UNIVERSITE JOSEPH FOURIER - GRENOBLE I

DEA Microélectronique
filière Conception de Systèmes Intégrés

Rapport du projet
réalisé au sein du Laboratoire VERIMAG

Analyse temporelle des circuits à travers le formalisme des automates temporisés

Réalisé par :
Ramzi BEN SALAH
(Ramzi.Salah@imag.fr)

Encadré par :
Oded MALER
(Oded.Maler@imag.fr)

juin 2003

Résumé

Avec l'avènement des technologies fortement submicroniques, la vérification est devenue le vrai handicape des concepteurs. Ce qui a poussé plusieurs industriels et laboratoires de recherches à s'investir pour pallier aux défaut des outils existants.

Dans ce projet nous proposons une nouvelle technique basée sur le concept des *automates temporisés*. C'est un formalisme qui peut inclure à la fois l'aspect fonctionnel et temporel du système à vérifier, et qui est assez efficace dans la gestion de l'indéterminisme dans les modèles de retard.

L'*explosion d'état* reste le principal défaut de cette méthode qui l'a toujours mis hors d'usage. Ce travail propose aussi une technique pour diminuer l'impact de ce problème.

Mots Clés : *vérification, simulation, automates temporisés, explosion d'états*

Abstract

With the advent of strong submicronic technologies, checking has become the principle handicap for conceptors. Consequently, this has caused industrialists and research laboratories to devote their research to solving the defects of existing tools.

In this project, we propose a new technique based on the concept of the *timed automata*. It is a formalism that could include at the same time the functional and temporal system aspects, this being effective in the management of indeterminism in the models of delay.

The *state explosion* remains the principle defect of this method. This work also proposes a technique that decreases the impact of this problem.

Keywords : *verification, simulation, timed automata, state explosion.*

Remerciements

Le présent travail à été réalisé au sein de l'équipe *"Systèmes Hybrides et Temporisés"* au laboratoire *VERIMAG* sous la direction de *M. Oded Maler*, Directeur de recherche au CNRS. Je le remercie vivement de m'avoir intégré à son équipe et permis de réaliser ce travail. Je lui suis également reconnaissant pour m'avoir encadré et orienté durant tout le projet ;

Je tiens aussi à exprimer mes remerciements envers toute autre personne qui a contribué, de prés ou de loin, au bon déroulement de ce stage. Je remercie :

- *M. Marius Bozga*, Ingénieur de recherche à VERIMAG, pour sa disponibilité et ses précieux conseils théoriques et techniques qui m'ont été d'un grand apport dans mon travail ;
- *M. Joseph Sifakis*, Directeur de VERIMAG, pour m'avoir accepté dans ce laboratoire et d'avoir réuni les bonnes conditions de déroulement du stage ;
- *M. Chaker Nakhli*, étudiant en doctorat à VERIMAG, pour son soutien amical et son aide et ses commentaires concernant quelques aspects techniques du travail ;
- Tout le reste de l'équipe du Laboratoire, un environnement de travail chaleureux et enthousiaste, que j'ai beaucoup apprécié au cours de ces moins.

Table des matières

Table des figures

Chapitre 1

Introduction

Du fait de l'évolution de la technologie et de l'augmentation rapide de la complexité des circuits, la seule issue qui restait devant des concepteurs, toujours soumis à la pression du "time to market", était de s'orienter vers les méthodologies et les outils de conception. Malheureusement ces derniers ont, aujourd'hui, du mal à suivre ce rythme exponentiel de croissance.

Parmi ces outils, ceux réservés à la vérification représentent le majeur handicape des concepteurs. Ils ont dépassé les 70% du temps global du processus de conception, sans pourtant arriver à couvrir un champ de test acceptable.

Dans plusieurs applications, comme les systèmes de contrôle d'énergie nucléaire ou les systèmes avioniques, un minimum de sûreté est indispensable. Une erreur non détectée à la phase de conception peut coûter beaucoup en terme de temps, d'argent, et surtout de vie humaine. La conception de tels systèmes, assez complexes et critiques a poussé vers l'exploration de nouvelles méthodologies de vérification.

Ce projet se place dans ce même cadre de recherche, en proposant une nouvelle technique basée sur le formalisme des automates temporisés.

1.1 État de l'art

1.1.1 Vérification formelle

Les méthodes de *vérification formelle* [Bar98, Bar00, CW96, CST01, Gup93] peuvent être classées en deux grandes catégories : la vérification basée sur le *theorem provers* et celle basée sur l'exploration de traces d'automates. Ces deux techniques ont trouvé beaucoup de succès et ont été intégrées dans plusieurs outils de vérification de circuits VLSI, voir tableau dans [Bla96].

La technique du *theorem proving* consiste à prouver des propriétés au système considéré à partir de ses axiomes et de ses règles d'inférences. La principale caractéristique de cette approche est sa capacité de décrire le circuit à plusieurs niveaux d'abstraction. Cela peut être un avantage, dans le sens où la plupart des circuits sont conçus d'une manière architecturale. Mais ce n'ai pas toujours le cas, surtout après une optimisation agressive du circuit. Son inconvénient

majeur qui la rend souvent inadaptée, est qu'elle n'est pas totalement automatisable, ce qui exige beaucoup d'expertise du côté de l'utilisateur.

Cette technique a été appliquée avec succès pour la vérification de plusieurs systèmes assez complexes, comme certains processeurs d'IBM (comme PowerPC et System/390) [KLS95, AK95], le CAP[1] de Motorola [BY96], les processeur AAMP5 [MS95], AMD5K86 [MKL96], etc.

Le *model-checking* comme l'*equivalence-checking* font partie de la catégorie de vérification basée sur les automates. Elles requièrent une exploration exhaustive de l'espace d'états du modèle. Pour des systèmes de grandes tailles, cette exploration amène souvent à un nombre très grand d'états. Ce phénomène, connu sous le nom d'*explosion d'états*, a été souvent la principale limitation à l'usage de ces méthodes. Pour remédier à ce problème, plusieurs tentatives d'amélioration ont été proposées [KM91] et de grands progrès ont été réalisés, surtout dans la vérification haut-niveau.

Contrairement au theorem-proving, le model-checking est une technique qui se prête bien à l'automatisation. Pour cette raison cette méthode a pu intègrer plusieurs outils CAO pour aider à la conception des circuits les plus complexes.

Comme exemple de l'application de cet approche (une liste plus complète est disponible dans [CW96]), nous citons celle du laboratoire VERIMAG de 1995. En effet, à travers leurs techniques implémentées dans la boite à outil CADP[2], des chercheurs du laboratoire ont contribué à la vérification du fonctionnement de l'architecture multiprocesseur appelée PowerScale, utilisée par des serveurs de Bull, et basée sur le processeur PowerPC d'IBM. Tout ce processus de vérification n'a pris que quelques minutes [CGM+96].

Souvent les méthodes formelles ont été appliquées dans le cadre de la vérification haut-niveau[3]. Aussi, elles ont servi à la vérification fonctionnelle au niveau circuit. Mais très rares sont les travaux, dans ce domaine, qui ont traité l'analyse temporelle des circuits. D'autres techniques sont plus utilisées pour ce traitement.

1.1.2 Analyse temporelle des circuits

Traditionnellement, la *simulation dynamique* a été souvent employée par les concepteurs pour la vérification des propriétés fonctionnelles et temporelles des circuits. Ce type de vérification simule le comportement d'un circuit donné soumis à une suite de vecteurs de test, et les résultats sont comparés à des valeurs prévues dans les mêmes conditions de manipulation. Avec les circuits industriels d'aujourd'hui, le nombre et la complexité de ces vecteurs ont beaucoup augmentés jusqu'à ce que le temps de vérification d'un circuit a dépassé les limites acceptables. Le majeur problème de cette méthode est sa dépendance de l'entrée[4], ce qui rend le test exhaustif hors du domaine de l'imaginaire. A titre

[1] Complex Arithmetic Processor
[2] CAESAR/ALDEBARAN Developement Packege.
[3] Système ou RTL
[4] L'état de l'environnement

d'exemple, si le temps de simulation d'une seule entrée d'une UAL[5], 32-bits à 77 entrées, prend 10^{-6} sec, la simulation exhaustive de cette dernière nécessite $10^{-6} * 2^{77} \approx 4,8 * 10^{9}$ ans [Bor00].

Cette applicabilité qui devient de plus en plus risquée et coûteuse de point de vue temps, a poussé les concepteurs à chercher d'autres techniques plus adaptées. Parmi ces techniques, *l'analyse temporelle statique* (*STA*[6]) [Dio98, Muk02, Dav01, DGL98, Syn99] a connu un grand succès. Le STA est une méthode très exhaustive pour un traitement purement temporel. Elle permet la vérification de toutes les traces d'exécution du modèle conçu indépendamment d'une entrée spécifique, une couverture que la simulation dynamique ne pourra jamais atteindre. Le STA a beacoup attirée les concepteurs de circuit cyclique, où il permet de vérifier certaines contraintes temporelles tel que le *hold* et le *setup* au niveau des registres. Il sert même à étalonner la fréquence de fonctionnement du circuit et ainsi sa performace.

Cela ne veut pas dire que le STA est arrivé à prendre la place de la simulation dynamique, surtout que *le STA se limite à l'aspect temporel* du circuit, et ne tient pas compte de son côté fonctionnel. Cette propriété met hors de sa portée tout phénomène incluant l'aspect logique du circuit. Ainsi, malgré tous ses vices, la simulation dynamique garde encore sa place, surtout pour la conception des circuits asynchrones.

Pour bénéficier des avantages de ces deux mondes, plusieurs ont essayé d'intégrer les deux méthodes dans un même outil, dit *hybride*. Malheureusement, une vérification exhaustive en un temps raisonnable, reste encore inatteignable.

1.2 Objectif de ce travail

Jusqu'à aujourd'hui les techniques formelles, et surtout les méthodes basées sur les automates, attirent beaucoup d'attention à travers les performances dont elles ont fait preuve. Leur application a été souvent limitée aux niveaux système et RTL. A ce niveau d'abstraction, la vérification est de grand intérêt, puisqu'elle permet la détection et la correction d'erreurs à un stade très avancé du processus de conception. Malheureusement, à ce niveau plusieurs propriétés du circuit restent invisibles. Ainsi, élargir la visibilité du model-checking, et étendre son domaine d'application vers la vérification au niveau circuit représente le premier défi de ce travail.

Au niveau porte, la plupart des outils d'analyse et de vérification existant gardent encore la séparation entre la logique fonctionnelle du circuit et ses propriétés temporelles. Cette séparation est parfois inacceptable, comme il est le cas pour les *circuits asynchrones* où l'on préfère garder encore la simulation dynamique comme seul moyen de vérification, malgré toutes ses faiblesses. Même dans le cadre des circuits synchrones, à cause de la négligence de la logique dans le calcul des chemins critiques, le STA peut facilement tomber sur des

[5]Unité Arithmétique Logique
[6]Static Timing Analysis

faux chemins, et par conséquence produire des estimations très pessimistes sur la performence des circuits.

La nouvelle technique que nous proposons dans ce travail, est basée sur le concept des *automates temporisés* noté aussi TA[7]. C'est un formalisme qui peut inclure à la fois l'aspect fonctionnel et temporel du système à vérifier et qui est assez efficace dans la gestion de l'indéterminisme dans les modèles de retard. Nous revindrons plus loin sur ces notions qui distinguent notre technique par rapport aux autres outils de vérification existants.

L'objectif global de cette recherche est de pouvoir appliquer cette méthode à l'analyse des circuits de grandes tailles, et de les intégrer dans des outils de CAO industriels. Le présent projet de DEA ne sort pas du cadre des circuits acycliques. Son but est de mettre la première pierre de cette nouvelle technique.

1.3 Organisation du rapport

Dans le *chapitre 2* nous introduisons tout d'abord le formalisme des automates temporisés. Après, nous expliquons le concept du retard dans les circuits pour leur proposer une modélisation en terme de TA. Nous présentons enfin la boite à outil IF qui servira à la génération du graphe d'atteignabilité des circuits traités. Déjà, l'étude de quelques petits circuits est possible à ce niveau.

Le *chapitre 3* servira à exprimer les limites de la méthode pour le traitement des circuits de grandes tailles. Dans ce même chapitre nous proposons une solution pour détourner ce problème et nous analysons son efficacité.

Le *chapitre 4* a comme principal but de résumer toute la procédure de vérification proposée dans ce travail. Il servira en même temps à rappeler une contribution effectuée dans ce stage sur le plan pratique, en vu d'automatiser tout le cycle d'analyse.

