondante.

On voit que pour les contraintes 1~4, les nombres de cycles à dérouler sont presque identiques pour les trois programmes et que la mémoire utilisée par P3 est généralement inférieure à celle utilisée par P1 et P2. Mais la variation de mémoire utilisée n'est pas proportionnelle au nombre de cycles à dérouler pour P3 car après avoir utilisé une mémoire de taille 3051 KB pour la contrainte numéro 3 avec un nombre de cycles à dérouler 35, la mémoire utilisée devient 656 KB pour la contrainte numéro 4 avec un nombre de cycles à dérouler 38. Pour le temps d'exécution, le programme P2 est nettement meilleur au programme P3 surtout pour la contrainte numéro 3.

Pour les contraintes 5~9, le nombre de cycles à dérouler utilisé pour P1 et P2 est inférieur à celui utilisé dans P3 et les mémoires utilisées pour ces trois programmes sont comparables. On constate que le programme P2 est environ 2-250 fois plus rapide que le programme P3.

Quant à la dernière contrainte, le nombre de cycles à dérouler pour P1 et P2 est plus grand que celui utilisé dans P3, P2 est 3 fois plus rapide que P3 avec une mémoire utilisée toujours comparable.

On voit que pour toutes les contraintes, le programme P2 est plus rapide que le programme P1 avec une mémoire utilisée légèrement inférieure. Ce résultat prouve ce qu'on a dit à propos de la représentation abstraite du temps physique dans la page 2 : un temps dense est plus complexe à manipuler.

Pour voir l'influence de l'ordre de vérification de chaque contrainte sur le temps d'exécution, on a fait un autre essai avec le programme P2 (entier) suivant l'ordre de vérification dérivé exactement de la bloc-machine du tableau 4.7. (indiqué par l'ordre 1) Le résultat obtenu est présenté dans le tableau 4.9. Les résultats du tableau 4.8 correspondant au programme P2 sont recopiés dans le tableau 4.9 (indiqué par l'ordre 2) pour la comparaison. Dans le tableau 4.9, le temps CPU et la mémoire utilisés sont représentés respectivement par deux chiffres pour chaque contrainte *commit* qui correspondent à deux ordres de vérification différents.

Ordre		Contraintes	Temps CPU (seconde)		Mémoire (KB)	
1	2		1	2	1	2
1	1	(AEN0, Dtrin0, [0, 5])	0.02	0.01	382	382
2	2	(Dtrin0, Dtackn0, [40, 600])	0.12	0.13	3208	3208
4	3	(Dtackn0, Dtrin1, [0, 1e8]) (AEN1, Dtrin1, [0, 5])	0.73	1.00	3536	3537
3	4	(AEN1, Up_axs0, [0, 80])	2.71	3.66	3935	3936
7	5	(Up_ack0, Up_axs1, [0, 60])	675	96	7385	5789
5	6	(AEN2, Dtrin2, [0, 5])	66	76	4543	5789
6	7	(Up_ack0, Dtackn2, [0, 1.e100]) (Dtrin2, Dtackn2, [40, 600])	109	103	7385	7385
10	8	(AEN3, Dtrin3, [0, 5]) (Dtackn2, Dtrin3, [0, 1.e100])	25827	3234	7720	7590
8	9	(AEN3, Up_axs2, [0, 80])	966	3408	8048	7968
9	10	(Up_ack2, Up_axs3, [0, 60])	9510	36309	9720	9724

Tableau 4.9 Les résultats obtenus avec l'ordre de vérification différent

Selon les résultats présentés dans le tableau 4.9, on constate que la mémoire utilisée pour chaque contrainte *commit* ne change pas beaucoup pour les deux ordres de vérification parce que le Maxcycle de chaque contrainte est inchangé, et que la variation du temps d'exécution est importante, surtout pour les trois dernières contraintes. Si l'on compare le temps total de vérification, la vérification suivant l'ordre 1 est environ 4650 secondes plus rapide que celle suivant l'ordre 2.

Une autre expérience effectuée est de modifier la valeur représentant le nombre réel infini et la précision des expressions du type flottant dans le programme P1. Le tableau 4.10 suivant montre les résultats de cette expérience avec une spécification (S1) : 1.e8 comme nombre infini et 1.e-8 comme précision. Les résultats du P1 extraits du tableau 4.8 avec 1.e100 comme infini et 1.e-4 comme précision (S2) sont également présentés dans le tableau 4.10. La première colonne du tableau 4.10 indique l'ordre de vérification.

Index	**Contrainte**	**Temps CPU (seconde)**		**Mémoire (KB)**	
		S1	**S2**	**S1**	**S2**
1	(AEN0, Dtrin0, [0, 5])	0.02	0.06	390	382
2	(Dtrin0, Dtackn0, [40, 600])	0.14	0.32	3256	3252
3	(Dtackn0, Dtrin1, [0, 1.e100]) (AEN1, Dtrin1, [0, 5])	1.05	3.00	3594	3586
4	(AEN1, Up_axs0, [0, 80])	2.79	7.62	3996	3988
5	(Up_ack0, Up_axs1, [0, 60])	96	265	5873	5861
6	(AEN2, Dtrin2, [0, 5])	120	312	5873	5861
7	(Up_ack0, Dtackn2, [0, 1.e100]) (Dtrin2, Dtackn2, [40, 600])	104	280	7485	7465
8	(AEN3, Dtrin3, [0, 5]) (Dtackn2, Dtrin3, [0, 1.e100])	3350	20720	7698	7678
9	(AEN3, Up_axs2, [0, 80])	3521	9289	8080	8048
10	(Up_ack2, Up_axs3, [0, 60])	37147	97493	9853	9813

Tableau 4.10 Les résultats avec la valeur infinie et la précision différente

On voit que dans le tableau 4.10, la mémoire utilisée pour vérifier une contrainte est (presque) la même pour les deux spécifications puisque le Maxcycle de chaque contrainte reste inchangé, mais le temps d'exécution varie beaucoup surtout pour les trois dernières contraintes. Bien qu'on ait pris une précision plus fine 1.e-8 qui rend le calcul plus long, mais ce ralentissement est récompensé par une valeur infinie plus petite 1.e8. Par conséquent, la spécification S1 (infinie = 1.e8, précision = 1.e-8) économise un temps CPU de 84000 secondes par rapport à la spécification S2 (infinie = 1.e100, précision = 1.e-4) pour vérifier ces 10 contraintes.

Pour mettre en évidence l'influence de différents facteurs sur le temps CPU total utilisé, le tableau 4.11 présente une comparaison de temps CPU total utilisé pour les différentes expériences présentées dans les tableaux 4.8, 4.9 et 4.10.

Index	**Logiciel**	**Spécification**	**Temps CPU total (seconde)**
1	Version 1 (flottant)	Ordre de vérification tableau 4.8 Infinie = 1.e100 précision = 1.e-4	128366
2	Version 1 (flottant)	Ordre de vérification tableau 4.8 Infinie = 1.e8 précision = 1.e-8	44341
3	Version 2 (entier)	Ordre de vérification tableau 4.8 Infinie = 100000000	41812
4	Version 2 (entier)	Ordre de vérification tableau 4.9 Infinie = 100000000	37159
5	Programme utilisé dans [3] (entier)	Ordre de vérification tableau 4.8 Infinie = 2147483647	160003

Tableau 4.11 Une comparaison de temps CPU total utilisé dans les différentes expériences

Avec ces différentes expériences, on peut obtenir les conclusions suivantes :

Tout d'Abord, l'augmentation de la mémoire utilisée est relativement stable pour notre outil car après la vérification de chaque contrainte, le but est retiré du système de contraintes et un nouveau but est ajouté dans le système de contraintes. On constate que la mémoire utilisée est proportionnelle à *Maxcycle* de l'événement cible de la contrainte *commit*. L'ordre de vérification, la valeur qui représente l'infini et la précision des expressions du type flottant ne changent (presque) pas la mémoire utilisée pour vérifier une contrainte *commit*.

L'augmentation du temps CPU est causée par de différents facteurs. On peut identifier au moins les facteurs suivants :

- La précision des expressions du type flottant : dans le tableau 4.8, on choisit la précision = 1.e-4 avec la méthode *setDefaultPrecision* de la classe *IlcManager* ; Plus cette valeur est petite, plus le calcul est précis. Par conséquent, le temps de calcul est plus long quand cette valeur est petite.

- Les constantes utilisées pour représenter un nombre réel ou entier infini : dans notre exemple : 1.e100 pour un nombre réel infini et 100000000 pour un nombre entier infini et dans [3], le langage BNR(Prolog) utilise 2147483647 (2^{31}-1) pour représenter un nombre entier infini ; Puisque le Solver ne prévoit pas la situation de "*overflow*", on ne peut pas utiliser la valeur maximum exprimable comme BNR(Prolog). Le tableau 4.10 montre pour la version 1 (nombre réel) l'influence de la combinaison de ces deux facteurs précédents sur la performance de l'outil.

- Le nombre de contraintes accumulées qui est proportionnel au nombre de cycles à dérouler Maxcycle ;

- La bloc-machine obtenue ainsi que l'ordre de vérification de chaque contrainte qui influencent les contraintes accumulées pour vérifier une contrainte, le tableau 4.9 montre l'influence de ce facteur sur la performance de l'outil.

- La caractéristique du problème de vérification de contrôleurs : on cherche exhaustivement un contre-exemple dans tout l'espace de recherche ; en effet, le nombre d'échecs rencontrés au long de la procédure de vérification est proportionnel à la taille de cet espace de recherche. On constate que l'exécution d'un but qui instancie le temps d'occurrence des événements consomme la plupart du temps d'exécution pour vérifier une contrainte.

4.4 Conclusion

Dans ce chapitre, nous avons montré avec un exemple d'application, le fonctionnement et la performance de notre outil. Avec cette démonstration, on constate qu'il est facile de définir un chronogramme et une FSM à l'aide de l'interface simple de l'outil. Bien que la lisibilité des fichiers définissant un chronogramme ou une FSM reste à améliorer, la méthode de modélisation des composants essentiels tels qu'un événement, une contrainte, une affectation etc., et celle de transformation de ces composants en contraintes acceptables par Solver sont intuitives et efficaces. On constate également qu'il n'est pas évident de comparer le résultat obtenu avec celui présenté dans [3] pour les raisons suivantes :

- Différents langages de programmation sont utilisés dans ces deux expériences : ILOG Solver et BNR-Prolog;

- Certaines contraintes sont modifiées dues aux erreurs du chronogramme présentées dans [3]. Par conséquent, le *Maxcycle* et l'ordre de vérification de chaque contrainte sont différents ;

- La modélisation de la contrainte est différente. Dans [3], chaque relation temporelle est considérée comme une contrainte. Dans notre l'outil, la composition de différentes relations temporelles est prise en compte.

- Le résultat dans [3] est obtenu par 13 lancements du programme écrit spécifiquement pour l'exemple considéré. Le calcul de *Maxcycle* et la recherche de la bloc-machine ne sont pas prise en compte dans le temps CPU et la mémoire utilisée. Notre résultat est obtenu avec un programme général qui accepte un chronogramme et une FSM comme paramètre et calcule automatiquement les *Maxcycles* et la bloc-machine. Les 10 contraintes *commit* sont vérifiées par un seul lancement du programme. En effet, notre programme est générique au problème et non une instance du problème.

Chapitre 5 Conclusions

Au niveau de l'interface d'un système matériel, il est possible d'observer la communication entre celui-ci et son environnement. Le domaine de recherche sur la vérification temporelle des interfaces tente de trouver les méthodes permettant de répondre entre autres aux questions telles que :

- Est-ce que le système de contraintes temporelles obtenu à partir d'un chronogramme est consistant, c'est-à-dire, il admet au moins une solution ?

