École Polytechnique

Promotion 2006

Jean Xavier

Rapport de stage de recherche

Étude des performances dans les systèmes embarqués, entre simulation numérique et solution équationnelle

non confidentiel

Option : Informatique

Champ de l'option : Systèmes embarqués

Directeur de l'option : Daniel Krob

Directeurs de stage : Matthieu Moy, Karine Altisen

Dates du stage : 6 avril au 29 juillet 2009

Adresse de l'organisme de stage :

VERIMAG

Centre Équation - 2, avenue de Vignate

38610 GIÈRES

Résumé : Les systèmes embarqués actuels sont de plus en plus complexes, et les cycles de développement sont de plus en plus courts. Dès la phase de conception, on souhaite avoir une estimation des performances du système alors qu'aucun prototype physique n'est réalisé, et que le système est seulement décrit à un haut niveau d'abstraction. Pour répondre à la complexité croissante des systèmes, un cadre d'analyse de systèmes modulaires a été introduit : le *Modular Performance Analysis* (ou MPA). Dans ce cadre d'analyse, les systèmes sont décomposés en modules qui communiquent via des canaux sur lesquels circulent des flots infinis d'événements. Ce cadre est fortement lié à une méthode analytique : le *Real-Time Calculus* (ou RTC). Des travaux récents visent à connecter RTC à d'autres formalismes permettant de décrire des systèmes abstraits (dits *modèles computationnels*), comme par exemple le langage Lustre et les automates.

Après avoir décrit le principe général de RTC et des méthodes d'analyses de modèles computationnels, ce rapport soulève la question de la pertinence du formalisme RTC pour abstraire des flots d'événements dans le cadre du MPA, et propose une nouvelle abstraction pour décrire ces flots d'événements, ainsi qu'une méthode permettant de déterminer la transformation d'un flot d'événements représenté par cette abstraction par un système modulaire. Il décrit enfin un outil implémentant cette méthode en utilisant le moteur d'analyse de *NBac*.

Abstract : Current embedded systems are getting more and more complex, whereas their developement cycles are getting shorter and shorter. During the industrial design phase, engineers expect an approximation of the system's performances, before any physical prototype has been made, and while the system has only a high level abstraction description. In order to answer to the growing complexity of systems, an analysis framework has been introduced : the *Modular Performance Analysis* (or MPA). In this framework, systems are represented by components which communicate through channels with infinite event flows. This framework is strongly linked with an analytical method : the *Real-Time Calculus* (or RTC). Recent work aim at connecting RTC to other formalisms which allow to describe abstract systems (named *computational models*), such as the language Lustre and automata.

After describing the main ideas of RTC and of analysis methods on computational models, this report rises the issue of the relevance of RTC to describe abstractions of event flows in the MPA framework, and it proposes a new abstraction to describe those event flows, with a method allowing to determine the transformation of an event flow described by this abstraction, by a modular system. Finally, this report describes a programm which implements this method using the analysability power of *NBac*.

Table des matières

Chapitre 1

Introduction

1.1 Les systèmes embarqués

Les systèmes embarqués prennent de plus en plus d'importance dans les objets de la vie quotidienne (téléphones portables, appareils photos, etc.) mais aussi dans les systèmes critiques (contrôleurs de vol des avions, maîtrise du réacteur dans les centrales nucléaires, etc.). Les systèmes actuels sont de plus en plus complexes, mais doivent être produits à un coût de plus en plus bas et suivant des cycles de plus en plus courts. L'évolution des techniques de virtualisation et de simulation a permis le développement de méthodes de conception de systèmes embarqués décrits à un haut niveau d'abstraction, et l'application de méthodes d'analyse dès la phase de conception des systèmes.

Ces méthodes d'analyse ont pour but de prévoir le plus tôt possible dans la réalisation d'un système le comportement du système réel. Elles ont également pour but de prévenir des problèmes inhérents au développement d'un programme. Par exemple, dans certains avions récents (airbus A340 ou A380 par exemple), le pilote agit sur une manette reliée à un ordinateur, qui commande les gouvernes. Un bug dissimulé dans le logiciel de vol peut alors avoir des conséquences catastrophiques. De même, un système dont on s'aperçoit après réalisation qu'il est trop gourmand en consommation électrique est non compétitif (on a du mal à imaginer un téléphone portable qui tienne moins d'une demi-journée sur sa batterie). Ce genre de problèmes peuvent s'anticiper lors de la conception par des méthodes d'analyse de systèmes décrits à un haut niveau d'abstraction.

C'est dans ce contexte que s'est déroulé ce stage. La section suivante décrit plus en avant les différents types de méthodes d'analyse.

1.2 L'analyse de performances dans les logiciels embarqués

L'analyse de performances est un domaine très large. Chaque système a ses caractéristiques. Pour un capteur sur batterie, on mesurera la consommation électrique, pour le logiciel de contrôle des barres de commandes d'un réacteur nucléaire, on s'intéressera plutôt au temps de réponse du système. Les méthodes d'analyse de performances doivent donc être très générales. On distinguera l'analyse de performances de la vérification de propriétés de sûreté et de vivacité. Une *propriété de sûreté* certifie que "quelque chose

de mauvais ne peut jamais arriver au système" (ce quelque chose de mauvais étant une caractéristique, par exemple un dépassement de capacité). Une *propriété de vivacité* certifie que "quelque chose de bon se passera inévitablement au cours de l'exécution du système" (par exemple, le système se mettra en veille au bout d'un certain temps).

L'analyse de performance répond à des questions plus générales sur le comportement global du système. Si par exemple on est capable de prouver que "le système se mettra en veille au bout d'un certain temps", une analyse de performance peut dire "le système restera éveillé entre 5 et 10 secondes, puis se mettra en veille pour une durée comprise entre 3 et 7 secondes". On remarquera qu'ici on aurait pu exprimer plusieurs propriétés de sûreté, par exemple "le système ne peut pas rester réveillé plus de 10 secondes". En prouvant cette propriété, on aurait une idée des performances du système. Nous verrons dans ce rapport qu'analyse de performances et vérification de propriétés de sûreté sont fortement liées.

Certaines méthodes d'analyse de performances sont basées sur la simulation d'un grand nombre d'exécutions d'un système et sur l'étude statistique des résultats obtenus (On peut appeler cela le risque calculé). Nous ne nous intéressons pas à ces méthodes dans ce rapport. Ces méthodes s'opposent aux méthodes dites formelles qui donnent des résultats exacts sur les modèles, mais qui peuvent être moins précis sur le système réel puisqu'ils sont souvent basés sur des approximations.

1.3 L'analyse modulaire

Souvent les systèmes entiers sont trop complexes pour être directement analysables. Il faut donc les décrire de façon modulaire, et ainsi décomposer un problème impossible à résoudre en sous-problèmes plus simples. Le projet "Modular Performance Analysis of Distributed Embedded Real-Time Systems" mené à l'ETH Zurich et présenté dans [Zür] propose un formalisme nommé *Modular Performance Analysis* (ou MPA). Dans la suite du rapport nous nous placerons dans ce cadre d'analyse, dont nous allons donner les grandes lignes.

Le principe du MPA est de représenter un système par des modules (qui sont des sous-systèmes) qui sont connectés entre eux et qui communiquent via des canaux. Sur ces canaux circulent des flots de données sur une durée infinie. Par la suite on représentera un module avec un canal entrant et un canal sortant.

1.4 Deux catégories de méthodes d'analyse

On distingue deux catégories de méthodes d'analyse : les méthodes dites *analytiques* et les méthodes dites *par les modèles* ou *computationnelles*.

Les méthodes analytiques sont basées sur des formalismes purement équationnels. L'idée est de décrire un composant par un ensemble d'équations que l'on peut résoudre. La résolution de ces équations donne le meilleur et le pire cas d'exécution. Le modèle analytique que nous présenterons par la suite est le *Real-Time Calculus*, et s'avère efficace puisque les algorithmes qui le mettent en oeuvre sont pour la plupart quadratiques.

Les méthodes d'analyse de modèles computationnels sont plus anciennes. Elles sont en général basées sur une technique de simulation d'exécution et de vérification exhaustive

d'un système exprimé dans un formalisme défini par le modèle. Elles sont en général plus difficiles à mettre en oeuvre puisque les algorithmes sont pour la plupart NP-complets. On présentera dans ce rapport quelques formalismes basés sur des représentations par des *automates*, ainsi que le langage *Lustre*, qui implémente un formalisme dit de flot de données.

1.5 Objectifs

Actuellement le MPA est uniquement utilisé avec du Real-Time Calculus, qui utilise les courbes d'arrivée pour décrire les flots d'événements circulant sur les canaux. L'objectif de ce stage est de proposer nouvelle abstraction à base d'automates pour décrire ces flots d'événements, de proposer une technique pour déterminer la transformation de cette abstraction par un module, et d'implémenter un outil mettant en oeuvre cette technique. Le but final est de proposer une méthode pour faire du MPA en analysant des systèmes exprimés sous forme d'automates, de programmes Lustre, ou dans le formalisme RTC, tout en décrivant les flots de données de manière plus expressive que par le formalisme de RTC.

Dans ce rapport, nous présenterons dans un premier temps le formalisme RTC, puis les automates et le langage Lustre. Nous évoquerons l'hypothèse synchrone, ainsi que quelques outils permettant de faire de l'analyse de modèles computationnels. Nous achèverons l'état de l'art par la présentation de quelques techniques permettant de faire la connexion entre les modèles computationnels et RTC.

Nous présenterons quelques contributions à l'une de ces connexions : l'outil *ac2lus*, qui connecte RTC et Lustre, puis nous expliquerons les motivations qui nous ont poussé à proposer une nouvelle abstraction pour décrire les flots d'événements, ainsi qu'une méthode détaillée permettant de déterminer la transformation d'un flot d'événement, abstrait dans notre formalisme, par un système modulaire représenté par un automate. Nous terminerons par la présentation de l'outil *ROBERT* qui implémente notre méthode avec l'outil d'interprétation abstraite *NBac*.

Première partie

Deux approches opposées pour l'analyse de performances dans les systèmes embarqués

Résumé :

Cette partie bibliographique expose les grandes lignes des deux familles de méthodes d'analyse de performances dans les systèmes embarqués. On s'intéresse d'abord au Real-Time Calculus, puis après avoir introduit quelques modèles computationnels, on s'intéresse aux méthodes d'analyses sur ces modèles. Enfin on décrit quelques connexions entre les deux familles de méthodes, en particulier on s'attarde sur l'outil *ac2lus* sur lequel nous reviendrons dans les contributions.

Certains éléments de cette étude bibliographique ne resservent pas dans la seconde partie. Ils ne sont donc pas indispensables à la compréhension des contributions de ce stage, mais sont évoqués à titre culturel. Cela concerne essentiellement le paragraphe sur les automates temporisés. Sont indispensables pour la compréhension des contributions de ce stage les paragraphes sur l'outil *NBac*, sur l'outil *ac2lus* et sur les *Event Count Automata*.

Chapitre 2

Le Real Time Calculus

De nombreux travaux ont été effectués sur des systèmes temps réel dans le cadre d'analyse MPA, où chaque module communique avec les autres par le biais de canaux sur lesquels circulent des données. Chaque module ayant des spécifications connues à l'avance (consommation en ressources, temps de traitement des données, etc.), il est naturel de se demander comment vont réagir ces modules interconnectés. Le Real-Time-Calculus introduit dans [CKT03] propose un cadre général pour exprimer, connecter et analyser de façon globale de tels systèmes.

2.1 Motivations

Le Real-Time Calculus (ou RTC) est né de plusieurs travaux visant à inclure la notion du temps dans l'analyse de systèmes modulaires abstraits. Lors de la phase de conception d'un système, on peut s'attendre à avoir un cadre d'analyse à un haut niveau d'abstraction pour déterminer, avant même la réalisation physique du système, certaines propriétés sur son fonctionnement au cours du temps. Certains travaux comme par exemple [RE02] proposent un cadre d'analyse pour des systèmes dont les composants ont des réponses périodiques (éventuellement à une incertitude près) ou explosives (capables d'envoyer une rafale d'événements en un temps très court).

RTC généralise ces travaux en s'inspirant du Network Calculus, théorie qui modélise les files d'attentes dans les réseaux de communications, et qui est décrite dans [LBT04]. La figure 2.1 illustre le principe de la représentation abstraite des composants. RTC est constitué :

- d'un cadre de description abstrait pour des composants à partir d'équations qui décrivent leur comportement et d'un modèle d'allocation de ressources.
- d'un cadre de description abstrait pour des flots de données.
- d'un jeu de formules permettant de déterminer la transformation de ces flots de données par des modules.

Il faut pour cela pouvoir décrire un composant dans le bon formalisme. Nous verrons par la suite que certains systèmes ne sont pas facilement modélisable en RTC alors qu'ils le sont trivialement dans d'autres formalismes.

Remarque 2.1. On étudiera pour l'instant des systèmes formés d'un seul module. Pour

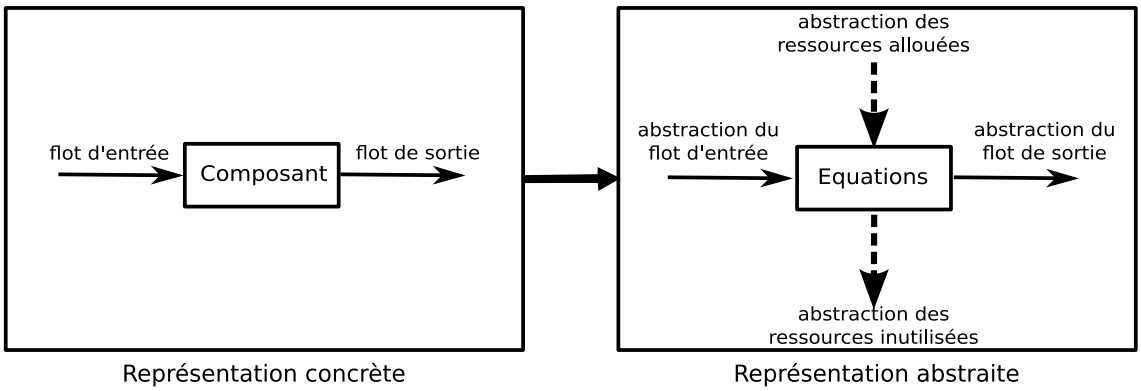


FIG. 2.1 – Abstraction d'un composant

simplifier les notations on désignera par "système" ou "composant" cet unique module.

2.2 Principes

Cette section expose les bases du RTC. Le lecteur intéressé pourra se référer à [LBT04] et à l'annexe A pour plus de détails. Dans cette partie, on utilisera le terme "flot d'événemtents" comme suit : un flot d'événements est une suite (éventuellement infinie) d'instances d'événements qui circulent sur un canal.

2.2.1 Courbes d'arrivée

Dans la plupart des cas, on ne connait qu'un ensemble de contraintes caractérisant un flot d'événements. Mais cet ensemble de contraintes peut correspondre à une infinité de flots possibles. Considérons par exemple un générateur d'événements périodique, avec une incertitude sur la période, disons de période $\pi = 10sec$ avec une incertitude $\epsilon = 1sec$. On notera que dire qu'il y a une incertitude sur la période revient à dire dans ce cas "la durée entre l'envoi de deux événements est comprise entre 9 et 11 secondes". La figure 2.2 montre plusieurs flots possibles correspondant à cette description. Sur cette figure, une flêche correspond à l'envoi d'un événement. Il y a clairement une infinité de flots possibles, et même une infinité non dénombrable dans ce cas, même si on ne considère qu'une section de durée finie du flot (il y a une infinité de valeurs dans l'intervalle [9, 11]).

Comment alors formuler un type de contrainte général (tel que tout flot puisse être compatible avec une contrainte de ce type, et tel qu'une contrainte puisse décrire une infinité de flots) ? Une vision simpliste serait de borner le nombre d'événements arrivant dans un intervalle de temps de taille fixée (ou instantanément si on se place dans une représentation continue). En reprenant notre exemple précédent on peut dire "Il arrive en une seconde entre 0 et 1 événements". Mais en n'utilisant que cette information, que peut-on dire sur un intervalle de 5 secondes ? Dans ce cas là, on ne peut rien dire de plus que "il arrive en cinq secondes entre 0 et 5 événements" alors que pour ce système on reste à un nombre d'événements émis compris entre 0 et 1. Si on continue l'expérience en regardant le comportement sur 11 secondes, on trouve avec cette représentation que le

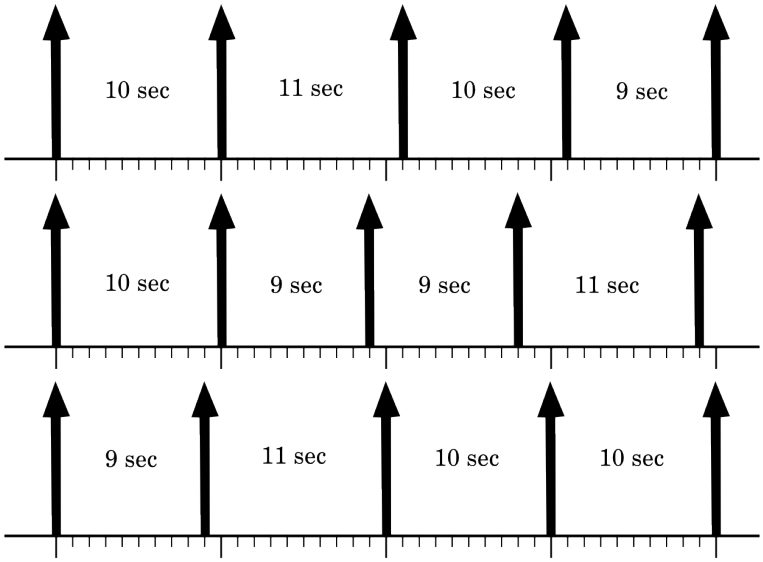


FIG. 2.2 – Divers flots comptaibles avec la contrainte énoncée ci-dessus

nombre d'événements est compris entre 0 et 11, alors que le nombre réel est compris entre 1 et 2 (il y a eu au moins un et au plus deux envois dans un intervalle de 11 secondes). Les résultats émis dans la première représentation restent vrais, mais font perdre beaucoup d'information, notemment sur le comportement du flot sur une durée plus grande.

La notion de courbe d'arrivée reprend la représentation précédente en ajoutant la notion de mémoire au cours du temps des valeurs prises par le flot. On dit souvent par abus de langage de courbe d'arrivée pour parler de paire de courbes d'arrivées (une correspondant à une borne supérieure, l'autre correspondant à la borne inférieure, on parle aussi de courbe haute et de courbe basse).

Definition 2.2. Courbe d'arrivée discrète : Soit $t \rightarrow R(t)$ un flot d'événements discrets cumulatif avec $t \in \mathbb{N}$ (il est arrivé $R(t)$ événements entre l'instant 0 et l'instant t). Les fonctions $x \rightarrow \alpha_u(x)$ (resp $x \rightarrow \alpha_l(x)$) sont courbe d'arrivée haute (resp basse) pour le flux r ssi $\forall s, t \in \mathbb{N}$, $s \leq t$, $\alpha_u(t-s) \geq R(t-s)$ (resp $\forall s, t \in \mathbb{N}$, $s \leq t$, $\alpha_l(t-s) \leq R(t-s)$). On remarquera que cette définition prend un temps avec une granularité fixe, et que l'on compte pour chaque intervalle de temps de cette granularité le nombre d'événements qui arrivent (qui est aussi entier). Certaines modélisations font l'hypothèse que le temps est discret, mais que les événements ne peuvent arriver qu'un par un. On a choisi ici de donner cette définition car en pratique, c'est celle qui correspond à nos implémentations.

Intuitivement, cela signifie que pour $x \in \mathbb{N}$, $\alpha_u(x)$ est une borne supérieure pour le nombre d'événements pouvant arriver dans une fenêtre de taille x et $\alpha_l(x)$ est une borne inférieure. Cette définition, valable pour des représentations discrètes s'adapte aux représentations continues.

Reprennons l'exemple ci-dessus. Une bonne courbe d'arrivée correspondant à ce système est représentée sur la courbe 2.3 :

On remarquera que passer d'un flot f (qui est une instance) à une paire de courbes d'arrivée revient à faire une abstraction. L'opération inverse consiste a partir d'une paire

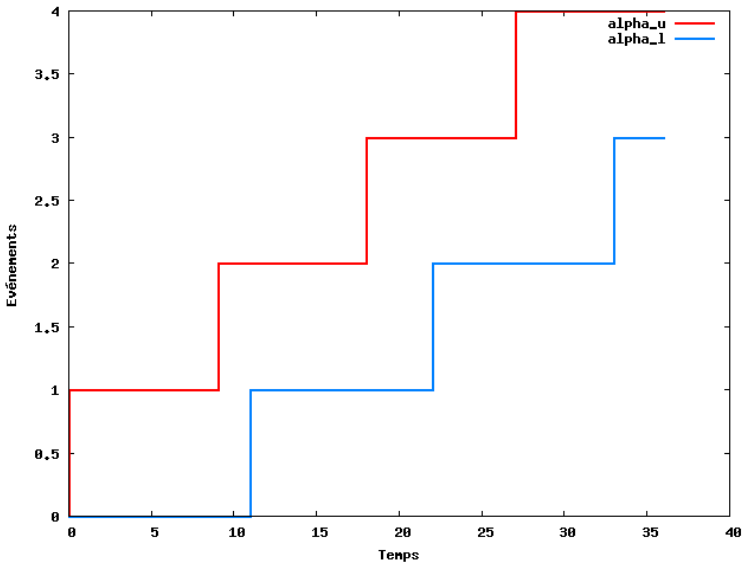


FIG. 2.3 – une paire de courbes d'arrivée

de courbes d'arrivée à considérer la classe de flots compatibles avec cette courbe d'arrivée. C'est la concrétisation.

2.2.2 Courbes de services

La notion de courbe de service est analogue à celle de courbe d'arrivée. Au lieu de répondre à la question "Combien d'événements arrivent dans un intervalle donné ?" elle répond à la question "Combien de ressources peut allouer le système pendant un intervalle de temps donné ?" Une unité de ressource traite instantanément un événement.

La définition est donc naturellement la suivante :

Definition 2.3. Courbe de service discrète (Service Curve) : Soit $t \rightarrow F(t)$ la fonction représentant le nombre de ressources allouées à un système entre l'instant 0 et l'instant t. La fonction $x \rightarrow \beta_u(x)$ (resp $x \rightarrow \beta_l(x)$) est courbe de service haute (resp basse) pour F ssi $\forall s,t \in \mathbb{N},\ s \leq t,\ \beta_u(t-s) \geq F(t-s)$ (resp $\forall s,t \in \mathbb{N},\ s \leq t,\ \beta_l(t-s) \leq F(t-s)$)

Comme pour les courbes d'arrivée,on fait l'abus de langage consistant à désigner par "courbe de service" la paire haute et basse de courbes de service. On peut formuler une définition analogue dans un espace continu. On peut également considérer la courbe de service correspondant à une classe de fonctions correspondant à des allocations de ressources.

2.2.3 Principe schématisé du RTC

Etant donné une paire de courbes d'arrivée modélisant un flot d'événements d'entrée, et un composant modélisé par une courbe de service et un ensemble d'équations, on peut calculer par des formules d'algèbre min-plus les courbes d'arrivée du flot de sortie ainsi que les courbes de service résiduelles. Le lecteur intéressé pourra se référer au théorème

A.16 pour plus de détails. La figure 2.4 donne une représentation courante des composants RTC.