Nous finissons par *conclure* et présenter les perspectives de la méthode.

[7] Timed Automata

Chapitre 2

Automates temporisés et circuits

Dans ce chapitre nous introduisons en premier lieu le formalisme des automates temporisés qui sera à la base de tout notre travail. Nous nous consacrerons après à la modélisation des circuits et de leurs retards en terme d'automates temporisés.

2.1 Introduction aux automates temporisés

La théorie des automates temporisés[Yov97] fournit un cadre formel de modélisation et d'analyse du comportement des systèmes temps réels. Ce formalisme a été introduit en premier lieu en[AD90] comme une extension de l'approche théorique basée sur les automates. La caractéristique principale des automates temporisés, et qui leur donne cette appellation, sont les horloges.

Les horloges seront utilisées comme variables d'état mesurant la progression du temps. La valuation des horloges est notée v. Pour tout réel positif t, on note par $v+t$ la valuation des horloges obtenues depuis v en augmentant de t la valeur de toutes les horloges.

Les définitions faisant l'objet de cette section sont tirées de [Bor98].

2.1.1 Automates Temporisés

Définition 2.1 (Automate Temporisé).
Un Automate Temporisé se compose de :

- *Un système de transitions étiqutées $(S, \rightarrow, A)$ où :*
  - *S est un ensemble fini d'états*
  - *A est un vocabulaire fini d'actions*
  - *$\rightarrow \subseteq S \times A \times S$ est une relation de transition*
- *Un vecteur X d'horloges, variables à valeurs réelles positives. L'ensemble des valuations de X, isomorphe à $\mathbb{R}_+^n$, est noté V.*
- *Une fonction d'étiquetage h qui transforme les transitions non temporisées de $\rightarrow$ en des transitions temporisées : $h(s, a, s') = (s, (a, g, d, f), s')$, où :*

- *g et d sont des prédicats sur X appelés respectivement la* garde *et le* deadline *de la transition. Ils doivent satisfaire $d \Rightarrow g$.*
- *$f : V \to V$ est un* saut.

□

On peut déduire de la définition précédente qu'un automate temporisé peut être obtenu à partir d'un automate non temporisé en associant à chaque action a, une action temporisée $b = (a, g, d, f)$.

Définition 2.2 (Sémantique des Automates Temporisés).
Un état d'un automate temporisé est une paire (s, v) où $s \in S$ est un état de contrôle et $v \in V$ une valuation des horloges. On associe au automate temporisé une relation de transition $\to \subseteq (S \times V) \times (A \cup \mathbb{R}_+) \times (S \times V)$. Les transitions étiquetées par un élément de A sont des transitions discrètes tandis que les transitions étiquetées par des réels positifs sont des pas de temps.

Etant donné $s \in S$, si $\{(s, a_i, s_i)\}_{i \in I}$ est l'ensemble de toutes les transitions discrètes issues de s et $h(s, a_i, s_i) = (s, (a_i, g_i, d_i, f_i), s_i)$ alors :

- *$\forall i \in I \; \forall v \in \mathbb{R}_+^n \;\; (s, v) \xrightarrow{a_i} (s_i, f_i(v))$ si $g_i(v)$*
- *$(s, v) \xrightarrow{t} (s, v + t)$ si $\forall t' < t \; c_s(v + t)$ où $c_s = \neg \bigvee_{i \in I} d_i$*

□

Lorsque le prédicat g est vrai, la transition discrète associée devient franchissable. Le prédicat d est le deadline, il exprime la *date-limite* : lorsque d devient vrai, la transition est forcée. La garde g caractérise l'ensemble des états depuis lesquels la transition est possible alors que le deadline d caractérise le sous ensemble d'états où la transition est forcée impliquant l'arrêt de la progression du temps. La position de d par rapport à g détermine l'*urgence* de l'action. Pour une garde g donnée le deadline d peut prendre deux valeurs extrèmes : $d = g$ ce qui veut dire que la transition est franchie dès qu'elle est possible, ce type d'action est appelé *eager*. L'autre cas est $d = false$, l'action n'est donc jamais forcée et le temps peut s'écouler indéfiniment. Ce type d'action est appelée *lazy*. Un autre cas intéressant est quand le deadline est égal au front descendant de la garde : $d = g \downarrow$, l'action ne peut être désactivée sans qu'elle ne soit forcée. Ce type d'action est appelé *delayable*.

Notation En pratique, on utilise la notation g^ϵ, g^δ et g^λ qui dénotent respectivement la garde g avec les types d'urgence *eager*, *delayable* et *lazy*.

Un circuit, comme la plupart des systèmes temps réel, inclut souvent plusieurs composants que nous appellerons dans notre contexte *processus*. Ces processus évoluent en parallèle et peuvent inter-communiquer durant l'exécution du système. La composition de ces processus donnera le comportement du système global.

Définition 2.3 (Composition des Systèmes Temporisés).
Soit Aut_1 et Aut_2 deux automates temporisés, $Aut_i = (S_i, A_i, \to_i, X_i, h_i)_{i \in \{1,2\}}$

On appelle l'automate produit, noté $Aut = Aut_1 \parallel Aut_2$ l'automate temporisé $Aut = (S, A, \to, X, h)$ telque :

- $S = S_1 \times S_2$
- $X = X_1 \cup X_2$
- *Si* $s_i \xrightarrow{a_i}_i s'_i$ *et* $h_i(s_i, a_i, s'_i) = (s_i, b_i, s'_i)$ *avec* $b_i = (a_i, g_i, d_i, f_i)$ *et* $i \in \{1, 2\}$, *alors :*
  - $((s_1, s_2), a_1, (s'_1, s'_2)) \in \rightarrow$
  - $((s_1, s_2), a_2, (s'_1, s'_2)) \in \rightarrow$
  - $h((s_1, s_2), a_1, (s'_1, s'_2)) = ((s_1, s_2), b_1, (s'_1, s'_2))$
  - $h((s_1, s_2), a_2, (s'_1, s'_2)) = ((s_1, s_2), b_2, (s'_1, s'_2))$

□

2.2 Les circuits

Dans ce projet nous restons dans le cadre des circuits booléens acycliques sans points mémorisants. Comme nous l'avons signalé, nous traitons les circuits à leur niveau portes. Dans ces conditions, nous avons :

Définition 2.4 (circuit booléen acyclique).
Un circuit booléen acyclique est un tuple $C = (V, \rightsquigarrow, F)$ *où :*
- V *est un ensemble de nœuds (représentant les signaux)*
- $\rightsquigarrow$ *est une relation binaire antisymétrique et non réflexive.* $v_1 \rightsquigarrow v_2$ *veut dire* v_2 *est sensible á* v_1*.*
- F *est une fonction qui assigne à chaque nœud* v *(qui n'est pas une entrée du circuit considéré) une fonction combinatoire* $F_v : \mathbb{B}^{|\pi(v)|} \rightarrow \mathbb{B}$*, où* $\pi(v)$ *représente les prédécesseurs immédiats de* v *et* $|\pi(v)|$ *leur nombre.*

□

A titre d'exemple, pour le circuit donné par la figure 2.1, et qui nous servira d'exemple tout au long de ce rapport :

$$V = \{x_1, x_2, y_1, y_2, z_1, z_2\}$$
$$\rightsquigarrow = \{(x_1, y_1), (x_1, y_2), (x_1, z_1), (x_2, y_1), (x_2, y_2), (y_1, z_2), (y_2, z_1), (y_2, z_2)\}$$
$$F = \{f_1, f_2, f_3, f_4\}$$

$$\text{avec } y_1 = f_1(x_1, x_2) y_2 = f_2(x_1, x_2) z_1 = f_3(x_1, y_2) z_2 = f_4(y_1, y_2)$$

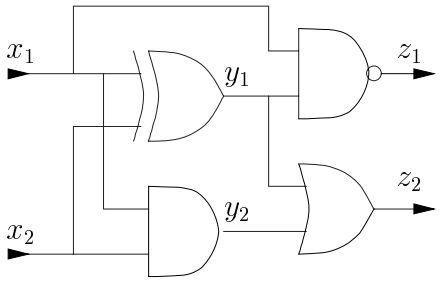

FIG. 2.1 – Exemple de circuit acyclique.

Cette définition est prise de l'article [SBM03] récemment écrit pour mieux formaliser notre travail. Jusqu'à maintenant nous avons défini le circuit comme une logique instantanée, ce qui n'est pas réellement vrai. Dans l'étape suivante, nous allons améliorer notre définition en introduisant la notion du temps.

2.2.1 Le retard dans les circuits

Avec l'évolution de la densité d'intégration et la miniaturisation des transistors, plusieurs nouveaux phénomènes parasites, que nous verrons plus loin, ont connu le jour. Ces phénomènes ne cessent de modifier le comportement temporel des nouveaux systèmes. Ainsi, la modélisation du retard des circuits microélectroniques a souvent été un problème d'actualité, un problème qui doit toujours dévisager le compromis entre la précision et l'efficacité.

La première cause du retard dans les circuits intégrés sont les capacités parasites :

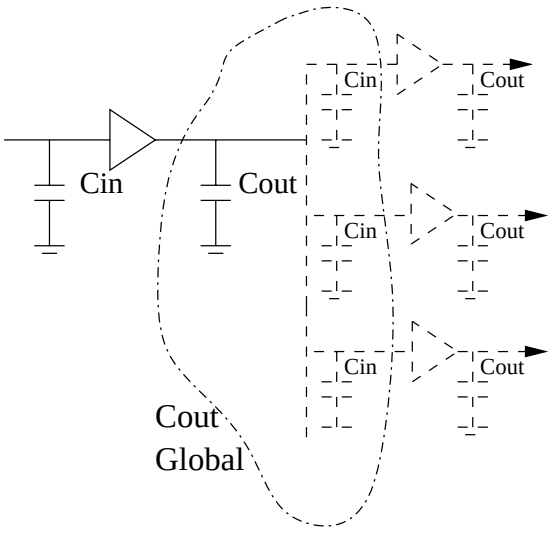

FIG. 2.2 – Les capacités parasites.

A travers l'exemple de la figure 2.2, on constate que le retard au niveau d'une porte ne dépend pas seulement de sa capacité de sortie mais aussi de sa sortance (fun-out). Plus précisément des capacités d'entrée de toutes les portes attachées à sa sortie. Cette remarque signifie que le vrai modèle d'une porte, n'a de sens que dans son circuit.

Avec les nouvelles technologies submicroniques les circuits ont connu de nouveaux phénomènes parasites qui ont beaucoup affecté les modèles de retard adoptés. *Les résistances d'interconnexion*, qui ont souvent été négligées dans les technologies inférieures à 0.5μ, causent aujourd'hui plus de 60% du retard des circuits. Ils tendent parfois à masquer l'effet des capacités parasites dont l'effet diminue avec la diminution de la taille du transistor. Une explication plus exhaustive de ce phénomène est donnée dans [FFG, CP].

Dans les circuits les plus récents, l'effet des capacités de couplage, encore connues sous le nom de *capacités de diaphonie* [AL02], commence à changer énormément les modèles de retard. Cette capacité représente le couplage électromagnétique entre un ou plusieurs conducteurs appelés agresseurs, et un conducteur appelé victime. L'influence de ce phénomène sur le modèle de retard niveau porte, a fait le sujet de [Avo03].

La conclusion que nous pouvons dégager de ce qui précède, est que le retard, dans ses trois formes mentionnées, provient de phénomènes qui sont entièrement inhérents aux circuits et non pas aux portes. Cette constatation explique l'importance de la simulation post-layout, souvent appelée *réannotation*. A cet étape interviennent ce que nous appelons communément les calculateurs de délai *delay calculators*.

Un calculateur de délais, tel que *DelayMill* [Dat], est un programme intégré dans un outil de vérification. Son rôle est d'effectuer une bonne estimation des délais de chaque cellule *dans son circuit*. Tous en cachant leur complexité et leurs modèles de retard qui ne cessent d'évoluer avec l'évolution de la technologie, ces calculateurs présentent leurs résultats sous une forme souvent faciles à interpréter. Le *format standard de delais*, *SDF*[1] [Soc00], sert souvent à représenter une telle information. Ce format normalisé, et assez portable, a souvent permis de faire le pont entre les calculateurs de délais et les outils de simulation et de vérification, tel que ModelSim [Com] et PrimeTime [Syn99].