- Est-ce que la spécification temporelle d'une interface sous forme de chronogramme est réalisable par une machine causale, c'est-à-dire, la spécification est causale ?

- Si la spécification d'un contrôleur d'interface est causale, étant donnée une de son implantation sous forme de FSM, est-ce que cette implantation satisfait à toutes les contraintes spécifiées dans le chronogramme correspondant ?

Ce mémoire présente un outil qui permet de répondre aux questions ci-dessus. Pour répondre aux deux premières questions, l'entrée de cet outil est le nom d'un fichier texte qui définit une spécification temporelle de l'interface sous forme de chronogramme. Et pour répondre à la troisième question, les entrées de cet outil sont :

- le nom du fichier définissant le chronogramme ;
- le nom du fichier définissant la FSM implantation du contrôleur d'interface ;

Si la bloc-machine est générée par l'outil, les paramètres suivants sont nécessaires :

- le nombre initial de blocs à générer pour construire une bloc-machine causale;
- la direction de recherche ;
- le temps maximum utilisé pour la recherche d'une bloc-machine pour chaque nombre de blocs.

Si l'on veut utiliser une partition d'événements spécifique, on donne simplement le nom du fichier qui définit cette partition.

Pour vérifier la consistance d'un chronogramme, l'outil calcule, à l'aide de la méthode *findDistances* de la classe *TimingDiagram*, les séparations temporelles des événements en prenant chaque événement comme référence.

L'outil effectue une vérification de causalité en appelant certains modules développés à cette fin. Particulièrement, le module *generateBlockMachine* est utilisé pour générer automatiquement une bloc-machine et le module *testPartition* est utilisé pour tester la causalité d'une partition particulière.

Pour la vérification de l'implantation d'un contrôleur d'interfaces, l'outil appelle la méthode *findDistances* en prenant l'événement original du chronogramme comme référence pour déterminer *Maxcycle* de chaque événement. Puis l'outil appelle selon le choix de l'utilisateur, soit le module *generateBlockMachine* soit le module *testPartition* afin de trouver une causale bloc-machine. Après ces pré-traitements, l'outil vérifie chaque contrainte *commit* en appelant le *module generate_constraint_system* qui se charge de mettre en œuvre la méthode de vérification de l'implantation du contrôleur d'interface présentée dans [3].

L'objectif de cette recherche est de rendre la méthode de vérification de l'implantation du contrôleur d'interface présentée dans [3] utilisable. Concrètement, les contributions originales des travaux rapportés dans ce mémoire sont les suivantes :

1) Choix d'un langage de programmation plus adéquat. Tout d'abord, les méthodes de vérification temporelle proposées dans [2] et celle proposée dans [3] sont indépendantes et développées en langage BNR_Prolog. Notre outil est développé sur ILOG Solver qui est un langage de programmation par contrainte plus puissant quant à la richesse de contraintes prédéfinies et plus facile au niveau d'interface et du contrôle de la recherche de solution. En profitant de la méthodologie orientée objet fournie par ILOG Solver, la modélisation de connaissance d'un domaine d'application est plus facile en comparant avec le langage BNR-Prolog.

2) Mise en œuvre des méthodes proposées dans [2] et [3] dans ILOG Solver ainsi qu'une interface permettant de définir un chronogramme et une FSM : Nous avons développé toutes les classes nécessaires dans le but de vérification temporelle des interfaces telles que : *Event*, *TimingRelation*, *TimingConstraint*, *TimingDiagram* et *FSM* etc. L'implantation de certaines classes est réalisée de manière plus générale, par exemple dans la classe *Port* on considère également le type *inout* ; dans la classe *TimingConstraint* on prend en compte non seulement les contraintes du type conjonctive mais aussi des types non linéaires *min* et *max*. La classe FSM permet de prendre en compte tous les aspects de l'implantation d'un contrôleur. En utilisant ces classes, on peut facilement définir un chronogramme ou une machine à états finis à l'aide d'un fichier texte très simple.

3) Pré-traitement automatique. Nous avons fait coopérer activement les méthodes proposées dans [2] pour la vérification de consistance et la vérification de causalité d'un chronogramme et celle proposée dans [3] pour la vérification de l'implantation d'un contrôleur d'interfaces. Avec les méthodes proposées dans [2], on peut déterminer *Maxcycle* et une causale bloc-machine utilisés par la méthode de vérification de contrôleur.

4) Développement d'une méthode de transformation. Vu que l'expression logique constitue un modèle important dans la modélisation d'une FSM, nous avons déve-

loppé une méthode qui permet de transformer une expression logique en contrainte acceptée par Solver.

Plusieurs travaux futurs sont envisageables.

Puisque l'implantation de contrôleurs d'interfaces est souvent décrite par un programme de description de matériel tels que VHDL et Verilog, il serait plus appréciable de créer un programme qui permet de générer automatiquement le fichier définissant une FSM à partir des programmes de description écrits en VHDL ou Verilog.

Une étude plus approfondie de l'implantation de ILOG Solver, des techniques de programmation par contrainte et du domaine de vérification temporelle des interfaces est nécessaire dans le but d'utiliser judicieusement chaque méthode définie dans ILOG Solver et de représenter de manière plus adéquate les connaissances de ce domaine d'application afin d'améliorer la performance de l'outil. Par exemple, les questions suivantes peuvent être intéressantes :

- Quelle est la valeur plus adéquate pour représenter un nombre infini dans la pratique ?
- Est-ce qu'il existe encore d'autres contraintes implicites qui permettent d'accélérer la procédure de vérification de contrôleur ?
- Parmi les algorithmes de recherche fournis par Solver, quel est le plus adéquat pour la vérification du contrôleur ?

On peut enrichir le modèle de contrainte en prenant en compte la corrélation des délais selon la méthode proposée dans [2].

Plus d'expérimentation doit aussi être effectuée avec notre outil afin de valider et améliorer la méthode de vérification de contrôleurs proposée.

Bibliographie

[1] E. Cerny, B. Berkane, P. Girodias et K. Khordoc, "Hierarchical Annotated Action Diagrams – An Interface-Oriented Specification Method", Kluwer Academic Publishers 1998.

[2] P. Girodias, "Vérification des Propriétés Temporelles des interfaces Matérielles à l'aide de la Programmation Logique avec Contraintes" , Thèse de doctorat, Université de Montréal, 1997.

[3] F. Jin, "Timing Verification of Interface Specification and Controllers", Thèse de doctorat, Université de Montréal, 2000.

[4] P. A. Babkine, "Un outil pour la Spécification de Matériel et la Génération de Modèles Exécutables", Mémoire de maîtrise, Université de Montréal, 1996.

[5] A. Tarnauceanu, "Logiciel pour la Vérification par Simulation de la Spécification de Haut Niveau de Systèmes Matériels", Mémoire de maîtrise, Université de Montréal, 1997.

[6] C. A. Mead and L. A. Conway. "An introduction to VLSI Systems". Addison Wesley, Readings, Massachusetts, 1980.

[7] ILOG Solver User's Manual Version 4.4 – http://www.ilog.com

[8] K. Khordoc et E. Cerny, "Semantics and Verification of Action Diagram with Linear Timing Constraints", ACM Transaction on Design of Electric System, Vol. 3, No. 1, pp. 21-50, January 1998.

[9] P. Girodias, "Interface Timing Verification CLP Prolog Library User's Manual – ICLP version 2.0 alpha", Rapport interne, Université de Montréal, 1997.

[10] F. Benhamou et W. J. Older, "Apply Interval Arithmetic to Real, Integer and Boolean Constraints", Journal of Logic Programming, Volume 32, numbers 1, 2 & 3, pp. 1-24, 1997.

[11] W. J. Older et A. Vellino, "Constraint Arithmetic on Real Intervals", in Constraint Logic Programming: Selected Research, F. Benhamou and A. Colmerauer (eds), MIT Press, pp. 175-195, 1993, Cambridge, MA (USA).

[12] K. McMillian et D. Dill, "Algorithms for Interface Timing Verification", Proceedings of IEEE International Conference on Computer Design, ICCD'92, pp. 48-51, 1992.

[13] G. Borriello, "A New Interface Specification Methodology and its Application to Transducer Synthesis", Thèse de doctorat, University of California, Berkeley, 1988.

[14] A. J. Gahlinger, "Coherence and Satisfiability of Waveform Timing Specifications", Research Report CS-90-11, Thèse de doctorat, University of Waterloo, 1990.

[15] T. Amo, "Specification, Simulation and Verification of Timing Behaviour", Thèse de doctorat, University of Washington, 1993.

[16] P. Vanbekbergen, G. Goossens, et H. De Man, "Specification and Analysis of Timing Constraints in Signal Transitions Graphs", Proceedings of the European Conference on Design Automation, EDAC'92, pp. 302-306, Brussels, Belgique, 1992.

[17] T. Amon, H. Hulgaard, G. Borriello et S. Burns, "Timing Analysis of Concurrent Systems", Proceedings of 1993 IEEE International Conference on Computer Design; VLSI in Computers & Processors, ICCD'93, pp. 166-173, October 1993, Cambridge (MA), USA.

[18] T. Y. Yen, A. Ishii, A, Casavant et W. Wolf, "Efficient Algorithms for Interface Timing Verification", Proceedings of the European Design Automation Conference, pp. 34-39, 1994.

[19] W. J. Older, "Delay Networks and Intervals", publication interne, Bell-Northern Research, 1991.

[20] T. Amon et G. Borriello, "An Approach to Symbolic Timing Verification", Proceedings of the 29th ACM/IEEE Design Automation Conference, DAC'92, pp. 410-413, 1992, Anaheim (CA), USA.

Annexe A

Le fichier définissant la FSM illustrée dans la figure 4.2

```
example 10. 12 1 1 0 0 12 1 0
S0 S1 S2 S3 S4 S5 S6 S7 S8 S9 S10 S11
X Y
tr1 2 2 2 2 2 1 0 1 0      % 12 transitions
1 S0 X 1 1 S6 X 1
1 1 1 1 S1
tr2 2 2 2 2 2 1 0 1 0
1 S0 X 0 1 S6 X 0
1 1 1 1 S0
tr3 2 2 2 2 2 1 0 1 0
1 S1 X 1 1 S7 X 1
1 1 1 1 S2
tr4 2 2 2 2 2 1 0 1 0
1 S1 X 0 1 S7 X 0
1 1 1 1 S8
tr5 2 2 2 2 2 1 0 1 0
1 S2 X 1 1 S8 X 1
1 1 1 1 S3
tr6 2 2 2 2 2 1 0 1 0
1 S2 X 0 1 S8 X 0
1 1 1 1 S9
tr7 2 2 2 2 2 1 0 1 0
1 S3 X 1 1 S9 X 1
1 1 1 1 S4
tr8 2 2 2 2 2 1 0 1 0
1 S3 X 0 1 S9 X 0
1 1 1 1 S10
tr9 2 2 2 2 2 1 0 1 0
1 S4 X 1 1 S10 X 1
1 1 1 1 S5
tr10 2 2 2 2 2 1 0 1 0
1 S4 X 0 1 S10 X 0
1 1 1 1 S6
tr11 2 2 2 2 2 1 0 1 0
1 S5 X 1 1 S11 X 1
1 1 1 1 S11
tr12 2 2 2 2 2 1 0 1 0
1 S5 X 0 1 S11 X 0
1 1 1 1 S7
assign1 Y 6 0 6 1 1 1 1 1 1 1 1 1 1 1  % une affectation décrvant
1 S6 1 S7 1 S8 1 S9 1 S10 1 S11          % la fonction de sortie
```