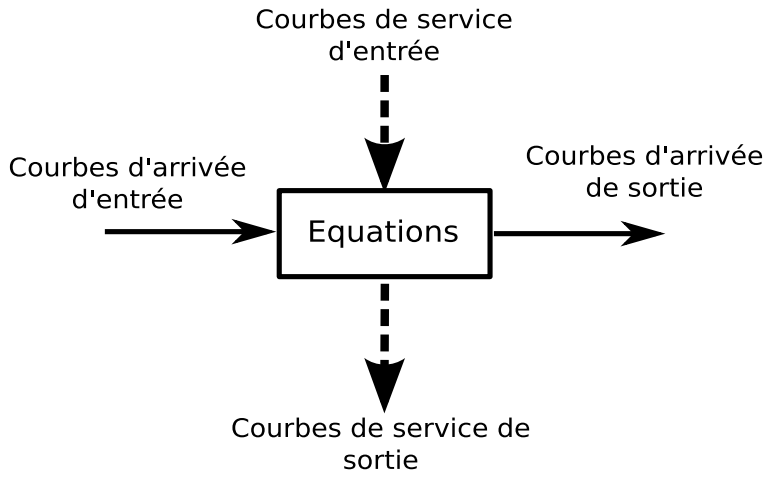


FIG. 2.4 – Schéma d'un composant RTC

2.3 Aspects algorithmiques

On donnera dans cette section les formules permettant de calculer en algèbre min-plus et max-plus une convolution et une déconvolution, qu'on utilisera pour calculer les courbes d'arrivée et de service sortantes. Nous verrons après comment l'application de ces formules peut se faire en temps polynomial.

2.3.1 Préliminaires mathématiques

L'algèbre min-plus (resp max plus) s'écrit en faisant l'analogie suivante : on part de l'algèbre usuelle, dans laquelle on remplace les additions par des minima (resp des maxima) et les multiplications par des additions.

Soient f et g deux fonctions à valeurs réelles. Les opérations de base s'écrivent alors comme suit :

Definition 2.4. Convolution en algèbre min-plus : $f \otimes g(x) = \inf_{i \in [0,x]} \{f(i) + g(x-i)\}$

Definition 2.5. Convolution en algèbre max-plus : $f \overline{\otimes} g(x) = \sup_{i \in [0,x]} \{f(i) + g(x-i)\}$

Definition 2.6. Déconvolution en algèbre min-plus : $f \oslash g(x) = \sup_{i \geq 0} \{f(x+i) - g(i)\}$

Definition 2.7. Déconvolution en algèbre max-plus : $f \overline{\oslash} g(x) = \inf_{i \geq 0} \{f(x+i) - g(i)\}$

2.3.2 Complexité

RTC s'est avéré être une méthode légère à implémenter. Dans les implémentations, on considère par simplicité des courbes à valeurs entières (en temps et en événements) et finies. Les opérations à effectuer sont les suivantes :

– des convolutions
– des déconvolutions
– des clotures sub-additives (ou super-additives)

Les algorithmes de convolution sont clairement quadratiques (Pour $f \otimes g(n)$, il faut tester toutes les combinaisons possibles entre les deux fonctions jusqu'à n, ce qui est en $O(n)$, au final, le calcul complet est en $O\left(\frac{N(N+1)}{2}\right)$ opérations où N est le nombre total de points à déterminer.

A priori, la déconvolution est plus difficile à implémenter puisque la formule $f \oslash g(x) = \sup_{i \geq 0} \{f(x+i) - g(i)\}$ implique une infinité de tests suivant les valeurs de i. [LBT04] propose d'utiliser la représentation de la déconvolution par inversion temporelle :

Definition 2.8. Représentation de la déconvolution par inversion temporelle : Soit g une fonction constante a partir d'un certain temps (par exemple une fonction finie prolongée par une fonction constante). Soit $T > 0$. Soit $\Phi_T(g)(t) = g(+\infty) - g(T-t)$. Soit f une fonction telle que $lim_{x->+\infty} f(x) = +\infty$. La relation suivante est vraie : $g \oslash f = \Phi_T\left(\Phi_T(g) \otimes f\right)$

Le calcul de la déconvolution est donc ramené au calcul d'une convolution et de deux inversions temporelles, ce qui est quadratique.

Calculer une cloture sub-additive finie est également quadratique. A priori calculer une cloture sous-additive demande également un nombre infini d'opérations. Si la courbe initiale est finie, rien ne permet d'affirmer que sa cloture sous-additive le sera (on peut très bien prolonger la courbe initiale par $+\infty$). Un résultat des travaux présentés dans [Bou05] est le suivant :

Théorème 2.9. ***Pseudo-périodicité des clotures sub-additives des courbes finies :*** *La cloture sub-additive d'une courbe finie est ultimement pseudo-périodique.*

Il suffit donc de calculer la partie apériodique ainsi qu'une période pour avoir toute l'information sur la courbe. Cependant, le problème reste ouvert concernant une borne acceptable sur la mise en place du régime pseudo-périodique, la meilleure borne actuelle étant en $O(N^2)$ pour une courbe à N points, ce qui donne un algorithme final en $O(N^4)$.

Les algorithmes implémentant RTC sont polynomiaux, ce qui est un luxe en vérification de systèmes embarqués, la plupart des méthodes étant basées sur des algorithmes NP-complets ¨.

Une des limites de RTC est le manque d'expressivité des systèmes. Si un système a un comportement particulier (par exemple un système qui induit un retard dans un flot d'événements), on ne pourra pas donner de composant RTC le décrivant avec précision. De même, et ce sera l'objet des parties suivantes, RTC ne s'applique pas bien aux machines à états, alors qu'elles sont très utilisées en modélisation de systèmes.

Chapitre 3

Quelques modèles computationnels

Les modèles dit “computationnels” visent à décrire le comportement d'un système de manière explicite. [Del] propose une définition des modèles computationnels proche de celle-ci : “Un modèle exécutable, réalisable, capable de prédire le comportement d'un système”. Ces modèles sont dotés d'un formalisme et souvent d'un outil implémentant ce formalisme. Dans cette partie nous présentons les modèles à base d'automates, puis nous explicitons plus en détail l'hypothèse synchrone, nous décrivons le langage Lustre, et nous donnons quelques éléments sur les automates.

3.1 généralités sur les automates

3.1.1 Motivations

Tous les systèmes (ou presque) à l'heure actuelle ont un mode d'économie d'énergie et un mode où ils sont réveillés et travaillent, mais consomment plus. La toquestion de l'économie d'énergie est légitime : de nombreux systèmes ont une batterie intégrée qui n'est pas changeable (ou du moins pas rentablement). C'est le cas par exemple de capteurs implantés dans des réseaux de capteurs. Une batterie doit durer plusieurs années, et avoir un logiciel économe en ressources permet de faire durer plus longtemps le système. Une manière intuitive de modéliser un tel système est de le décrire par un automate doté d'un état “en veille” et d'un état “réveillé”.

Les automates s'avèrent utiles dans de nombreux domaines de l'informatique (modélisation de systèmes, langages de programmation, etc...). De nombreux modèles ont leur format d'automate avec les outils adaptés. Les décrire tous serait inintéressant dans ce rapport. On donnera dans la partie suivante une définition formelle la plus simple possible des automates, ainsi que des définitions informelles de modèles d'automates utilisés par la suite.

3.1.2 Eléments de définitions

Un automate dans sa version la plus simple est un ensemble d'états et un ensemble de relations de transitions. On nomme *structure de contrôle* le graphe formé par ces états et ces transitions. On donne la définition formelle suivante :

Definition 3.1. Automates : Un automate dans sa version minimaliste est un triplet (Q, τ, Q_{init}) où Q représente l'ensemble des états, $\tau \subset Q \times Q$ est un ensemble de transitions, et $Q_{init} \subset Q$ est un ensemble d'états initiaux.

On appelle *trace* de l'automate la suite des états visités lors d'un parcours de ce graphe (plus généralement la trace d'un automate correspond à une information dépendant directement du parcours de cet automate, ce peut être l'évoution d'une variable par exemple). On a deux façons d'intuiter le lien entre un automate et une trace (qui sont heureusement équivalentes) :

- La manière passive : on donne une trace, un automate, et on vérifie si il existe un parcours de l'automate qui correspond à cette trace, en d'autres termes on vérifie que la trace est acceptée par l'automate.
- La manière active : l'automate définit un ensemble de traces qui correspondent à tous ses parcours possibles. On pourra parler de l'ensemble des éxécutions de l'automate.

Les automates décorés avec des variables

Dans les modèles d'automates qui nous intéressent, on ajoute des variables numériques et booléennes dans les automates, et on se donne un jeu d'opérations sur ces variables (affectation, test, ...). La trace d'un automate peut donc être la succession de valeurs prises par une (ou plusieurs) variable lors du parcours de l'automate.

On peut préciser plus en avant comment franchir une transition dans un automate. Une transition peut n'être franchissable que si une condition sur les variables est réalisée. On parlera de *garde*. De même, lors d'un franchissement d'une transition, les variables d'un automate peuvent changer. On pourra alors préciser leur nouvelle valeur. On parlera d'*action*. La figure 3.1 montre un exemple d'automate décoré d'une variable numérique (x) et de deux variables booléennes (a, b). La figure 3.2 montre un automate plus compliqué suivant le même principe.

Les variables d'un automate peuvent être des horloges (en pratique des variables réelles) qui avancent toutes à la même vitesse, et qui peuvent être réinitialisées. On parlera alors d'*automate temporisé*. La vitesse des horloges peut être la solution d'une équation différentielle. On parlera alors d'*automate hybride*. De même ces variables peuvent être des compteurs qui sont tous incrémentés d'une grandeur arbitraire ensembles à chaque pas de temps, et qui peuvent être réinitialisés. On parlera alors d'*event count automata*. On peut décorer un automate avec des variables d'entrée, qui sont non-déterministes, ou bien avec des variables internes dites variables d'état. Intuitivement, un automate ne contrôle pas ses variables d'entrées : il peut les lire mais pas leur affecter une valeur. Inversement, il contrôle ses variables d'état. Il peut donc à la fois lire et écrire dessus.

Composer plusieurs automates

Etant donné deux systèmes modélisés chacun par un automate, on souhaite savoir comment vont réagir ces deux systèmes l'un à coté de l'autre. On définit alors un opérateur de composition qui va faire un produit cartésien des deux automates. Dans l'automate produit, un état correspond à un couple d'états des deux automates initiaux, un état

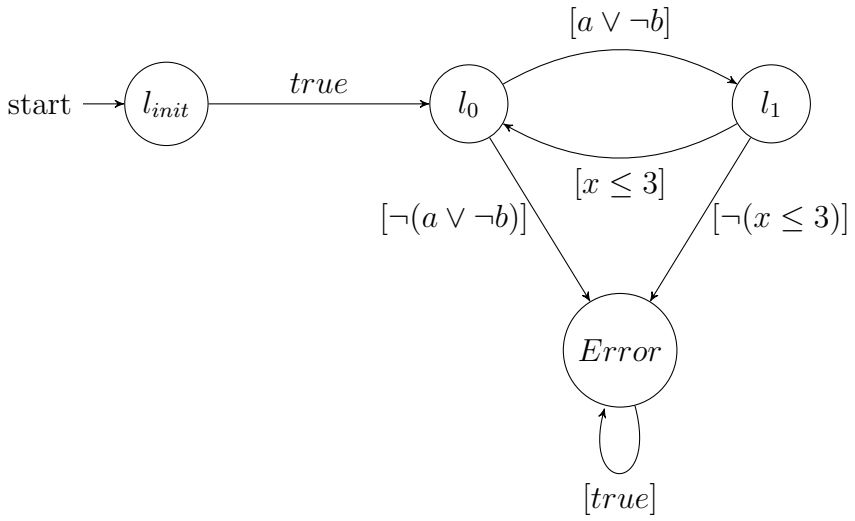


FIG. 3.1 – Exemple d'automate décoré

initial correspond à un couple d'états initiaux. L'ensemble des transitions est déterminé à partir des transitions des automates du produit, de différentes manières selon le type de produit (synchrone ou asynchrone). Intuitivement, le produit cartésien de deux automates décrit les éxécutions possibles des deux systèmes en parallèle, et de manière décorellée. On pourra éventuellement faire des produits d'automates partageant des variables.

3.2 L'hypothèse synchrone

3.2.1 L'approche synchrone dans les systèmes embarqués

On peut vouloir éxécuter plusieurs systèmes en parallèle. Cependant plusieurs problèmes se posent :

- Les différentes entrées du système peuvent arriver à des dates différentes (comment différencier une absence de signal d'un retard ?).
- Les tâches à effectuer sont difficiles à ordonner, puisque les durées d'éxécutions des systèmes sont a priori inconnues. Il peut s'ensuivre des problèmes d'accès concurrents à des données, etc.
- Il est difficile de prévoir la durée des communications, et d'interpréter l'absence de communication dans le système.

Bref, le comportement de deux systèmes peut se révéler non-déterministe. L'hypothèse synchrone répond à ce problème de la manière suivante : on abstrait le temps en une suite d'instants, on suppose que les traitements sont effectués infiniment vite sur un instant, se répètent à chaque instant, et sont simultanés. En pratique on vérifie cette hypothèse en s'assurant que le temps de traitement des données est petit devant la granularité adoptée dans l'abstraction du temps en instants.

Dans les cycles de conception actuels, un modèle synchrone doit passer à travers un compilateur qui génère du code usuel (en général du C). Il faut écrire un programme

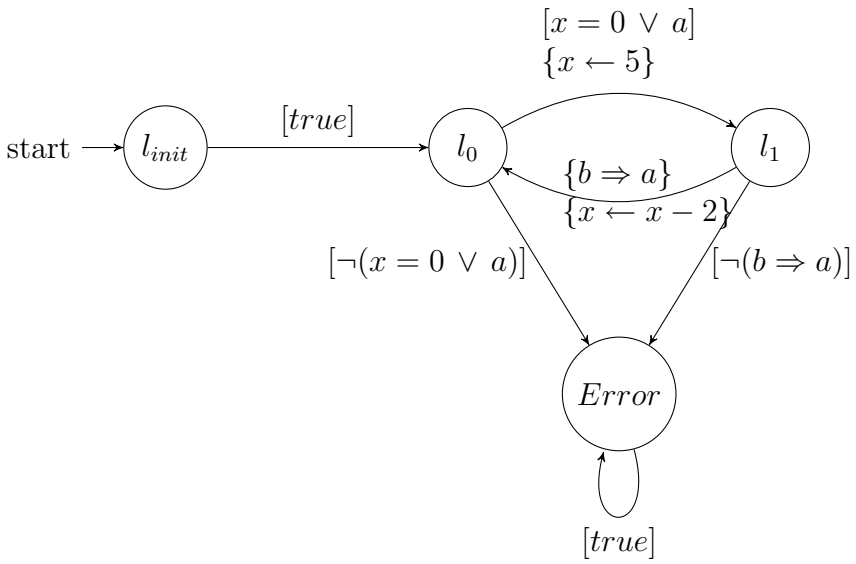


FIG. 3.2 – Autre exemple d'automate décoré, avec des affectations sur les transitions

d'intégration qui joue le rôle du "chef d'orchestre". Il rend artificiellement synchrone un système (et un environnement) asynchrone. Entre autres, il gère la lecture des capteurs, l'appel du programme synchrone et les actions résultantes sur les commandes. Au final, on évite tous les problèmes non-déterministes posés ci-dessus grâce au programme d'intégration. Par conséquent, le programme généré peut fonctionner dans la plupart des cas sur une machine nue (sans système d'exploitation) ou sur une machine avec un système d'exploitation léger.

3.2.2 L'approche synchrone comme outil de modélisation

Cette approche a plusieurs avantages :

- Elle permet la conception de programmes et de modèles synchrones abstraits, simples à concevoir
- Elle permet de faire de l'analyse de performance, ainsi que de la preuve formelle de programme
- Elle s'intègre bien dans le cycle de développement d'un système, comme le montre la figure 3.3

3.2.3 L'approche synchrone dans la composition d'automates

On peut faire l'hypothèse synchrone dans un produit d'automates. Sous cette hypothèse, les automates dont on fait le produit doivent "avancer ensemble". Il s'ensuit que certains couples d'états sont incompatibles car non atteignables en même temps. Nous réutiserons cette notion par la suite. On donne sur la figure 3.4 en exemple le produit synchrone des deux automates présentés sur les figures 3.1 et 3.2

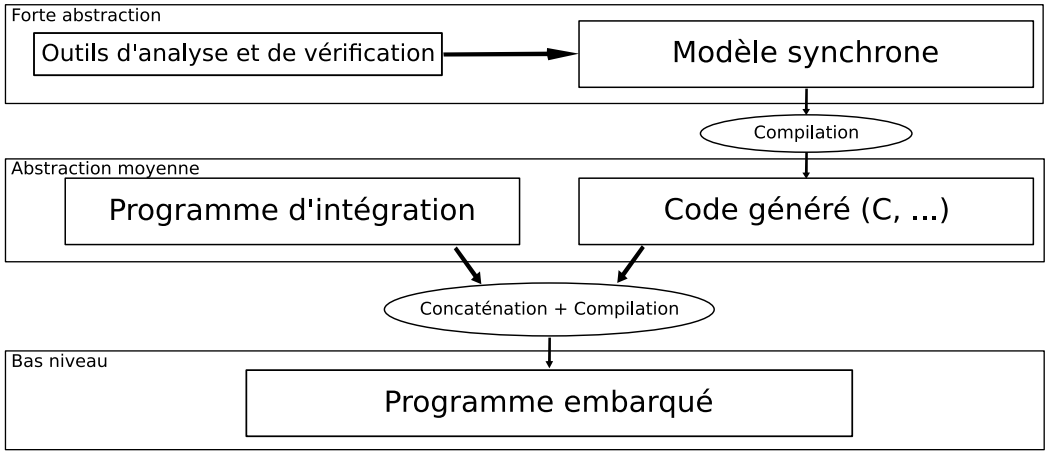


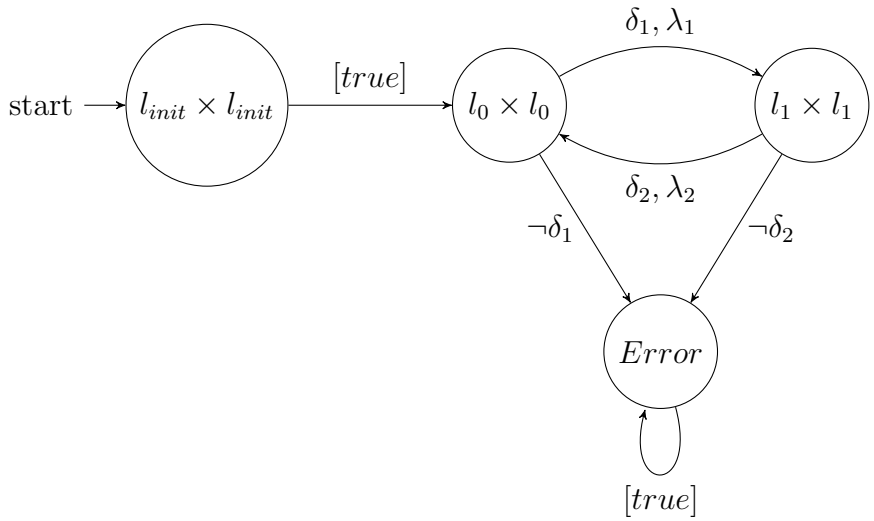
FIG. 3.3 – Chaine de conception d'un système synchrone

3.3 Le langage Lustre

3.3.1 Généralités

Lustre est défini comme "un langage de programmation synchrone déclaratif, et par flot". La première version date de 1987 et est présentée dans [CPHP87] et dans [HCRP91] . La version actuelle (lustre v6) est maintenue à Verimag. Scade, la version industrielle de Lustre existe depuis 1993 et est commercialisée par Esterel Technologies. Elle est entre autres utilisée par Airbus pour l'informatique embarquée sur les airbus A340/600 et A380, ainsi que par Schneider Electric pour l'informatique implantée dans les centrales nucléaires. Lustre permet notamment la génération de code C directement implémentable (ou presque) sur des systèmes. Il existe deux autres langages synchrones similaires à Lustre : Esterel et Signal.

En Lustre les variables manipulées sont des flots de données. On notera x une variable $t \rightarrow x(t)$ où t correspond au temps discret. Un programme Lustre se décompose en noeuds, chaque noeud correspondant à une transformation d'un (ou plusieurs) flot d'entrée en un (ou plusieurs) flot de sortie, c'est à dire à une fonction F telle que si X représente l'ensemble des variables d'entrée et Y représente l'ensemble des variables de sortie, un noeud décrit la relation $Y(t) = F(X(t), X(t-1), X(t-2), ..., X(t-n))$. On remarquera qu'en Lustre, la mémoire est bornée statiquement. Mathématiquement, un nœud correspond à un système non ordonné d'équation entre sorties, entrées et variables internes. Lustre implémente notamment un opérateur *pre* qui retarde un signal d'une unité de temps. La figure 3.5 montre un exemple simple de noeud lustre (les flêches représentent les différentes arrivées et départs de paquets d'événements, plus la flêche est grande, plus le nombre d'événements contenu dans le paquet est grand, ici le flot d'arrivée serait [2, 3, 1, 2] et le flot de départ serait [1, 3, 3, 2] en lisant le flot de droite à gauche). Le lecteur interessé par la programmation en Lustre pourra se référer à [Hal93].



$$\delta_1 = [(a \vee \neg b) \wedge (x = 0 \vee a)]$$
$$\delta_2 = [(b \Rightarrow a) \wedge (x \leq 3)]$$
$$\lambda_1 = \{x \leftarrow 5\}$$
$$\lambda_2 = \{x \leftarrow x - 2\}$$

FIG. 3.4 – Exemple de produit synchrone d'automates

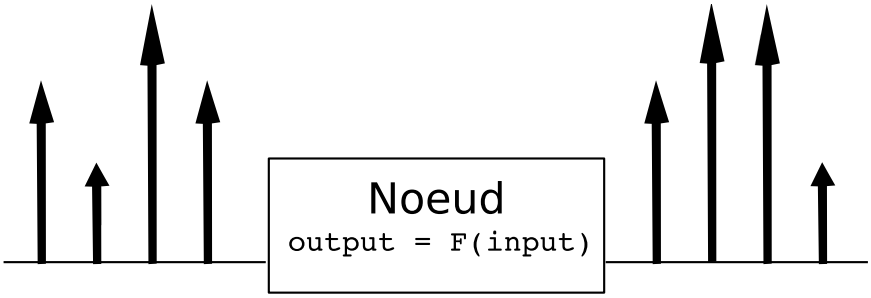


FIG. 3.5 – Exemple de noeud transformant un flot d'entiers réalisable en Lustre

3.3.2 Exemple de programme

Cette section présente un programme modélisant une bascule RS écrit en Lustre. Ce noeud décrit un interrupteur que l'on peut allumer en mettant à *true* le canal *set*, éteindre en mettant à *true* le canal *reset* et dont la valeur initiale est celle du canal *in*. La valeur du signal de sortie (*level*) ne change pas en cas d'action simultanée sur *set* et *reset*. La figure 3.6 donne le code source du noeud décrit ci-dessus ainsi qu'un exécution possible de ce noeud.

```
node SWITCH (set, reset, in : bool) returns (level : bool);
let level = in -> if reset and not set then false
                 else if set and not reset then true
                 else pre(level);
tel
```

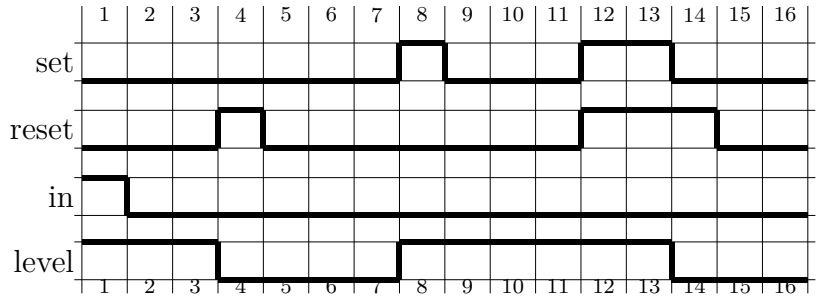


FIG. 3.6 – Exemple de programme en Lustre et d'une exécution de ce programme

Chapitre 4

Méthodes d'analyse sur les modèles computationnels

Les problèmes d'analyse de modèles sont très souvent compliqués. Certains problèmes de décision en analyse de modèles sont indécidables, et on ne peut trouver qu'une solution partielle. Il existe de nombreuses méthodes d'analyse automatiques ou semi-automatiques. Dans cette partie nous présentons deux méthodes : le model-checking et l'interprétation abstraite, ainsi que des outils les implémentant basés sur les modèles présentés dans la partie précédente.