Le format SDF peut décrire le circuit à travers l'ensemble de ses cellules. Le comportement temporel d'une cellule est décrit à travers celui de chaque couple entrée-sortie. Si nous restons dans la logique binaire[2], le format SDF décrit séparément la montée (rising time) et la descente (falling time). Chacune est représentée par deux valeurs, le délai proprement dit et le délai limite de rejet, souvent noté *r-limit*[3]. Nous revenons sur la signification de ces délais plus tard.

2.2.2 Le modèle de retard adopté

Dans ce travail nous considérons le circuit comme un ensemble de cellules communiquantes entre elles à travers des interconnexions de résistances nulles. Par contre, toute la notion du temps est renfermée dans les cellules. Le modèle de retard d'une cellule sera défini à travers deux valeurs réelles positives, qui seront notées dans tout le reste du rapport, l pour *"lower bound"* et u pour *"upper bound"*.

- u représente la largeur minimale d'une impulsion à l'entrée pour influer, à coup sûr, la valeur logique de la sortie.
- l représente la largeur d'une impulsion à l'entrée, au-dessous de laquelle cette dernière sera complètement rejetée. C'est le *r-limit* que nous avons cité dans le SDF.

[1] Standard Delay Format

[2] Le format SDF peut représenter d'autres logiques tel que la logique d'ordre quatre(0,1,X,Z), ou d'ordre trois(0,1,X).

[3] Pulse rejection limit

Autrement dit, un changement à l'entrée sera filtré si sa durée est inférieure à l. S'il persiste au délà de u, il atteint certainement la sortie. Entre ces deux valeurs la sensibilité de la sortie à cette entrée est *incertaine*. Contrairement aux modèles *déterministes* utilisés dans la simulation SPICE, la capacité d'exprimer et d'analyser ce *non-déterminisme* temporel fera partie des caractéristiques principales de notre technique.

Dans ce travail, nous supposons que toutes les entrées d'une cellule ont la même valeur pour l et la même valeur pour u. C'est une abstraction *conservative*[4] du modèle le plus complexe qui traite différemment les différentes entrées. Elle peut être réalisée en prenant l comme le *minimum* de tous les l_i, et u comme le *maximum* de tous les u_i.

Enfin, nous définissons un circuit acyclique booléen temporisé de la manière suivante :

Définition 2.5 (circuit booléen acyclique temporisé).
Un circuit booléen acyclique temporisé est un tuple $C = (V, \rightsquigarrow, F, I)$ où :
- *$(V, \rightsquigarrow, F)$ est un circuit booléen acyclique.*
- *I est une fonction qui assigne à chaque nœud v, qui n'est pas une entrée, un retard I_v.*
  I_v est définie par un couple de réels $\{l_v, u_v\}$ tel que $0 < l_v \leq u_v < \infty$.

□

A partir de cette définition une porte logique apparaît comme une composition d'une partie fonctionnelle, représentée par une fonction f_i, et d'une partie temporelle, représentée par le retard $[l_i, u_i]$.

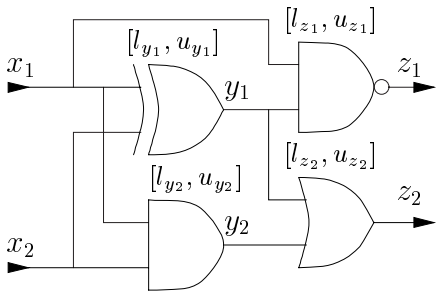

FIG. 2.3 – Circuit acyclique augmenté par la notion de retard.

2.3 Modélisation des circuits avec les automates temporisés

Après avoir introduit la notion de temps et le modèle de retard que nous considérons, nous utilisons maintenant le formalisme des automates temporisés à urgence, développé dans le paragraphe 2.1, pour modéliser le comportement d'un circuit. Nous commençons en premier lieu par introduire le modèle d'une porte logique, pour s'étendre ensuite au modèle global d'un circuit.

[4]Le modèle conservatif inclus tout comportement du vrai modèle

2.3.1 Modèle d'une porte logique

Dans ce paragraphe, nous donnons le modèle, en terme de TA, d'une porte logique, unité de base de nos circuits. Ce modèle a été introduit dans [MP95]. Pour une porte dont la fonction logique est donnée par f_i, nous pouvons faire la distinction entre le retard de la montée "↑", et le retard de descente "↓". $[l_\uparrow, u_\uparrow]$ représentera le retard de montée et $[l_\downarrow, u_\downarrow]$ celui de la descente. Dans ces conditions le TA qui décrit le comportement de cette porte est celui donné ci-dessous par la figure 2.4.

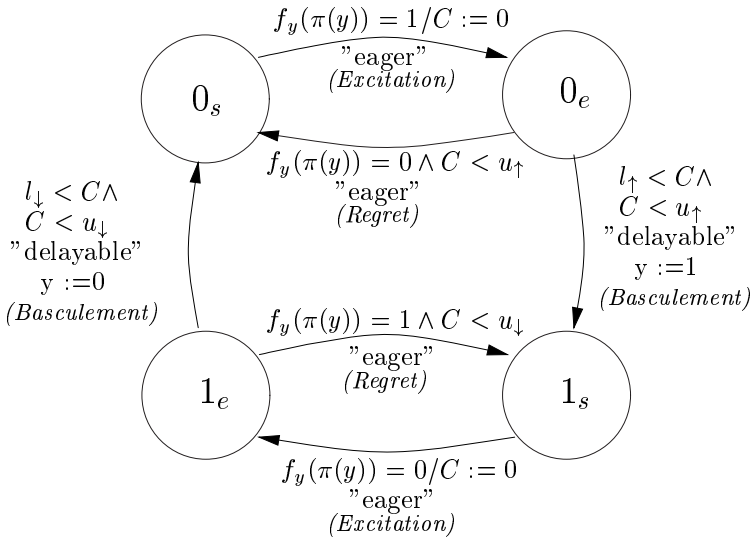

FIG. 2.4 – Automate temporisé d'une porte logique avec des délais différents à la montée et à la descente

Ainsi, on distingue quatre états de la porte. Les états 0_s et 1_s représentent l'état stable de la porte, au niveau logique 0, ou bien au niveau 1. A état stable, la sortie de la porte correspond à la valeur de sa fonction logique appliquée aux valeurs actuelles de ses entrées. Autrement dit $y = f_y(\pi(y))$, avec y représentant la sotie de la porte logique, et $\pi(y)$ ses entrées. Le passage entre ces deux états stables ne se fait pas directement, La porte doit passer par un état *excité* (0_e ou 1_e).

A un état excité, deux comportements peuvent avoir lieu :

- Jusqu'à un délai u, un retour immédiat à l'état stable d'origine est possible suite à une deuxième modification de l'entrée qui annule l'effet de la première. Cela exprime le filtrage des changements bref à l'entrée.
- Une fois la borne l est atteinte, le basculement devient possible, il sera obligatoire dès qu'on atteint la borne u. Cette idée est exprimée par le mot clé *'delayable'* expliqué au paragraphe 2.1.1 (et qui veut dire que le deadline sur la transition suit le front descendant de la garde).

Dans le reste du travail nous considérons un même retard $[l, u]$ pour la montée et la descente. Cette considération ramène ce dernier modèle d'une

porte logique à l'automate temporisé donné ci-dessous par la figure2.5. Ce TA ne contient plus que deux états qui correspondent aux états stable et excité de la porte. Une porte passe à son état excité dès que sa sortie y est tel que $y \neq f_y(\pi(y))$. A cet état, nous pouvons avoir un *regret* (la porte revient à son état stable sans changement de la valeur logique de sa sortie), ou bien un *basculement* de la valeur logique de la sortie, et la porte retrouve aussi l'état stable.

Nous soulignons ici que la valeur logique de la porte n'est plus représentée par l'état, comme dans le modèle précédent, mais elle est véhiculée par une variable interne. La différence entre ces deux modèles est plutôt syntaxique : Si on étend l'automate avec la valeur de y on obtient un automate similaire à celui de la figure 2.4.

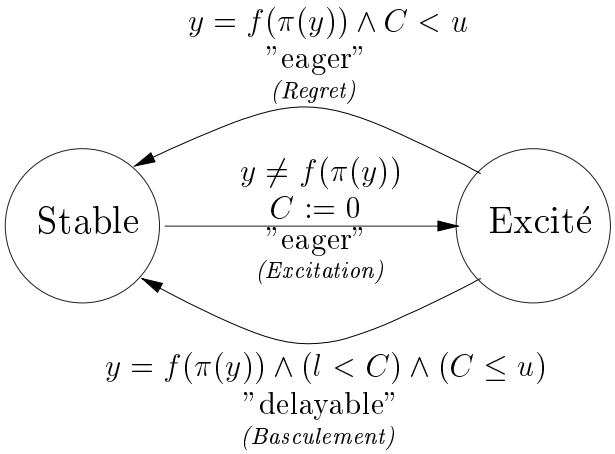

FIG. 2.5 – Modèle réduit d'une porte logique

Nous rappelons que le choix des modèles de base dans ce domaine de vérification revient toujours à un problème de *compromis entre complexité et efficacité*. Toutefois, ce choix est orthogonal avec le reste de notre méthodologie, et pourrait bien être raffiné plus tard.

2.3.2 Le graphe de simulation d'un circuit

Le comportement d'un circuit est défini comme celui des différents composants évoluants individuellement et communiquants via des canaux, qui sont les interconnexions dans notre cas. Ces composants restent toujours à l'entente de l'entrée globale de ce circuit. Cela se traduit en un système comportant autant de processus que le circuit comporte de composants, avec un processus supplémentaire qui représente la conduite de l'entrée. Nous traduisons chaque processus en un automate temporisé. Le produit de tous ces automates représentera tous les comportements possibles du circuit considéré.

Dans le paragraphe précédent nous avons introduit le modèle individuel d'une porte logique. Connaissant le comportement de l'entrée, il ne reste qu'à

réaliser le produit déjà donné par la définition 2.3 pour déduire le modèle du circuit global.

A titre d'exemple, si nous considérons le circuit, à trois portes, donné dans la figure 2.6, le produit parallèle de ses différents composants et du comportement à son entrée, génère un automate globale dont la sémantique est donnée par le graphe de la même figure. C'est ce graphe que nous appelons *"graphe de simulation"*, et qui représente tout le comportement possible du circuit global. A travers ce graphe, plusieurs propriétés critiques, peuvent être vérifiées.

A noter, qu'à part les transitions discrètes (excitation, basculement, regret), ce graphe présente un autre type de transitions, qu'on a déjà invoqué, et qui est spécifique aux TA ; ce sont les transitions qui sont étiquetées par *"time"*, et qui expriment l'écoulement du temps.

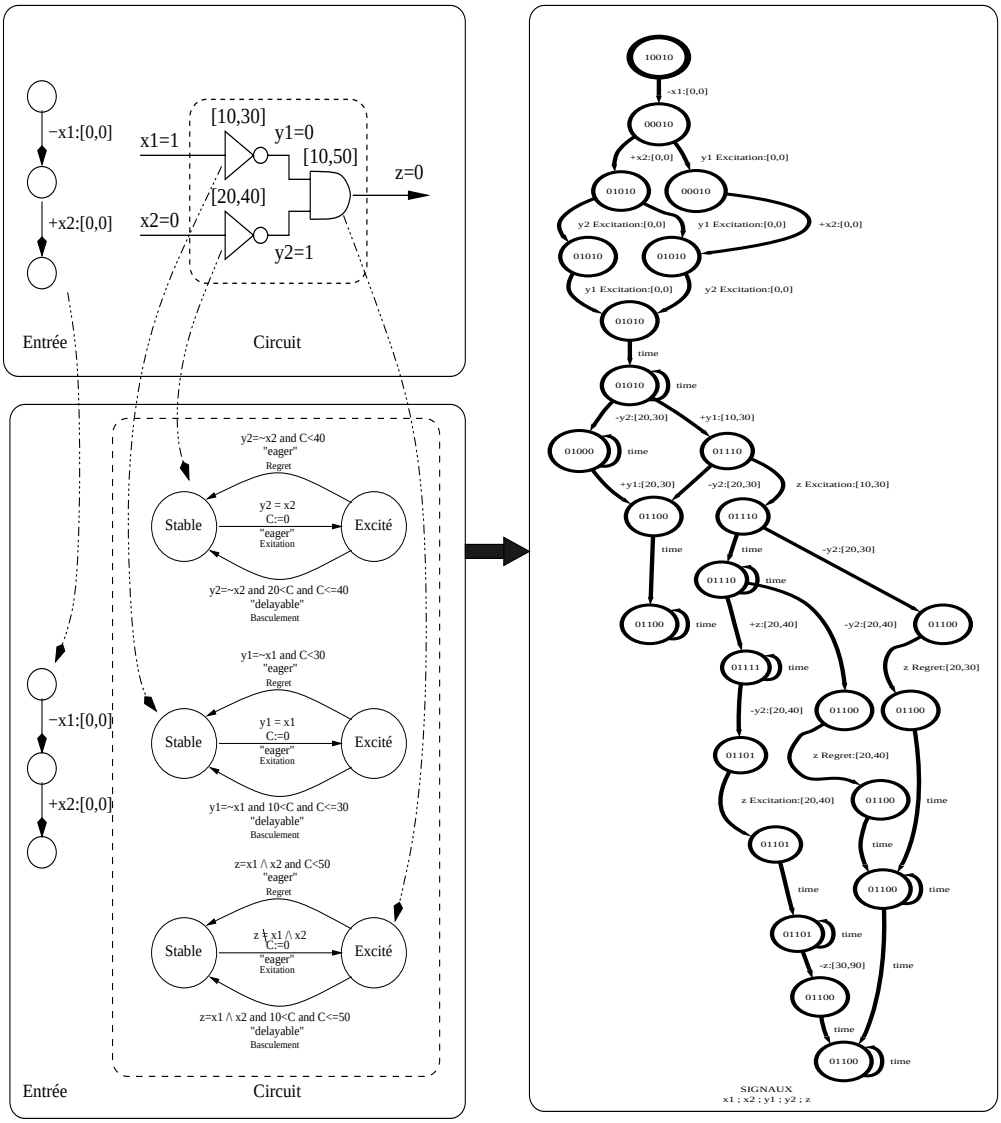

FIG. 2.6 – Composition d'automates temporisés et graphe de simulation

La composition d'automates effectuée ci-dessus devient impossible manuellement pour des exemples plus grands. Pour cette raison, nous utiliserons la boite à outils IF, dédiée à ce type de manipulations de graphes.