Annexe B

B.1 Le fichier décrivant le chronogramme illustré dans la figure 4.2.

```
7 22              % 7 ports et 22 contraints
CSN in 5 1
AEN in 5 1
RW in 5 1
Up_axs out 5 0
Up_ack in 5 0
Dtackn out 5 1
Dtrin out 5 0
asm1 linear 3
rel77  AEN -1 AEN 0 19 19   % contrainte supplémentaire
rel7 CSN 0 AEN 0 0 10
rel8 RW 0 AEN 0 4 10
asm2 linear 3
rel11 CSN 2 AEN 2 0 10
rel12 AEN 1 AEN 2 40 200
rel13 RW 2 AEN 2 4 10
asm3 linear 2
rel9 AEN 0 AEN 1 60 1.e100
rel10 Dtackn 0 AEN 1 0 40
cmt0 linear 1 AEN 0 Dtrin 0 0 5
asm4 linear 1 AEN 1 CSN 1 0 10
asm5 linear 1 AEN 1 RW 1 0 10
cmt1 linear 1 AEN 1 Up_axs 0 0 80
cmt2 linear 2
rel1 AEN 1 Dtrin 1 0 5
rel2 Dtackn 0 Dtrin 1 0 1.e100
asm6 linear 2
rel14 AEN 2 AEN 3 60 1.e100
rel15 Dtackn 2 AEN 3 0 40
cmt3 linear 1 AEN 2 Dtrin 2 0 5
cmt4 linear 2
rel3 AEN 3 Dtrin 3 0 5
rel4 Dtackn 2 Dtrin 3 0 1.e100
asm7 linear 1 AEN 3 CSN 3 0 10
asm8 linear 1 AEN 3 RW 3 0 10
cmt5 linear 1 AEN 3 Up_axs 2 0 80
asm9 linear 1 Up_axs 0 Up_ack 0 0 320
asm10 linear 1 Up_axs 2 Up_ack 2 0 320
asm11 linear 1 Up_ack 0 Up_ack 1 20 20
cmt6 linear 1 Up_ack 0 Up_axs 1 0 60
cmt7 linear 2
rel5 Up_ack 0 Dtackn 2 0 1.e100
rel6 Dtrin 2 Dtackn 2 40 600
cmt8 linear 1 Up_ack 2 Up_axs 3 0 60
asm12 linear 1 Up_ack 2 Up_ack 3 20 20
cmt9 linear 1 Dtrin 0 Dtackn 0 40 600
```

B.2 Le fichier définissant l'implantation du contrôleur décrite dans la section 4.2.

```
controller 20. 4 4 3 8 4 4 11 6
IDLE WAIT ACKW REND    % 4 états de FSM
CSN AEN RW Up_ack      % 4 entrées
Up_axs Dtackn Dtrin    % 3 sorties
CSN_R1 CSN_R2 AEN_R1 AEN_R2     % 7 registres
Up_ack_R Up_wrnlocal Up_axspending
1 1 1 1 0 0 0     % valeurs initiales des registres
1 2 1 3 1 0 0     % 4 expressions logiques internes
1 ACKW Up_wrnlocal 1 Up_ack_R 1
1 2 1 3 1 0 0
1 IDLE CSN_R2 0 AEN_R2 0
2 5 2 4 3 1 0 0 0 1 0 0
1 IDLE CSN_R2 0 AEN_R2 0 RW 0
1 ACKW Up_wrnlocal 0 Up_ack_R 1
2 4 2 4 2 1 0 0 0 1 0
1 IDLE CSN_R2 0 AEN_R2 0 RW 1
1 WAIT AEN_R2 1

% 4 fonctions de transition
tr1 4 5 4 2 2 2 3 1 0 1 0 1 0 1 0 0
1 IDLE CSN_R2 1 1 IDLE AEN_R2 1 1 REND AEN_R2 1
1 ACKW Up_wrnlocal 1 Up_ack_R 1 1 1 1 1 IDLE
tr2 2 4 2 2 4 1 0 1 0 0 0
1 WAIT AEN_R2 0 1 IDLE CSN_R2 0 AEN_R2 0 RW 0 1 1 1 1 WAIT
tr3 3 5 3 4 2 2 1 0 0 0 1 0 1 0
1 IDLE CSN_R2 0 AEN_R2 0 RW 1 1 ACKW Up_ack_R 0
1 WAIT AEN_R2 1 1 1 1 1 ACKW
tr4 2 3 2 3 2 1 0 0 1 0
1 ACKW Up_wrnlocal 0 Up_ack_R 1
1 REND AEN_R2 0 1 1 1 1 REND
link1 CSN_R1 1 1 0 1 0 CSN 1   % 11 instances de Assignment
link2 CSN_R2 1 1 0 1 0 CSN_R1 1
link3 AEN_R1 1 1 0 1 0 AEN 1
link4 AEN_R2 1 1 0 1 0 AEN_R1 1
link5 Up_ack_R 1 1 0 1 0 Up_ack 1
link6 Up_wrnlocal 2 2 0 3 3 2 2 0 2 2 0 0 0 1 1
RW 0 0 0 0 1 Up_wrnlocal 1
link7 Up_axspending 2 2 0 2 2 2 0 0 0
1 3 Up_ack 0 Up_axspending 1 Up_ack 0
link8 Dtackn 3 3 0 1 1 2
0 0 0 2 CSN 1 AEN 1 AEN 0 0 2
link9 Dtrin 1 2 0 2 0 0 CSN 0 AEN 0
link10 Up_axs 2 1 0 1 1 2 0 1 3 Up_axspending 1
link11 RW_R 1 1 0 1 0 RW 1
Dtrin0 max 2 CSN0 AEN0   % 6 instances de AsynchronousOutput
Dtrin2 max 2 CSN2 AEN2
Dtrin1 min 2 CSN1 AEN1
Dtrin3 min 2 CSN3 AEN3
Dtackn1 min 2 CSN1 AEN1
Dtackn3 min 2 CSN3 AEN3
```