4.1 Le Model-Checking

Le model-Checking représente une large classe de méthodes visant à démontrer des propriétés logiques dépendant du temps sur des systèmes exprimés par des automates. Il existe de nombreuses variantes du model-checking suivant la logique utilisée et le type de propriétés à prouver. On présentera de manière simplifiée dans cette section le type de model-checking utilisé dans l'outil *Lesar*, puis l'outil *Lesar*, qui fait de la vérification de programmes écrits en Lustre.

Les premiers travaux sur le model-checking ont été entrepris par Edmund Clarke et Allen Emerson en 1981 dans [CE82], ainsi que par Jean-Pierre Queille et Joseph Sifakis en 1982 dans [QS82]. Le model-checking est aujourd'hui très largement utilisé dans les programmes d'analyse de systèmes. Le lecteur intéressé par le model-checking dans plus de détails pourra se référer par exemple aux ouvrages de Edmund Clarke, comme [CGP00].

4.1.1 Principes

Le model-checking a pour but de prouver une propriété logique sur un système en prennent en compte son évolution au cours du temps. Cette propriété peut être une propriété de vivacité ("quelque chose de bon va obligatoirement se produire lors de l'exécution") ou une propriété de sûreté ("quelque chose de mauvais ne peut jamais arriver lors de l'exécution"). Dans le contexte qui nous intéresse, on cherche à prouver des propriétés de sûreté uniquement. On s'intéressera donc uniquement aux méthodes qui traitent ce problème, que l'on appellera quand même model-checking par abus de langage.

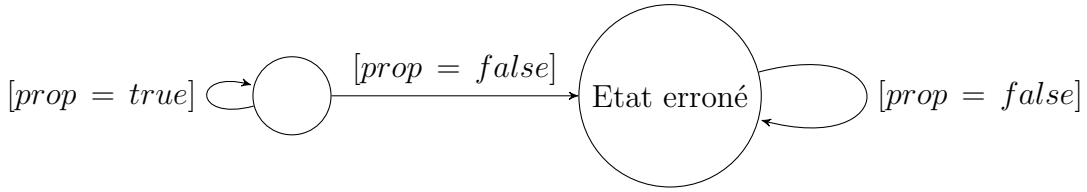


FIG. 4.1 – Exemple d'observateur pour une propriété *prop*

L'idée simplifiée du model-checking est de faire une analyse d'accessibilité sur le produit du système étudié et d'un observateur (ou un autre automate exprimant la négation de la formule) de la propriété que l'on veut prouver (cf figure 4.1). Si aucun des états de l'automate produit où la propriété est fausse (on parle d'*états erronés*) n'est accessible depuis les états initiaux, alors la propriété est vraie pour le système étudié. On pourra éventuellement faire des hypothèses sur le système. On parlera d'*assertions* sur le système. Comme on peut le voir sur la figure 4.2, qui représente un exemple de produit d'automates, une analyse d'accessibilité (et éventuellement de coaccessibilité, c'est à dire d'accessibilité en arrière en partant des états erronés) montre que certaines transitions (barrées sur la figure) sont irréalisables. Sur cet exemple on peut voir que les états erronés sont inaccessibles (les transitions qui y mènent sont symboliquement barrées), et que la propriété est vraie.

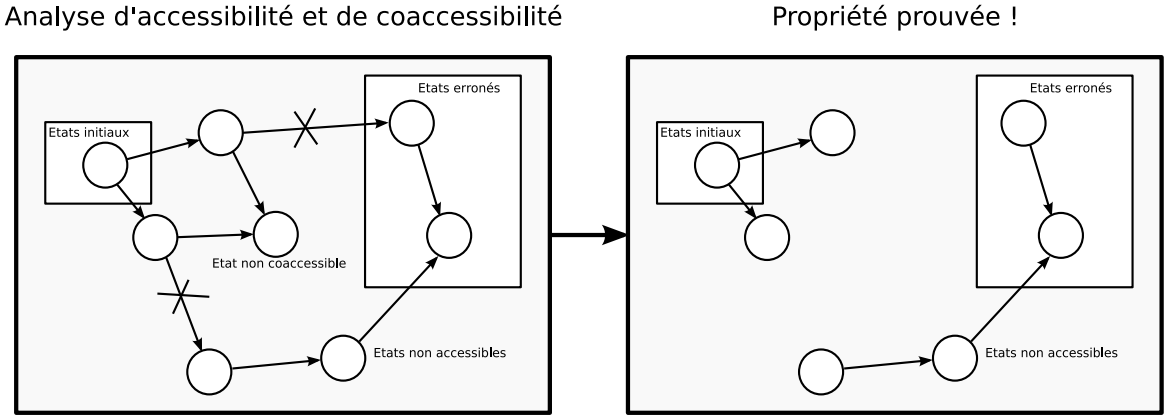


FIG. 4.2 – Exemple d'analyse de système

4.1.2 La nécessité de faire des abstractions

Certains problèmes sont trop complexes pour pouvoir être traités de manière exacte. Dans certains cas, le problème exact peut être indécidable. Beaucoup d'algorithme d'exploration d'espace d'états ont une complexité exponentielle par rapport au nombre d'états ou au nombre de variables. Dans certains cas, on peut réduire le problème considéré en considérant une approximation du problème, c'est à dire un sur-ensemble des comportements possibles pour le système étudié, ce surensemble étant plus simple à étudier. On peut par exemple éliminer une variable booléenne en considérant comme satisfiables toutes

les clauses où elle apparaît. Ici les approximations que l'on fait sont des abstractions conservatives, c'est à dire des abstractions dont les propriétés de sûreté sont des propriétés de sûreté du système.

Le model-checking est efficace pour traiter des problèmes finis (en particulier booléens). Cependant les model-checkers ne traitent pas les problèmes infinis (comme par exemple certains problèmes numériques, contrairement aux interpréteurs abstraits, ce que nous verrons dans la section suivante). Dans *Lesar*, qui est le model-checker que nous décrirons par la suite, on fait l'approximation suivante : les propriétés booléennes sur des variables numériques (par exemple $x \leq 2$) sont toujours considérées comme vraies dans l'analyse d'atteignabilité de l'automate. D'une manière générale, les model-checkers et les outils d'analyse d'automates ont recours à des abstractions.

En supposant par exemple toutes les propriétés numériques comme vraies, on surapproxime l'ensemble des comportements du systèmes (dans le système réel, toutes les propriétés numériques ne sont heureusement pas forcément vraies), mais on réduit l'ensemble des comportements du système abstrait à un ensemble de variables booléennes, ce qui est décidable de manière exacte. Dans certains cas, l'approximation sera trop grossière le model-checker ne pourra pas prouver la propriété, alors qu'elle pourrait être vraie. Le compromis entre précision de l'analyse et simplicité est un problème récurent en analyse de systèmes. La section suivante présente l'outil *Lesar*, ainsi qu'une structure efficace pour représenter des fonctions sur des variables booléennes : les *Diagrammes de Décision Binaire* (ou BDD).

4.1.3 L'outil *Lesar*

L'outil *Lesar* est un model-checker basé sur le langage Lustre. Plus précisément, c'est la concaténation des outils *lus2ec* et *ecverif*, *ecverif* étant le model-checker. Lesar possède plusieurs algorithmes de model-checking, et utilise des diagrammes de décision binaire pour représenter les fonctions booléennes. On peut faire :

- une analyse d'atteignabilité en calculant par un parcours en profondeur un invariant pour chaque état (sous la forme d'un BDD), et en déterminant si un état atteignable viole la propriété. C'est l'*algorithme énumératif*.
- une analyse déterminant une fonction booléenne représentant l'ensemble des états accessibles, qui doit toujours rester disjointe de l'ensemble des états erronés. C'est l'*algorithme symbolique en avant*.
- une analyse déterminant une fonction booléenne représentant l'ensemble des états coaccessibles (qui mènent à des états erronés), qui doit toujours rester disjointe de l'ensemble des états initiaux. C'est l'*algorithme symbolique en arrière*.

Tous ces algorithmes doivent faire des opérations sur des fonctions booléennes. Une des clés de l'efficacité de ces algorithmes est donc l'efficacité de ces opérations. Les BDD permettent d'effectuer de manière efficace ces opérations. Ils ont été introduits dans [Bry86]. Aujourd'hui, la bibliothèque CUDD implémente de manière efficace les opérations sur les BDD. Des extensions aux BDD ont été développées, comme par exemple les MTBDD qui ajoutent les fonctions booléennes à partir de variables numériques.

4.2 L'interprétation abstraite

L'interprétation abstraite est en quelque sorte le complément numérique du model-checking. Elle détermine pour chaque variable numérique un ensemble de valeurs admissibles au cours de l'exécution du programme, et confronte cet ensemble de valeurs à des fonctions booléennes basées sur ces variables (par exemple $x \geq 5 \wedge y = 0$). Cette section présente quelques principes de l'interprétation abstraite, ainsi que l'outil *NBac* qui implémente ces méthodes.

4.2.1 Principes

Domaines abstraits

On ne peut pas énumérer les valeurs possibles que peut prendre une variable numérique puisqu'il y en a a priori une infinité, et on ne peut pas représenter exactement de manière finie n'importe quel ensemble de valeurs. On doit donc avoir recours à des abstraction. On définit une *valeur abstraite* comme étant une abstraction d'un ensemble de valeur (par exemple l'intervalle [0, 1] est une valeur abstraite de l'ensemble des réels compris entre 0 et 1). On définit un *domaine abstrait* comme un ensemble des valeurs abstraites. Il y a plusieurs sortes de domaines abstraits. Les plus utilisés sont les polyèdres convexes, les intervalles et les octogones. Ils doivent être clos pour les opérations usuelles (unions convexes, intersection, ...).

Problèmes rencontrés en analyse d'automates

Lorsqu'on fait une analyse d'automate, on a deux cas de figure, comme illustré sur la figure 4.3 :

- Tant qu'on ne rencontre pas de cycle dans la structure de contrôle, on détermine sans problèmes les images successives des valeurs abstraites des variables dans chaque état en appliquant les fonctions de transition (quitte à faire une union convexe des domaines obtenus si deux transitions arrivent sur un état).
- Dès que l'on rencontre un cycle, on bloque sur un problème : on peut avoir besoin d'un nombre infini d'itérations pour déterminer la valeur abstraite finale.

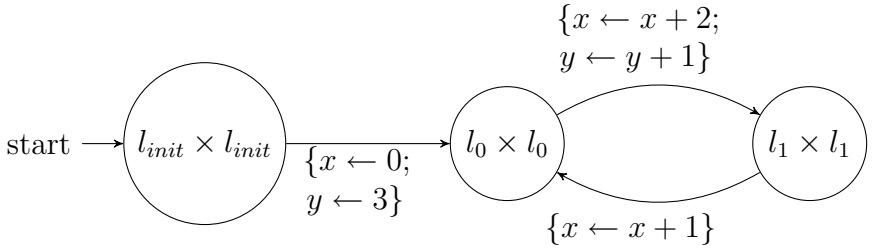


FIG. 4.3 – Exemple d'automate contenant un cycle

Ce deuxième point est un problème fondamental en interprétation abstraite. Il s'agit de résoudre de manière itérative une équation de plus petit point fixe du type $X = F(X)$ où X est un domaine abstrait et F est la fonction de transition de l'automate étudié.

L'analyse des relations linéaires

Un cas intéressant est celui où l'ensemble des relations manipulées sont linéaires. Il s'avère que l'ensemble des polyèdres convexes permet de décrire exactement ces relations. Cependant,la méthode itérative décrite ci-dessus appliquée aux polyèdres convexes peut ne pas converger. On doit donc avoir recours pour résoudre l'équation de point fixe ci-dessus à un opérateur d'élargissement, introduit dans [CH78] entre autres. Un opérateur d'élargissement (noté en général ∇) appliqué lors de la méthode itérative (c'est à dire en prenant $X_0 = \top, X_{n+1} = \nabla(F(X_n), X_n)$) assure la convergence de la méthode en un nombre fini d'itérations.

Intuitivement un opérateur d'élargissement "anticipe" la dynamique d'évolution du domaine abstrait obtenu au cours de la méthode itérative. L'opérateur d'élargissement implémenté dans l'outil *NBac*, que nous verrons plus tard, fonctionne de la manière suivante : lorsque le domaine a tendance à grossir dans une certaine direction, il suppose qu'il va continuer à évoluer dans cette direction et détermine le domaine correspondant si cette évolution était infinie. On peut aussi voir cela comme le fait d'enlever une contrainte au polyèdre représentant la domaine dans lequel les variables "vivent". Les opérateurs d'élargissement simples sont d'ailleurs basés sur ce principe : si en une itération le nombre de contraintes du polyèdre n'a pas évolué, c'est que l'on a atteint un point fixe. Sinon, il diminue. Le résultat converge forcément en un nombre fini d'itérations. La figure 4.4 montre un exemple d'application d'opérateur d'élargissement : si on n'applique pas l'opérateur d'élargissement, la contrainte C se transforme en la contrainte C'. Une itération plus loin, on pourrait avoir C'' à la place de C', et un nouveau domaine abstrait D''. Si en revanche on applique l'opérateur, la contrainte C disparaît, de telle sorte qu'en une seule itération, on a une surapproximation du domaine final.

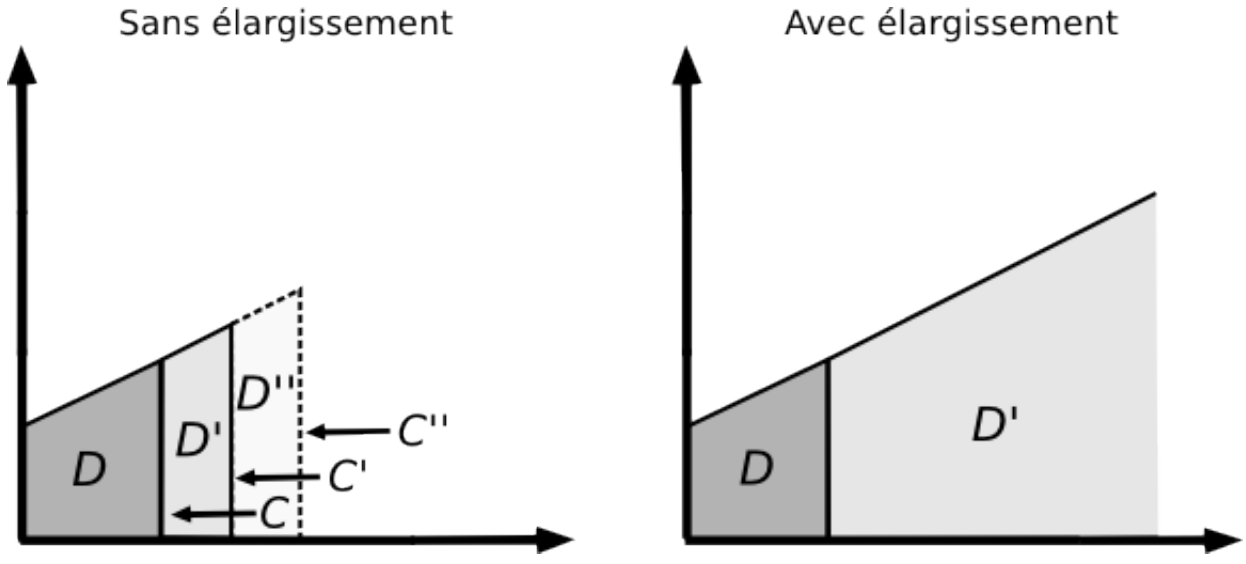


FIG. 4.4 – Exemple d'application d'un opérateur d'élargissement à l'automate décrit sur la figure 4.3

4.2.2 L'outil *NBac*

Cette section présente l'outil *NBac* développé au laboratoire Verimag dans le cadre de la thèse de Bertrand Jeannet. NBac implémente les méthodes d'interprétation abstraites présentées ci-dessus (avec des optimisations), ainsi que la technique du *partitionnement dynamique*. Décrire en détail cette technique n'est pas le but de ce rapport. Le lecteur intéressé est invité à lire [Jea00].

NBac fait de la preuve de propriété de sûreté sur des automates. Dans ce cadre il est amené à effectuer une analyse d'accessibilité en avant et en arrière. On peut spécifier la structure de contrôle de l'automate à analyser, ou bien il peut en déterminer une abstraction, c'est à dire une approximation de l'automate réel, ayant pour *places initiales* l'ensemble des approximations contenant des états initiaux, et pour *places finales* l'ensemble des approximations contenant des états erronés. C'est notamment le cas lors de l'analyse de programmes en Lustre (l'outil *lus2nbac* permet d'exprimer des programmes lustre directement sous le format NBac). En ce qui concerne les variables booléennes, NBac implémente un algorithme de model-checking.

NBac manipule d'un côté un programme réactif (définissant les relations de transitions des variables), et de l'autre une structure de contrôle correspondant à ce programme. Il effectue une analyse d'atteignabilité avant-arrière des états erronés à partir des données du programme. S'il arrive à prouver la propriété demandée, il s'arrête, sinon il utilise le principe du partitionnement dynamique.

Le principe du partitionnement dynamique peut s'écrire comme suit : il peut s'avérer que la structure de contrôle est trop grossière pour parvenir à prouver la propriété. On peut alors raffiner l'automate analysé en séparant des états. NBac implémente des heuristiques pour trouver les états pour lesquels il est pertinent de raffiner l'approximation (il faut que le fait de raffiner simplifie de façon significative le problème à échelle locale). Une fois les états raffinés NBac réitère son algorithme d'analyse avant-arrière, ce jusqu'à ce qu'il arrive à prouver la propriété, ou le cas échéant jusqu'à ce qu'il ait raffiné un certain nombre de fois.

L'interprétation abstraite est un problème très vaste, pour lequel plusieurs méthodes et outils ont été développés. NBac est celui que nous utiliserons par la suite.

Chapitre 5

Connexions entre les deux approches

Certains systèmes sont faciles à décrire dans le formalisme RTC. D'autres sont très faciles, voire triviaux à décrire sous forme de programme synchrone, ou bien sous forme d'automates alors qu'ils sont très difficiles, voire impossibles (le problème est ouvert) à décrire en formalisme RTC. Des travaux récents visent à connecter ces deux approches, pour pouvoir à terme les utiliser ensembles dans l'analyse de systèmes réels. L'idée générale de ces méthodes est de fournir des "adaptateurs", c'est à dire des composants qui interfacent le formalisme RTC avec un autre formalisme. A partir d'une paire de courbes d'arrivée, ils déterminent un objet f_{inp} qui décrit un ensemble de flots d'événements admissibles. On dispose d'une méthode permettant à partir d'un système et d'un objet f_{inp} de déterminer un objet f_{out} qui décrit un ensemble de flots de sortie possibles. Le schéma général est présenté sur la figure 5.1. Cette partie présente plusieurs méthodes et outils qui connectent ces approches. Il est conseillé de bien lire la partie sur les *Event Count Automata* puisqu'elle resservira beaucoup dans la seconde partie de ce rapport.

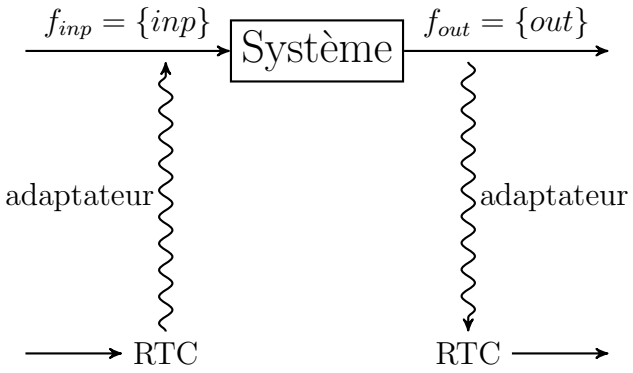


FIG. 5.1 – Schéma général des méthodes de connexions entre RTC et d'autres formalismes

5.1 Les *Event Count Automata*

Un Event Count Automaton (ou ECA) est un automate synchrone auquel on a ajouté des compteurs qui s'incrémentent tous ensembles d'une grandeur arbitraire qui correspond

au nombre d'événements qui arrivent à chaque unité de temps (que nous appellerons "flot d'entrée"). On peut choisir de réaffecter un compteur (éventuellement de le réinitialiser) lors d'une transition. Le lecteur intéressé par les event count automata pourra se référer à [CPT05].

Les Event count automata peuvent servir à représenter sous forme d'automate des paires de courbes d'arrivée discrètes et finies. Étant donné une paire de courbes d'arrivée discrète (par rapport au temps), on peut créer un event count automaton dont l'ensemble des traces admissibles pour le flot d'entrée (c'est à dire l'ensemble des flots d'entrée compatibles avec le parcours de l'automate) est égal à l'ensemble des flots compatibles avec la paire de courbes d'arrivée donnée. cette technique est présentée dans [PCTT07]. Nous en présenterons ici une version simplifiée sous les hypothèses suivantes (qui en pratique ne sont pas gênantes pour des modèles synchrones) :

- La granularité du temps est fixe, c'est à dire que les événements arrivent périodiquement (ce qui permet de ne pouvoir décrire une courbe α qu'avec la donnée des points $\alpha(1)$, $\alpha(2)$, ...$\alpha(n)$).
- La courbe donnée est subadditive si c'est une courbe haute, super-additive si c'est une courbe basse, et minimale (ie le dernier point de la courbe ne peut pas être déduit des précédents par clôture sub-additive pour les courbes hautes, ou super-additive pour les courbes basses), sinon au besoin on peut retirer les derniers points tant que la courbe ne l'est pas.