2.3.3 Boîte à outils IF

L'architecture globale de cette boite à outils et les connexions entre ses différents composants sont représentées dans la figure 2.7.

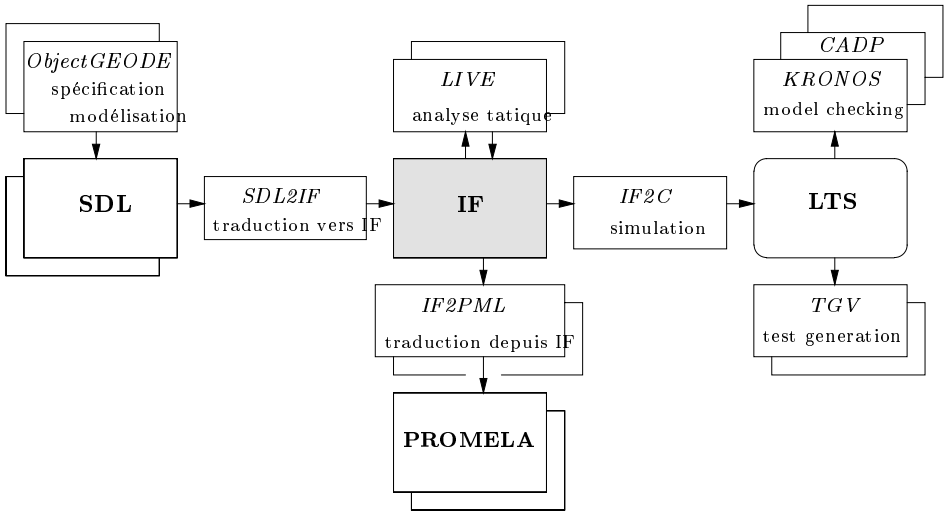

FIG. 2.7 – Environnement de validation IF

L'environnement de validation IF repose sur trois niveaux de représentation :

- *La spécification* : C'est la description initiale du système. Elle est exprimée dans un formalisme de haut niveau en utilisant, par exemple, l'un des standards internationaux. Afin d'être analysée, cette description est *automatiquement* traduite en la représentation IF correspondante.
- *Le format intermédiaire* IF : qui correspond à la représentation IF. Dans IF, un système est exprimé par un ensemble de processus représentés par des automates temporisés - *parallèles* communicants :
  - d'une manière asynchrone via des files (avec ou sans perte, bornées ou non). Plusieurs politiques d'empilement sont possibles (fifo, pile, ...).
  - d'une manière communication synchrone par rendez-vous via des portes.
  - via des variables partagées.
- *Le modèle sémantique* LTS [5] : Le modèle sémantique sous forme de LTS représente les exécutions possibles de la spécification décrite dans le programme IF.

De cette boite à outil, nous avons utilisé le composant *If.open*. Ce programme réalise une simulation exhaustive et génère le LTS correspondant à une spécification donnée en format intermédiaire IF(.if). Le LTS résultant est

[5] *Labeled Transition System* : Système à Transitions Etiquetées.

restauré dans la format d'*Aldebaran*[6](.aut).

Pour qu'il réponde mieux aux besoins de notre travail, nous avons modifié dans ce projet certains algorithmes de cet outil. Nous revenons sur cette modification ultérieurement.

2.4 Conclusion

Nous avons présenté comment nous parvenons à modéliser un circuit en automates temporisés, et le traduire en processus exprimés en format IF. Enfin, nous appliquons le programme *If.open* modifié pour généré le LTS, un graphe qui simule tous les comportements possibles du circuit. A partir de ce graphe, plusieurs propriétés logico-temporelles peuvent être vérifier.

La méthode, comme elle a été introduite à ce niveau, a pu être appliquée pour des circuits de quelques dizaines de portes. Il s'avère que pour des exemples plus grands cette technique n'est plus adaptée. Le chapitre suivant sert à mieux exposer cette limitation et essaye de lui proposer une solution.

[6]Un package faisant partie de la boite à outil CADP.

Chapitre 3

Limitations et solutions

Comme nous l'avons dévoilé depuis le début, tous les méthodes de vérification basées sur les FSM sont limitées par un problème commun qui est *l'explosion d'état*. Ce problème est plus grave dans notre cas, puisque nous analysons le comportement logique et temporel du circuit à son niveau porte.

Cet handicape sera examiné de plus près dans la première partie de ce chapitre. Aprés, nous nous consacrerons à expliquer l'idée proposée pour dépasser cet obstacle.

3.1 Performances de la méthode

Pour mieux exprimer la complexité de notre méthode, nous proposons dans cette section une première expérience qui consiste à incrémenter la taille du circuit traité, et voir quel effet cela peut donner.

Nous allons considérer le circuit de la figure 2.3 à l'état stable $\{x_1 = 0, x_2 = 0, y_1 = 0, y_2 = 0, z_1 = 1$ et $z_2 = 0\}$. Ce circuit présente autant d'entrées que de sorties, ce qui permet de le doubler, le tripler, et ainsi de suite. A chaque fois, nous examinons le nombre d'état du graphe, ainsi que le temps pris pour sa génération. Les résultats trouvés sont résumés dans le tableau ci-dessous.

Portes	Etats	Temps($10^{-2}s$)
4	126	72
8	1074	110
12	8172	897
16	137548	26751
20	$+3,6.10^6$	$+4heures$

TAB. 3.1 – Évolution de la taille du graphe et du temps de traitement en fonction de la taille du circuit traité

Nous constatons qu'à partir de 20 portes le calcul a pris un temps énorme sans aboutir à sa fin. Le graphe partiellement généré a dépassé en taille la

mémoire de notre machine qui est de $512Mo$.

Cette borne qui dépend principalement des performances de la machine utilisée, peut donner une idée de la complexité de la méthode. Mais la conclusion la plus importante que nous pouvons tirer de cette expérience est donnée par la figure 3.1.

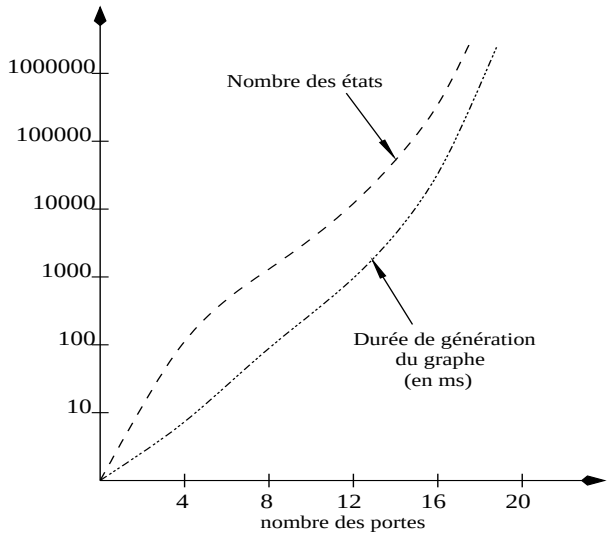

FIG. 3.1 – Explosion de la taille du graphe et du temps de sa génération en fonction du nombre de portes par circuit.

À travers la courbe ci-dessus, qui n'est autre qu'une représentation plus lisible du tableau précédent, nous constatons que la taille du graphe, aussi bien que le temps de sa génération, évoluent d'une manière exponentielle en fonction du nombre de portes du circuit. C'est cela qui représente la vraie défaillance de cette technique. Car même si le changement de machine permet de traiter des circuits plus grands, le problème finira par échapper à notre contrôle. Le temps de traitement peut atteindre des limites inacceptables, et la taille de notre graphe finira par dépasser la mémoire, peu importe la taille de cette dernière. Il faut souligner que ce phénomène, bien connu dans la vérification fonctionnelle (où il est traité par des méthodes symboliques) est beaucoup aggravé dans les TA à cause de la nécessité de représenter et de stocker l'ensemble des valeurs d'horloges.

Ce problème a été prévu dès le début, et ces résultats ne représentent pas, pour nous, cette grande surprise. Nous avons accepté de prendre ce défi, et nous commencerons dès le prochain paragraphe à exposer nos solutions.

3.2 Solutions

L'efficacité des méthodes d'exploration d'états et du model-checking dépendent énormément de la taille du graphe d'atteignabilité. Ce dernier atteint rapidement des valeurs significatives avec la complexité du système traité, et exige

ainsi un temps de traitement et une mémoire plus grande. Ce problème, connu sous le nom d'*explosion d'états*, a fait le sujet de plusieurs recherche, mais une solution universelle n'existe toujours pas. Tout dépend du type du problème traité.

Nous nous intéressons dans ce projet à des circuits *acycliques*, une caractéristique que nous allons exploiter pour pouvoir éviter l'analyse du circuit en entier. Nous partitionnons ce dernier en des fragments que nous traitons un à un tout en rejetant, chaque fois, l'information inutile. Pour la mieux l'expliquer nous allons illustrer cette technique à travers un exemple.

3.2.1 Exemple introductif

Dans ce paragraphe nous allons considérer le circuit de la figure 3.2.

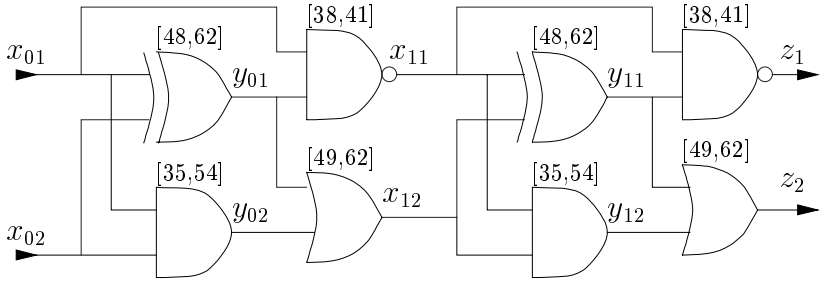

FIG. 3.2 – Exemple de circuit acyclique.