Annexe C

Définition de classes et méthodes principales en ILOG Solver pour la vérification temporelle des interfaces (Version 2 : entier)

```
#include <ilsolver/ilcany.h>
#include <ilsolver/ilcfloat.h>
#include <ilsolver/ilcint.h>
#include <ilsolver/basic.h>
#include <ilsolver/ilcset.h>
#include <ilsolver/setcst.h>
#include <ilsolver/search.h>
#include <ilsolver/ilctrace.h>
#include <string.h>
#include <strings.h>
#include <iostream.h>
#include <stdlib.h>
#include <fstream.h>
#include <math.h>

enum modeType {in=0, out=1, inout=2, none=3};
enum intentType {assume=0, commit=1};
enum compositeType {linear=0, min=1, max=2};

IlcInt period;
const IlcInt infinie = 100000000;

//=========== Event class definition ===============//

class Event {
private:
   IlcManager _manager;
   const char* _name;
   modeType _mode;
   IlcBool _initialValue;
public:
   Event(IlcManager m, const char* name, modeType mode, IlcBool
initialValue): _manager(m), _name(name), _mode(mode),
_initialValue(initialValue) {};
   const char* getName() {return _name;}
   modeType getMode() {return _mode;}
   IlcBool getInitialValue() {return _initialValue;}
   void display();
};

//=================Port class definition=================//

class Port {
private:
   IlcManager _manager;
   const char* _name;
   modeType _mode;
```

```
   IlcInt _nbStates;
   IlcIntArray _indEventsIn;
   IlcBool _initialValue;
   IlcAnyArray _events;
public:
   Port(IlcManager m, const char* name, modeType mode, const IlcInt
          nbStates, const IlcBool  initialValue);
   Port(IlcManager m, const char* name, modeType mode, const IlcInt
          nbStates, const IlcBool  initialValue, IlcIntSet eventsIn);
   Port(IlcManager m, const char* name, IlcAnyArray events);
   void display();
   IlcInt getNbStates() { return _nbStates; }
   const char* getName() { return _name; }
   modeType getMode() { return _mode; }
   IlcBool getInitialValue() { return _initialValue; }
   IlcAnyArray getEvents() { return _events; }
   Event* getEvent(IlcInt index) { return (Event*)_events[index]; }
   IlcIntArray getIndEventsIn() { return _indEventsIn;}
};

// constructor for in or out type port
Port::Port(IlcManager m, const char* name, modeType mode, const IlcInt
nbStates, const IlcBool initialValue):
_manager(m), _name(name), _mode(mode), _nbStates(nbStates),
_initialValue(initialValue) {
     if ( _mode == in || _mode == out ) {
       IlcAnyArray events(_manager, _nbStates-1);
       if ( _mode == in ) {
          IlcIntArray ind(_manager, _nbStates -1);
          _indEventsIn = ind;
       } else _indEventsIn = IlcIntArray();
       IlcBool b = _initialValue;
       char t = ' ';
       for (IlcInt i = 0; i< _nbStates-1; i++) {
           char* nm = new (m.getHeap()) char(strlen(_name)+5);
           strcpy(nm, _name);
           char* str = ulltostr(i, &t);
           if(i <= 9) strncat(nm, str, 1);
           else if(i < 100) strncat(nm, str, 2);
           events[i] = new(m.getHeap())Event(_manager, nm, _mode, b);
           b = !b;
       }
       _events = events;
    } else
       _manager.out() << "this constructor is only used to construct
                           the ports of type in or out" << endl;
}

// constructor for inout type port
Port::Port(IlcManager m, const char* name, modeType mode, const IlcInt
nbStates, const IlcBool initialValue, IlcIntSet eventsIn):
_manager(m), _name(name), _mode(mode), _nbStates(nbStates),
_initialValue(initialValue) {
       if ( _mode == inout) {
         IlcIntArray ind(_manager, eventsIn.getSize());
         IlcAnyArray events(_manager, _nbStates-1);
```

```
          IlcBool b = _initialValue; IlcInt j = 0; char t = ' ';
          for (IlcInt i = 0; i< _nbStates-1; i++) {
            char* nm = new char(strlen(_name) +5);
            char* str = ulltostr(i, &t);
            strcpy(nm, _name);
            if ( i <= 9 ) strncat(nm, str, 1);
            else if(i < 100) strncat(nm, str, 2);
          if ( eventsIn.isIn(i) ) {
             events[i] = new (m.getHeap()) Event(_manager, nm, in, b);
             ind[j] = i; j++;
          } else
            events[i] = new (m.getHeap()) Event(_manager, nm, out, b);
            b = !b;
          }
            _events = events; _indEventsIn = ind;
        } else
            _manager.out() << "this constructor is only used to
                      construct the ports of type inout" << endl;
}

//============= TimingRelation class definition ===============//

class TimingRelation {
private:
     IlcManager _manager;
     const char* _name;
     Event* _source;
     Event* _sink;
     IlcInt _min;
     IlcInt _max;
public:
     TimingRelation(IlcManager m, const char* name, Event* source,
                    Event* sink, IlcInt minValue, IlcInt maxValue);
     void display();
     IlcManager getManager() {return _manager;}
     const char* getName() { return _name; }
     Event* getSource() { return _source; }
     Event* getSink() { return _sink; }
     IlcInt getMin() { return _min; }
     IlcInt getMax() { return _max;}
};

TimingRelation::TimingRelation(IlcManager m, const char* name, Event*
source, Event* sink, IlcInt minValue, IlcInt maxValue): _manager(m),
_name(name), _source(source), _sink(sink), _min(minValue),
_max(maxValue) {}


//=============== TimingConstraint class definition ==========//

class TimingConstraint {
private:
    IlcManager _manager;
    IlcAnyArray _composants;
    const char* _name;
    intentType _intent;
```

```
    compositeType _composition;
    IlcInt _nbComposants;
public:
    TimingConstraint(IlcManager m, const char* name,
                compositeType composite, IlcAnyArray composants);
    TimingConstraint(IlcManager m, const char* name, Event* source,
                Event* sink, IlcInt minValue, IlcInt maxValue);
    IlcAnyArray getComposants() { return _composants; }
    const char* getName() {return _name; }
    intentType getIntent() { return _intent; }
    IlcAnySet getSource();
    Event* getSink();
    compositeType getCompositeType() { return _composition; }
    IlcInt getNbComposants() { return _nbComposants; }
    void display();
    IlcBool checkComposition(IlcAnyArray composants);
};

TimingConstraint::TimingConstraint(IlcManager m, const char* name,
compositeType composite, IlcAnyArray composants):
_manager(m), _name(name), _composition(composite),
_composants(composants) {
     if (checkComposition(composants)) {
         TimingRelation* tc = (TimingRelation*) _composants[0];
       Event* ev = tc->getSink();
       if(ev->getMode() == out)
          _intent = commit;
       else _intent = assume;
       _nbComposants = _composants.getSize();
     } else  m.out() << "Invalide composants: sink event is not
                          unique" << endl;
}

//=============== TimingDiagram class definition ===============//

class TimingDiagram {
private:
   IlcManager _manager;
   Event* _origine;
   IlcInt _nbPortsIn;
   IlcAnyArray _portsIn;
   IlcInt _nbEventsIn;
   IlcAnyArray _eventsIn;
   IlcInt _nbPortsOut;
   IlcAnyArray _portsOut;
   IlcInt _nbEvents;
   IlcAnyArray _events;
   IlcInt _nbConstraintsAsm;
   IlcAnySet _constraintsAsm;
   IlcInt _nbConstraintsCmt;
   IlcAnySet _constraintsCmt;
   IlcInt _nbAddedConstraintsAsm;
   IlcAnySet _addedConstraintsAsm; // for inout type ports
   IlcInt _nbAddedConstraintsCmt;
   IlcAnySet _addedConstraintsCmt; // for inout type ports
   IlcBool _withMinConstraints;  // contains min constraints ?
```

```
   IlcBool _withMaxConstraints;  // contains max constraints ?
   IlcInt _nbMinConstraints;
   IlcInt _nbMaxConstraints;
public:
   TimingDiagram(IlcManager m, Event* origine, IlcAnyArray ports,
                 IlcAnySet constraints);
   void display();
   IlcManager getManager() { return _manager; }
   Event* getOrigine() { return _origine; }
   IlcInt getNbPortsIn() { return _nbPortsIn; }
   IlcAnyArray getPortsIn() { return _portsIn; }
   IlcInt getNbPortsOut() { return _nbPortsOut; }
   IlcAnyArray getPortsOut() { return _portsOut; }
   IlcInt getNbEventsIn() { return _nbEventsIn;}
   IlcInt getNbEventsOut() { return _nbEvents - _nbEventsIn;}
   IlcAnyArray getEventsIn() { return _eventsIn;}
   IlcAnyArray getEvents() { return _events;}
   IlcInt getNbConstraintsAsm() { return _nbConstraintsAsm; }
   IlcInt getNbConstraintsCmt() { return _nbConstraintsCmt; }
   IlcAnySet getConstraintsAsm() { return _constraintsAsm; }
   IlcAnySet getConstraintsCmt() { return _constraintsCmt; }
   IlcInt getNbAddedConstraintsAsm(){return _nbAddedConstraintsAsm;}
   IlcInt getNbAddedConstraintsCmt(){return _nbAddedConstraintsCmt;}
   IlcAnySet getAddedConstraintsAsm() {return _addedConstraintsAsm;}
   IlcAnySet getAddedConstraintsCmt() {return _addedConstraintsCmt;}
   IlcBool withMinConstraints() {return _withMinConstraints;}
   IlcBool withMaxConstraints() {return _withMaxConstraints;}
   IlcInt getNbMinConstraints() { return _nbMinConstraints;}
   IlcInt getNbMaxConstraints() { return _nbMaxConstraints;}
   IlcAnySet getConstraintsWithSink(Event* event);
   IlcAnySet getConstraintsWithSource(Event* event);
   IlcAnySet getEventsLinkedTo(Event* event);
   IlcAnySet getEventsLinkingWith(Event* event);
   IlcAnyArray splitInoutPort(Port* port);
   IlcAnySet addConstraints(Port* port);
   IlcIntVarArray findDistances(Event* reference);
   IlcIntVarArray build_constraint_system(IlcIntVarArray occur);
   IlcIntVarArray build_constraint_system(compositeType cmp,
                  IlcIntVarArray occur);
   IlcInt findEvent(Event* event);
   IlcIntVarArray applyLinearConstraint(TimingConstraint* tc,
                                        IlcIntVarArray occur);
   IlcIntVarArray applyNonLinearConstraint(TimingConstraint* tc,
                  compositeType composite, IlcIntVarArray occur);
   IlcIntVarArray decomposition(TimingConstraint* tc,
                  compositeType cmp, IlcIntVarArray occur);
};

TimingDiagram::TimingDiagram(IlcManager m, Event* origine, IlcAnyArray
ports, IlcAnySet constraints):
_manager(m), _origine(origine) {
      IlcInt np = ports.getSize();
      IlcInt nc = constraints.getSize();
      IlcInt nbmin = 0; IlcInt nbmax = 0;
      IlcInt nasm = 0; IlcInt ncmt = 0; IlcAny val;
```

```
for (IlcAnySetIterator iter(constraints); iter.ok(); ++iter) {
   val = *iter;
   TimingConstraint* tc = (TimingConstraint*) val;
   if ( tc->getIntent() == assume) nasm++;
   else ncmt++;
}
IlcAnyArray constraintsAsm = IlcAnyArray();
IlcAnyArray constraintsCmt = IlcAnyArray();
if (nasm) constraintsAsm = IlcAnyArray(m, nasm);
if (ncmt) constraintsCmt = IlcAnyArray(m, ncmt);
IlcInt indasm = 0; IlcInt indcmt = 0;
for (IlcAnySetIterator iter(constraints); iter.ok(); ++iter) {
   val = *iter;
   TimingConstraint* tc = (TimingConstraint*) val;
   if ( tc->getIntent() == assume) {
     constraintsAsm[indasm] = val; indasm++;
   } else {
     constraintsCmt[indcmt] = val; indcmt++;
   }
   if ( tc->getCompositeType() == min ) nbmin++;
   else if ( tc->getCompositeType() == max ) nbmax++;
}
   _nbMinConstraints = nbmin;_nbMaxConstraints = nbmax;
   if ( nbmin ) _withMinConstraints = IlcTrue;
   else _withMinConstraints = IlcFalse;
   if ( nbmax ) _withMaxConstraints = IlcTrue;
   else _withMaxConstraints = IlcFalse;
if (nasm) {
  _constraintsAsm = IlcAnySet(m, constraintsAsm);
  _nbConstraintsAsm = constraintsAsm.getSize();
} else {
  _constraintsAsm = IlcAnySet();
  _nbConstraintsAsm = 0;
}
if (ncmt) {
   _constraintsCmt = IlcAnySet(m, constraintsCmt);
   _nbConstraintsCmt = constraintsCmt.getSize();