Quelques remarques sur cette méthode :

- Cette méthode est basée sur une représentation des courbes d'arrivée finies. Aussi on pourra appliquer le théorème 2.9 qui dit que la clôture sub-additive d'une courbe d'arrivée discrète et finie est ultimement pseudo-périodique. On pourra donc réduire la courbe à la partie apériodique et une pseudo-période au besoin.
- On remarquera que l'on peut déterminer un ECA décrivant la courbe haute, un ECA décrivant la courbe basse, et faire un produit synchrone des deux pour obtenir un ECA conforme à la paire de courbes.

On présentera un exemple simple de cette méthode appliquée à la courbe C (représentée sur la figure 5.2) dont les coordonnées des points sont : $\alpha_u = [0, 3, 5, 7, 8]$ et $\alpha_l = [0, 1, 2, 4, 5]$. Les compteurs de l'ECA sont notés $\alpha_1, \alpha_2, ...$ (intuitivement, α_i correspondra à la mémoire du nombre d'événements arrivés dans les i derniers instants). Conformément à la sémantique des ECA, on n'indiquera pas le flot d'entrée comme une variable du système. Les compteurs seront incrémentés après chaque transition de la valeur du flot d'entrée. L'ECA correspondant est représenté sur la figure 5.3 (sur cette figure, les fonctions δ_i représentent les gardes, et les fonctions λ_i les actions).

Ce modèle d'ECA sera par la suite repris dans la partie 7.

L'ensemble des traces admissibles par l'ECA décrit ci-contre correspond exactement à l'ensemble des flots compatibles avec la paire de courbes d'arrivée représentée sur la figure 5.2. Cet ECA peut être composé avec des systèmes modélisés par des automates. En revanche, il n'est pas toujours possible de représenter le résultat de la composition du système à étudier et de l'ECA d'entrée directement par un ECA. [PCTT07] propose alors une technique pour déterminer à partir d'un automate une paire de courbes d'arrivée qui lui correspond. On notera qu'un automate contient plus d'information qu'une paire de

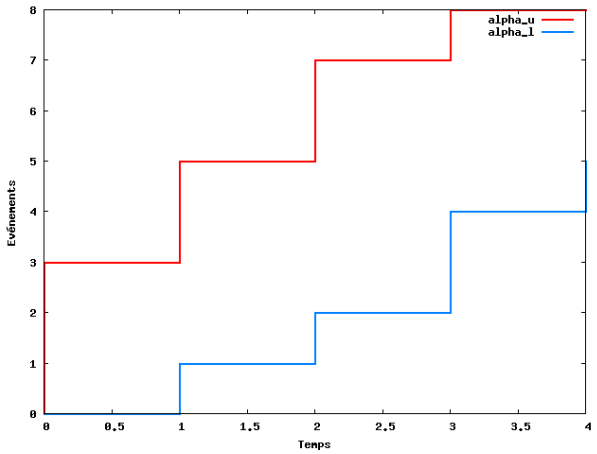


FIG. 5.2 – Courbe d'arrivée correspondant à l'ECA

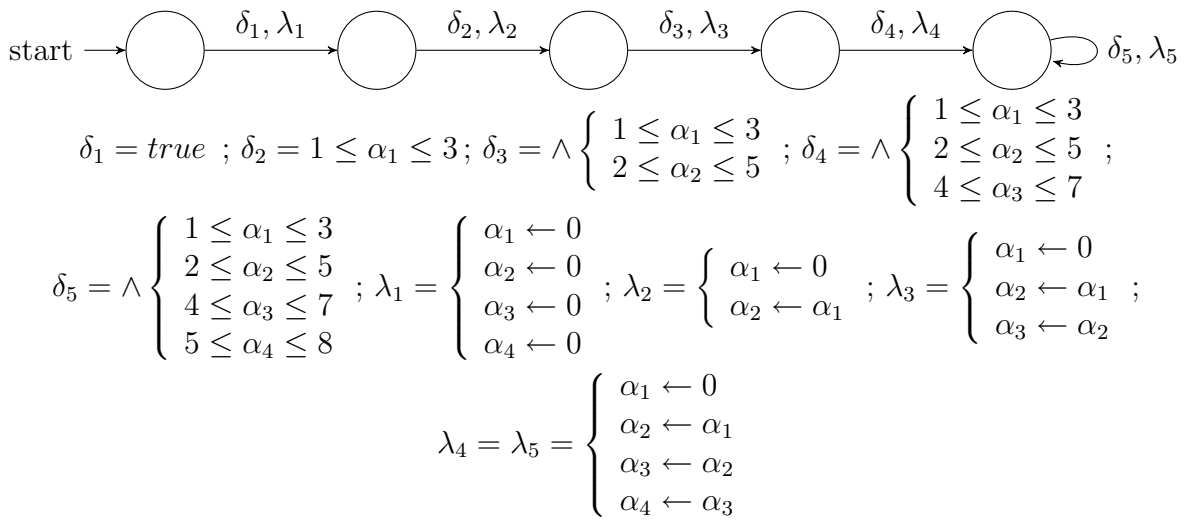


FIG. 5.3 – Event count automata correspondant

courbes d'arrivée, il y a donc dans ce cas une perte d'informations sur le comportement du système.

5.2 Les automates temporisés

Les automates temporisés sont des automates classiques auxquels on a rajouté des horloges qui avancent toutes en même temps. Les contraintes sur les horloges ξ_i sont de la forme $\xi_i \prec a$ ou $\xi_i - \xi_j \prec a$, avec a une constante et $\prec \in \{=, \leq, \geq, >, <\}$. On peut lors d'une transition réinitialiser une ou plusieurs horloges. On étudie en général les automates temporisés sur le domaine qui correspond à ces contraintes (il est appelé "ensemble des zones" dans la littérature). Une particularité de ce domaine est que si les horloges sont toutes bornées, il est fini, et donc l'analyse est exacte. L'outil *CATS* présenté dans [PLW] implémente ces techniques.

La figure 5.4 montre le principe d'analyse de système à partir d'automates temporisés. On sait exprimer un automate temporisé à partir d'une courbe d'arrivée (en utilisant un adaptateur qui utilise représentation proche de celle des ECA). Le lecteur intéressé pourra se référer à [PLW]. Pour déterminer une paire de courbe d'arrivée à partir d'un automate temporisé, on a recours à un observateur, dont nous décrivons le principe dans la suite de cette section.

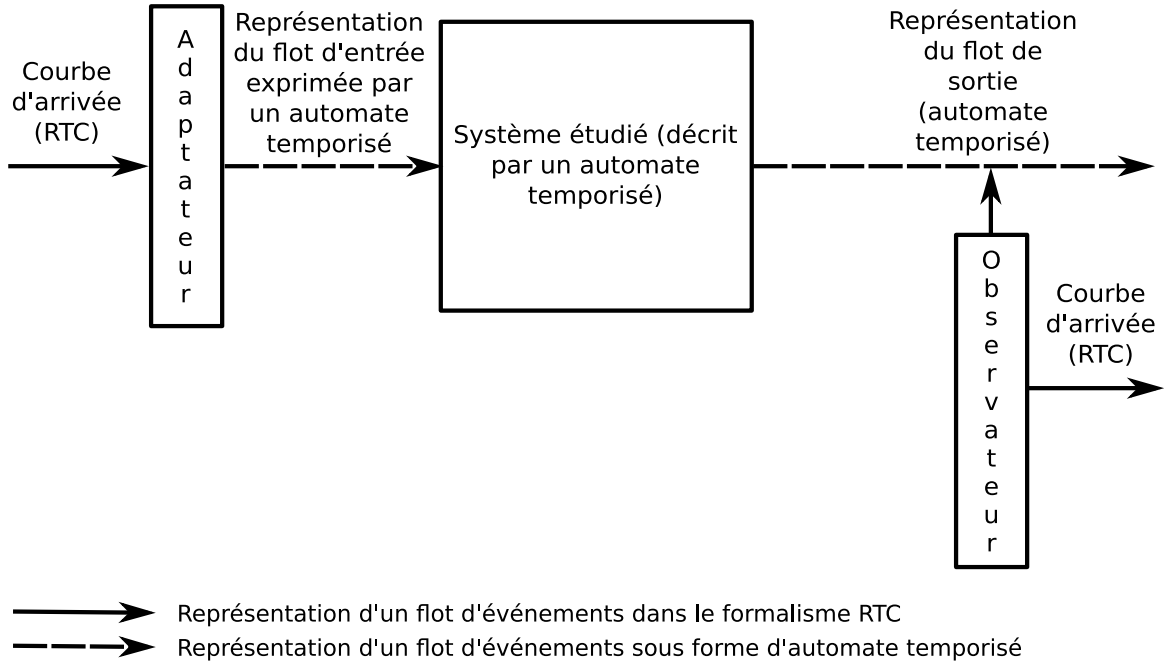


FIG. 5.4 – Chaîne d'analyse d'un système modélisé par un automate temporisé

Le principe d'un observateur est le suivant : étant donné une grandeur a, on détermine la durée minimale d dans laquelle le système peut envoyer a événements, ainsi que la durée maximale D pendant laquelle le système peut n'émettre pas plus de a événements. En répétant cette méthode pour plusieurs valeurs a_i, on obtient une paire de courbes formée par les points (d_i, a_i) et (D_i, a_i) qui est une surapproximation de la courbe réelle. La figure 5.5 illustre cette méthode.

5.3 le multimode RTC

Le multimode RTC est une théorie récente développée dans [PCT08]. L'idée est de prendre un automate, et d'associer à chacun de ses états une paire de courbes d'arrivée, ainsi qu'une paire de courbes de service. Cette théorie, encore immature, n'est pas

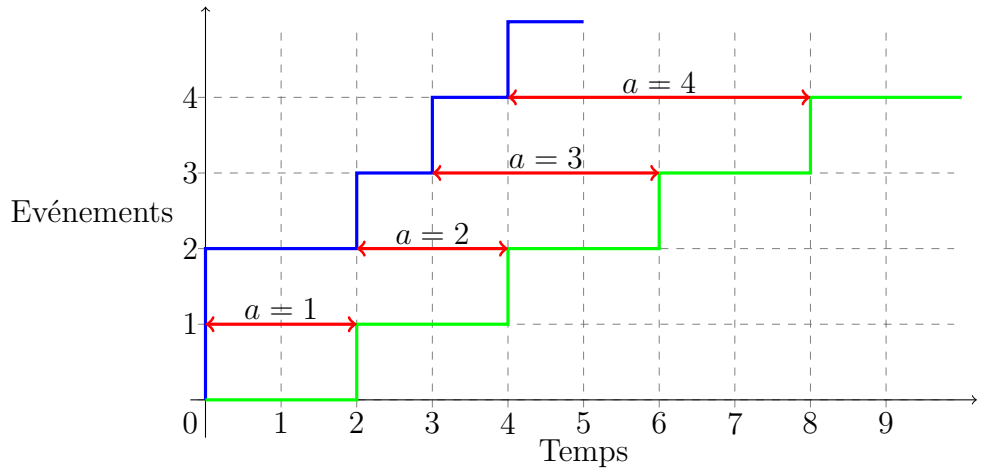


FIG. 5.5 – Exemple de courbe d'arrivée produite par un observateur à partir d'un automate temporisé

développée dans ce rapport. Le lecteur intéressé pourra se référer à l'article de référence donné ci-dessus.

5.4 L'outil *ac2lus*

5.4.1 Présentation

L'outil *ac2lus* est une connexion entre RTC et Lustre. Il a été développé à Verimag par Matthieu Moy en 2008. Il est présenté dans [MA08]. Le principe est proche de celui des analyseurs à base d'automates temporisés :

1. On a un système à étudier sous la forme d'un programme Lustre.
2. Un adaptateur va transformer la courbe d'arrivée d'entrée en un programme Lustre d'entrée (qui sera le noeud *input_observer* ou *input_generator*, ce que nous décrirons plus tard)
3. Un noeud Lustre nommé *binary_search* va concaténer le programme d'entrée, le système à étudier, ainsi qu'un *observateur synchrone* de sortie, qui, étant donné une paire de courbe d'arrivée qui lui est proposée va dire si oui ou non le flot de sortie est compatible avec cette courbe.
4. Un module d'analyse va rechercher de manière dichotomique la meilleure courbe à proposer à l'observateur de sortie.

Le schéma de fonctionnement est décrit sur la figure 5.7. L'outil *ac2lus* prend en entrée une paire de courbes d'arrivée à partir de laquelle il génère un programme lustre. On peut générer deux types de programmes :

– un générateur d'entrée qui crée de façon non déterministe un flot d'entrée compatible avec la courbe d'arrivée (l'ensemble des flots pouvant être générés doit être égal à l'ensemble des flots compatibles avec la courbe d'arrivée). Ce sera le noeud *input_generator*.

```
-- Observateur correspondant à la courbe d'arrivée
node binary_search  (sequence : int)
returns (ok : bool)
var
 ok : bool; in_ok : bool; out_ok : bool;
 out_seq : int;
let
  --Propriété à prouver
  ok = out_ok or not (in_ok);
  --Observateur de sortie
  out_ok = output_observer(out_seq);
  --Transformation du flot par le système
  out_seq = systeme(sequence);
  --Observateur d'entrée
  in_ok = input_observer(sequence);
tel

-- Exemple de système étudié : un calculateur
-- infiniment rapide
node systeme (in_seq  : int)
returns (out_seq  : int)
let
  out_seq = in_seq;
tel
```

FIG. 5.6 – Code source du noeud binary_search et exemple de programme Lustre étudié

– un observateur synchrone qui observe un flot d'entrée non déterministe et renvoie *true* si le flot est compatible avec la courbe d'arrivée, *false* sinon. Ce sera le noeud *input_observer*.

Si on est dans la configuration avec un observateur en entrée, et si on note F le flot d'entrée, $S(F)$ le flot de sortie généré par le système, $I(F(t))$ la valeur de l'observateur d'entrée jusqu'à la date t, et $O(S(F(t)))$ la valeur de l'observateur de sortie jusqu'à la date t. On note $I(F) = \forall t,\ I(F(t))$ et $O(S(F)) = \forall t,\ O(S(F(t)))$. On cherche à prouver la propriété suivante : $\forall F,\ I(F) \Rightarrow O(S(F))$, où de manière équivalente $\forall F,\ \neg I(F) \vee O(S(F))$. Dans le cas ou l'on a un générateur en entrée, la propriété à prouver est $\forall F,\ O(S(F))$ (ici, le flot d'entrée est forcément compatible avec la courbe d'arrivée d'entrée, on a donc seulement à vérifier la compatibilité du flot de sortie avec l'observateur). La figure donne l'allure du code Lustre du noeud binary_search dans la version avec observateur d'entrée.

Concrètement *ac2lus* est un programme réalisé en C++ qui fait environ 1500 lignes de code. La version actuelle est encore à l'essai. Une des contributions de ce stage est l'ajout d'optimisations à *ac2lus*. Ce sera l'objet de la partie suivante.

5.4.2 Les observateurs synchrone

Un observateur synchrone est un programme exprimant une propriété logique $P(t)$ dépendant de ses entrées et/ou de ses sorties. Un analyseur peut alors prouver une propriété du type $\forall t$, $P(t)$. Dans *ac2lus*, les observateurs synchrones vérifient des propriétés de sûreté sur un flot d'entiers. Cependant on est confronté à un problème de logique lorsqu'on utilise deux observateurs : Si on note $I(F(t))$ la propriété "Le flot F (d'entrée) est conforme par rapport à l'observateur d'entrée jusqu'à la date t", et $O(S(F(t)))$ la propriété "Le flot $S(F)$ (de sortie) est conforme par rapport à l'observateur de sortie jusqu'à la date t", l'analyseur (qui va analyser le programme entier) va prouver $\forall F$, $\forall t$, $[I(F(t)) \Rightarrow O(S(F(t)))]$, alors que l'on cherche à prouver $\forall F$, $I(F) \Rightarrow O(S(F))$, ou de manière équivalente $\forall F$, $[\forall t, I(F(t))] \Rightarrow [\forall t, O(S(F(t)))]$. Ces deux propositions sont équivalentes si et seulement si les trois assertions suivantes sont respectées :

- Quand une propriété devient fausse, elle le reste pour toujours, on dit que l'échec doit être permanent.
- La précondition (ici $I(F(t))$) doit être causale, c'est à dire que "toute séquence compatible avec la courbe d'arrivée jusqu'à une certaine date est prolongeable en une s'équence éternellement compatible".
- La précondition doit être déterministe.

La notion de causalité est abordée dans [MA08]. Nous n'y reviendrons pas. Le problème de l'échec permanent est résolu dans le code Lustre en remplaçant la propriété *prop* par $prop \wedge (true \rightarrow pre(prop))$ en Lustre. Enfin, l'observateur d'entrée est toujours déterministe. On remarquera que dans la version de *ac2lus* munie d'un générateur d'entrée, le problème précédent ne se pose pas puisque l'on cherche à prouver $\forall F$, $O(S(F))$.

La figure 5.8 donne le code Lustre d'un observateur d'entrée correspondant à la courbe d'arrivée $\alpha_u = [0, 3, 5, 7]$; $\alpha_l = [0, 1, 2, 4]$. La propriété à prouver est *ok*.

5.4.3 La détermination de la meilleure courbe de sortie

Un observateur de sortie tel qu'on l'utilise dans *ac2lus* traduit les performance performances du système en une propriété de sûreté ("le système émet au plus 5 événements par seconde"). Cette propriété est donnée à l'outil *NBac* qui dit si elle est vraie sur le système, ou indéterminée. Dans le cas où elle est vraie, on peut traduire la propriété de sûreté en une courbe d'arrivée. En pratique, il est plus simple de déterminer un par un les points de la courbe de sortie. Il est en effet plus simple de prouver séparément des propriétés du type "le système émet au plus 5 événements en une seconde" et "le système émet au plus 7 événements en deux secondes" que lorqu'elles sont réunies (en cas d'échec de la preuve de la propriété globale, on ne sait pas quel point est mauvais). On s'intéresse donc à un observateur de sortie sur une seule courbe correspondant à un seul point. La figure 5.9 montre le code source d'un observateur de sortie qui exprime la propriété "le système émet au plus un événement en deux unités de temps".

Pour déterminer la meilleure courbe possible, on procède par dichotomie : le principe est le suivant (pour déterminer par exemple le point $\alpha_u(i)$:

1. On initialise $\alpha_u(i)$ à 1

2. Tant que *NBac* n'arrive pas à prouver la propriété $P(\alpha_u(i))$: "Le système émet au plus $\alpha_u(i)$ événements en i unités de temps", on fait l'opération $\alpha_u(i) \leftarrow 2\alpha_u(i)$
3. Dès que *NBac* arrive à prouver la propriété $P(\alpha_u(i))$, on considère le segment $[\alpha_u(i)/2, \alpha_u(i)]$. On effectue alors une recherche dichotomique classique sur ce segment.

La procédure de test d'une valeur pour un point est la suivante :

1. Détermination de la valeur à tester pour le point considéré
2. Génération de l'observateur de sortie suivant cette valeur
3. Soumission du noeud *binary_search* à *NBac*

Une contribution de ce stage est la détermination d'heuristiques permettant d'accélérer significativement la recherche dichotomique. Nous présenterons cela dans la partie suivante.

L'outil *ac2lus* fournit donc à partir d'une paire de courbe d'arrivée et d'un système modélisé par un programme en Lustre, la meilleure courbe d'arrivée déterminable par *NBac* qui correspond au flot de sortie. La partie suivante présente des optimisations apportées à *ac2lus* dans le cadre de ce stage.

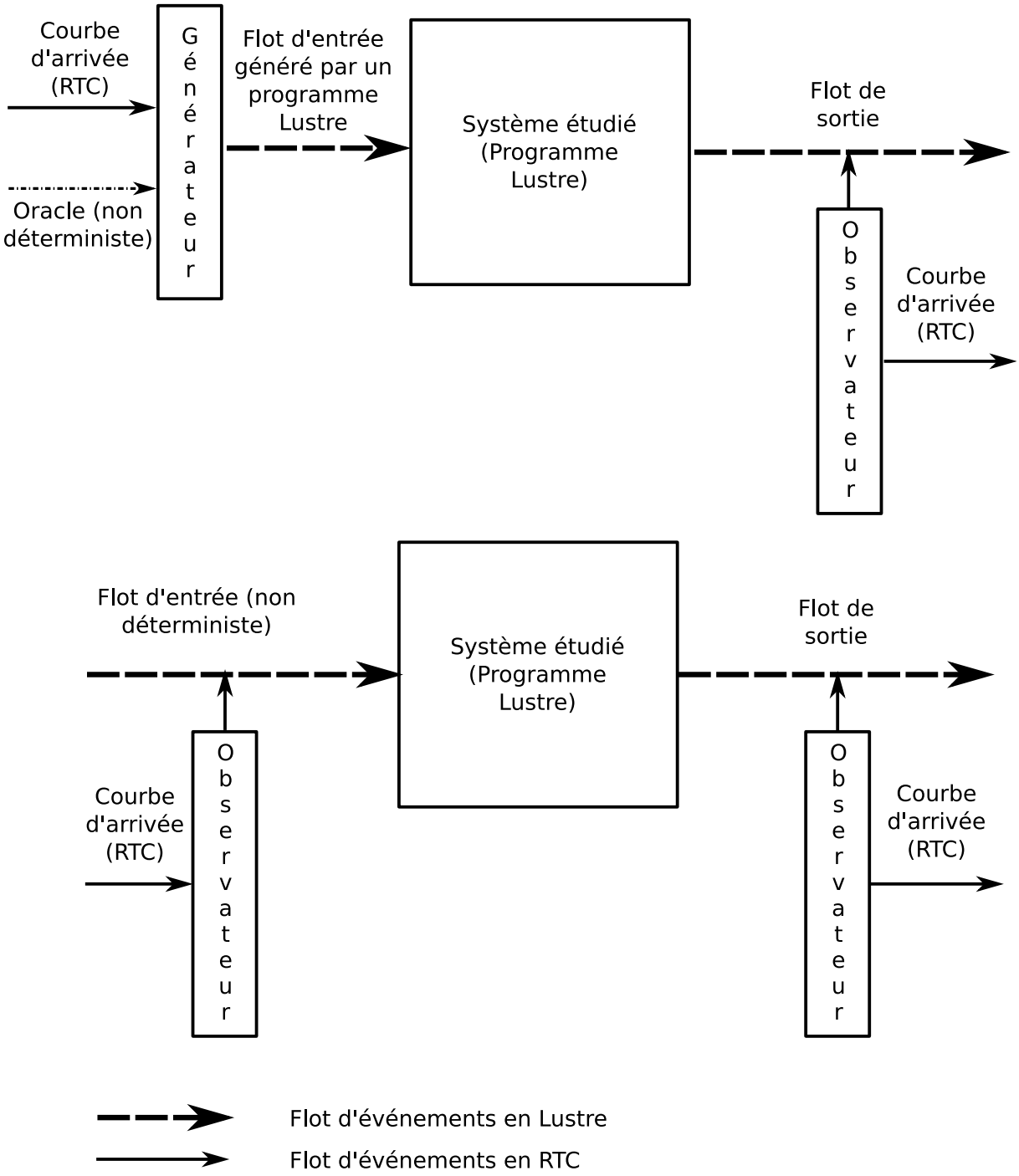


FIG. 5.7 – Principe de fonctionnement de l'outil *ac2lus* dans la version générateur et observateur d'entrée

```
-- Observateur correspondant à la courbe d'arrivée
node input_observer  (sequence : int)
returns (ok : bool)
var
  ok_now : bool;
  count1 : int; inter1 : bool; count2 : int;
  inter2 : bool; count3 : int;
let
  count1 = sequence;
  inter1 = true -> false;
  count2 = sequence -> pre(count1) + sequence;
  inter2 = true -> pre(inter1);
  count3 = sequence -> pre(count2) + sequence;
  ok_now = (1 <= count1 and count1 <= 3)
    and (if inter1 then true else (2 <= count2 and count2 <= 5))
    and (if inter2 then true else (4 <= count3 and count3 <= 7))
  ok = ok_now and (true -> pre(ok));
tel
```

FIG. 5.8 – Code source en Lustre d'un observateur synchrone

```
-- Observateur de sortie correspondant à la courbe haute [?, ?, 1]
node output_observer  (sequence : int)
returns (ok : bool)
var
  ok_now : bool;
  count1 : int; inter1 : bool;
  count2 : int; inter2 : bool;
  count3 : int;
let
  count1 = sequence;
  inter1 = true -> false;
  count2 = sequence -> pre(count1) + sequence;
  inter2 = true -> pre(inter1);
  count3 = sequence -> pre(count2) + sequence;
  ok_now = (if inter2 then true else (count3 <= 1));
  ok = ok_now and (true -> pre(ok));
tel
```

FIG. 5.9 – Code source en Lustre d'un observateur de sortie

Deuxième partie

Proposition d'une méthode d'analyse de performances pour des systèmes modulaires

Résumé :

Cette partie décrit en détail l'ensemble des contributions de ce stage. Elle suit l'ordre chronologique des différentes pistes explorées, ainsi que des outils sur lesquels j'ai travaillé. La partie 7 décrit le problème posé durant ce stage. La partie 8 décrit une piste suivie pendant un certain temps, que nous avons finalement décidé d'abandonner. Cette partie peut donc aisément être sautée, tout comme la partie 6, qui présente quelques travaux effectués au début du stage. La partie 9 propose une solution au problème posé dans la partie 7. C'est la partie la plus intéressante de ce rapport. Enfin, la partie 10 présente l'outil *ROBERT* développé durant ce stage.