Pour vérifier une propriété de ce circuit, nous commençons par générer son graphe de simulation en combinant les modèles de ses différents composants avec le comportement à son entrée. Dans la figure 3.3, nous avons considéré un exemple d'entrée pour générer *"Graphe A"* qui contient 1047 états et 1498 transitions. Ce graphe renferme tous les comportements possibles de tous les signaux du circuit(Excitation, Regret, Basculement et Ecoulement du temps). Cette information, très exhaustive, peut ne pas être en totalité utile, car souvent les propriétés à vérifier dans un circuit n'impliquent qu'un ensemble réduit de signaux. A titre d'exemple, dans le reste de cette partie nous nous intéressons à l'analyse de la sortie de ce circuit, c'est à dire le comportement mutuel de z_1 et z_2. Cette information peut-être extraite à partir du comportement global de tout le circuit représenté à travers son graphe de simulation(*"Graphe A"*), un graphe dont la taille peut former le majeur handicape de cette méthode. Pour éviter cet obstacle, notre idée consiste à *exploiter le caractère acyclique* de nos circuits. En effet, au lieu d'analyser ce circuit d'un seul coup, nous proposons de le partitionner en deux fragment, comme le montre la figure 3.3. Nous commençons par traiter la première partition en la combinant avec la même entrée précédente. Le graphe généré (voir *"Graphe B"*) sera minimisé pour en extraire uniquement le comportement de $\{x_{11}, x_{12}\}$. Maintenant, connaissant la conduite de $\{x_{11}, x_{12}\}$, rien ne nous empêche de générer le graphe de simulation de la deuxième partition (voir *"Graphe C"*). Ce dernier graphe sera minimisé de la même manière, pour ne laisser enfin que le comportement de $\{z_1, z_2\}$. Le

graphe final de six états peu maintenant servir à vérifier les propriétés voulues. Ce même graphe final a pu être déduit, comme le montre la figure 3.3, du *"Graphe A"*, du circuit global.

Autrement dit, l'idée consiste à arriver au résultat final en passant par *"Graphe B"* et *"Graphe C"* au lieu de *"Graphe A"* qui est de taille importante, et qui exige des performances beaucoup plus importantes pour sa génération et son analyse.

Avant de passer, je voudrais bien comparer *"Graphe A"* avec *"Graphe B"* et *"Graphe C"* pour mieux conclure sur le gain réalisé à travers cette méthode d'abstraction. La première remarque évidente est que le nombre d'états et de transitions a été presque divisé par dix. Cela ne reflète pas la totalité de l'astuce, car un état du *"Graphe A"* ne présente pas la même taille et la même complexité que celui de *"Graphe B"* ou de *"Graphe C"*. En effet, l'information principale encapsulée dans un état est celle concernant l'écoulement du temps. Cette information se présente comme une matrice d'ordre n. Où n représente le nombre d'orloge du système, qui est dans notre cas, le nombre de portes. Ainsi, la taille d'un état est une puissance de deux de la taille du circuit traité. Alors, pour notre exemple : *Taille de "Graphe A"* $\approx (10+2^2)\times$ *Taille de "Graphe B"*. (de même pour *"Graphe C"*)

3.2.2 Généralisation

Les résultats qui ont été constatés nous ont encouragés à passer vers des circuits plus importants en tailles. Un grand circuit ne sera pas traité d'un seul coup. Il sera tout d'abord découpé en une succession de sous-circuits. Nous les déroulons ensuite l'un après l'autre dans l'ordre défini par l'acyclicité du circuit d'origine. A chaque fois, comme on l'a vu, toute information inutile sera rejetée.

Les contraintes qui règlent notre partitionnement sont surtout relatives aux performances de la machine qui sert à ce traitement. Plus ces performances sont faibles et plus on est obligé à réduire les taille des partitions.

Le premier algorithme de partitionnement que nous avons implémenté dans ce projet consiste à progresser en profondeur depuis l'entrée du circuit. Chaque fois qu'on atteint un nombre n de portes, donné par l'utilisateur, nous commençons une nouvelle partition. Nous verrons plus loin que cet algorithme, qui est le plus intuitif, ne représente pas la meilleure solution.

3.3 Réduction de graphe

Par soucie de clarté, nous avons, à plusieurs reprises, invoqué le concept de réduction de graphes sans lui donner d'explication. Ce paragraphe sert à mieux exposer cette étape qui ne manque pas d'importance.

Si nous revenons à l'exemple précédent, le comportement de la deuxième partition n'est sensible qu'à son entrée, plus précisément, à la conduite de $\{x_{11}, x_{12}\}$. Cette information existe dans le graphe de simulation de la partition précédente (voir *"Graphe B"* de la figure 3.3). Mais ce graphe, que l'on

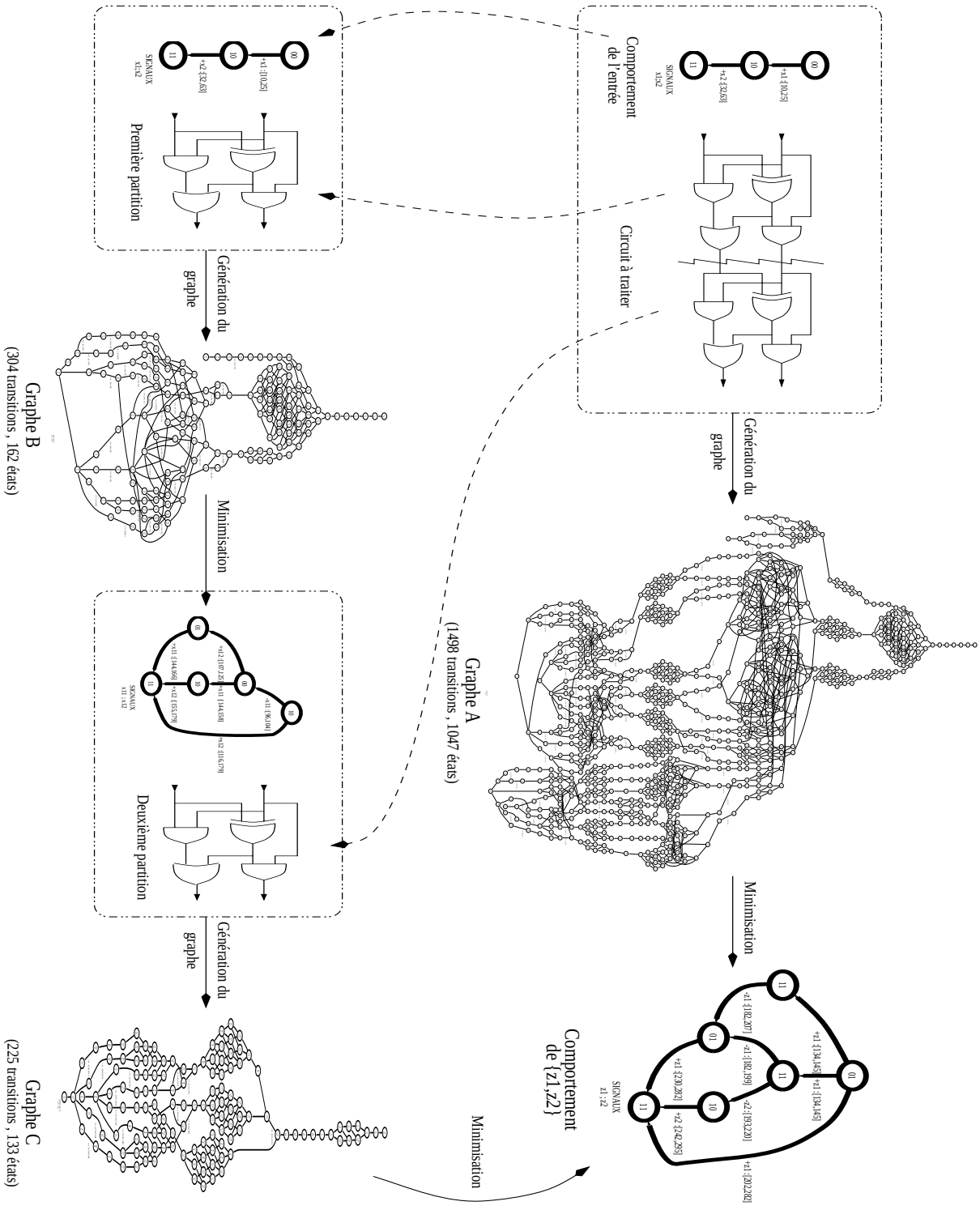

FIG. 3.3 – Résumée de la technique d'abstraction utilisée dans le cas des circuits acycliques

notera par la suite G_{init}, comporte aussi une grande quantité d'information inutile pour notre traitement. Pour cette raison nous avons décidé de construire à partir de G_{init} un autre graphe G_{min}, qui a deux principales caractéristiques :

- G_{min} est beaucoup plus petit que G_{init}.
- G_{min} est une *sur-approximation*[1] de la projection de G_{init} sur les signaux d'entréee de la prochaine partition (x_{11} et x_{12}).

De cette façon, si nous remplaçons, G_{init} par G_{min}, comme modèle pour la première partition, nous pouvons garantir une *sur-approximation* du modèle de la prochaine partition.

La première étape dans la procédure de construction G_{min} est de ramener G_{init} à un automate géré par une unique horloge. C'est une horloge absolue, que nous ajoutons au système, et qu'on appellera dorénavant $\mathcal{H}$. Ce dernier automate aura la même structure de transition que G_{init}. Chaque transition $(s, v) \rightarrow (s', v')$ inclura dans son étiquette un intervalle $[t_1, t_2]$ obtenu à partir de G_{init} par projection de v sur $\mathcal{H}$. Il est clair qu'une transition qui prend cette étiquette n'est franchissable qu'entre t_1 et t_2 du temps absolu.

A noter que cette abstraction peut ajouter un comportement qui est impossible dans l'automate d'origine. A titre d'exemple, dans l'automate (a) de la figure 3.4, la première transition peut avoir lieu dans l'intervalle de temps $[l_1, u_1]$. Elle sera suivie par la deuxième transition entre l_2 et u_2, comptée à partir de cet instant. En appliquant la procédure décrite ci-dessus, nous obtenons l'automate de la figure 3.4-(b), où la deuxième transition peut être effectuée à tout moment entre $[l_1 + l_2, u_1 + u_2]$, indépendamment du temps de franchissement de la première. La figure 3.5 représente l'automate sur-approximé, à

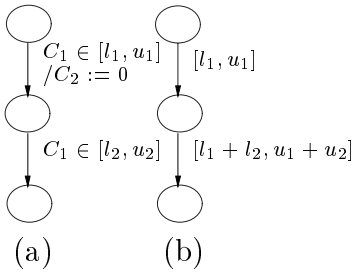

FIG. 3.4 – La projection sur le temps absolu génère une sur-approximation.

horloge unique, obtenu pour la première partition du circuit exemple.

Toutes les transitions non étiquetées dans ce dernier automate représentent les événements internes à cette partition de circuit. Ces événements sont invisibles de l'extérieur, et ne présentent aucun intérêt pour le reste de notre traitement. Pour cette raison, la deuxième étape de notre technique d'abstraction consiste à appliquer un algorithme de minimisation qui émerge les états indistingtifs par rapport aux transitions observables. Plus formellement, nous considérons la relation de congruence $\sim$ définie sur les nœuds du graphe étiqueté par :

[1] Un modèle A *sur-approxime* un deuxième modèle B, si tout comportement décrit dans B existe dans A