} else {
   _constraintsCmt = IlcAnySet();
   _nbConstraintsCmt = 0;
}
IlcInt ncadd = 0; IlcInt npIn = 0; IlcInt npOut = 0;
IlcAnySet inoutPort(_manager, ports, IlcFalse );
for (IlcInt i =0; i < ports.getSize(); i++) {
   Port* port = (Port*) ports[i];
   if ( port->getMode() == inout) {
      inoutPort.add(port); npIn++; npOut++;
      ncadd = ncadd + (port->getNbStates())/2;
   } else if ( port->getMode() == in ) npIn++;
     else npOut++;
}
IlcInt nbinout = inoutPort.getSize();
IlcAnyArray totalPortsIn = IlcAnyArray();
if (npIn) totalPortsIn = IlcAnyArray(_manager, npIn);
IlcAnyArray totalPortsOut = IlcAnyArray();
if(npOut) totalPortsOut = IlcAnyArray(_manager, npOut);
```

```
 IlcInt added = 0;
 for (IlcAnySetIterator iterc(inoutPort); iterc.ok(); ++iterc) {
      IlcAny p = *iterc;
      Port* pt = (Port*) p;
      added = added + (pt->getNbStates()-1)/2;
 }
 IlcAnyArray possible = IlcAnyArray();
 if (added) possible = IlcAnyArray(m, added);
 IlcInt indIn = 0; IlcInt indOut = 0; IlcInt add = 0;
    for (IlcInt i =0; i < ports.getSize(); i++) {
       Port* port = (Port*) ports[i];
       if ( nbinout > 0 && inoutPort.isIn(port) ) {
       IlcAnyArray splited = splitInoutPort(port);
       totalPortsIn[indIn] = splited[0];
       totalPortsOut[indOut] = splited[1];
       indIn++; indOut++;
       IlcAnySet addC = addConstraints(port);
       IlcAny valc;
       for (IlcAnySetIterator iterc(addC);iterc.ok();++iterc) {
          valc = *iterc;
          possible[add] = valc; add++;
       }
    } else {
        if (port->getMode() == in ) {
           totalPortsIn[indIn] = port; indIn++;
        } else {
           totalPortsOut[indOut] = port; indOut++;
        }
     }
  }
  _addedConstraintsAsm = IlcAnySet(); _nbAddedConstraintsAsm = 0;
  _addedConstraintsCmt = IlcAnySet(); _nbAddedConstraintsCmt = 0;
 if (possible.getSize() ) {
   IlcAnySetVar AddedConstraintsAsm(m, possible);
   IlcAnySetVar AddedConstraintsCmt(m, possible);
   TimingConstraint* tcst = (TimingConstraint*) possible[0];
   for (IlcInt i = 0; i< possible.getSize(); i++) {
        tcst = (TimingConstraint*) possible[i];
        if ( tcst->getIntent() == assume )
           AddedConstraintsAsm.addRequired(tcst);
        else
           AddedConstraintsCmt.addRequired(tcst);
   }
   _addedConstraintsAsm = AddedConstraintsAsm.getRequiredSet();
   _nbAddedConstraintsAsm = _addedConstraintsAsm.getSize();
   _addedConstraintsCmt = AddedConstraintsCmt.getRequiredSet();
   _nbAddedConstraintsCmt = _addedConstraintsCmt.getSize();
}
_nbPortsIn = totalPortsIn.getSize(); _portsIn = totalPortsIn;
_nbPortsOut = totalPortsOut.getSize(); _portsOut = totalPortsOut;
indIn = 0; indOut = 0;
for (IlcInt i = 0; i < _nbPortsIn; i++) {
    Port* p = (Port*) _portsIn[i];
    indIn = indIn + p->getNbStates() - 1;
}
```

```
    for (IlcInt i = 0; i < _nbPortsOut; i++) {
        Port* p = (Port*) _portsOut[i];
        indOut = indOut + p->getNbStates() - 1;
    }
    _nbEventsIn = indIn;
    _nbEvents = indIn + indOut;
    IlcAnyArray eventsIn(_manager, indIn);
    IlcAnyArray events(_manager, indIn+indOut);
    IlcInt ind1 = 0;
    for (IlcInt i = 0; i < _nbPortsIn; i++) {
          Port* p = (Port*) _portsIn[i];
          IlcAnyArray e = p->getEvents();
          for (IlcInt j = 0; j < e.getSize(); j++) {
            eventsIn[ind1 + j] = e[j];
            events[ind1+j] = e[j];
          }
          ind1 = ind1 + e.getSize();
    }
    IlcInt ind2 = ind1;
    for (IlcInt i = 0; i < _nbPortsOut; i++) {
          Port* p = (Port*) _portsOut[i];
          IlcAnyArray e = p->getEvents();
          for (IlcInt j = 0; j < e.getSize(); j++) {
            events[ind2 + j] = e[j];
          }
          ind2 = ind2 + e.getSize();
    }
    _eventsIn = eventsIn;
    _events = events;
}

IlcIntVarArray TimingDiagram::findDistances(Event* reference) {
  IlcIntVarArray distances(_manager,_nbEvents+1,-infinie,infinie);
  // set reference event's distance 0
     IlcInt ind = findEvent(reference);
     if ( ind >= 0) _manager.add(distances[ind] == 0);
     else {
       cout << "cannot find reference event!" << endl;
       return distances;
     }
 // add implicite constraints of timing diagram
      for(IlcInt i=0; i< _nbPortsIn; i++) {
         Port* pt = (Port*) _portsIn[i];
         IlcAnyArray events = pt->getEvents();
         for(IlcInt j=0; j<events.getSize(); j++) {
           if(j==0)  {
             Event* ev = pt->getEvent(j);
             _manager.add(distances[findEvent(ev)] >  distances[0]);
           } else {
             Event* ev1 = pt->getEvent(j-1);
             Event* ev2 = pt->getEvent(j);
             _manager.add(distances[findEvent(ev2)] >
                          distances[findEvent(ev1)]);
           }
       }
     }
```

```
      for(IlcInt i=0; i< _nbPortsOut; i++) {
          Port* pt = (Port*) _portsOut[i];
          IlcAnyArray events = pt->getEvents();
          for(IlcInt j=0; j<events.getSize(); j++) {
            if(j==0)  {
              Event* ev = pt->getEvent(j);
              _manager.add(distances[findEvent(ev)] > distances[0]);
            } else {
              Event* ev1 = pt->getEvent(j-1);
              Event* ev2 = pt->getEvent(j);
              _manager.add(distances[findEvent(ev2)] >
                          distances[findEvent(ev1)]);
            }
        }
      }
      if (!(_withMinConstraints || _withMaxConstraints)) {
           return build_constraint_system(distances);
      } else if (_nbMinConstraints > _nbMaxConstraints)
           return build_constraint_system(max, distances);
      else
           return build_constraint_system(min, distances);
}

IlcIntVarArray TimingDiagram::build_constraint_system(IlcIntVarArray
distances) {
   _manager.out() << "maximum separation with LINEAR, Min or Max
                     constraints" << endl;
   for(IlcAnySetIterator iterc(_constraintsAsm);iterc.ok();++iterc) {
      IlcAny val = *iterc;
      TimingConstraint* tc = (TimingConstraint*) val;
      compositeType cmp = tc->getCompositeType();
      if (cmp == linear)
        distances = applyLinearConstraint(tc, distances);
      else if (cmp == min)
        distances = applyNonLinearConstraint(tc, min, distances);
      else
        distances = applyNonLinearConstraint(tc, max, distances);
   }
   for(IlcAnySetIterator iterc(_constraintsCmt);iterc.ok();++iterc) {
      IlcAny val = *iterc;
      TimingConstraint* tc = (TimingConstraint*) val;
      compositeType cmp = tc->getCompositeType();
      if (cmp == linear)
        distances = applyLinearConstraint(tc, distances);
      else if (cmp == min)
        distances = applyNonLinearConstraint( tc, min, distances);
      else
        distances = applyNonLinearConstraint(tc, max, distances);
   }
   for(IlcAnySetIterator iterc(_addedConstraintsAsm); iterc.ok();
                                    ++iterc) {
      IlcAny val = *iterc;
      TimingConstraint* tc = (TimingConstraint*) val;
      compositeType cmp = tc->getCompositeType();
      if (cmp == linear)
        distances = applyLinearConstraint(tc, distances);
```

```
      else if (cmp == min)
        distances = applyNonLinearConstraint(tc, min, distances);
      else
        distances = applyNonLinearConstraint(tc, max, distances);
   }
   for (IlcAnySetIterator iterc(_addedConstraintsCmt); iterc.ok();
                                          ++iterc ) {
      IlcAny val = *iterc;
      TimingConstraint* tc = (TimingConstraint*) val;
            compositeType cmp = tc->getCompositeType();
      if (cmp == linear)
        distances = applyLinearConstraint(tc, distances);
      else if (cmp == min)
        distances = applyNonLinearConstraint(tc, min, distances);
      else
        distances = applyNonLinearConstraint(tc, max, distances);
   }
   return distances;
}

IlcIntVarArray TimingDiagram::applyLinearConstraint(TimingConstraint*
tc, IlcIntVarArray occur) {
   IlcAnyArray comp = tc->getComposants();
   IlcInt nc = comp.getSize();
   TimingRelation* tr = (TimingRelation*) comp[0];
   Event* sink = tr->getSink();
   IlcInt indSink = findEvent(sink);
   if ( indSink > 0) {
     for (IlcInt i = 0; i < nc; i++) {
        tr = (TimingRelation*) comp[i];
        Event* source = tr->getSource();
        IlcInt indSource = findEvent(source);
        IlcIntVar delay(_manager, tr->getMin(), tr->getMax());
        if ( indSource >= 0 ) {
           _manager.add(occur[indSink] == occur[indSource] + delay);
        }  else {
           _manager.out() << "Source event cannot be found for
                   constraint " << tr->getName() << endl;
           break;
        }
      }
    } else
        _manager.out() << "Sink event cannot be found for constraint"
                       << tc->getName() << endl;
    return occur;
}

IlcIntVarArray TimingDiagram::applyNonLinearConstraint(
   TimingConstraint* tc, compositeType ct, IlcIntVarArray occur) {
   IlcAnyArray comp = tc->getComposants();
   IlcInt nc = comp.getSize();
   IlcIntVarArray interval(_manager, nc, -infinie, infinie);
   TimingRelation* tr = (TimingRelation*) comp[0];
   Event* sink = tr->getSink();
   IlcInt indSink = findEvent(sink);
   if (indSink > 0) {
```

```
      for (IlcInt i = 0; i < nc; i++) {
        tr = (TimingRelation*) comp[i];
        Event* source = tr->getSource();
        IlcInt indSource = findEvent(source);
        IlcIntVar delay(_manager, tr->getMin(), tr->getMax());
        if ( indSource >= 0 ) {
           _manager.add(interval[i] == occur[indSource] + delay);
        } else {
           _manager.out() << "Source event cannot be found for the
                             constraint " << tr->getName() << endl;
           break;
        }
       }
       if (ct == min) {  // apply a min constraint
            _manager.add(occur[indSink] == IlcMin(interval));
       } else {
            _manager.add(occur[indSink] == IlcMax(interval));
       }
  } else
       _manager.out() << "Sink event cannot be found for the
                         constraint " << tc->getName() << endl;
       return occur;
}
// decomposition necessary
IlcIntVarArray TimingDiagram::build_constraint_system(compositeType
                comp, IlcIntVarArray distances) {
   _manager.out() << "maximum separation with linear, min and max
                     constraints" << endl;
   for(IlcAnySetIterator iterc(_constraintsAsm);iterc.ok();++iterc) {
      IlcAny val = *iterc;
      TimingConstraint* tc = (TimingConstraint*) val;
      compositeType cmp = tc->getCompositeType();
      if (cmp == linear)
        distances = applyLinearConstraint(tc, distances);
      else if ( cmp != comp)
        distances = applyNonLinearConstraint(tc, cmp, distances);
      else
        distances = decomposition( tc, cmp, distances);
   }
   for(IlcAnySetIterator iterc(_constraintsCmt);iterc.ok();++iterc) {
      IlcAny val = *iterc;
      TimingConstraint* tc = (TimingConstraint*) val;
      compositeType cmp = tc->getCompositeType();
      if (cmp == linear)
        distances = applyLinearConstraint( tc, distances);
      else if ( cmp != comp)
        distances = applyNonLinearConstraint( tc, cmp, distances);
      else
        distances = decomposition(tc, cmp, distances);
   }
   for (IlcAnySetIterator iterc(_addedConstraintsAsm); iterc.ok();
                                          ++iterc ) {
      IlcAny val = *iterc;
      TimingConstraint* tc = (TimingConstraint*) val;
      compositeType cmp = tc->getCompositeType();
      if (cmp == linear)
```

```