Chapitre 6

Participation au développement de l'outil *ac2lus*

Au début de mon stage, j'ai travaillé sur des optimisations de l'outil *ac2lus*. Ce travail avait deux buts : me familiariser avec la programmation synchrone et l'analyse de performance, et essayer effectivement de nouveaux outils, comme par exemple les observateurs non-déterministes. Dans cette partie on présentera les optimisations apportées, ainsi que des comparaisons des différents modes de fonctionnement de *ac2lus*, avec et sans les optimisations.

6.1 Optimisations apportées

6.1.1 Une heuristique pour la recherche dichotomique

La recherche dichotomique des meilleurs points des courbes d'arrivée était implémentée de façon basique. Par exemple dans le cas d'une courbe haute, pour déterminer un point α, on initialisait la recherche à $\alpha = 1$. Puis tant que le point n'était pas valide, on faisait l'opération $\alpha \leftarrow 2\alpha$. Ceci jusqu'à trouver une valeur α_0 qui soit valable. On effectuait ensuite une recherche dichotomique entre $\alpha_0/2$ et α_0.

Un inconvénient de cette méthode est qu'elle ne tient pas compte des valeurs des points précédents. Par exemple si on a déjà pour la courbe haute les valeurs $[\alpha_u(0), \alpha_u(1), \alpha_u(2)]$, une valeur possible de $\alpha_u(3)$ se déduit par cloture sub-additive (on la notera $\overline{\alpha_u(3)}$), et on peut s'attendre à avoir $\alpha_u(3) \geq \alpha_u(2)$ (à supposer que l'on ait correctement déterminé $\alpha_u(2)$). L'heuristique est donc d'effectuer la recherche dichotomique entre $\alpha_u(2)$ et $\overline{\alpha_u(3)}$. La courbe obtenue n'est pas forcément la meilleure : on peut avoir mal déterminé la valeur de $\alpha_u(2)$ et obtenir une valeur de $\alpha_u(3)$ plus petite. Ce cas de figure peut se présenter et entraîne alors une perte de précision dans le résultat.

Cette technique permet de réduire significativement le nombre d'analyses effectuées, et apporte donc un facteur d'accélération assez important, comme nous le verrons dans la partie suivante.

6.1.2 L'observateur non-déterministe

Une autre optimisation que j'ai apportée durant ce stage est l'utilisation d'un observateur non-déterministe. Intuitivement un observateur non déterministe observe une seule fenêtre de temps d'une durée fixée. Cette fenêtre de temps peut débuter à n'importe quel moment. En pratique un oracle déclenche le début de la période d'observation. L'allure du code d'un observateur non-déterministe pour une fenêtre de temps n, pour une courbe haute, et pour valider un point de valeur 10 par exemple, est la suivante :

```
node NDET (const n, flot : int; oracle : bool)
returns (property_proved : bool);
var A : bool^n; compteur : int;
let
  A[0] = oracle -> oracle or pre (A[0]);
  --A est un tableau de n booléens qui sert à
  --modéliser une fenêtre de temps de taille n
  -- A[0] vaudra (et restera) true dès que
  -- oracle vaudra true.

  A[1..n-1] = (false^(n-1)) -> pre (A[0 .. n-2]);
  -- A[i] sera égal à A[0] décalé de i unités de temps.

  compteur = 0 -> if(A[0] and not A[n-1])
            then pre (compteur) + flot
            else pre (compteur);

  property_proved = (not A[n-1]) or compteur <= 10;
  -- tant que A[n-1] est faux, on n'a pas fini
  -- de compter les événements, donc il ne faut pas
  -- tester la valeur de compteur.
tel
```

Fig. 6.1 – Code Lustre d'un observateur non-déterministe

L'observateur non-déterministe permet de simplifier le noeud à une seule variable numérique, ce qui accélère asymptotiquement la recherche de la meilleure courbe. Il est plus compliqué que l'observateur déterministe dans les cas simples, mais s'avère plus efficace si on va loin dans la recherche de la meilleure courbe. Plus précisément, l'observateur non-déterministe s'avère plus efficace pour prouver qu'une propriété est vraie, mais moins efficace en cas d'échec de la preuve. Pour la recherche dichotomique, les résultats sont assez mitigés étant donné qu'une grande partie de l'algorithme consiste à "trouver le premier point tel que la propriété est vraie".

On remarquera que l'observateur non-déterministe ne peut être placé qu'en sortie. En effet dans le cas déterministe, *NBac* prouve la propriété $\forall F, \forall t, [I(F(t)) \Rightarrow O(S(F(t)))]$ comme expliqué dans la partie précédente. Si l'observateur d'entrée est non-déterministe, on donnera à *NBac* la propriété $\forall F, \forall oracle, \forall t, [I(F(t), oracle) \Rightarrow O(S(F(t)))]$. Or ce n'est pas équivalent à la propriété $\forall F, [\forall oracle, \forall t, I(F(t), oracle)] \Rightarrow [\forall t, O(S(F(t)))]$, qui

est elle équivalente à la propriété $\forall F$, $\exists oracle$, $[\forall t, I(F(t), oracle)] \Rightarrow [\forall t, O(S(F(t)))]$. Le fait d'avoir le symbole $\exists oracle$ indique qu'on a là une propriété de vivacité et non de sûreté du système, ce que *NBac* ne sait pas prouver.

6.2 Performances

On présente dans cette section les temps d'éxécution de l'outil `ac2lus` dans sa version observateur d'entrée sur plusieurs programmes test. On donne au préalable une description de ces programmes.

6.2.1 Programmes de test

On présente ici les programmes qui ont été analysés par *ac2lus* pour les mesures de performances.

Retardateur

Un retardateur (ou delayer) est un exemple de programme trivial à écrire en Lustre, et difficile à exprimer dans le formalisme RTC. Il consiste à retarder un flux d'une unité de temps. Le code source est donné sur la figure 6.2 :

```
node delayer (flot : int) returns (output : int);
let
        output = 0 -> pre input;
tel
```

FIG. 6.2 – Code Lustre d'un retardateur

Power manager

On s'intéresse à un power manager. Il contient deux modes : veille et marche. En veille, il accumule les paquets d'événements, et se réveille quand il en a reçu un certain nombre (5 dans l'exemple). Alors il les traite à une vitesse fixe (5 événement par unité de temps), puis quand sa pile est vide, il se rendort. Le code source est donné sur la figure 6.3 :

6.2.2 Temps d'éxécution

Le tableau ci-dessous donne les temps d'éxécution de *ac2lus* pour la courbe d'arrivée représentée sur la figure 5.2 dont on rappelle qu'elle correspond aux courbes suivantes : $\alpha_u = [0, 3, 5, 7, 8]$ et $\alpha_l = [0, 1, 2, 4, 5]$.

```
node power_manager (flot : int) returns (output : int);
var buffer : int; veille : bool;
let
  buffer = flot -> pre (buffer) + flot - pre output;
  veille = flot <= 4 -> if (pre (veille) and buffer >= 5)
                then false
       else if (not pre (veille) and buffer = 0)
                then true
       else pre (veille);
  output = if veille then 0
       else if buffer >= 5 then 5
       else buffer;
tel
```

FIG. 6.3 – Code Lustre d'un power manager

	delayer	power manager
Pas d'optimisation	6	44
Observateur non déterministe	36	170
Heuristique	11	15
Observateur non déterministe + heuristique	33	82

FIG. 6.4 – Temps d'éxécution (en secondes) pour calculer un point de la courbe

	delayer	power manager
Pas d'optimisation	41	464
Observateur non déterministe	354	non mesuré
Heuristique	44	83
Observateur non déterministe + heuristique	75	non mesuré

FIG. 6.5 – Temps d'éxécution (en secondes) pour calculer trois points de la courbe

	delayer	power manager
Pas d'optimisation	326	1113
Observateur non déterministe	893	non mesuré
Heuristique	110	178
Observateur non déterministe + heuristique	157	non mesuré

FIG. 6.6 – Temps d'éxécution (en secondes) pour calculer cinq points de la courbe

Chapitre 7

Changer d'abstraction

Cette partie décrit le problème qui nous a motivé durant ce stage. Elle présente en premier lieu les motivations qui ont amené à proposer une abstraction pour décrire les flots d'événements entre les systèmes, puis la démarche globale qui a dirigé les travaux décrits dans les parties suivantes.

7.1 motivations

Dans l'outil *ac2lus* et dans les outils analogues (par exemple sur les automates temporisés dans CATS), on essaie de connecter des méthodes d'analyse de modèles computationnels au Real-Time Calculus, comme le montre la figure 7.1. Or, un automate (ou un programme Lustre) contient beaucoup plus d'informations sur le comportement du système qu'il décrit qu'une paire de courbes d'arrivée correspondant à la sortie du système. En particulier, la notion de retard d'un signal est très mal exprimée en RTC alors qu'elle est triviale en Lustre. La notion de mode d'économie d'énergie est très peu représentable en RTC alors qu'elle s'exprime très naturellement avec un automate.

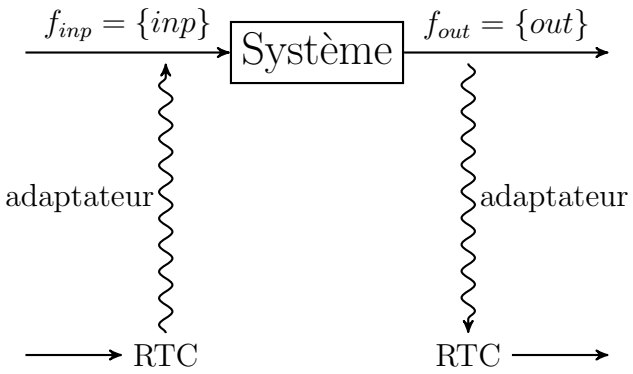


FIG. 7.1 – Principe de la connexion entre RTC et les modèles computationnels

Lors d'une analyse, la complexité du problème peut exploser. Pour éviter cela, on fait des abstractions, et donc on accepte de perdre de l'information sur le système qu'on étudie. La question que nous nous sommes posée est "Pourquoi s'obstiner à vouloir repasser dans

le formalisme RTC ?" Certes, une paire de courbes d'arrivée obtenue avec *ac2lus* est une abstraction du comportement d'un programme Lustre, mais cette abstraction n'est-elle pas trop forte ? Ne perd-on pas trop d'informations sur le comportement du système, notamment la notion d'état ?

7.2 Démarche globale

Nous avions envie de changer d'abstraction pour représenter les flots d'événements, et pour ce faire il fallait introduire des objets pour représenter ces flots dans un certain formalisme, et être capable de déterminer dans ce formalisme la transformation d'un flot par un système. La figure 7.2 illustre cette démarche. Sur cette figure, on a un système exprimé dans un certain formalisme, et deux objets f_{inp} et f_{out} qui représentent l'ensemble des flots d'entrée et de sortie du système.

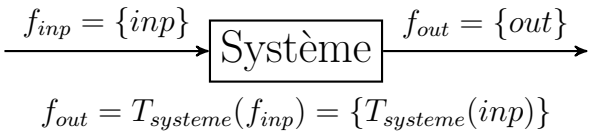


FIG. 7.2 – Illustration de la démarche globale

f_{inp} et f_{out} sont liés par la relation $f_{out} = T_{systeme}(f_{inp})$, sachant que $T_{systeme}$ est décrite dans le système (on dit que $T_{systeme}$ est la fonction de transformation du flot par le système, ou bien que le système est un transformateur). Mais si on fait un calcul exact de f_{out} connaissant f_{inp}, on se heurte au problème de l'explosion de la complexité du problème : si on manipule des automates par exemple, on a une explosion du nombre d'états et de variables. Il faut donc en déterminer une abstraction, donc perdre de l'information sur f_{out} afin de limiter sa complexité. Ce sera l'un de nos objectifs dans la suite de ce rapport.

Pour déterminer une abstraction de la représentation du flot de sortie f_{out}, nous avons utilisé l'outil d'interprétation abstraite *NBac*, qui est beaucoup utilisé à Verimag. La partie suivante décrit les premiers essais à la main, ainsi qu'un embryon de méthode d'abstraction qui s'est avéré trop compliquée, et les raisons qui nous ont poussé à développer notre méthode d'analyse de système basée sur le moteur d'analyse de *NBac*.

Chapitre 8

Premiers essais

Dans la partie précédente, nous avons présenté une démarche globale. Le but est d'utiliser NBac pour déterminer une abstraction d'un ensemble de flots de sortie f_{out} à partir d'un système et d'un ensemble de flots d'entrée f_{inp}, comme illustré sur la figure 7.2 page 45. Cette partie décrit l'ensemble des travaux effectués dans ce sens qui n'ont pas réellement abouti, et les raisons pour lesquelles nous avons décidé d'implémenter un outil ab nihilo pour mettre en oeuvre notre technique. Cette partie assez technique décrit chronologiquement l'évolution de la manière dont nous avons abordé le problème, ainsi que les diverses idées que nous avons eu. Elle n'est pas indispensable pour comprendre la suite du rapport, puisque la partie suivante décrit la solution retenue, qui est issue des essais que nous allons maintenant décrire.

8.1 Premières expériences avec NBac

On peut utiliser NBac sans préciser de structure de contrôle. Charge à NBac d'en déterminer une à partir d'un système de transitions temporel sur les variables du problème. C'est le cas lors de l'analyse de programmes en Lustre. On peut également spécifier une structure de contrôle et laisser NBac faire une analyse d'accessibilité dessus. C'est ce qui nous intéresse ici. On décrira dans cette section le langage *AutoC* qui nous a permis de donner des structures de contrôle à NBac, puis un premier outil que nous avons implémenté et les résultats obtenus.

8.1.1 L'utilisation de l'outil *NBac*

En utilisant *ac2lus*, nous faisions des appels récurrents à *NBac* pour effectuer la partie d'analyse dans la recherche dichotomique. L'impression que nous avions était que *NBac* effectuait à chaque fois à peu près la même chose (les invariants déterminés pour chaque états sont a priori les mêmes lors de chaque analyse). Il est possible de ne pas donner de propriété à prouver à *NBac*, l'analyse se limite alors à la détermination d'invariants pour le système.

Nous avons donc cherché comment déterminer à la main une représentation du flot de sortie du système constitué d'un observateur d'entrée (écrit en Lustre) sur le flot d'entrée d'un composant lui aussi écrit en Lustre, directement à la suite de l'analyse de ce système par *NBac*. Nous nous sommes aperçus que la structure de contrôle générée par

NBac à partir d'un programme Lustre n'était pas intuitive. Nous avons alors essayé de spécifier nous-mêmes une structure de contrôle, en utilisant le format AutoC. Ces essais sont décrits dans la partie suivante. Dans la suite de cette partie, on proposera un format d'automates qui permet de décrire des flots d'événements. On proposera également une méthode permettant d'effectuer l'analyse d'un système dont on décrit le flot d'entrée par un automate de ce genre.

Ces propositions, qui sont des heuristiques, sont issues des essais effectués à la main avec *NBac* durant ce stage. Finalement, le format d'automates introduit ici, trop compliqué, a été modifié vers un format plus simple qui sera dans la version finale et que nous décrirons dans la partie suivante.

8.1.2 Le langage *AutoC*

Le langage *AutoC* a été développé par Bertrand Jeannet. Il permet de décrire des automates hybrides à rendez-vous valué. Nous n'avons pas utilisé l'aspect hybride (qui consiste à autoriser plusieurs horloges à des vitesses différentes dans un même automate) qui ne nous intéressait pas dans notre situation. *AutoC* permet de faire des produits asynchrones d'automates qui peuvent être synchronisés sur des canaux. Lors de ce produit, on peut expliciter la structure de contrôle de l'un des automates et traduire les états de l'autre automate en variables. On appelera cela le *produit implicite*. *AutoC* a pour cela un format intermédiaire appelé *Auto* qui a a peu près la même syntaxe et décrit l'automate produit. Une documentation succinte du langage *Autoc/Auto* est proposée en annexe de ce rapport. Nous reviendrons dans la partie suivante sur la notion de produit implicite.

Auto a l'avantage d'être interfacé avec NBac, de telle sorte qu'on peut générer un automate analysable par NBac avec un automate décrit en Auto, et que l'on peut "annoter" en retour le fichier *Auto* a partir de l'analyse de NBac. Nous nous sommes cependant aperçus que les anotations effectuées sur le fichier Auto étaient moins précises que les invariants donnés en sortie de NBac (peut-être à cause d'un bug de l'outil *nbac2auto*). Nous n'avons donc pas utilisé de fichiers Auto annotés.

Dans la syntaxe du langage AutoC, on inclue les synchronisations sur les canaux dans les transitions. Par exemple une transition qui se fait sur synchronisation sur le canal $input()$ ayant pour garde δ et pour action λ sera notée $\{\delta \text{ and } sync\ input()\}, \{\lambda\}$. Par convention, on mettra un canal $start()$ dans la première transition, ce qui permettra à tous les automates que l'on considère de "démarrer en même temps".

Tous les essais présentés dans cette partie sont partis d'automates codés en AutoC. AutoC ne permet pas de modéliser des systèmes synchrones directement. Nous avons donc, dans nos essais, synchronisé les systèmes étudiés grâce à des rendez-vous sur des canaux. C'est en grande partie ce problème qui nous a fait abandonner cette piste, ce que nous verrons par la suite.

8.1.3 Premier format d'automates permettant de décrire un flot

Pour décrire un flot de données, nous avons voulu reprendre le modèle des *Event Count Automata* (ou ECA), qui ont été présentés dans le chapitre 5. On rappelle que les ECA sont des automates qui observent un flot (que l'on notera *input*), et qui sont dotés d'un ensemble de compteurs δ_i, de telle sorte qu'après chaque transition (y compris chaque

nouvelle affectation des compteurs), on ait la nouvelle affectation $\delta_i \leftarrow \delta_i + input$. On notera que les ECA sont des accepteurs. Un flot non accepté fait passer l'ECA dans un état erroné.

Tel quel, les ECA ne peuvent pas être codés en AutoC, ce qui était notre but. Nous avons donc modifié leur syntaxe en explicitant cette mise à jour des compteurs. Nous avons également utilisé le fait qu'AutoC est un format d'automates à rendez-vous valué, mais qu'il n'a pas de notion d'entrée (un automate ne peut pas avoir de variable d'entrée comme dans Lustre ou *NBac*). Nous avons donc contourné le problème en synchronisant l'automate considéré sur un canal à valeur entière.

On donne sur la figure 8.1 un exemple d'automate codé en AutoC, ainsi qu'une représentation de cet automate. Il décrit un ensemble de flots correspondant à la courbe d'arrivée finie $\alpha_u = [0, 5, 9, 14]$ et $\alpha_l = [0, 0, 1, 2]$.

8.2 Exemple d'analyse de système avec *NBac*

8.2.1 Programme test

Le système étudié pour les tests est le delayer étudié avec l'outil *ac2lus*, et qui est présenté à la page 42. On l' exprimera ici sous forme d'automate sur la figure 8.2. On prendra en entrée l'automate décrit sur la figure 8.1. On prendra pour convention que la variable d'état correspondant à la sortie du système sera notée *sortie*, et que les canaux d'entrée et de sortie seront notés *input* et *output*.

8.2.2 Résultats obtenus après analyse par *NBac*

On a effectué un produit d'automates totalement explicite entre l'automate de la figure 8.1 et le delayer représenté sur la figure 8.2. On a ensuite soumis ce produit d'automates à NBac pour déterminer pour chaque couple d'état un domaine dans lequel l'ensemble des variables des deux automates vivent.

8.2.3 La nécessité d'un outil d'analyse des sorties de *NBac*

NBac permet de faire une abstraction d'un système représenté par un automate en déterminant pour chaque état un invariant. Cependant cet invariant peut rapidement être compliqué et difficile à intuiter. Nous avons donc développé un premier outil d'analyse des sorties de NBac pour nous assister dans nos expériences visant à trouver des abstractions des flots de sortie.

Dans notre premier outil d'analyse nous avons implémenté un parseur succint (essentiellement un lecteur d'expressions linéaires) pour un fichier produit par NBac. Nous avions déjà un opérateur de produit (l'outil *autoc2auto*), il restait à faire un outil de projection sur des variables choisies par l'utilisateur. C'est donc ce que nous avons fait. L'outil terminé faisait 1000 lignes de c++. Ce module sera repris dans *ROBERT*, nous le décrirons donc plus en détail dans la partie suivante. Enfin, l'outil pouvait afficher les résultats sous la forme d'un automate au format *dot*.

Pour illustrer le fonctionnement de cet outil, on donnera ici les résultats finaux de l'exemple précédent sur la figure 8.3. Ici, on n'a gardé que les variables de sortie.

8.3 Les limites de cette approche

Il était assez délicat de décrire des systèmes synchrones par des automates à rendez-vous. Il fallait en effet respecter "manuellement" certaines propriétés qui sont triviales sous l'hypothèse synchrone, notamment le fait d'écrire sur les sorties après avoir lu sur les entrées (ce qui n'est pas un problème facile lorsqu'un système a plusieurs sorties et/ou plusieurs entrées). Dans certains cas, le simple ordre de lecture/écriture pouvait entraîner un blocage du système. De plus, l'utilisation de canaux de synchronisations nuisait beaucoup à la rapidité de l'analyse de *NBac*, ce qui ne se produisait pas pour des automates de complexité similaire sans rendez-vous. Pour ces raisons, nous n'avons pas voulu continuer sur cette piste, que nous avions initialement empruntée parce que nous avions des outils déjà existants (composition d'automates, langage *Auto* connecté à *NBac*, etc...).