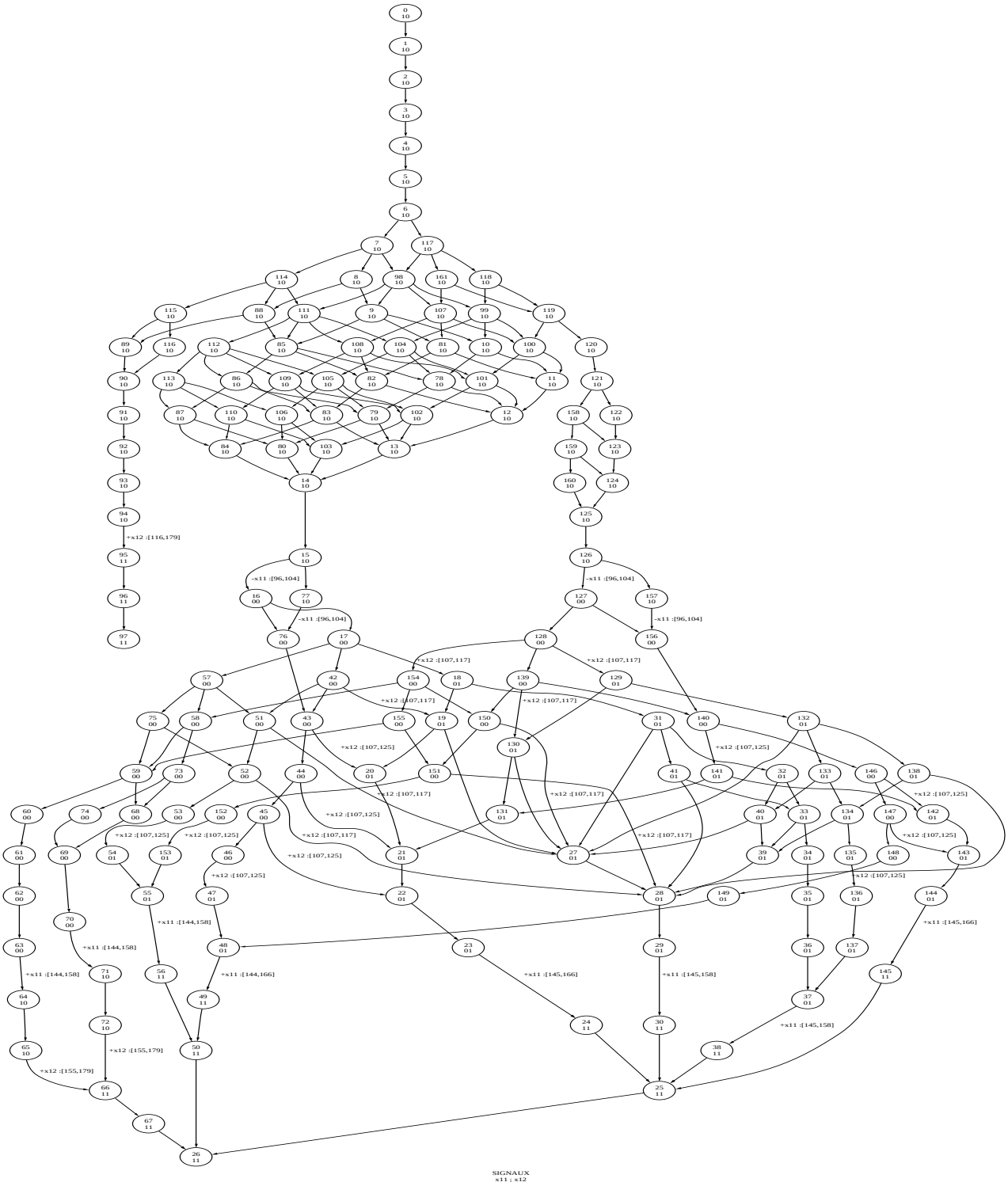

FIG. 3.5 – Automate déduit du graphe d'atteignabilité de la deuxième partition du circuit exemple, à travers une projection sur une horloge absolue. Seulement les valeurs des signaux x_{11} et x_{12} qui sont visibles pour la deuxième partition, restent dans les états et seulement les gardes des transitions qui changent leurs valeurs qui apparaissent.

$$s_1 \sim s_2 \text{ ssi } \forall \delta, I \; s_1 \xrightarrow{\tau^*.(\delta,I)} s_1' \Rightarrow \; (\exists s_2' \text{ tel que } s_2 \xrightarrow{\tau^*.(\delta,I)} s_2' \text{ et } s_1' \sim s_2')$$

Où (δ, I) est une transition avec intervalle, et τ^* est une séquence arbitraire de transition non observable. Cette relation est la *"safety bisimulation"* de [BFG+91] Cette deuxième étape de minimisation nous ramène vers l'automate (A) de la figure 3.6

Notons bien que la relation ci-dessus voit les labels des transitions d'une manière purement syntaxique. Ainsi "$+x_{11} : [145, 158]$", "$+x_{11} : [145, 166]$", "$+x_{11} : [144, 158]$", "$+x_{11} : [144, 166]$" sont vus comme des événements totalement indépendants. Pour cette raison les états colorés de l'automate (A) ne sont pas fusionnés. Pour obtenir une abstraction plus agressive, nous avons défini une nouvelle relation d'équivalence $\sim'$ qui ignore les différences entre les intervalles :

$$s_1 \sim' s_2 \text{ ssi } \forall \delta, I \; s_1 \xrightarrow{\tau^*.(\delta,I)} s_1' \Rightarrow \; (\exists s_2', I' \text{ tel que } s_2 \xrightarrow{\tau^*.(\delta,I')} s_2' \text{ et } s_1' \sim' s_2')$$

Les états de l'automate minimisé représentent les classes d'équivalences de $\sim'$. Une transition entre deux classes est étiquetée par $(\delta, \bar{I})$, où $\bar{I}$ est l'union convexe de tout les intervalles I_i des transitions étiquetées par δ, I_i et reliant ces deux dernières classes (voir figure 3.7). Le résultat de la minimisation par rapport à $\sim'$ apparaît sur la figure 3.6-(B), et nous pouvons voir que le comportement de x_{11} et x_{12} est bien sur-approximé.

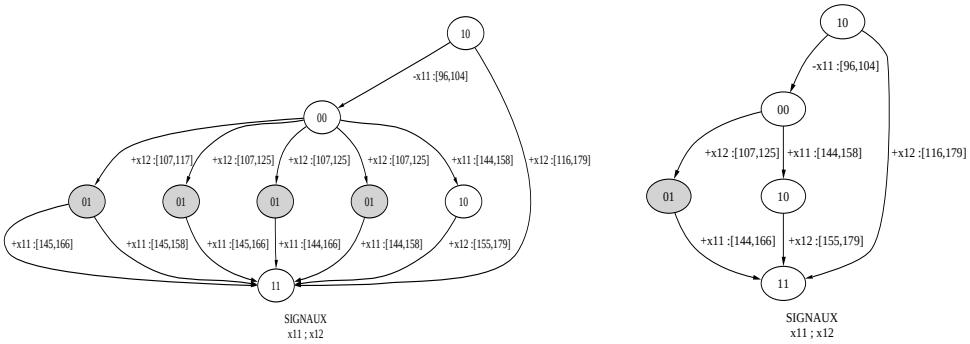

FIG. 3.6 – (A) Minimisation par rapport aux transitions observables ; (B) Minimisation en tenant compte de la sémantique des intervalles sur les transitions

Tout le processus décrit ci-dessus est implémenté. Nous revenons sur cette partie pratique dans le prochain chapitre.

3.4 Apport de la nouvelle technique d'abstraction

Pour mieux évaluer les performances de cette nouvelle technique d'abstraction nous avons refait l'expérience du paragraphe 3.1. Les nouveaux résultats

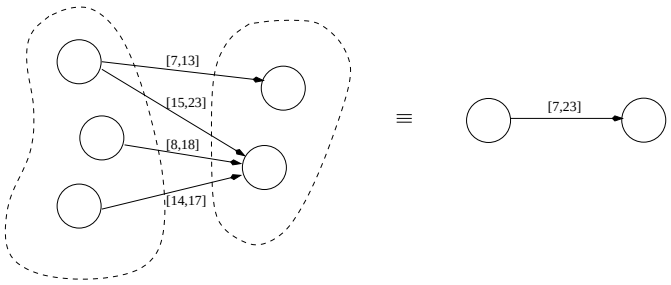

FIG. 3.7 – Minimisation en tenant compte de la sémantique des labels sur les transitions

obtenus sont représentés par le tableau 3.2 ci-dessous. Dans chaque ligne de ce tableau nous donnons, pour un nombre de porte donné, le nombre d'états du graphe généré, le nombre d'états du graphe déduit par minimisation, et le temps nécessaire pour le traitement.

Portes	Global	Minimisé	Temps
4	304	5	0.58 s
8	133	6	1.28 s
16	4080	11	3.24 s
20	2690	16	5.13 s
24	21498	30	11.41 s
28	50543	39	29.47 s
32	73502	48	39.57 s
36	95619	57	1 min 53.68 s
40	117736	66	3 min 12.01 s
44	139853	75	5 min 07.23 s
48	161970	84	7 min 31.53 s
52	184087	93	10 min 04.82 s
56	206204	102	14 min 41.57 s
60	228321	111	20 min 38.43 s
64	250438	120	28 min 14.93 s
68	272555	129	37 min 45.58 s
72	294672	138	49 min 35.27 s
76	316789	147	1 h 04 min 03.58 s
80	338906	156	1 h 21 min 47.88 s
84	361023	165	1 h 42 min 58.68 s
88	383140	174	1 h 58 min 55.42 s
92	405257	183	2 h 30 min 30.8 s
96	?	?	+10 h 11 min 6.74 s

TAB. 3.2 – Les nouvelles limites à travers l'utilisation de l'abstraction

Une première constatation est qu'on a pu pousser notre limite de 20 portes à presque 100 portes. Mais le vrais apport de cette technique d'abstraction est mieux visible dans le graphe de la figure 3.8.

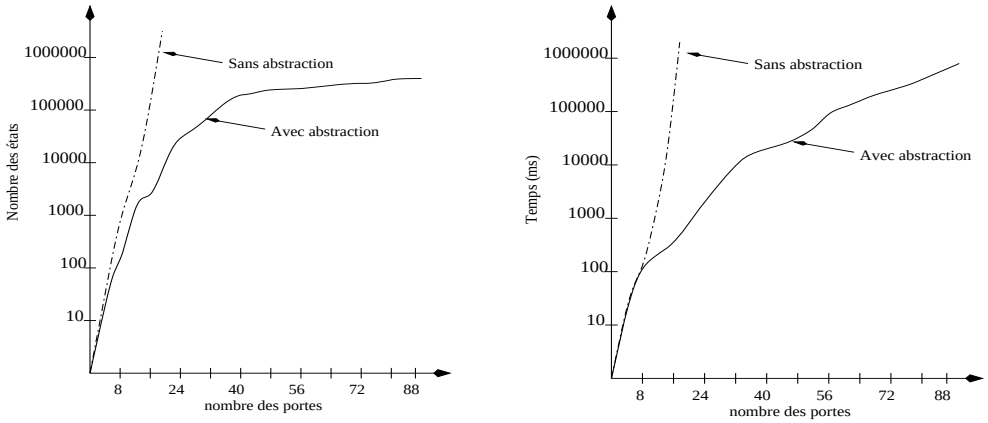

FIG. 3.8 – Amélioration de la complexité de la méthode à travers la technique d'abstraction.

Nous y remarquons que cette abstraction tend à *linéariser* l'évolution de la taille du graphe généré, avec le nombre de portes traitées. Cela représente un exploit contre le problème de l'explosion d'états.

Aussi, le temps global de traitement s'est beaucoup amélioré. Cela malgré que notre nouvelle méthode est une articulation d'un nombre d'étapes qui peut beaucoup augmenter avec la finesse du partitionnement et la profondeur du circuit traité.

3.5 Limites de la nouvelle technique d'abstraction

Regardons de plus près les résultats données par le tableau 3.2, et plus précisément la troisième colonne. Nous constatons qu'à chaque étape, la partition de circuit traitée est combinée avec une entrée plus complexe que la partition qui la précède. On a commencé par une première partition dont le modèle de l'entrée contient 3 états (entrée globale du circuit) pour arriver au niveau de la 23^{eme} partition à une entrée modélisée par un automate à 183 états.

Pour maîtriser ce problème, nous somme actuellement en train d'explorer de nouveaux algorithmes de partitionnment plus raffinés que celui présenté jusqu'à maintenant et implémenté dans notre travail. Cette direction nous a été dictée par d'autres exemples plus grands qui présentent mieux le problème que l'exemple de quatre portes utilisé dans ce rapport (voir figure 3.2).

La méthode de partitionnement qui nous a attirés le plus, consiste à isoler à l'intérieur des partitions les cellules les plus inter-connectées entre elles. Cela à deux avantages :

– Le premier est que plus on a de relations entre les cellules d'une même

partition plus le graphe généré est petit (la figure 3.9 représente cette idée).

– Le deuxième intérêt de cette méthode est de ne laisser entre les différentes partitions que le minimum de liaison, ce qui rend notre technique de réduction de graphe très efficace.

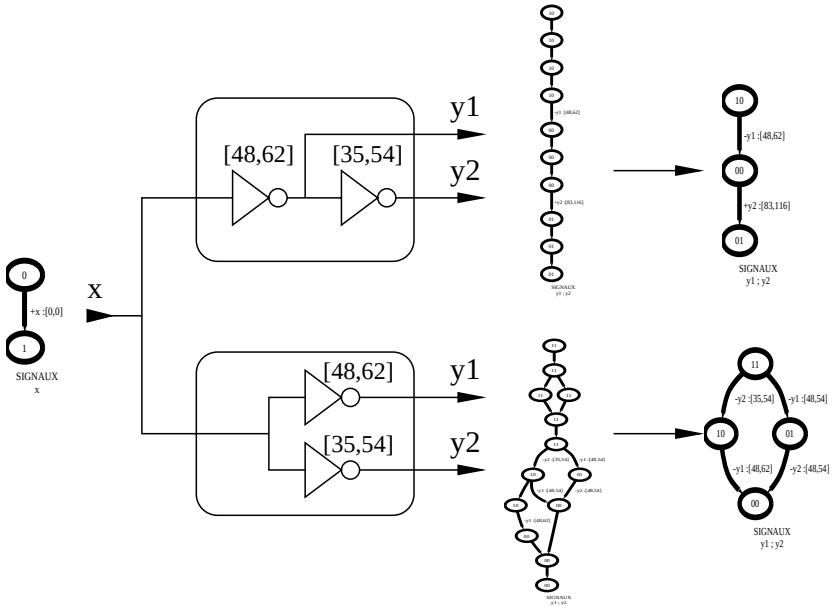

FIG. 3.9 – Effet des interconnexions entre les cellules d'une même partition, sur la taille du graphe généré

Par faute de temps, ces algorithmes n'ont pas encore été implémentés. Pour cette raison, nous n'allons pas s'attarder plus sur cette partie que nous gardons parmi nos perspectives.