        distances = applyLinearConstraint( tc, distances);
      else if ( cmp != comp)
        distances = applyNonLinearConstraint(tc, cmp, distances);
      else
        distances = decomposition(tc, cmp, distances);
   }
   for (IlcAnySetIterator iterc(_addedConstraintsCmt); iterc.ok();
                                          ++iterc ) {
      IlcAny val = *iterc;
      TimingConstraint* tc = (TimingConstraint*) val;
      compositeType cmp = tc->getCompositeType();
      if (cmp == linear)
        distances = applyLinearConstraint(tc, distances);
      else  if ( cmp != comp)
        distances = applyNonLinearConstraint(tc, cmp, distances);
      else
        distances = decomposition(tc, cmp, distances);
   }
   return distances;
}

IlcIntVarArray TimingDiagram::decomposition(TimingConstraint* tc,
               compositeType cmp, IlcIntVarArray distances) {
     cout << "decomposition of timing constraint" << endl;
     Event* sink = tc->getSink();
     IlcInt indsink = findEvent(sink);
     if (indsink) {
       IlcInt nbcomp =  tc->getNbComposants();
       IlcIntVarArray interval(_manager, nbcomp, -infinie, infinie);
       IlcAny val; IlcInt i = 0;
       for (IlcAnySetIterator iter(tc->getComposants()); iter.ok();
                                          ++iter ) {
          val = *iter;
          TimingRelation* tr = (TimingRelation*) val;
          Event* source = tr->getSource();
          IlcInt j = findEvent(source);
          IlcIntVar delay(_manager, tr->getMin(), tr->getMax());
          if ( j >= 0 ) {
             _manager.add(interval[i] == delay + distances[j]);
             i++;
          } else {
            cout << "source event cannot be found" << endl;
            return distances;
          }
       }
       IlcBoolVarArray bools(_manager, nbcomp);
       IlcIntVarArray integers(_manager, nbcomp, 0, 1);
       IlcIntVarArray somme(_manager, nbcomp, -infinie, infinie);
       for (IlcInt i = 0; i< nbcomp; i++) {
           bools[i] = IlcBoolVar(_manager);
           if ( cmp == min ) {
              IlcIntSetVar ind = find_min(_manager, interval);
              IlcIntSet indpos = ind.getPossibleSet();
              IlcBool bv = indpos.isIn(i);
              if(!bv) _manager.add(bools[i] != 1);
           } else {
```

```
                IlcIntSetVar ind = find_max(_manager, interval);
                IlcIntSet indpos = ind.getPossibleSet();
                IlcBool bv =  indpos.isIn(i);
                if(!bv) _manager.add(bools[i] != 1) ;
              }
              _manager.add(integers[i] == bools[i]);
            _manager.add(somme[i] == interval[i]*integers[i]);
        }
        IlcIndex ii(_manager);
        _manager.add(IlcCard(ii, integers[ii] == 1) == 1);
        _manager.add(distances[indsink] == IlcSum(somme));
        _manager.add(IlcGenerate(bools, IlcChooseFirstUnboundBool,
                     IlcFalse));
        } else cout << "sink event cannot be found" << endl;
        return distances;
}

//============= Block class definition ====================//

class Block {
private:
    IlcManager _manager;
    IlcIntVar _rank;
    IlcIntVar _dir;
    IlcIntVar _type;
    IlcAnySetVar _evList;
    IlcAnySetVar _trigList;
public:
  Block(IlcManager m, IlcIntVar rk, IlcIntVar dir, IlcIntVar comp,
        IlcAnySetVar ev, IlcAnySetVar trig):_manager(m), _rank(rk),
        _dir(dir), _type(comp), _evList(ev), _trigList(trig){}
  IlcManager getManager() { return _manager;}
  IlcIntVar getRank() {return _rank; }
  IlcIntVar getDirection() { return _dir; }
  IlcIntVar getType() {return _type; }
  IlcAnySetVar getEvents() {return _evList; }
  IlcAnySetVar getTriggers() {return _trigList; }
  void display();
  void setConstraints();
};

void Block::setConstraints() {
    // min or max block can contain at most one event
    IlcIfThen(_type == 1 || _type ==2, IlcCard(_evList) == 1);
    // local event list should be different from trigger events list
    _manager.add(IlcNullIntersect(_evList, _trigList));
}

//============== modules for causality checking ==============

IlcAnyArray partition;
IlcIntArray maxcycle;
IlcIntArray mincycle;

IlcIntVarArray rank_events(TimingDiagram td, IlcIntVarArray ranks) {
    IlcManager m = td.getManager();
```

```
    IlcAnyArray events = td.getEvents();
    IlcInt evsize = events.getSize();
    for (IlcInt i = 0; i< evsize; i++) {
       Event* ev = (Event*) events[i];
       modeType mde = ev->getMode();
       IlcAnySet withSink = td.getEventsLinkedTo(ev);
       IlcInt sz = withSink.getSize();
       Port* port = findPortForEvent(td, ev);
       if(port) {
         IlcInt index = findRankForEvent(port, ev);
         if(index >=0) {
            if(sz == 0 && index == 0) m.add(ranks[i] == 0);
            else {
              if(index > 0) {
                Event* evprec = findEventWithRank(port, index-1);
                m.add(ranks[td.findEvent(evprec)-1] <= ranks[i]);
              }
              IlcAny val;
              for (IlcAnySetIterator iterc(withSink); iterc.ok();
                                                  ++iterc) {
                 val = *iterc;
                 Event* evs = (Event*)val;
                 if(ev->getMode() != evs->getMode() && evs->getMode()
                           != none)
                    m.add(ranks[td.findEvent(evs)-1] < ranks[i]);
                 else
                    m.add(ranks[td.findEvent(evs)-1] <= ranks[i]);
              } }}}}
    return ranks;
}

IlcIntVarArray execute_blocks(TimingDiagram td, IlcAnyArray partition,
IlcInt rk, IlcIntVarArray dis, IlcBool con) {
    IlcManager m = td.getManager();
    IlcInt found = -1;
    if(rk == 0) m.add(dis[0] == 0);    // 14/02/2001
    for (IlcInt i=0; i<partition.getSize();i++) {
       Block* b = (Block*)partition[i];
       IlcIntVar rank = b->getRank();
       if(rank.getValue() == rk) {
            found = i;
            break;
       }
    }
    if (found>=0) {
        Block* blk = (Block*)partition[found];
        IlcAnySet events = (blk->getEvents()).getRequiredSet();
        IlcAny vale;
        for (IlcAnySetIterator iter(events); iter.ok(); ++iter) {
           vale = *iter;
           Event* ev = (Event*) vale;
           Port* pt = findPortForEvent(td, ev);
           IlcInt ind = findRankForEvent(pt, ev);
           IlcAnySet cstSet = td.getConstraintsWithSink(ev);
           IlcAny valc;
           for (IlcAnySetIterator iterc(cstSet);iterc.ok();++iterc) {
```

```
            valc = *iterc;
            TimingConstraint* cst = (TimingConstraint*) valc;
            compositeType comp = cst->getCompositeType();
            if (comp == linear) {
                  dis = td.applyLinearConstraint(cst, dis);
            } else if (comp = min)
                  dis = td.applyNonLinearConstraint(cst, min, dis);
            else
                  dis = td.applyNonLinearConstraint(cst, max, dis);
        }}}
    if (rk > 0 && con)
        dis = execute_blocks(td, partition, rk-1, dis, con);
    return dis;
}

ILCGOAL2 (not_well_defined, IlcIntVar, var1, IlcIntVar, var2) {
  IlcManager m = getManager();
  m.add(var2 == 0);
  m.add(var1 >= var2);
  return 0;
}

ILCGOAL4 (not_past_dominated, IlcIntVar, var1, IlcIntVar, var2,
IlcIntVar, var3, IlcIntVar, var4) {
  IlcManager m = getManager();
  m.add(var2 == 0); m.add(var4 == 0);
  IlcInt maxvar3 = var3.getMax();
  IlcIntVar var3max(m, maxvar3, maxvar3);
  m.add(var3 == var3max);
  m.add(var1-var2 >= var3 -var4);
  return 0;
}

ILCGOAL2(valid_triggers, TimingDiagram, td, IlcAnyArray, partition) {
    IlcManager m = getManager();
    IlcAnyArray events = td.getEvents();
    IlcInt evsize = events.getSize();
    for (IlcInt i=0; i<partition.getSize(); i++) {
       Block* b = (Block*)partition[i];
       b->display();
       IlcInt type = (b->getType()).getValue();
       IlcAnySet eventList = (b->getEvents()).getRequiredSet();
       IlcAnySet trigList = (b->getTriggers()).getRequiredSet();
       IlcInt eventsize = eventList.getSize();
       IlcInt trigsize = trigList.getSize();
       IlcIntVarArray dis(m, evsize+1, -infinie, infinie);
       IlcAny vale;
       if(type == 0 && trigsize > 0 ) {
          dis = execute_blocks(td, partition,
                 b->getRank().getValue(), dis, IlcTrue);
          IlcIntVarArray localEvents(m,eventsize, -infinie, infinie);
          IlcIntVarArray triggers(m, trigsize, -infinie, infinie);
          IlcInt i=0;
        for(IlcAnySetIterator itere(eventList);itere.ok(); ++itere) {
            vale = *itere;
            Event* ev = (Event*)vale;
```

```
            IlcInt id = td.findEvent(ev);
            m.add(localEvents[i] == dis[id]);
            i++;
         }
         IlcIntVar minev(m, -infinie, infinie);
         m.add(minev == IlcMin(localEvents));
         i=0;
         IlcBoolVarArray bl(m, trigsize);
         for (IlcAnySetIterator itere(trigList); itere.ok();
         ++itere) {
            vale = *itere;
            Event* ev = (Event*)vale;
            IlcInt id = td.findEvent(ev);
            m.add(triggers[i] == dis[id]);
            if(m.solve(not_well_defined(m, triggers[i], minev))) {
              m.out() << "event " << ev->getName() << " is not a
                         well_defined trigger" << endl;
              m.fail();
            }
            i++;
         }
      }
   }
   return 0;
}

ILCGOAL2(valid_blocks, TimingDiagram, td, IlcAnyArray, partition) {
   IlcManager m = getManager();
   IlcAnyArray events = td.getEvents();
   IlcInt evsize = events.getSize();
   for (IlcInt i=0; i<partition.getSize(); i++) {
      Block* b = (Block*)partition[i];
      IlcInt rk = b->getRank().getValue();
      IlcInt type = (b->getType()).getValue();
      IlcAnySet eventList = (b->getEvents()).getRequiredSet();
      IlcAnySet trigList = (b->getTriggers()).getRequiredSet();
      IlcInt eventsize = eventList.getSize();
      IlcInt trigsize = trigList.getSize();
      IlcIntVarArray dist(m, events.getSize()+1, -infinie, infinie);
      IlcIntVarArray dist1(m, events.getSize()+1,-infinie, infinie);
      if(type == 0 && trigsize > 1 ) {
         dist = execute_blocks(td, partition, rk-1, dist, IlcTrue);
         dist1 = execute_blocks(td, partition, rk, dist1, IlcFalse);
         IlcIntVarArray trig1(m, trigsize, -infinie, infinie);
         IlcIntVarArray trig2(m, trigsize, -infinie, infinie);
         IlcInt i = 0;
         IlcAny vale;
       for (IlcAnySetIterator itere(trigList);itere.ok(); ++itere) {
            vale = *itere;
            Event* ev = (Event*)vale;
            IlcInt id = td.findEvent(ev);
            m.add(trig1[i] == dist[id]);
            m.add(trig2[i] == dist1[id]);
            i++;
         }
         for (IlcInt i=0; i< trigsize; i++) {
```

```
            for (IlcInt j=0; j< trigsize; j++) {
               if( i != j) {
                 if(m.solve(not_past_dominated(m, trig1[i],
                    trig1[j], trig2[i], trig2[j]))) {
                    m.out() << "past_dominated checking is failed "
                            <<  endl;
                    m.fail();
                 }
               }
            }
         } }}
    return 0;
}

ILCGOAL5(generate_blocks, TimingDiagram, td, IlcIntVar, nbBlocks,
   IlcIntVarArray, ranks, IlcIntArray, modes, IlcIntArray, types) {
      IlcManager m = getManager();
      IlcInt nblks = nbBlocks.getValue();
      IlcAnyArray events = td.getEvents();
      IlcInt evsize = events.getSize();
      IlcIntVarArray rk(m, nblks, 0, nblks-1);
      IlcIntVarArray dir(m, nblks, 0, 1);