Nous avons donc décidé de créer l'outil *ROBERT*, qui implémente les outils que nous avions dans le format *AutoC*, directement dans le format de *NBac*. La partie suivante décrit plus précisément la méthode finale retenue pour déterminer à partir d'un flot d'entrée d'un système, une abstraction du flot de sortie du système. Puis on décrira dans une autre partie l'outil *ROBERT* plus en détail.

```
channel input(int);
channel start();

process f_inp_representation {
sync input, start;
var m :int ;m2 :int ;
init pc = linit;
loc linit :
  when  true sync start() goto l0 assign { m  := 0; m2 :=0 };
loc l0 :
  when  0<=x and x<=5 sync input(x) goto l1 assign { m  := x };
loc l1 :
  when  0<=x and x<=5 and  1<= x+m and x+m <= 9
        sync input(x) goto l2 assign {m2 :=m;  m  := x};
loc l2 :
  when  0<=x and x<=5 and  1<= x+m and x+m <= 9
        and 2<= x+m+m2 and x+m+m2 <= 14 sync input(x)
        goto l2 assign {m2 :=m;  m  := x};
}

system f_inp_representation;
explicit f_inp_representation;
```

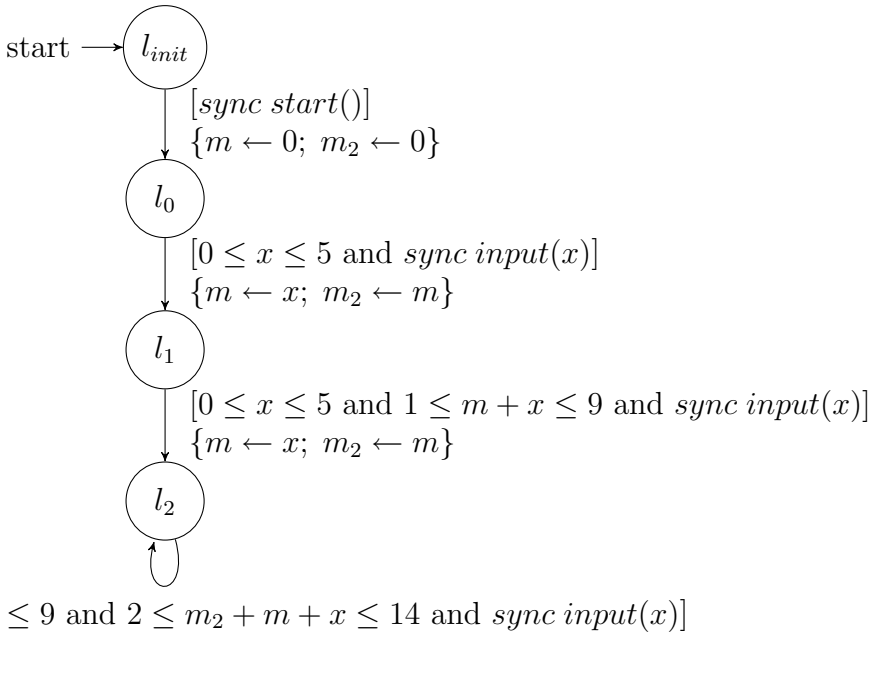


FIG. 8.1 – Automate décrivant un ensemble de flots d'événements, en code AutoC et représenté schématiquement

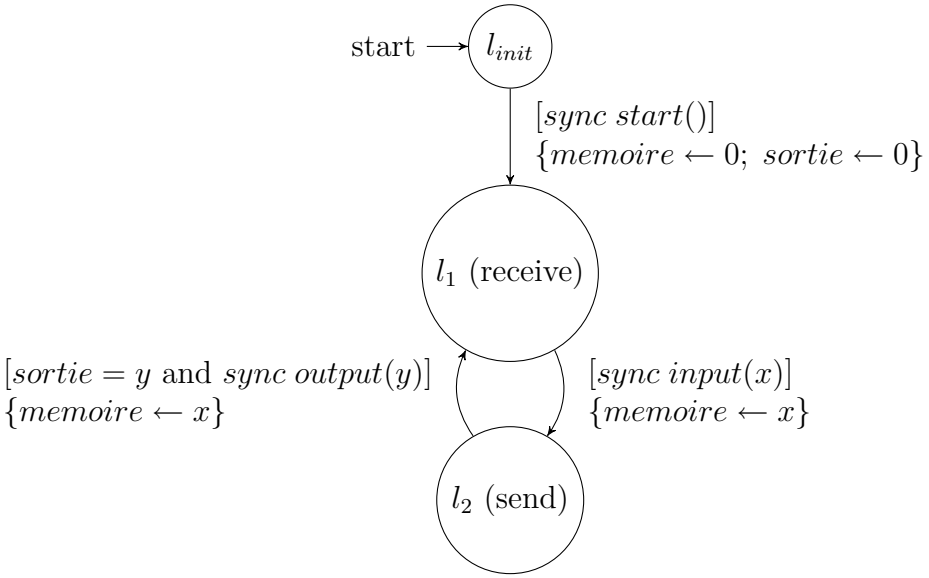


FIG. 8.2 – Représentation d'un delayer

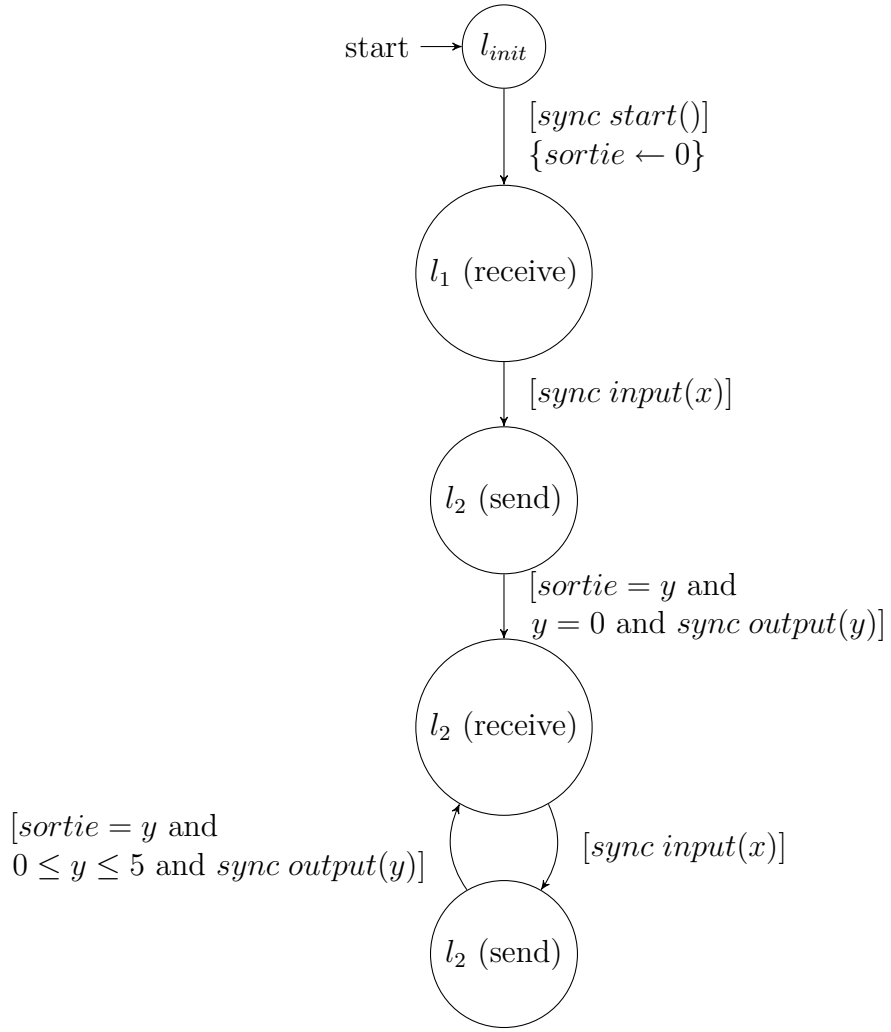


FIG. 8.3 – Résultat final après analyse d'un delayer

Chapitre 9

Approche adoptée

Cette partie décrit la méthode qui a été développée durant ce stage. Les choix qui ont été faits proviennent de l'étude de la bibliographie, des essais décrits dans la partie précédente et des conseils de personnes travaillant ou ayant travaillé à Verimag.

Le but est d'expliciter tous les éléments exposés sur la figure 9.1 qui a été présentée précédemment. On décrira tout d'abord le format d'automates utilisé pour représenter les flots d'événements (f_{inp} et f_{out}), puis une première méthode "naïve" permettant de calculer la transformation d'une représentation d'un flot par un système ($T_{systeme}(f_{inp})$), puis d'en déterminer une abstraction grâce à *NBac*. On terminera sur l'ajout à cette méthode d'heuristiques permettant de guider NBac à la main afin d'obtenir des résultats exploitables par un humain. La figure 9.2 illustre les différentes étapes de la méthode "naïve" de calcul de l'abstraction du flot de sortie d'un système, qui sera présentée dans la deuxième section. La figure 9.3 montre des raffinements de cette méthode "naïve" amenant à une méthode plus complète, fournissant des résultats plus exploitables. Ces techniques de raffinement, ainsi que le calcul exact du flot de sortie (l'opérateur de composition) ont été implémentés dans l'outil *ROBERT*, que nous présenterons dans la partie suivante.

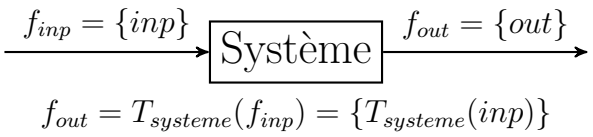


FIG. 9.1 – Illustration de la méthode proposée

9.1 Proposition d'une nouvelle abstraction pour décrire les flots de données

Nous proposons d'abstraire les flots de données circulant sur des canaux par des automates dits *automates à compteurs*. Cette section décrit les raisons qui nous ont amené à adopter ce format d'automates, puis la syntaxe de ces automates.

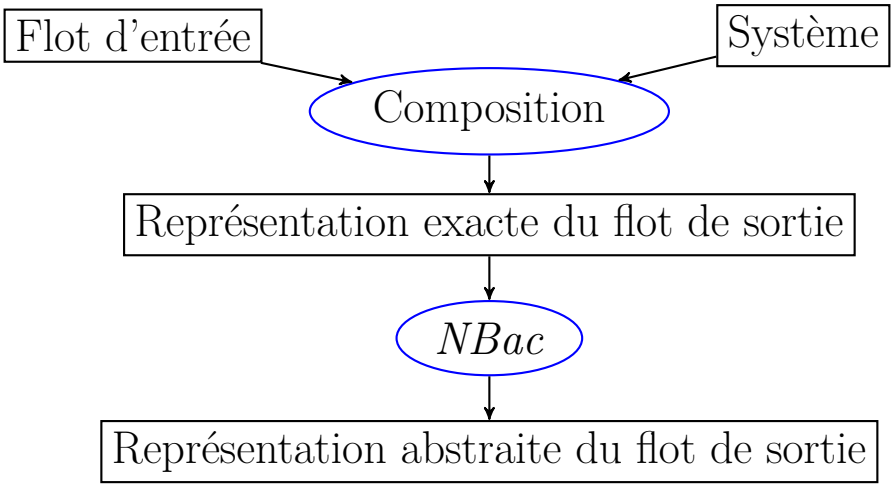


FIG. 9.2 – Vue d'ensemble de la méthode "naïve" de calcul d'une abstraction du flot de sortie d'un système

9.1.1 Principe

Nous avons décidé de reprendre le principe de l'observateur synchrone utilisé dans *ac2lus*, en nous inspirant des *Event Count Automata* qui sont présentés dans la partie 5.

On décrira les flots de données par des automates dits *accepteurs*, c'est à dire dont l'ensemble des traces acceptées est l'ensemble des traces ne menant pas à un état erroné. Sur les représentations de ces automates, l'état erroné est implicite. Les Event Count Automata présentés dans la partie 5 sont pratiques pour décrire des flots, mais ils ne sont pas programmables en tant que tel dans le format *NBac*. Nous nous sommes donc inspirés de ce format pour écrire des automates qui décrivent des flots, et qui sont programmables dans le format *NBac*.

9.1.2 Description des automates à compteurs

La syntaxe des automates à compteurs est celle du format d'entrée de *NBac*, qui est décrite dans [Jea]. Les automates à compteurs représentent un sous ensemble des automates analysables par *NBac*. Ce sont des automates décorés des variables suivantes :

- Une variable d'entrée notée en général *flot* qui correspond au flot d'événements observés par l'accepteur. Cette variable se remet à jour à chaque transition (dans certains exemples nous notons ce flot *output* en faisant référence à un flot de sortie).
- Une série de compteurs notés en général m_i (ou n_i quand on voudra décrire un flot de sortie) qui mesurent le nombre d'événements arrivés depuis i unités de temps. On aura aux initialisations près, lors de chaque transition l'affectation $m_0 \leftarrow input$, $\forall i \geq 1$, $m_i \leftarrow m_{i-1} + input$.

A priori, rien n'empêche un automate à compteurs d'avoir des variables internes qui n'ont aucun rapport avec le flot. Nous n'en avons cependant pas utilisé dans nos exemples.

La figure 9.4 donne un exemple d'automate à compteurs. On a choisi ici de représenter le même ensemble de flots que l'automate à rendez-vous présenté sur la figure 8.1 page 50.

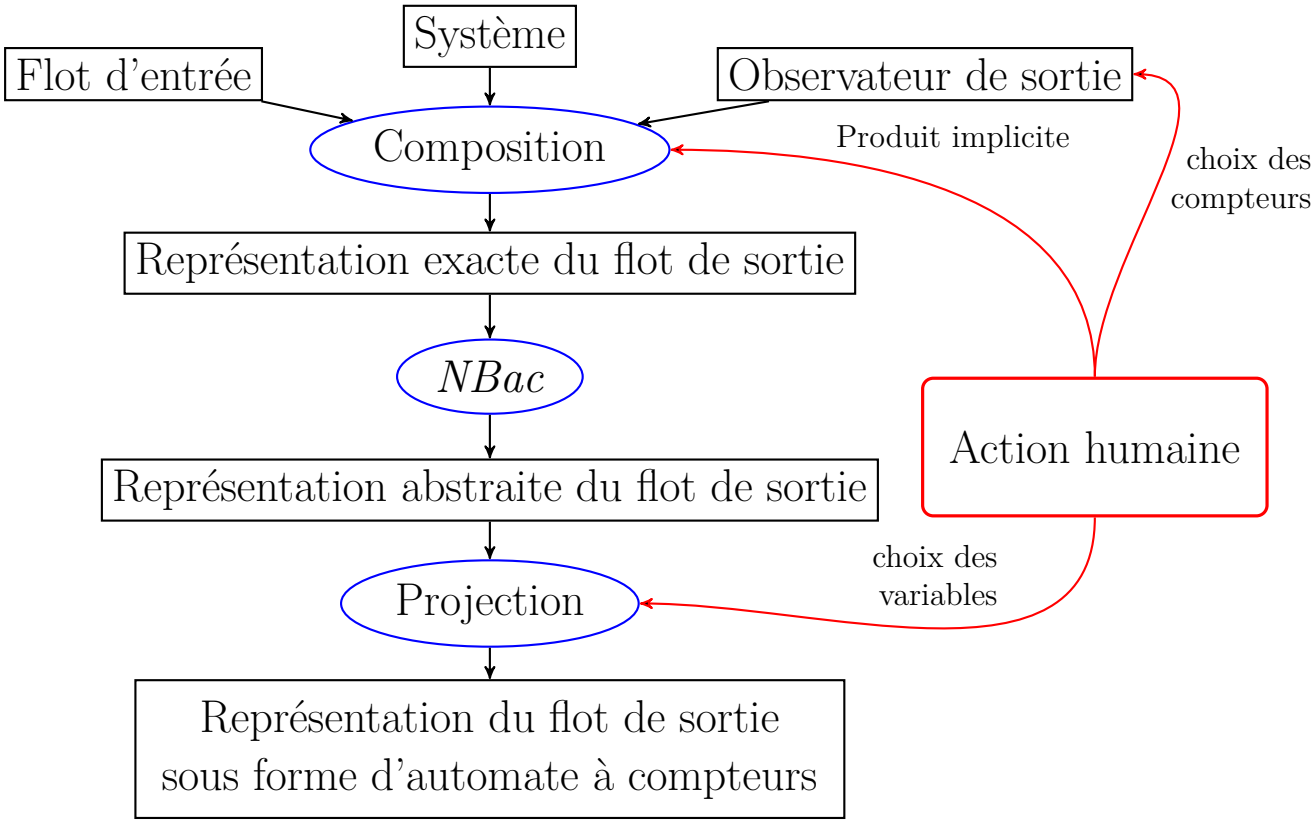


FIG. 9.3 – Vue d'ensemble de la méthode globale de calcul d'une abstraction du flot de sortie d'un système

On rappelle qu'il décrit également l'ensemble de flots décrit par la courbe d'arrivée finie $\alpha_u = [0, 5, 9, 14]$ et $\alpha_l = [0, 0, 1, 2]$.

9.2 Proposition d'une méthode "naïve" d'abstraction et de transformation d'un automate à compteurs

9.2.1 Principe

La transformation d'un flot décrit par un automate à compteurs par un système représenté par un automate est un produit synchrone d'automates. On notera que le système doit être un automate dont la syntaxe et la sémantique doivent être compatibles avec *NBac*.

Nous allons donc utiliser *NBac* pour déterminer une abstraction de ce produit synchrone. Pour cela, on donne à *NBac* le produit synchrone de l'automate à compteurs d'entrée et du système. Une option de *NBac* lui permet de faire une analyse sans avoir de propriété à prouver (on se réfèrera à l'annexe technique sur *ROBERT* pour plus d'informations). Concrètement, *NBac* détermine pour chaque état un invariant, c'est à dire un domaine dans lequel l'ensemble des variables de l'automate va vivre. L'abstraction du flot de sortie que nous obtenons alors est la suivante : pour chaque transition, la garde est égale à

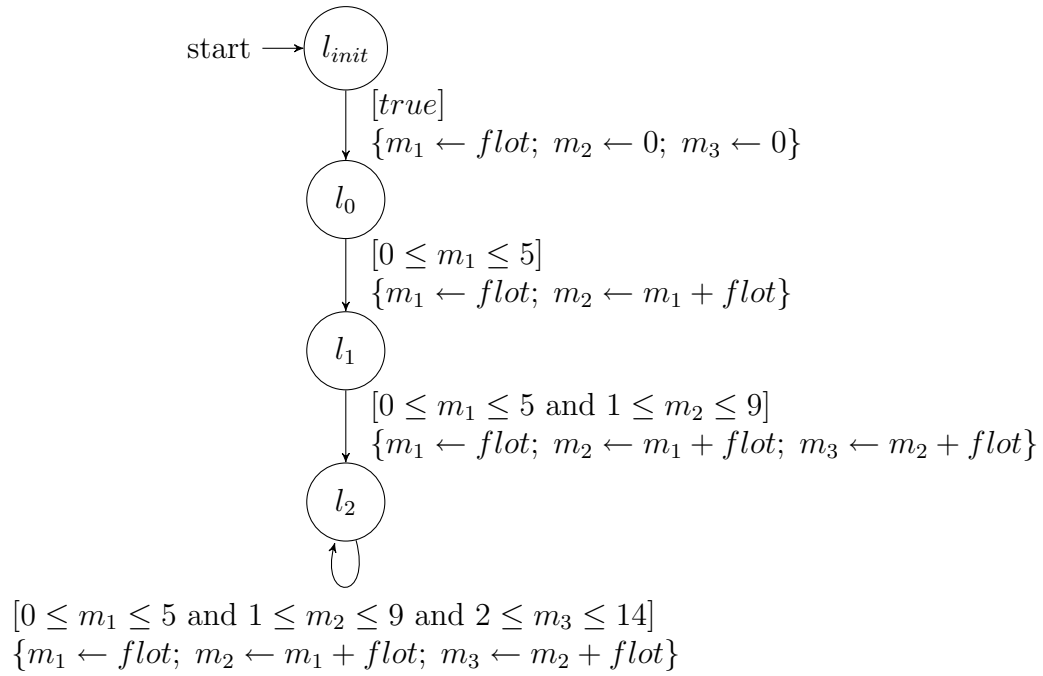


FIG. 9.4 – Exemple d'automate à compteurs

l'invariant de l'état de départ, l'action ne change pas.

Cette méthode possède plusieurs inconvénients :

- Elle conserve toutes les variables internes du système, alors que certaines sont clairement superflues. Il faut un moyen de ne conserver que les variables qui nous intéressent.
- Elle ne sort pas directement un automate à compteurs (lorsqu'on décrit le système, on n'écrit pas les compteurs sur sa sortie). Nous avons résolu ce problème par l'ajout d'un *observateur de sortie* (nous préciserons leur définition, qui est différente de celle des observateurs de sortie utilisés dans *ac2lus*), que nous présenterons dans la section 9.3.2
- elle est sensible au phénomène d'explosion de complexité du problème (par exemple, une explosion du nombre d'états). Nous verrons donc quelques méthodes heuristiques permettant de limiter cette explosion de complexité.

Le résultat obtenu n'est donc pas directement exploitable à la main. Nous avons donc développé des méthodes heuristiques qui viennent compléter cette méthode pour produire en sortie des automates à compteur correctement formatés, pour maîtriser l'explosion de la complexité, et pour ne garder qu'un certain nombre de variables "pertinentes" dans l'automate produit en sortie.

9.2.2 Exemple d'application de la méthode naïve

On reprend l'exemple du power manager déjà utilisé dans les tests sur ac2lus. On en rappelle le principe : ce système a deux états *sleep* et *busy*, et une file d'attente (une variable *buffer*) qui dans l'état *sleep* se remplit avec le flot entrant, dès qu'elle dépasse

un certain seuil (ici 5), le système passe dans l'état *busy*, et traite la file d'attente à une vitesse constante (5 événements par unité de temps), jusqu'à ce qu'il ait traité toute sa file d'attente, auquel cas il retourne dans l'état *sleep*. On donne sur la figure 9.5 une représentation de ce système. On notera le flot d'entrée *input* et le flot de sortie *output* (le flot de sortie sera une variable d'état).