3.6 Conclusion

A part l'étude théorique effectuée, nous avons développé dans ce projet, un système d'outils qui sert à dérouler d'une manière automatique tout le processus de vérification proposé. Ce travail pratique nous a beaucoup aidé à avancer dans ce travail, puisqu'il a permis l'analyse de plusieurs exemples de circuits, et d'en déduire rapidement les lignes directrices de nos perspectives.

Le prochain chapitre servira à résumer les différentes étapes du processus de vérifications, et les différents composants logiciel que nous avons mis en place dans ce projet pour concrétiser nos idées.

Chapitre 4

Automatisation du processus de vérificaion

Ce dernier chapitre sert à résumer le processus de vérification proposé dans le présent travail. Il permettra en même temps de présenter tout un effort réalisé sur le plan pratique pour automatiser tout le cycle d'analyse.

4.1 Vue externe du système

Comme il est représenté par la figure 4.1, vu de l'extérieur, le système peut avoir jusqu'à cinq entrées :

1. Le fichier qui contient la description (fonctionnelle et temporelle) du circuit.
2. Un entier naturel qui indique la taille limite, en terme de nombre de porte, des sous-circuits obtenus aprés l'étape de partitionnement.
3. Un fichier qui décrit la conduite de l'entrée global du circuit.
4. Un fichier qui impose un état initial au circuit. A titre d'exemple, dans le paragraphe 3.1 nous avons effectué le traitement par rapport à l'état initial de l'entrée $\{x_1 = 0, x_2 = 0\}$. Toutefois, cette information est optionelle, son absence implique un traitement paralèl de tous les états initiaux stables possibles.
5. Un fichier qui donne la liste des signaux dont dépend la propriété du circuit que nous voulons arriver à vérifier. Cette entrée est optionnelle. Par défaut l'analyse sera réalisée sur la sortie globale du circuit.

Le résultat généré à partir de ce système se présente comme un graphe réduit du comportement des signaux à superviser, donné par la cinqième entrée décrite ci-dessus. A partir de ce graphe, toute propriété concernant ces dernier sigaux est vérifiable.

Il est à noter dans cette partie que le fichier décrivant le comportement du circuit est un fichier que nous avons conçu pour mieux décrire le circuit dans le cadre de notre modèle. Comme le montre la figure 4.2, ce fichier donne en

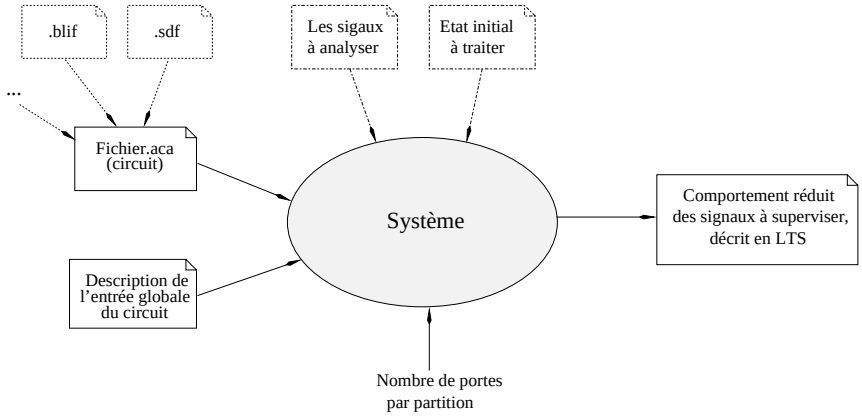

FIG. 4.1 – Vu externe du système conçu

premier lieu la liste des entrées et des sorties du circuit concerné. Il décrit ensuit l'ensemble des cellules de ce dernier. Il fourni pour chacune sa fonction logique, et son modèle temporel donné par les deux bornes l et u précédemment définies. Ce fichier prendra dans le reste du rapport l'extension *".aca"*

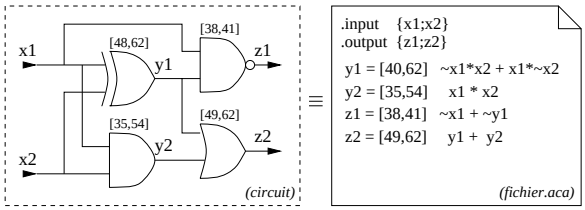

FIG. 4.2 – Le format du fichier *aca*.

Tout circuit décrit en un autre format, SDF, $BLIF$,etc., doit passer par ce dernier format pour être traiter par notre outil.

4.2 Le processus de vérification

Etant donné que la vérification concerne toujours un ensemble fini de signaux, la première étape à faire est de rejeter du circuit global toute la partie inutile. Il s'agit d'extraire *le cône d'influence des signaux à analyser* (voir 4.3), qui constitue le vrai circuit à traiter pour le reste du processus.

Dans un deuxième temps, nous partitionnons le circuit extrait lors de l'étape précédente tout en tenant compte des performances de la machine sur laquelle se déroule le test. Plus ces performances sont faibles, et plus on est amené à diminuer la taille des partitions. Comme nous l'avons vu au paragraphe 3.3, au niveau de chaque partition nous effectuons une réduction qui introduit une sur-approximation du modèle. Pour cette raison nous avons intérêt à diminuer le nombre des partitions en augmentant leurs tailles. Cela à chaque fois que les

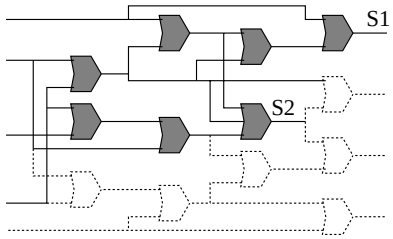

FIG. 4.3 – Extraction du cône d'influence

performances de la machine de test le permette.

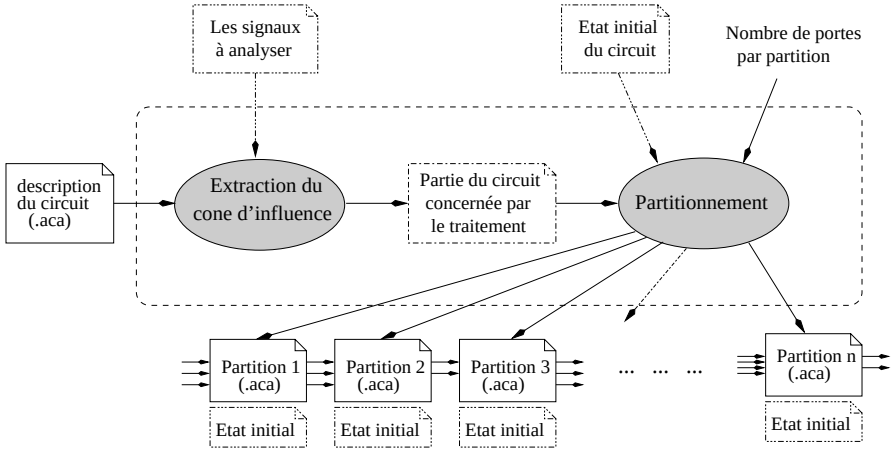

FIG. 4.4 – Extraction et patitionnement du circuit

Maintenant, après avoir divisé le circuit en une succession de fragments, nous allons dérouler le cycle d'analyse qui commence par la première partition et se déroule jusqu'à atteindre la dernière. Au niveau de chaque partition le traitement effectué, comme il est décrit par la figure 4.5 comporte les étapes suivantes :

1. **Génération du fichier IF**
   Cette étape consiste à traduire en premier lieu la définition du sous-circuit en son modèle équivalent décrit en terme de processus IF. Cette information sera complétée en une deuxième phase par le processus qui décrit le comportement de l'entrée de cette partition. Le fichier "global.if" de la figure 4.5 représente le résultat de cette étape.
2. **Génération du graphe de simulation**
   A cette étape nous générons à partir de la description IF du sous-circuit son graphe d'atteignabilité. Cette génération sera faite à l'aide de l'outil "if.open" que nous avons modifié au cours de ce projet pour l'adapter à notre travail. A l'origine, l'outil représente l'information du graphe généré sur deux parties. La première concerne uniquement les transitions, et la

deuxième concerne les états du graphe. Cette dernière comporte toute l'information temporelle sous forme de matrices d'horloges souvent de très grandes tailles. Pour cette raison nous avons ajouté au système une horloge absolue sur laquelle nous avons effectué la projection décrite dans le paragraphe 3.3. Les intervalles de temps qui restent de cette projection sont ensuite introduits dans les labels des transitions qui seront décrits par le fichier "Global.aut" généré en sortie.

3. **Minimisation**
   Cette étape sert à réaliser la réduction agressive introduite par le paragraphe 3.3. Cela nous amène au fichier "Minimised.aut" de la figure 4.5. Cela représente la fin de notre processus si on est sur la dernière partition. Sinon ce fichier représentera le comportement à l'entrée de la prochaine partition.
4. **Passage du format *aut* d'Aldebaran au format intermédière de IF**
   Ce passage est nécessaire pour pouvoir réaliser la concaténation entre deux partitions successives. En effet, comme le montre la figure 4.5, l'entrée de chaque partition (différente de la première) provient de la partition qui la précède. Cette dernière produit cette information dans le format *aut* d'Aldebaran. Par contre cette même information est consommée par la partition suivante en tant que *processus IF*

A part son aspect théorique, ce projet comporte aussi une grande partie d'implémentation. En effet, dans les schémas 4.5 et 4.4, tous les nœuds grisés représentent des composants logiciels développés au cours de ce travail.

Le composant *"aut2ps"* qui figure dans le schémas 4.5 n'a pas été introduit dans ce qui précède parce qu'il n'est pas en relation avec le processus d'analyse. En effet les fichiers ".aut" décrivent les automates d'une manière pas humainement lisible. L'outil *"aut2ps"* sert à ramener cette description vers une représentation schématique plus claire et facile à interpréter. Les images des graphes données dans ce rapport, telque celui de la figure 3.5, représentent un bon exemple des résultats donnés par cet outil.

4.3 Conclusion

Dans ce chapitre nous avons reprsenté les différentes étapes du processus de vérification, et des différents composants implémentés pour cet effet, à savoir :

- L'extraction du cône d'influence de l'ensemble des signaux à analyser.
- Le partitionnement du circuit.
- La traduction du format *".aca"* au format *IF*.
- La traduction du format *".aut"* au format *IF*.
- La projection du graphe sur une horloge absolue (Modification introduite à l'outil *"if open"*).
- La minimisation de graphe basée sur la prise en considération des intervalles de délai ajoutés aux transitions du graphe.
- La visualisation du graphe initiallement décrit dans le format *".aut"*.

Toute la complexité du processus de vérification décrite dans ce rapport est encapsulée par un script perl. C'est ce dernier qui fait, en interne, appel à tous les composants du système qui restent transparents pour l'utilisateur.