      IlcIntVarArray type(m, nblks, 0, 2);
      IlcAnySetVarArray evList(m, nblks, events);
      IlcAnySetVarArray trigList(m, nblks, events);
      for (IlcInt i=0; i< nblks; i++) {
           Block* bk = new (m.getHeap())Block(m, rk[i], dir[i],
                        type[i], evList[i], trigList[i]);
           partition[i] = bk;
           bk->setConstraints();
      }
      for (IlcInt i=0; i< nblks; i++) {
         Block* blk = (Block*)partition[i];
         m.add(rk[i] == blk->getRank());
         m.add(dir[i] == blk->getDirection());
         m.add(type[i] == blk->getType());
         m.add(evList[i] == blk->getEvents());
         m.add(trigList[i] == blk->getTriggers());
         m.add(IlcCard(evList[i]) >= 1);
      }
      IlcAnySetVarArray srctrig(m, evsize, events);
      for(IlcInt i=0; i<evsize; i++) {
          Event* ev = (Event*)events[i];
          IlcAnySet trig = td.getEventsLinkedTo(ev);
          IlcAnySet sources = td.getEventsLinkingWith(ev);
          if(sources.getSize()>0) m.add(srctrig[i] == sources);
          else m.add(IlcNotMember(ev, IlcUnion(trigList)));
          for(IlcInt j=0; j< nblks; j++) {
              Block* blk = (Block*)partition[j];
              IlcIfThen(dir[j] != modes[i] || type[j] != types[i],
                           IlcNotMember(ev, evList[j]));
              IlcIfThen(IlcMember(ev, evList[j]), ranks[i] == rk[j]);
              IlcIfThen(IlcMember(ev, trigList[j]),ranks[i]< rk[j]);
            if(trig.getSize() >0)
              IlcIfThen(IlcMember(ev, evList[j]), IlcSubsetEq(trig,
                           IlcUnion(evList[j], trigList[j])));
```

```
            if(sources.getSize() > 0)
                IlcIfThen(IlcNullIntersect(evList[j], srctrig[i]),
                          IlcNotMember(ev, trigList[j]));
        }
    }
    m.add(IlcAllDiff(rk));
    m.add(IlcPartition(evList, events));
    m.add(IlcUnion(evList) == events);
    IlcGoal g1 = IlcAnd(IlcGenerate(evList, IlcChooseMinSizeAnySet),
                  IlcGenerate(trigList, IlcChooseMinSizeAnySet));
    IlcGoal g2 = IlcAnd(IlcGenerate(dir, IlcChooseMinSizeInt,
SelectMin(m)), IlcGenerate(type, IlcChooseMinSizeInt, SelectMin(m)));
    IlcGoal g3 = IlcGenerate(rk, IlcChooseMinSizeInt, SelectMin(m));
    IlcGoal valid_partition = IlcAnd(valid_triggers(m, td,
                partition),valid_blocks(m, td, partition));
    IlcGoal g4 = IlcAnd(IlcAnd(g1, g2), g3);
    m.add(IlcAnd(g4, valid_partition));
    return 0;
}

IlcBool generateBlockMachine(TimingDiagram td, IlcInt direction,
                             IlcInt initial, IlcFloat time) {
    IlcManager m = td.getManager();
    IlcAnyArray events = td.getEvents();
    IlcInt evsize = events.getSize();
    IlcInt cstMin = td.getNbMinConstraints();
    IlcInt cstMax = td.getNbMaxConstraints();
    IlcInt nbEventsIn = td.getNbEventsIn();
    IlcInt nbEventsOut = td.getNbEventsOut();
    IlcIntArray types(m, evsize);
    IlcIntArray modes(m, evsize);
    IlcIntVarArray ranks(m, evsize, 0 , evsize-1);
    ranks = rank_events(td, ranks);
    for (IlcInt i=0; i< evsize; i++) {
       Event* ev  = (Event*) events[i];
       modes[i] = ev->getMode();
       compositeType comp = linear;
       IlcAnySet cst = td.getConstraintsWithSink(ev);
       if (cst.getSize()) {
            IlcAny val;
            for (IlcAnySetIterator iter(cst); iter.ok(); ++iter ) {
               val = *iter;
               TimingConstraint* tcst = (TimingConstraint*) val;
               comp = tcst->getCompositeType();
               if (comp == min || comp == max) break;
            }
       }
       types[i] = (IlcInt)comp;
    }
    IlcIntVar nbBlocks;
    IlcInt lim = 0;
    if(nbEventsIn >0 && nbEventsOut>0)
         lim = IlcMax(2, cstMin+cstMax);
    else lim = IlcMax(1, cstMin+cstMax);
    if(direction == 1)
         nbBlocks = IlcIntVar(m, lim, IlcMin(initial, evsize));
```

```
      else nbBlocks = IlcIntVar(m, IlcMax(lim, initial), evsize);
      IlcGoal g0;
      if(direction == 0) g0 = IlcAnd(IlcInstantiate(nbBlocks,
           SelectMin(m)), create_partition(m, nbBlocks));
      else g0 = IlcAnd(IlcInstantiate(nbBlocks, SelectMax(m)),
                create_partition(m, nbBlocks));
      IlcGoal g1 = generate_blocks(m, td, nbBlocks, ranks, modes,
                                   types);
      IlcGoal construct_partition;
      if(time > 0) { // a limit search is executed
        m.setDfsOnly(IlcFalse);
        IlcGoal g2 = IlcLimitSearch(g1, IlcTimeLimit(m, time));
        construct_partition = IlcAnd(g0, g2);
      } else construct_partition = IlcAnd(g0, g1);
      IlcBool Causal=IlcFalse;
      if(m.solve(construct_partition)) {
        Causal = IlcTrue;
        IlcAny val;
        cout << "A solution is found " << endl;
        cout << "block number = " << partition.getSize() << endl;
           for (IlcInt i=0; i<partition.getSize(); i++) {
             Block* blk = (Block*) partition[i];
             IlcAnySet evs= (blk->getEvents()).getRequiredSet();
             if(evs.getSize()>0) blk->display();
           }
          m.printInformation();
      }
      if (!Causal) {
          m.out() << "Any causal partition can be found (within the
                      given time limit)" << endl;
          m.printInformation();
      }
      return Causal;
}


//============= FSM class definition ==================//

class FsmState {
public:
    IlcManager _manager;
    const char* label;
    const IlcInt no;
    FsmState(IlcManager m, char* nm, IlcInt number): _manager(m),
             label(nm), no(number){}
    void display();
};
class LogicExpr{
public:
   IlcManager _manager;
   IlcIntArray types;  //state (1), logical (0) and expr (2) element
   IlcInt nbNames;
   char** nameList;  //for WaveState label checking
   IlcInt nbTerms;
   IlcInt nbLogicElement;
   IlcInt nbStateElement;
```

```
   IlcIntArray nbElements; // delimits of each term
   IlcIntArray memLogical;  // index of each element to determine if
negative necessary
   IlcInt nbStates;
   char** stateList;
   IlcIntArray stateType;  // current (0) or next (1)
   IlcIntArray memState;   // same function as memLogical
   IlcAnyArray exprs;
   IlcIntArray memExpr;
   LogicExpr(IlcManager m, IlcInt nle, IlcInt nse, IlcIntArray ty,
IlcInt nr, char** ref, IlcInt ns, char** slist, IlcIntArray st, IlcInt
nt, IlcIntArray ne, IlcIntArray ml, IlcIntArray ms, IlcAnyArray eps,
IlcIntArray me):_manager(m), nbLogicElement(nle), nbStateElement(nse),
types(ty), nbNames(nr), nameList(ref), nbStates(ns), stateList(slist),
stateType(st), nbTerms(nt), nbElements(ne), memLogical(ml),
memState(ms), exprs(eps), memExpr(me){}
   void display();
};

class Transition {
public:
    IlcManager _manager;
    const char* label;
    LogicExpr* condition;
    LogicExpr* conclusion;
    Transition(IlcManager m, char* name, LogicExpr* ct1, LogicExpr*
ct2): _manager(m), label(name), condition(ct1), conclusion(ct2){ }
    void display();
};

class Assignment{
public:
    IlcManager _manager;
    const char* name;
    IlcInt nbNames;
    char** nameList;
    char* label;
    LogicExpr* value;
    Assignment(IlcManager m, char* nm, IlcInt nbNm, char** nList, char*
out, LogicExpr* le): _manager(m), name(nm),nbNames(nbNm),
nameList(nList), label(out), value(le){}
    void display();
};
class AsynchronousOutput {
public:
    IlcManager _manager;
    char* output;
    compositeType op;
    IlcInt nbInputs;
    char** events;
    AsynchronousOutput(IlcManager m, char* ev, compositeType oper,
         IlcInt nb, char** evs):_manager(m), output(ev), op(oper),
         nbInputs(nb), events(evs){}
    void display();
};
```

```
class FSM {
public:
   IlcManager _manager;
   const char* name;
   FsmState* original_state;
   IlcAnyArray states;
   IlcInt nbInputs;
   char** inputNames;
   IlcInt nbOutputs;
   char** outputNames;
   IlcInt nbRegisters;
   char** registerNames;
   IlcIntArray reginit;
   IlcAnyArray transitions;
   IlcAnyArray assignments;
   IlcAnyArray asynchrones;
   FSM(IlcManager m, char* nm, IlcAnyArray States, IlcInt Inputs,
char** iname, IlcInt Outputs, char** oname, IlcInt Registers, char**
rname, IlcIntArray regs, IlcAnyArray tr, IlcAnyArray as, IlcAnyArray
asyn, FsmState* original): _manager(m), name(nm), states(States),
nbInputs(Inputs), inputNames(iname), nbOutputs(Outputs),
outputNames(oname), nbRegisters(Registers), registerNames(rname),
reginit(regs), transitions(tr), assignments(as), asynchrones(asyn),
original_state(original) { }
    void display();
};

ILCSTLBEGIN
IlcInt Maxcycle;
TimingDiagram* TD = 0; FSM* fsm = 0;
IlcInt sizePin, sizePout, totalPsize;
IlcInt nbEventsIn, nbEventsOut;
IlcInt nbPortsIn, nbPortsOut;
IlcAnyArray portsIn, portsOut;
IlcIntArray initialValues;
IlcInt nbStatesFsm, nbTransitions, nbAssignments;
IlcAnyArray Transitions, Assignments;
IlcInt nbRegisters, nbAsynchrones;
char** registerList;
IlcAnyArray Asynchrones, events;
IlcInt finalState, position, indSink;
IlcInt indLogic = 0; IlcInt indState = 0; IlcInt indExpr = 0;

IlcConstraint getLogicElement(IlcIntArray memLogic, IlcBoolVarArray
vars) {
  IlcConstraint ct;
  IlcInt ind = memLogic[indLogic];
  if(ind > 1) {
    if(ind%2 == 0) ct = (vars[ind/2] != 1);
    else ct = (vars[ind/2] == 1);
  } else {
    if(ind == 0) ct = (vars[ind/2] != 1);
    else  ct = (vars[ind/2] == 1);
  }
  return ct;
}
```

```
IlcConstraint getStateElement(IlcInt stateType, IlcIntArray memState,
IlcIntVar current, IlcIntVar next) {
        IlcConstraint ct;
        IlcInt ind = memState[indState];
        if(ind > 1) {
          if(stateType == 0)
            if(ind%2 == 0) ct = (current != ind/2);
            else ct = (current == ind/2);
          else
            if(ind%2 == 0) ct = (next != ind/2);
            else ct = (next == ind/2);
        } else {
          if(stateType == 0)
            if(ind == 0) ct = (current != ind/2);
            else  ct = (current == ind/2);
          else
            if(ind == 0) ct = (next != ind/2);
            else  ct = (next == ind/2);
        }
        return ct;
}

IlcConstraint translate_logicExpr(LogicExpr* lexp, IlcBoolVarArray
vars, IlcIntVar current, IlcIntVar next);
IlcConstraint getExprElement(IlcIntArray memExpr, IlcAnyArray exprs,
IlcBoolVarArray vars, IlcIntVar current, IlcIntVar next) {
       IlcConstraint ct;
       IlcInt indlogic = indLogic;    //cache global variables
       IlcInt indstate = indState;
       IlcInt indexpr = indExpr;
       IlcInt ind = memExpr[indExpr];
       LogicExpr* lexp = (LogicExpr*) exprs[ind/2];
       if(ind > 1) {
         if(ind%2 == 1) ct = translate_logicExpr(lexp, vars,
                              current, next);
          else ct = !(translate_logicExpr(lexp, vars, current,
                        next));
        } else {
            if(ind == 0) ct = !(translate_logicExpr(lexp, vars,
                                 current, next));
            else  ct = translate_logicExpr(lexp, vars,current,next);
        }
        indLogic = indlogic; indState = indstate; indExpr = indexpr;
        return ct;
}

IlcConstraint getTerm(IlcInt nbElements, IlcIntArray types, IlcIntArray
stateType, IlcIntArray memLogic, IlcIntArray memState, IlcIntArray
memExpr, IlcAnyArray exprs, IlcBoolVarArray vars, IlcIntVar current,
IlcIntVar next) {
          IlcConstraint ct;
          IlcInt index = indLogic + indState + indExpr;
          if(types[index] == 0) {
             ct = getLogicElement(memLogic, vars);
             indLogic++;
          } else if(types[index] == 1) {