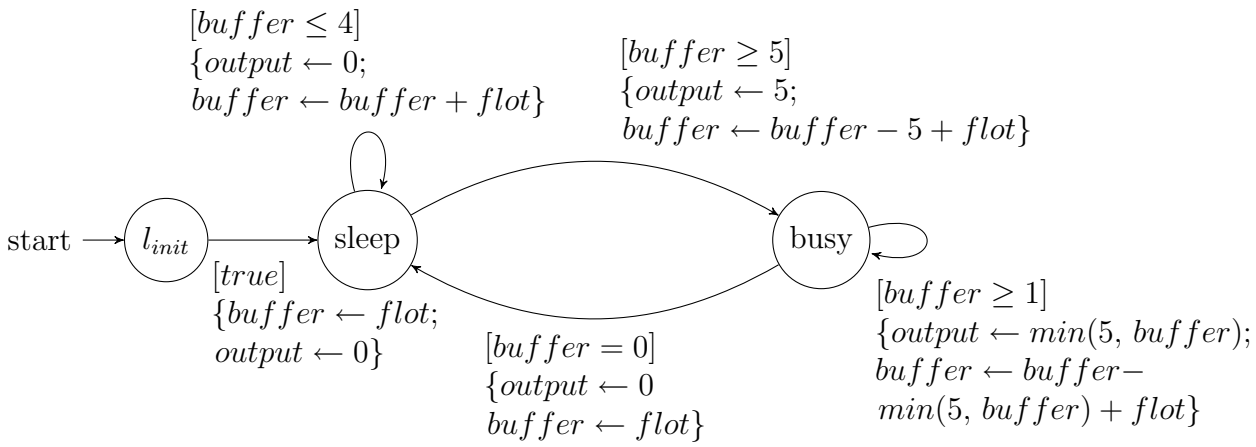


FIG. 9.5 – Représentation d'un power manager

On effectue le produit synchrone de ces deux automates (qui est représenté sur la figure B.1 page 78), puis on l'analyse avec *NBac*. Le résultat après analyse est donné sur la figure B.2 page 79.

La méthode d'abstraction proposée est la suivante : *NBac* détermine un invariant du système qu'il analyse (plus précisément un invariant pour chaque état de la structure de contrôle). On obtient une abstraction du système en partant de sa structure de contrôle et en remplaçant les gardes des transitions par les invariants de leurs états de départ.

L'abstraction du système obtenue par cette méthode naïve n'est pas intuitive, ni même réutilisable. La section suivante montre des méthodes basées sur des heuristiques permettant d'exploiter ce résultat pour trouver au final un automate à compteurs plus intuitif, et réutilisable.

9.3 Raffinements de la méthode naïve, vers une méthode plus complète

Cette section décrit l'ensemble des méthodes permettant de tirer parti de la méthode précédente. Ces méthodes sont essentiellement basées sur des heuristiques, et nécessitent l'action d'un être humain. L'idée générale est d'aider *NBac* à donner un résultat exploitable en modifiant astucieusement l'automate qu'on lui demande d'analyser.

ces méthodes sont en partie implémentées dans l'outil *ROBERT* que nous verrons dans la partie suivante. Nous proposerons successivement des solutions pour contrôler l'explosion du nombre d'états, pour ajouter des compteurs sur la sortie du système, et enfin pour éliminer les variables qui ne sont pas pertinentes dans le résultat final. Nous poursuivrons l'étude du power manager en ajoutant au fur et à mesure ces méthodes.

9.3.1 Le produit implicite

Un problème qui peut survenir lorsque l'on fait l'analyse de plusieurs systèmes concaténés est l'explosion du nombre de variables et/ou du nombre d'états. De nombreux travaux visent à trouver des méthodes pour limiter cette explosion, donc à trouver de nouvelles abstractions, on citera par exemple [MBBS] qui présente une technique visant à abstraire des automates temporisés en introduisant astucieusement des horloges dans le systèmes et en projetant l'automate résultant sur ces horloges.

Le produit implicite (parfois appelé *produit projeté*) est une variante du produit synchrone d'automates qui permet d'éviter l'explosion du nombre d'états. L'idée est lors du produit, d'expliciter la structure de contrôle de l'un des deux automates, et d'exprimer la structure de contrôle de l'autre automate sous forme de variables. Ce produit est exact (on ne perd pas d'informations), et de la complexité d'un des deux automates en nombre d'états. On peut étendre cette définition à un produit de $n \times m$ automates, avec n automates explicités, et m automates qui sont implicites. La structure de contrôle finale correspond à celle du produit explicite des n automates.

NBac analyse un programme qui décrit un système représenté par un automate, et se base sur une structure de contrôle. Nous déterminons une abstraction du système étudié par la même méthode que précédemment. Le rôle joué par la structure de contrôle est alors important : si on donne à *NBac* une structure de contrôle donnée, on obtient en sortie cette structure de contrôle analysée. Faire tourner NBac sur un produit implicite permet donc en sortie d'avoir une abstraction de ce produit implicite. On perd de l'information par rapport au produit synchrone explicite, puisque NBac détermine pour chaque état de la structure de contrôle un invariant qui prend en compte des informations dépendant de plusieurs états de l'automate qui est implicite. La figure 9.6 montre le résultat de l'analyse de *NBac* du produit implicite entre l'automate à compteurs de la figure 9.4, et le power manager de la figure 9.5. On a explicité la structure de contrôle du power manager. On appelle *pc* la variable qui prend comme valeur les états de l'automate à compteurs.

Plusieurs questions se posent alors :

- Est-ce que le choix d'expliciter l'un ou l'autre des automates a une influence sur la précision des résultats ? La réponse semble être oui. Par exemple, rendre implicite un système du type power manager fait perdre la notion de mode veille dans le système après analyse, puisque *NBac* considère pour chaque état de la structure de contrôle les deux possibilités (veille et réveillé).
- Y a-t-il un critère permettant alors de choisir quel automate on va rendre explicite ? Nous n'avons pas de réponse précise. Intuitivement, si le système a un comportement très différent d'un état à l'autre, il est préférable de l'expliciter pour retrouver ces comportements très différents dans le résultat final.
- Y a-t-il une structure de contrôle plus pertinente que celle de l'un des deux automates initiaux ? La question est ouverte. Nous n'avons pas étudié ce problème.

9.3.2 L'observateur de sortie

Un défaut de la méthode "naïve" est qu'elle ne retourne pas à priori d'automate à compteurs. Lorsqu'on décrit un système par un automate, on n'écrit pas de compteurs sur la sortie puisqu'ils ne servent à rien pour décrire le système.

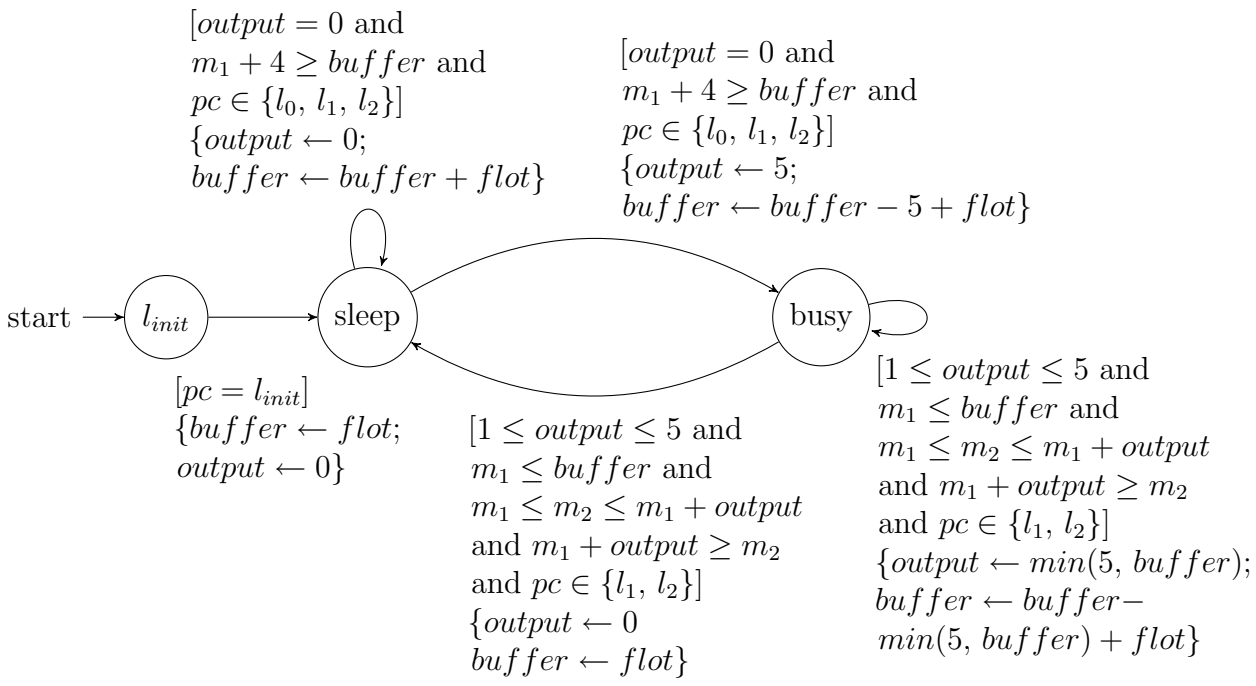


FIG. 9.6 – Analyse par *NBac* du produit implicite entre un power manager et un automate à compteurs

Nous avons introduit un objet que nous avons appelé *observateur de sortie* (bien que cette définition soit différente de celle utilisée dans *ac2lus* pour la recherche dichotomique). Le but d'un observateur de sortie est d'ajouter des variables que l'on juge pertinentes pour décrire le flot de sortie du système. Typiquement, pour avoir en sortie un automate à compteurs, il est judicieux de définir au préalable un observateur de sortie qui contient les compteurs. La figure 9.7 montre un exemple d'observateur de sortie observant un flot (ici noté $flot$, mais qui sera égal à *output* dans l'implémentation de l'exemple) et qui contient des compteurs (ici notés n_1, n_2, n_3). Pour ajouter cet observateur de sortie au système, il suffit de faire un produit implicite entre l'observateur et le système (où le système est explicité). Par la suite, nous réutiliserons cet observateur de sortie dans notre exemple. Le lecteur pourra se référer à l'annexe technique sur *ROBERT* pour avoir des détails sur la manière d'utiliser *ROBERT* avec des observateurs de sortie.

9.3.3 L'opérateur de projection

NBac lors de son analyse détermine un invariant par état. L'abstraction finale est obtenue en remplaçant la garde de chaque transition par l'invariant de l'état de départ. Cependant, cet invariant appartient à un espace qui contient toutes les variables. Or certaines variables sont superflues. De plus, le produit implicite permet de maîtriser l'explosion du nombre d'états, mais pas du nombre de variables. Dans notre méthode, un être humain décide donc quelles variables sont pertinentes, et quelles variables sont superflues pour décrire le flot de sortie. Il faut ensuite projeter l'invariant de chaque état dans un espace contenant uniquement les variables retenues. L'observateur de sortie s'il y

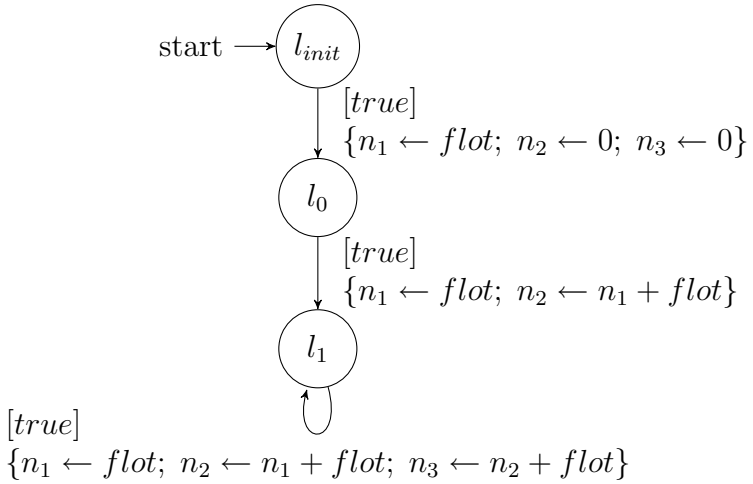


FIG. 9.7 – Exemple d'observateur de sortie

en a un contient normalement des variables pertinentes pour décrire le flot de sortie.

Ecrire des compteurs dans un observateur de sortie, puis projeter le résultat de *NBac* sur ces compteurs permet d'exprimer la sortie comme un automate à compteurs à un détail près : la sortie du système (*output* dans notre exemple) doit être exprimée comme une variable d'entrée, pour permettre au système d'être directement réutilisable.

La figure 9.8 montre l'automate finalement obtenu. Une remarque que nous pouvons faire est que nous n'avons pas d'information sur les compteurs n_2 et n_3. Nous pensons que cela est dû à une optimisation de *NBac* qui effectue une projection de chaque invariant sur un espace de dimension minimale, et donc occulte les variables qui dépendent linéairement des autres. Nous reviendrons sur ce point dans la partie suivante.

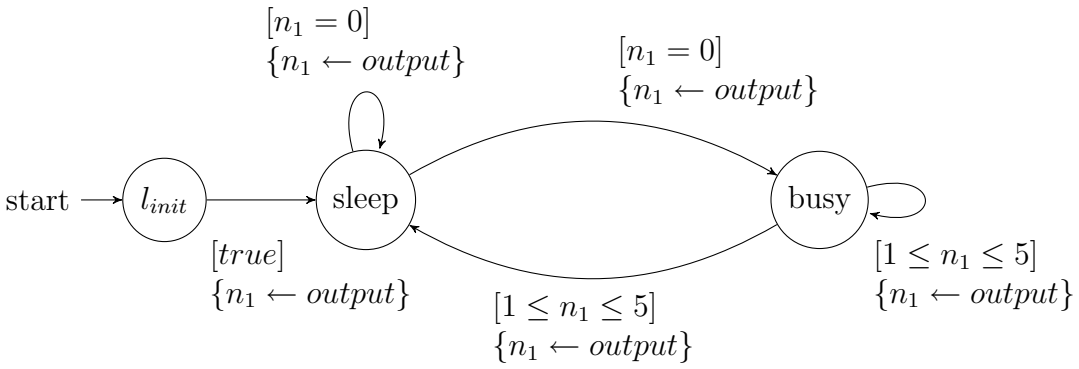


FIG. 9.8 – Automate à compteurs obtenu en sortie du power manager

Nous avons introduit dans cette partie une méthode permettant de déterminer une abstraction du flot de sortie d'un système connaissant son flot d'entrée sous la forme d'un automate à compteurs. Cette méthode permet de contrôler l'explosion de la complexité du problème en nombre d'états et en nombre de variables. La partie suivante décrit l'outil *ROBERT* qui implémente cette méthode.

Chapitre 10

L'outil *ROBERT*

*ROBERT*est le ROBot d'Extraction des Résultats Temporels. C'est une des contributions principales de ce stage. Cette section décrit les outils utilisés lors de son implémentation, ses diverses fonctionnalités déjà implémentées ainsi que ses extensions possibles. Le lecteur intéressé par l'utilisation technique de *ROBERT* pourra se référer à l'annexe B en complément.

10.1 Description

ROBERT implémente divers opérateurs de manière cohérente par rapport à la technique d'abstraction que nous avons présentée précédemment et qui est rapellée sur la figure 10.1 :

- un opérateur de produit synchrone (éventuellement implicite) d'automates exprimés dans le format utilisé par *NBac*.
- un opérateur de projection des résultats sur des variables choisies par l'utilisateur, qui, avec l'opérateur précédent, constitue le moteur d'analyse de *ROBERT*.

L'idée de bien séparer ces trois opérateurs est basée sur la volonté de rendre ROBERT facilement interfaçable avec d'autres formats. Étant donné un langage de description d'automates, il suffit de se doter d'une structure de données adaptée à ce langage, de programmer un opérateur de produit synchrone sur des automates exprimés dans cette structure de données, et d'interfacer cette structure de données avec la structure de données adaptée à ROBERT pour réutiliser le moteur d'analyse de résultats.

ROBERT est interfacé avec des librairies connues : Apron, introduite dans [JM09] pour l'interprétation des résultats de *NBac* et pour la réduction d'expressions numériques, et gmp pour l'arithmétique exacte sur les rationnels et les entiers. Il utilise également du code C++ généré par flex++ et bison++ pour parser la grammaire de *NBac*. La version actuelle de *ROBERT* compte 5700 lignes de C++, dont 2200 générées automatiquement. La figure 10.2 montre les divers morceaux de *ROBERT*.

Faute de temps, *ROBERT* n'a pas été entièrement développé. Nous n'avons donc pas pu effectuer d'étude de cas plus poussée que celle du power manager dans la partie précédente, contrairement à ce qui avait été prévu. Nous décrivons dans la section suivante quelques pistes pour étendre *ROBERT*, ainsi que quelques problèmes rencontrés.

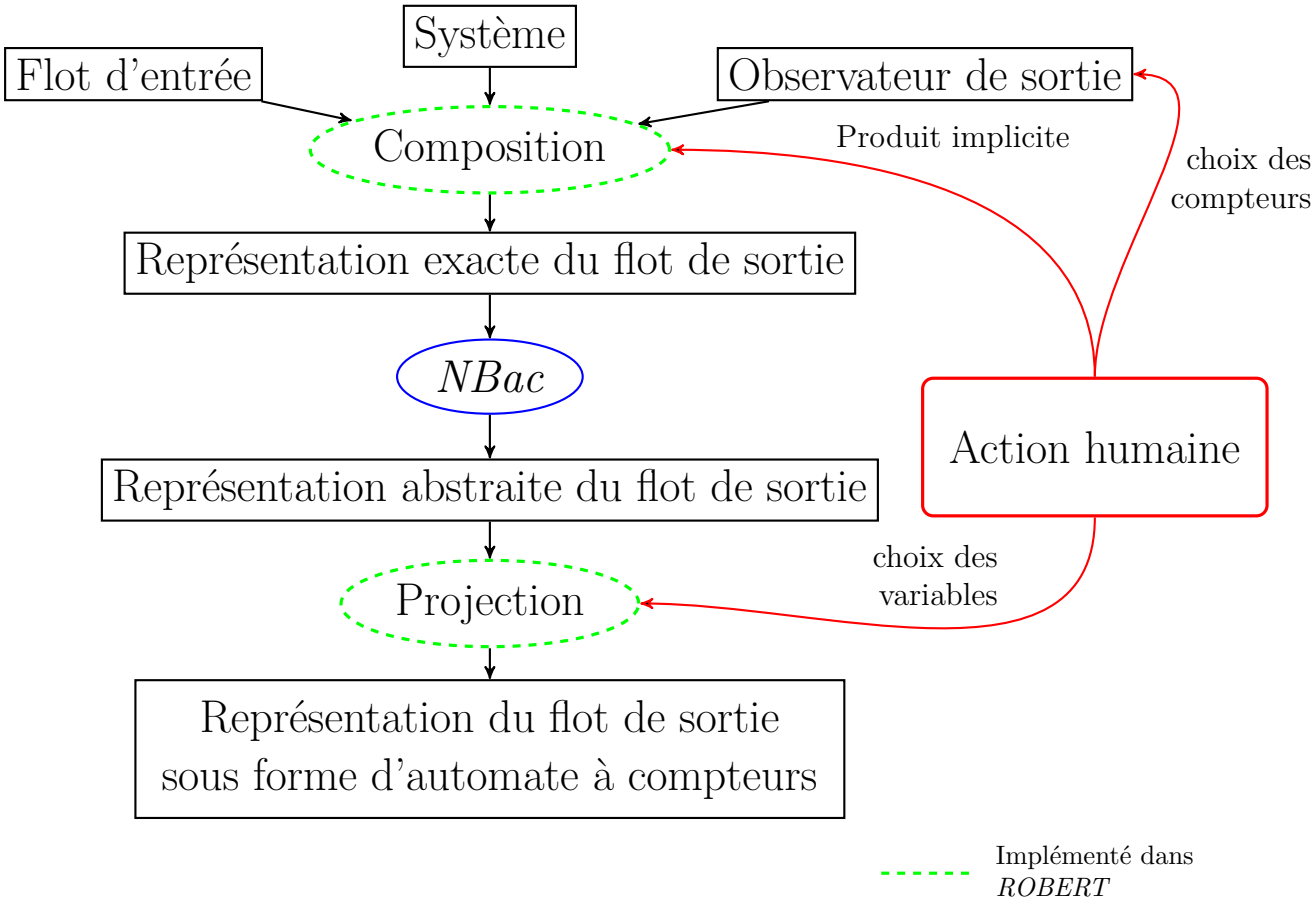


FIG. 10.1 – Position de *ROBERT* par rapport à *NBac*

10.2 Possibilités, limites

Un problème technique que nous avons rencontré est que *NBac* n'a pas de notion de sortie. Une des possibilités à laquelle nous avons pensé est qu'une optimisation interne de *NBac* consiste à raisonner sur des espaces de dimension minimale, et donc à occulter certaines variables qui dépendent simplement des autres, en particulier les variables de sortie. Dans l'exemple que nous avons développé dans la partie précédente, nous avons effectué une composition du système avec un observateur comptenant trois compteurs n_1, n_2, n_3. A la sortie, nous n'avons pu donner de résultat que sur n_1 car n_2 et n_3 avaient été occultés.

Nous avons envisagé de faire un opérateur d'analyse des résultats produits par *NBac*. Le but de cet opérateur est de redonner artificiellement une sortie à NBac, c'est à dire de "repêcher" certaines variables pertinentes. Faute de temps, il n'a pas été implémenté. Le principe est le suivant : on effectue un parcours en profondeur de l'automate en appliquant la procédure suivante pour chaque état de l'automate, et pour chaque variable que l'on veut "repêcher" (on discutera des éventuels problèmes de cette méthode après) :

1. Pour chaque prédécesseur de l'état en question, on détermine les nouvelles valeurs de la variable considérée au cours de chaque transition en fonction des autres variables.

2. A partir de l'invariant de chaque état prédécesseur et des relations entre la variable considérée et les autres variables, on détermine pour chaque transition un domaine abstrait correspondant à la variable considérée.
3. On fait une union convexe de tous les domaines abstraits déterminés à partir de chaque transition arrivant sur l'état en question.

Il demeure quelques problèmes : sur une transition, la nouvelle valeur d'une variable peut dépendre de son ancienne valeur. En pratique pour des mémoires de la sortie (ce qui nous intéressera dans les études de cas), ce n'est pas le cas. Mais si on est dans ce cas là, il faut pour déterminer l'invariant étendu à la nouvelle variable dans l'état courant avoir déterminé les invariants étendus à la nouvelle variable dans les états prédécesseurs. Se pose alors le problème du cycle dans un automate. Faut-il appliquer les méthodes de l'interprétation abstraite et appliquer un opérateur d'élargissement, ou bien vu que NBac a déjà effectué une analyse avec un opérateur d'élargissement des simples analyses locales suffisent ? Pour le moment, dans le cadre de ce stage, la question demeure ouverte. Il est souhaitable que ce problème soit résolu pour permettre à *ROBERT* d'être utilisable sur des automates assez généraux.

L'utilisation de *ROBERT* sur du code non généré automatiquement peut être incertaine, étant donné que *ROBERT* ne fait pas d'analyse contextuelle des fichiers qu'on lui donne, un bug peut très bien passer au travers des mailles et se répercuter dans le résultat final. D'une manière générale, le format *NBac* n'est pas vraiment conçu pour être utilisé comme format externe. Il est plus pratique de le générer à partir d'autres formats, plus faciles à manipuler, comme du Lustre ou du AutoC. On pourra se référer à l'annexe B pour plus de détails.