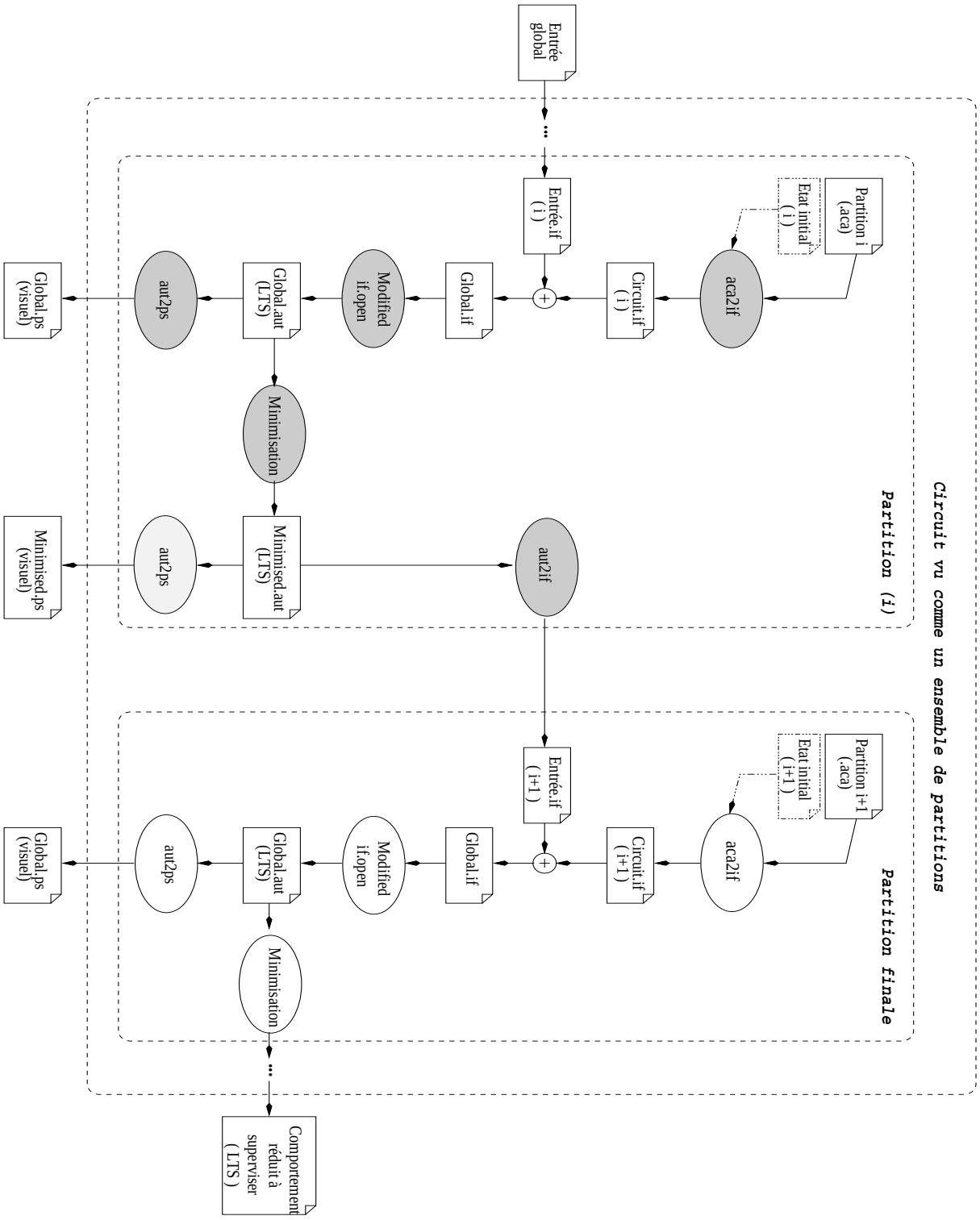

FIG. 4.5 – Processus de vérification : traitement de deux partitions consécutives

Chapitre 5

Conclusion

Ce projet à englobé trois grandes parties. La première est une étude du cadre formel du travail, à savoir les automates temporisés et la modélisation des circuits numériques à travers ce formalisme. Dans la deuxième partie, nous avons, premièrement, mis le point sur la principale limitation de cette méthode, qui est le phénomène d'explosion d'état, ensuite nous avons proposer une solution pour détourner ce problème. La troisième partie de ce projet représente une contribution sur le plan pratique. Elle a servi à concrétiser nos idées à travers l'implémentation d'une plate-forme de test qui automatise tous le processus d'analyse proposé.

La méthode de vérification exposée dans ce travail traite les circuits acycliques à leurs niveaux portes. Elle permet une analyse du comportement logiquo-temporel avec une bonne maîtrise de l'indéterminisme, qui est une caractéristique réelle des circuits.

A cette étape d'avancement, nous ne prétendons pas résoudre totalement le problème d'explosion d'état. Ce projet de DEA a juste avancé quelques pas dans ce domaine de recherche dont nous croyons beaucoup à son avenir.

Parmi les idées qui ne sont pas encore mises au point et implémentées, et qui ne figurent pas dans ce rapport :

- La modification de la philosophie de partitionnement. Nous pensons que cela va beaucoup améliorer nos résultats.
- Aussi, actuellement nous somme en train de travailler sur les algorithmes de IF. Ces algorithmes ont été conçus pour opérer sur tout type de graphe. Or dans notre cas les graphes ont une caractéristiques très intéressante : l'*acyclicité*. Cette caractéristique peut être exploiter pour libérer au fur et à mesure la mémoire, en rejetant des états qui ont été traités. De cette manière, nous espérons analyser des circuits de tailles plus grandes, sur des machines de performances plus limitées.
- La modification des modèles de portes pour pouvoir traiter un circuit comme système ouvert (réactif). Cela permet de considérer le circuit sur plusieurs niveaux d'abstractions, et bénéficier ainsi du caractère hiérarchique de ce dernier.

Bibliographie

[AD90] R. Alur and D. Dill. Automata for modeling real-time system. *Lecture Notes in Computer Scince, pages 332-335*, 1990.

[AK95] D.P. Appenzeller and A. Kuelhmann. Formal verification of powerpc microprocessor. *Porceeding of the IEEE International Conference on Computer Design(ICCD'95), Pages 79-84*, Oct 1995.

[AL02] G. Avot and M.M. Louerat. Influence et prise en compte des capacités de diaphonie . *Laboratoire Paris VI.*, 2002.

[Avo03] G. Avot. *Analyse temporelle des circuits intègrés digitaux CMOS, pour les technologies profondément submicroniques.* PhD thesis, Université Paris 6, février 2003.

[Bar98] M. Barjaktarovic. The state-of-the-art in formal methods. *WetStone Technologies, Inc.*, 1998.

[Bar00] L. Barakatain. *A Practical Model Checking Approach Using FormalCheck.* PhD thesis, Concordia University, Departement of Electrical and Computer Engineering, 2000.

[BFG+91] A. Bouajjani, J.C. Fernandez, S. Graf, C. Rodriguez, and J. Sifakis. Safty for branching time semantics,. *Proc, ICALP'91, LNCS 510*, Springer 1991.

[Bla96] D.C. Black. A first-time user of synopsys ' behavioral compiler describes the development of a high-performance asic. *EEdesign*, Juin, 1996.

[Bor98] S. Bornot. *De la Composition des Systèmes Temporisés.* PhD thesis, docteur de l'iniversité Joseph Fourier,, décembre 1998.

[Bor00] D. Borrione. Utilisation des méthodes formelles pour la vérification des systèmes intégrés digitaux. *Approches Formelles dans l'Assistance au Développement de Logiciels*, 2000.

[BY96] R.S. Boyer and Y. Yu. Automated proofs of object code for a widly used microprocessor. *Journal of the ACM*, Janvier 1996.

[CGM+96] G. Chehaibar, H. Garavel, L. Mounier, N. Tawbi, and F. Zulian. Specialisation and verification of the powerscale bus arbitration protocol : An industrual experiment with lotos. *Proceedings of FORTE/PSTV'96*, 1996.

[Com] Mentor Graphics Company. Model technology. http ://www.model.com/.

[CP] J. Cong and D.Z. Pan. Interconnect delay estimation models for synthesis and design planning. *UCLA Computer Science Department Los Angeles, CA 90095 Sponsored by SRC and Intel.*

[CST01] E. Cerny, X. Song, and S. Tahar. Formal verification of systems. Department of Electrical and Computer Engineering, 2001.

[CW96] E.M. Clarke and J.M. Wing. Formal methods : State of art and future directions. *CMU-CS-96-178*, 1996.

[Dat] DelayMill Datasheet. Nanometer-ready delay calculation. http ://idec.kaist.ac.kr/CAD/epic/delaymill.html.

[Dav01] M. Davidson. Timing verification. *NOKIA*, 2001.

[DGL98] K. Dioury, A. Greiner, and M. Louerat. Hierarchical static timing analysis for cmos vlsi circuits. *University Pierre et Marie Curie*, 1998.

[Dio98] K. Dioury. *Docteur*. PhD thesis, Université Paris VI, 1998.

[FFG] B. Franzini, C. Forzan, and C. Guardiani. Accurate and efficient macromodel of submicron digital standard cells. *SGS-THOMSON Microelectronics.*

[Gup93] A. Gupta. Formal hardware verification methods : A survey. *School of Computer Science, Carnegie Mellon University*, 1993.

[KLS95] A. Kuehlmann, D.P. Lapotin, and A. Srinivasan. Verity - a formal verification program for custom cmos circuits. *IBM Journal of Research and Developement*, 1995.

[KM91] R.P. Kurshan and K.L. McMillan. Analysis of digital circuits through symbolic reduction. *IEEE trans. on Computer-Aided Design, 10,1350-1371*, 1991.

[MKL96] J S. Moore, M. Kaufmann, and T. Lynch. A mechanically checked proof of the correctness of the amd5k86 floating point devision algorithm. 1996.

[MP95] O. Maler and A. Pnueli. Timing analysis of asynchronous circuits using timed automata. *P.E. Camurati, H. Eveking (Eds.), Proc. CHARME'95,LNCS 987,189-205*, Springer, 1995.

[MS95] S.P. Miller and M. Srivas. Formal verification of the aamp5 microprocessor. *Workshop on Industial-Strength Formal Verification Technique, Page 2-16*, 1995.

[Muk02] R. Mukherjee. Static timing of integrated circuits. *ECE Dept, Northwestern University*, 2002.

[SBM03] R.B. Salah, M. Bozga, and O. Maler. On timing analysis of large combinational circuits. May 2003.

[Soc00] Design Automation Standards Committee Of The IEEE Computer Society. P1497 draft standard for standard delay format (sdf) for the electronic design process. *American National Standards Institute*, Juin 2000.

[Syn99] Synopsys, USA. *PrimeTime User Guide : Advanced Timing Analysis*, 1999.

[Yov97] S. Yovine. Model checking, timed automata. 1997.

Annexe A

Syntaxe des fichiers utilisés

A.1 Syntaxe du langage IF

```
system ::=
             system system-id ;
            [ type type + ]
            [ gate gate + ]
            [ var variable + ]
            [ sync sync-expression end ; ]
            process +

predefined-type-id ::=
            bool | int | real | nat | pid

type ::=
            type-id = enum literal { , literal }* ;
            type-id = range literal .. literal ;
            type-id= record { field-id : type-id ; }+ end ;
            type-id = array [ literal .. literal ] of type-id ;

literal ::=
            identifier | integer

gate ::=
            gate-id [ ( type-id { , type-id }* ) ] ;

variable ::=
            variable-id [initializer] {, variable-id [ initializer ]}* : type-id ;

initializer ::=
            ( expression { , expression }* )

sync-expression ::=
```

```
            process-id |
            sync-expression sync-operator sync-expression |
            hide gate-id { , gate-id }* in sync-expression

sync-operator ::=
            |[ gate-id { , gate-id }* ]|   |  ||   |    |||

process ::=
            process process-id  ;
            var variable + ]
            state state +
            transition transition +

state ::=
            state-id [ :init ]  ;

transition ::=
            from state-id
            [ if expression ]
            [ trigger [ if expression ] ]
            [ do action ] to state-id ;

trigger ::=
            sync gate-id { ? left-expression | ! expression }*

action ::=
            action , action |
            left-expression := expression

expression ::=
            literal |
            unary-operator expression |
            expression binary-operator expression |
            expression ? expression : expression |
            left-expression

left-expression ::=
            variable-id |
            left-expression . field-id |
            left-expression [ expression ] |
            any

unary-operator ::=
            not | pred | succ | pos | + | -

binary-operator ::=
            or | and | = | <> | < | > | <= | >= | + | - | * | / | %
```

A.2 Syntaxe du format d'Aldebaran pour les LTS

Le fichier ".aut" décrit un LTS dans le format d'Aldebaran, ou chaque état est représenté par un entier naturel.

```
aut-file ::=
            descriptor
            [ transition-definition* ]

descriptor ::=
            des (first-state, number-of-transitions, number-of-states)

transition-definition ::=
            (from-state, ‘‘transition-label’’, to-state)
```

A.3 Syntaxe du format descriptif du circuit

```
circuit ::=
            inputs-definition
            outputs-definition
            [ gate-definition + ]

inputs-definition ::=
            .input {input-id {; input-id}*}

output-definition ::=
            .output {output-id {; output-id}*}

gate-definition ::=
            gate-id = delay-Model  logic-function

delay-Model ::=
            [lower-bound, upper-bound]

logic-function ::=
            sigal-id |
            unary-operator sigal-id |
            ( logic-function ) |
            logic-function binary-operator logic-function

unary-operator ::=
            ~

binary-operator ::=
            * | +
```