```

```
            ct = getStateElement(stateType[indState], memState,
            current, next);
            indState++;
          } else {
            ct=getExprElement(memExpr,exprs,vars,current,next);
            indExpr++;
          }
          for(IlcInt i=index+1; i< index+nbElements;i++) {
            if(types[i] == 0) {
                  ct = ct && getLogicElement(memLogic, vars);
                  indLogic++;
            } else if(types[i] == 1) {
                  ct = ct && getStateElement(stateType[indState],
                            memState, current, next);
                  indState++;
            } else {
                  ct = ct && getExprElement(memExpr, exprs, vars,
                             current, next);
                  indExpr++;
            }
          }
          return ct;
}

IlcConstraint translate_logicExpr(LogicExpr* lexp, IlcBoolVarArray
                  vars, IlcIntVar current, IlcIntVar next) {
          IlcInt nbterms = lexp->nbTerms;
          IlcIntArray nbElements = lexp->nbElements;
          IlcIntArray types = lexp->types;
          IlcIntArray memLogic = lexp->memLogical;
          IlcIntArray memState = lexp->memState;
          IlcIntArray stateType = lexp->stateType;
          IlcIntArray memExpr = lexp->memExpr;
          IlcAnyArray lexprs = lexp->exprs;
          IlcInt totalElements = IlcSum(nbElements);
          indLogic = 0; indState = 0; indExpr = 0;
          IlcConstraint ct = getTerm(nbElements[0], types, stateType,
           memLogic, memState, memExpr, lexprs, vars, current, next);
          for(IlcInt i=1; i< nbterms; i++) {
               ct = ct || getTerm(nbElements[i], types, stateType,
           memLogic, memState, memExpr, lexprs, vars, current, next);
          }
          return ct;
}

// create_constraint_system
ILCGOAL6(Create_constraint_system, IlcInt, clk, IlcIntVar, current,
IlcBoolVarArray, regOutput, IlcIntVarArray, presentStates,
IlcIntVarArray, Occurs, IlcIntVarArray*, clist ) {
      IlcManager m = getManager();
      if (clk < Maxcycle) {
          IlcBoolVarArray waveInputs(m, sizePin);
          IlcBoolVarArray waveOutputs(m, sizePout);
          IlcBoolVarArray regInput = IlcBoolVarArray();
          if(nbRegisters >0) {
          regInput = IlcBoolVarArray(m, nbRegisters);
```

```
            for(IlcInt i=0; i<nbRegisters; i++)
                regInput[i] = IlcBoolVar(m);
        }
        IlcInt nbleft = sizePout+nbRegisters;
        IlcInt nbright = sizePin+nbRegisters;
        IlcBoolVarArray leftside(m, nbleft);
        IlcBoolVarArray rightside(m, nbright);
        IlcIntVarArray nextStates(m, totalPsize);
        IlcIntVar  next(m, 0, nbStatesFsm-1);
        for (IlcInt i=0; i<totalPsize; i++) {
            for(IlcInt j=0; j<sizePin; j++) {
              waveInputs[j] = IlcBoolVar(m);
              Port* p = (Port*)portsIn[j];
              IlcInt nbStates = p->getNbStates();
              nextStates[j]= IlcIntVar(m, 0, nbStates-1);
            }
            for(IlcInt j=0; j<sizePout; j++) {
              waveOutputs[j] = IlcBoolVar(m);
              Port* p = (Port*) portsOut[j];
              IlcInt nbStates = p->getNbStates();
              nextStates[j+sizePin]= IlcIntVar(m, 0, nbStates-1);
            }
         }
          for(IlcInt i=0; i<nbleft; i++) {
               leftside[i] = IlcBoolVar(m);
               if(i<sizePout) m.add(leftside[i] == waveOutputs[i]);
               else m.add(leftside[i] == regInput[i-sizePout]);
          }
          for(IlcInt i=0; i<nbright; i++) {
               rightside[i] = IlcBoolVar(m);
               if(i<sizePin) m.add(rightside[i] == waveInputs[i]);
               else m.add(rightside[i] == regOutput[i-sizePin]);
          }
          if ( clk == 0) {
            m.add(current == 0);
            for(IlcInt i=0; i<totalPsize; i++)
                m.add(presentStates[i] == 0);
          }
// generate the input of the fsm from present input value
          for (IlcInt i=0; i<sizePin; i++) {
            Port* p = (Port*)portsIn[i];
            IlcInt nbStates = p->getNbStates();
            for(IlcInt j=0; j<nbStates; j = j+2) {
              IlcIfThen(presentStates[i] == j,
                        waveInputs[i]==initialValues[i]);
              if (j+1 < nbStates)
                  IlcIfThen(presentStates[i] == j+1,
                            waveInputs[i]!=initialValues[i]);
            }
          }
// generate assignments of fsm
            for(IlcInt i=0; i<nbAssignments; i++) {
              Assignment* assign = (Assignment*)Assignments[i];
              IlcConstraint as = translate_logicExpr(assign->value,
                                 rightside, current, next);
              char* label = assign->label;
```

```
            IlcInt ind = findLabel(label, assign->nbNames,
                                   assign->nameList);
            m.add(leftside[ind] == as);
        }
// set transition and output function
        for(IlcInt i=0; i<nbTransitions;i++) {
            Transition* tr = (Transition*)Transitions[i];
            LogicExpr* cd = tr->condition;
            LogicExpr* cl = tr->conclusion;
            IlcConstraint cond = translate_logicExpr(cd, rightside,
                                 current, next);
            IlcConstraint conclu = translate_logicExpr(cl, rightside,
                                 current, next);
            IlcIfThen(cond, conclu);
        }
// generate the next output waveform states
        for(IlcInt i=0; i<sizePout; i++) {
          Port* p = (Port*) portsOut[i];
          IlcInt nbStates = p->getNbStates();
          for(IlcInt j=0; j<nbStates; j = j+2) {
            IlcIfThen((presentStates[sizePin+i] == j &&
             waveOutputs[i] != initialValues[sizePin+i]) ||
             (presentStates[sizePin+i] == j+1 && waveOutputs[i] ==
             initialValues[sizePin+i]),
             nextStates[sizePin+i] == presentStates[sizePin+i]+1);
           IlcIfThen((presentStates[sizePin+i] == j && waveOutputs[i]
== initialValues[sizePin+i]) || (presentStates[sizePin+i] == j+1 &&
waveOutputs[i] != initialValues[sizePin+i]), nextStates[sizePin+i] ==
presentStates[sizePin+i]);
          }
        }
// detect an synchronous or asynchrnous output event
        IlcInt k = nbEventsIn+1;
        char* outname; IlcConstraint ct; IlcInt index = 0;
        IlcIntArray evs = IlcIntArray();
          IlcIntSet asynev = IlcIntSet();
          if(nbAsynchrones>0) {
          evs = IlcIntArray(m, nbAsynchrones);
          for(IlcInt i=0; i<nbAsynchrones; i++) {
            AsynchronousOutput* asyn = (AsynchronousOutput*)
                                       Asynchrones[i];
            outname = asyn->output;
            Event* ev = findEventWithName(outname, events);
            index = TD->findEvent(ev); evs[i] = index;
          }
          asynev = IlcIntSet(m, evs, IlcTrue);
        }
        for(IlcInt i=0; i<sizePout; i++) {
          Port* p = (Port*)portsOut[i];
          IlcInt nbStates = p->getNbStates();
          for(IlcInt j=0; j<nbStates-1; j++) {
            if(nbAsynchrones == 0 || !asynev.isIn(j+k))
              IlcIfThen(presentStates[sizePin+i] == j &&
                        nextStates[sizePin+i] == j+1,
                        Occurs[j+k] == (clk+1)*period);
             else {
```

```
                AsynchronousOutput* asyn = 0;
                for(IlcInt i=0; i<nbAsynchrones; i++) {
                  asyn = (AsynchronousOutput*) Asynchrones[i];
                  outname = asyn->output;
                  Event* ev = findEventWithName(outname, events);
                  index = TD->findEvent(ev);
                    if(index == j+k) break;
                }
                ct = translateAsynchronousOutput(asyn, TD, Occurs);
                IlcIfThen(presentStates[sizePin+i] == j &&
                          nextStates[sizePin+i] == j+1, ct);
              }
            }
            k = k+nbStates-1;
          }
// detect an input event
          k=1;
          for(IlcInt i=0; i<sizePin; i++) {
            Port* p = (Port*)portsIn[i];
            IlcInt nbStates = p->getNbStates();
            for(IlcInt j=k; j<k+nbStates-1; j++) {
              if(mincycle[j] <= clk && clk <= maxcycle[j]) {
              if(Occurs[j].getMin() >= clk*period && Occurs[j].getMax()
                       < (clk+1)*period)
                m.add(clist[j-1][clk] == 1);
              if(Occurs[j].getMin() >= (clk+1)*period ||
                       Occurs[j].getMax() < clk*period)
                m.add(clist[j-1][clk] == 0);
              IlcIntVar interval(m, clk*period, (clk+1)*period-1);
              IlcIfThen(clist[j-1][clk] == 1, Occurs[j]== interval );
              IlcIfThen(clist[j-1][clk] == 0,
                        !(Occurs[j] == interval));
              } else m.add(clist[j-1][clk] == 0);
            }
            k = k+ nbStates -1;
          }
// generate next input waveform states
          k = 0;
          for(IlcInt i=0; i< sizePin; i++) {
            Port* p = (Port*)portsIn[i];
            IlcInt nbStates = p->getNbStates();
            IlcIfThen(presentStates[i] == nbStates-1,
                      nextStates[i] == nbStates-1);
            for(IlcInt j=0; j<nbStates-1; j++) {
              IlcIfThen(presentStates[i] == j,
                        nextStates[i] == j + clist[k+j][clk]);
            }
            k = k+nbStates-1;
          }
         if(clk+2 == Maxcycle)
            m.add(nextStates[sizePin+position] == finalState);
            clk++;
            m.add(Create_constraint_system(m, clk, next, regInput,
               nextStates, Occurs, clist));
        } else {
            m.add(generate_clist(m, clist));
```

```
            cout << "for create_constraint_system : " << endl;
            m.printInformation();
        }
        return 0;
}

ILCGOAL4(generate_matrix, IlcIntVarArray*, clist, IlcInt, rank,
IlcAnyArray, Part, IlcAnyArray, events) {
        IlcManager m = getManager();
        for (IlcInt i=0; i< nbEventsIn; i++)
           clist[i] = IlcIntVarArray(m, Maxcycle, 0, 1);
        IlcIndex I(m);
        for (IlcInt i = 0; i < nbEventsIn; i++) {
           Block* b = findBlockForEvent(events[i], Part);
           Event* ev = (Event*)events[i];
           IlcInt evi = (b->getRank()).getValue();
           if(evi < rank) m.add(IlcCard(I, clist[i][I] == 1) == 1);
           if(evi > rank) m.add(IlcCard(I, clist[i][I] == 1) == 0);
        }
        return 0;
}

void calcul_cycle(TimingDiagram td, IlcInt evsize) {
        IlcManager m = td.getManager();
        IlcIntVarArray occurs = td.findDistances(td.getOrigine());
        maxcycle = IlcIntArray(m, evsize+1);
        mincycle = IlcIntArray(m, evsize+1);
        for(IlcInt i=0; i<evsize+1; i++) {
           mincycle[i] = IlcMax(0, floor(occurs[i].getMin()/period));
           maxcycle[i] = ceil(IlcFloat(occurs[i].getMax())/period)+1;
        }
}
```

Remerciements

Les travaux rapportés dans ce mémoire n'auraient pu être menés à terme sans le support et la direction d'Eduard Cerny. L'auteur tient donc à le remercier pour sa précieuse collaboration.

Doit également être remercié Gilles Pesant pour sa permission à l'ouvert d'un compte dans CRT pour accéder au logiciel ILOG Solver.

L'auteur veut aussi exprimer sa gratitude envers Fen Jin, Pierre Girodias et Yi Feng pour leur support dans le développement de ce projet.

Mon époux Hua Li, pour sa compréhension et son amour.

Mes chers enfants Victor et Linda pour leur sourire et leur amour.