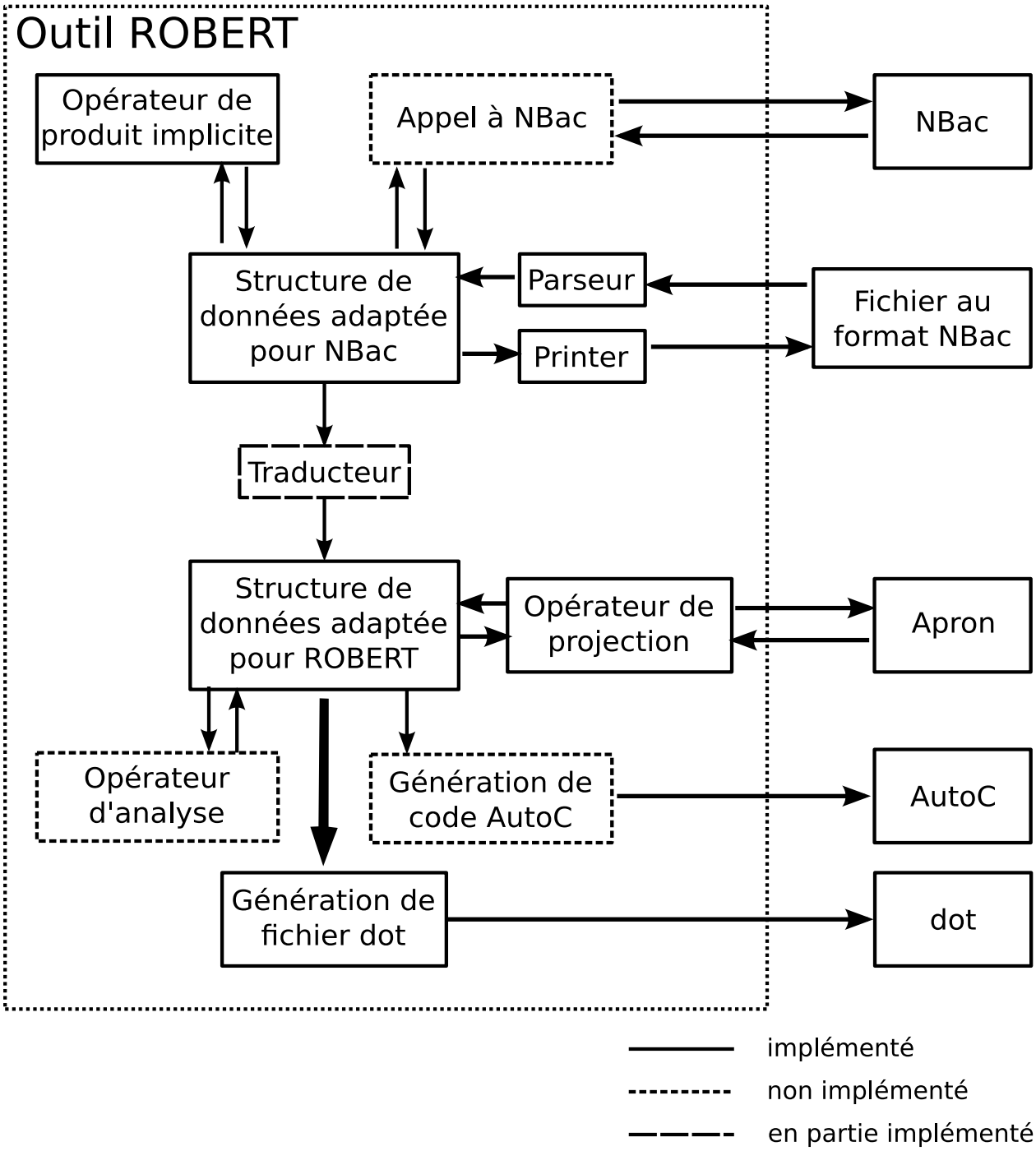


FIG. 10.2 – État d'avancement de *ROBERT*

Chapitre 11

Conclusion

11.1 Contributions de ce stage

La contribution principale de ce stage est l'introduction d'automates à compteurs pour représenter les flots d'événements qui circulent entre les différents composants d'un système modulaire. Cette abstraction a pour but de remplacer les courbes d'arrivée dans le cadre du Modular Performance Analysis. Elle s'accompagne d'une méthode expliquant comment, à partir d'un composant exprimé dans un certain formalisme et d'un automate à compteurs représentant l'ensemble des flots d'entrée admissibles, déterminer un automate à compteurs représentant une approximation de l'ensemble des flots de sortie du composant. Cette nouvelle abstraction permet de garder la notion d'état dans la représentation des sorties d'un système, ce que le Real-Time Calculus ne peut pas faire à l'heure actuelle.

Cette méthode est en partie implémentée dans l'outil *ROBERT*, que nous avons présenté en partie 10. L'implémentation de *ROBERT* sera probablement terminée par Matthieu Moy dans le courant de l'année 2009. On peut proposer de nombreuses extensions, qui pourront faire l'objet de travaux futurs. Une extension possible serait de rendre *ROBERT* plus indépendant de *NBac* pour pouvoir l'interfacer avec d'autres outils d'interprétation abstraite, comme par exemple *FAST* qui donne des résultats exacts, ou encore *Aspic* qui donne des résultats plus précis que *NBac* pour un temps d'analyse similaire. Il serait également intéressant d'interfacer *ROBERT* avec des formats de description d'automates plus pratiques que le format de *NBac* pour écrire des automates à la main.

Les autres contributions de ce stage ont été l'apport d'optimisations à l'outil ac2lus, autant dans le cadre théorique (quelques travaux sur le problèe de la causalité) que pratique (développement de deux optimisations).

11.2 Perspectives

Durant ce stage, nous avons adopté un point de vue assez externe sur le MPA, qui a été introduit comme un cadre d'analyse naturel pour le Real-Time Calculus. Le MPA est basé sur l'idée de faire de l'abstraction de composants et de l'abstraction de flots d'événements. D'après l'étude bibliographique, les connexions entre RTC et les modèles computationnels ne dissocient pas le MPA du RTC, dans le sens où elles prennent d'autres abstraction pour les composants, mais pas pour les flots d'événements. Le but est donc de

connecter un modèle computationnel à RTC et non au MPA. Le point de vue que nous avons adopté est différent : nous dissocions le MPA et RTC. Selon ce point de vue, il faut pour faire de l'analyse d'un composant une abstraction de ce composant, une abstraction des flots d'événements entrants et sortants, et éventuellement des méthodes pour changer d'abstraction pour décrire les flots d'événements (des adaptateurs en quelque sorte). Par conséquent il n'est pas a priori nécessaire de repasser dans le formalisme RTC après une analyse de système représenté par un modèle computationnel, à moins qu'on souhaite analyser ensuite un composant décrit dans le formalisme RTC.

Le cadre MPA et le Real-Time Calculus ont ouvert une nouvelle branche dans l'analyse de performances dans les systèmes embarqués. Les récents travaux visant à connecter le Real-Time Calculus aux méthodes plus classiques d'analyse de systèmes sont allés dans ce sens. La question soulevée dans ce rapport est la suivante : Le Real-Time Calculus est-il le formalisme le plus pertinent pour décrire les communications entre les différents composants d'un système ?

La réponse que nous proposons est la suivante : les automates sont plus expressifs que les courbes d'arrivée. Ils permettent de décrire une classe plus variée de flots d'événements que les courbes d'arrivée. Passer d'une abstraction à une autre implique une perte d'information. Si on veut analyser en série deux composants d'un système exprimés en Lustre avec l'outil *ac2lus*, il est dommage d'exprimer la sortie du premier sous la forme d'une courbe d'arrivée, pour repasser tout de suite après en Lustre. On perd alors beaucoup de précision sur l'analyse du système. L'abstraction que nous proposons limite cette perte de précision, et la technique de transformation proposée permet de limiter l'explosion de complexité du problème.

Une évolution possible de la recherche est le développement de connexions entre les formalismes visant à décrire les flots d'événements grâce auxquels les systèmes modulaires communiquent, comme le montre la figure 11.1. On pourrait très bien envisager par exemple qu'un certain nombre de formalismes bien connus (comme les automates temporisés, Lustre, RTC, etc) soient tous connectés entre eux. Lors de l'analyse d'un système dont les composants sont tous exprimés dans ces formalismes, on ne repasserait jamais dans un formalisme intermédiaire entre deux analyses de composants, et donc on limiterait la perte d'information.

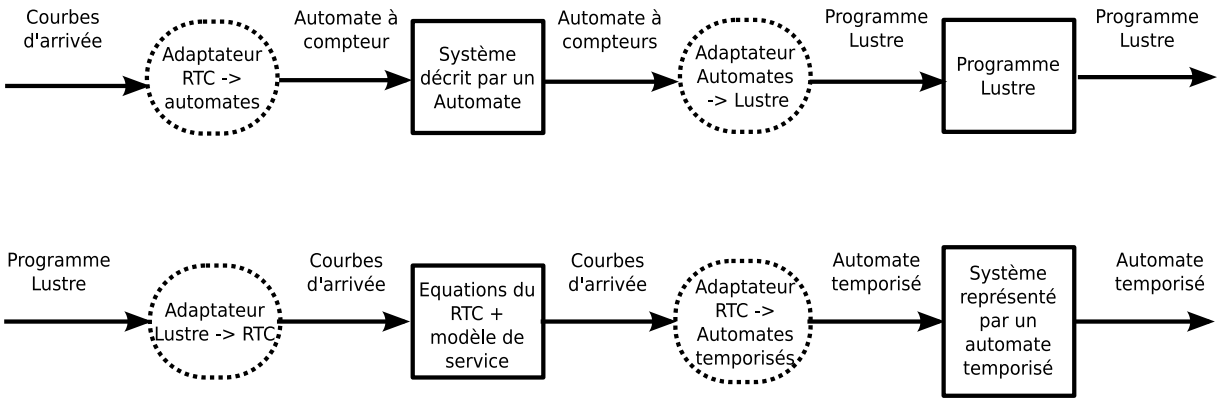


Fig. 11.1 – Chaine d'analyse d'un système modulaire

11.3 Remerciements

Je tiens particulièrement à remercier Matthieu Moy et Karine Altisen pour l'accueil qu'ils m'ont fait à Verimag, ainsi que pour leur disponibilité au quotidien durant mon stage, et leurs conseils toujours avisés. Je tiens également à remercier Bertrand Jeannet pour son aide sur l'utilisation de NBac et de AutoC. J'adresse enfin des remerciements à l'ensemble des thésards, stagiaires et permanents de Verimag que j'ai cotoyé pendant ces quatre mois et qui contribuent tous à mettre une très bonne ambiance dans le laboratoire.

Annexe A

Aspects théoriques en RTC

Cette annexe énonce sans démonstration quelques résultats intéressants liés à l'algèbre min-plus et max-plus, ainsi qu'au Real-Time Calculus. Des démonstrations de ces résultats pourront être trouvées dans [LBT04]. De même on présentera ici des résultats liés à des représentations continues des courbes d'arrivée. Ces mêmes résultats sont bien sûr vrais pour des représentations discrètes avec les formules correspondantes.

A.1 Délai maximal et file d'attente maximale

Un intérêt du Real-Time Calculus est de pouvoir borner les délais d'attente de traitement des événements, ainsi que donner des bornes supérieurs aux files d'attente qui sont implantées à l'entrée des modules. Cela permet entre autres de prouver grâce aux bornes sur les délais qu'il n'y aura pas de situation de famine, et grâce aux bornes sur les files d'attentes qu'il n'y aura pas de dépassement de capacité.

On a donc besoin de définir la notion de délai et de file d'attente à partir des courbes d'arrivée et de service d'un système :

Definition A.1. Délai maximal de traversée d'un système : On définit le délai maximal par la relation : $D \leq \sup_{t \geq 0} \{\inf \{\tau \geq 0 : \alpha_u(t) \leq \beta_l(t+\tau)\}\}$

Definition A.2. File d'attente maximale (Backlog) : On définit la file d'attente maximale par la relation : $B \leq \sup_{t \geq 0} \{\alpha_u(t) - \beta_l(t)\}$

A.2 Aspects mathématiques

A.2.1 Bases d'algèbre

Le Real-Time Calculus utilise l'algèbre min-plus et l'algèbre max-plus. On en rappelle ici les bases. Le lecteur intéressé pourra se référer à [LBT04] (chapitre 3) pour plus de détails.

L'algèbre min-plus (resp max plus) s'écrit en faisant l'analogie suivante : on part de l'algèbre usuelle, dans laquelle on remplace les additions par des minima (resp des maxima) et les multiplications par des additions.

Soient f et g deux fonctions à valeurs réelles. Les opérations de base s'écrivent alors comme suit :

Definition A.3. Convolution en algèbre min-plus : $f \otimes g(x) = \inf_{i \in [0,x]} \{f(i) + g(x-i)\}$

Definition A.4. Convolution en algèbre max-plus : $f \overline{\otimes} g(x) = \sup_{i \in [0,x]} \{f(i) + g(x-i)\}$

Definition A.5. Déconvolution en algèbre min-plus : $f \oslash g(x) = \sup_{i \geq 0} \{f(x+i) - g(i)\}$

Definition A.6. Déconvolution en algèbre max-plus : $f \overline{\oslash} g(x) = \inf_{i \geq 0} \{f(x+i) - g(i)\}$

On peut donner une interprétation physique de la convolution comme suit : prenons $x \leftarrow f(x)$ et $x \leftarrow g(x)$ deux fonctions correspondant au nombre maximum d'événements arrivant dans des fenêtres de taille x sur deux cannaux différents. On peut observer au choix l'un des deux canaux. Convoler f et g en algèbre min-plus correspond à regarder pendant un certain temps un canal puis l'autre dans le but de minimiser le nombre maximal d'événements observables (cela revient donc à trouver deux fenetres de temps dont la durée cumulée vaut x et à regarder pendant la première fevêtre un canal, puis l'autre dans la deuxième fenêtre). Convoler f et g en algèbre max-plus correspond à regarder pendant un certain temps un canal puis l'autre dans le but de maximiser le nombre maximal d'événements observables.

On peut continuer cette analogie pour donner une interprétation physique de la déconvolution (en algèbre min-plus). On fixe une fenêtre de temps de taille x, et on va observer un flot inconnu pendant cette fenêtre de temps. On aimerait de plus que cela soit rentable : sachant qu'on a un autre flot à regarder, si on ne regarde que celui-ci, peut-être qu'on va manquer une quantité intéressante d'événements sur l'autre canal. Si on regarde quasi-exclusivement l'autre canal (donc sur une fenêtre de temps grande devant x), on peut se dire qu'on aurait eu plutôt intéret à regerder ce canal plus longtemps, afin d'observer plus d'événements sur ce canal. Il faut trouver un juste milieu. La valeur $f \otimes g(x)$ correspond au nombre maximum d'événements observables sur le canal en ayant trouvé ce juste milieu.

Definition A.7. Convolution récursive : On notera $f^n = f \underbrace{\otimes \cdots \otimes}_{n-1 \text{ fois}} f$ la convolée n-ième de f en algèbre min-plus.
On notera $f^{\underline{n}} = f \underbrace{\overline{\otimes} \cdots \overline{\otimes}}_{n-1 \text{ fois}} f$ la convolée n-ième de f en algèbre max-plus

Definition A.8. Cloture sub-additive (resp super-additive) : Soit f une fonction à valeurs réelles. La cloture sub-additive (resp super-additive) de f notée $\overline{f}$ (resp $\underline{f}$) vaut : $\overline{f}(x) = \inf_{x \in \mathbb{N}} \{f^n(x)\}$ (resp $\underline{f}(x) = sup_{x \in \mathbb{N}} \{f^{\underline{n}}(x)\}$)

On dira que f est sub-additive (resp super-additive) ssi $f = \overline{f}$ (resp $f = \underline{f}$). $\overline{f}$ est bien évidemment sub-additive, et $\underline{f}$ est super-additive.

Remarque A.9. analogie avec les représentations discrètes : Les mêmes raisonnements donnent des formules et des définitions analogues dans des espaces discrets. En pratique ce sont celles là que l'on utilise dans les implémentations.

A.2.2 Résultats notables

Théorème A.10. ***Croissance des courbes d'arrivée et de service :*** *Soit α une fonction représentant une courbe d'arrivée (resp une courbe de service). Alors α est croissante au sens large.*

Démonstration. Si α est une courbe d'arrivée haute, $\alpha(x)$ correspond au nombre maximum d'événements arrivant dans une fenêtre de temps x, Or si $y \leq x$, si on prend $\epsilon \geq 0$ petit, il existe une fenêtre de temps de taille y au cours de laquelle $\alpha(y) - \epsilon$ événements arrivent. Cette fenêtre peut être prolongée à droite en une fenêtre de taille x, au cours de laquelle au moins $\alpha(y) - \epsilon$ événements seront donc arrivés. D'où la relation $\alpha(x) \geq \alpha(y) - \epsilon$. ϵ étant quelconque, il vient $\alpha(x) \geq \alpha(y)$, d'où le résultat. La preuve est analogue dans le cas où α est une courbe d'arrivée basse ou une courbe de service. □

Théorème A.11. ***Comparaison entre deux fonctions et leur convolée :*** *Soient f et g deux fonctions à valeurs réelles. Si $f(0) = 0$ et $g(0) = 0$, alors $f \otimes g \leq \min(f, g)$ et $f \overline{\otimes} g \geq \max(f, g)$*

Démonstration. On a $\forall x \in (R)$, $f \otimes g(x) = \inf_{i \in [0,x]} \{f(i) + g(x - i)\} \leq f(0) + g(x) = g(x)$ car $f(0) = 0$, de la même façon $f \otimes g(x) \leq f(x)$, d'où le résultat. La preuve est similaire pour la convolution en algèbre max-plus. □

Remarque A.12. Importance de la relation $f(0) = 0$: Cette hypothèse est souvent faite car elle implique que la convolée de f avec elle même est décroissante en algèbre min-plus, ce qui donne des résultats intéressants en Real-Time Calculus, sachant que cette hypothèse traduit le fait qu'un événement ne peut pas arriver instantanément.

Théorème A.13. ***Réduction aux courbes d'arrivée continues à gauche :*** *Soit $t \to f(t)$ un flot à valeurs réelles continu à droite ou à gauche, et α une courbe d'arrivée (haute ou basse) pour ce flot. Alors $x \to \alpha_g(x) = \lim_{y \to x} \alpha(y)$ est aussi une courbe d'arrivée du flot f (cette limite existe car α est croissante).*

Théorème A.14. ***Courbe d'arrivée et flot :*** *Soit f un flot à valeurs réelles, et soit α_u et α_l des fonctions à valeurs réelles. Les résultats suivants sont vrais :*

- *α_u est une courbe d'arrivée haute pour le flot f ssi $f \leq f \otimes \alpha_u$*
- *α_l est une courbe d'arrivée basse pour le flot f ssi $f \geq f \overline{\otimes} \alpha_l$*

Théorème A.15. ***Courbe d'arrivée sub-additive / super-additive :*** *Soient α_u et α_l des courbes d'arrivée hautes et basses du flot f. Les résultats suivants sont vrais :*

- *$\overline{\alpha_u}$ est une courbe d'arrivée haute pour f*
- *$\underline{\alpha_l}$ est une courbe d'arrivée basse pour f*

En d'autres termes, étant donné une paire de courbes d'arrivée haute/basse, leur cloture sub-additive/super-additive donne une nouvelle paire de courbes d'arrivée plus fine.

A.3 Règles de transformation

On s'intéresse ici à la transformation du flot d'entrée à travers le système. Le flot d'entrée est décrit par une paire de courbes d'arrivée, et le système est décrit par une

paire de courbes de service. On cherche à exprimer aussi finement que possible une paire de courbes d'arrivée décrivant le flot de sortie ainsi qu'une courbe de service donnant une estimation des ressources disponibles inutilisées (on parlera de courbe de service de sortie).

Le théorème suivant donne une formule de transformation :

Théorème A.16. ***Transformation d'un flot par un système :*** *Soient* (α_u, α_l) *une paire de courbes d'arrivée décrivant un flot, et* (β_u, β_l) *une paire de courbes de service décrivant un système. On pourra supposer sans perte de généralité que* α_u *et* β_u *sont sous-additives et que* α_l *et* β_l *sont super-additives. Soient* (α'_u, α'_l) *et* (β'_u, β'_l) *la paire de courbes d'arrivée du flot de sortie et la paire de courbes de service résiduelle. Les relations suivantes sont vraies pour tout* $\Delta \in \mathbb{R}$ *:*

- $\alpha'_u(\Delta) = \min\{\beta_u(\Delta),\ (\alpha_u \otimes \beta_u) \oslash \beta_l(\Delta)\}$
- $\alpha'_l(\Delta) = \min\{\beta_l(\Delta),\ (\alpha_l \oslash \beta_u) \otimes \beta_l(\Delta)\}$
- $\beta'_l(\Delta) = \sup_{0 \le \lambda \le \Delta}\{\beta_l(\lambda) - \alpha_u(\lambda)\}$
- $\beta'_l(\Delta) = \max\{0,\ \inf_{\Delta \le \lambda}\{\beta_u(\lambda) - \alpha_l(\lambda)\}\}$

On pourra interpréter grossièrement ce résultat comme suit pour α'_u : imaginons le cas où on atteint un maximum d'événements émis dans une fenêtre de temps Δ. Il faut pour cela qu'il en soit arrivé beaucoup auparavant (dans une fenêtre plus grande, $\Delta + \lambda$), et que peu d'entre eux aient été émis en dehors de la fenêtre (pour en avoir en réserve). Pour en émettre le moins possible en dehors de la fenêtre de taille Δ, il faut que $\beta_l(\lambda)$ soit petit. Cela justifie un résultat plus grossier que le précédent qui est : $\alpha''_u(\Delta) = \alpha_u \oslash \beta_l(\Delta) = \sup_{\lambda \ge 0}\{\alpha_u(\Delta + \lambda) - \beta_l(\lambda)\}$ est une courbe d'arrivée haute du flot de sortie.

Pour arriver au résultat final il faut pousser l'analogie plus loin : on peut vouloir recevoir beaucoup pendant une certaine période plus longue $(\Delta + \lambda + \mu)$ que la période où on veut émettre beaucoup (Δ), et vouloir émettre peu pendant la période $\lambda + \mu$, afin de garder de la réserve pour la période Δ. Le résultat est plus fin si on prend en compte les courbes de service haute et basse. On pourra donc considérer que pendant la fenêtre μ, β_u devra être petit, et que pendant la fenêtre λ, β_l le sera aussi, λ et μ étant des paramètres libres, qui servent à optimiser le résultat. Cela explique que $\alpha'_u(\Delta) = (\alpha_u \otimes \beta_u) \oslash \beta_l(\Delta) = \sup_{\lambda \ge 0}\{\inf_{0 \le \mu \le \lambda + \Delta}\{\alpha_u(\Delta + \lambda - \mu) + \beta_u(\mu)\} - \beta_l(\lambda)\}$

L'interprétation des autres résultats est basée sur le même principe.

Remarque A.17. Propriétés des courbes de sortie : On pourra remarquer que les définitions précédentes de (α'_u, α'_l) et (β'_u, β'_l) impliquent que α'_u et β'_u sont sous-additives et que α'_l et β'_l sont super-additives.

Théorème A.18. ***Concaténation de deux systèmes :*** *Soient deux systèmes munis des courbes de services* (β^1_u, β^1_l) *et* (β^2_u, β^2_l)*. Le système global formé de ces deux systèmes admet pour courbes de service* $\beta'_u = \beta^1_u \otimes \beta^2_u$ *et* $\beta'_l = \beta^1_l \overline{\otimes} \beta^2_l$

Ce résultat permet de considérer un système formé de plusieurs sous-systèmes analysés dans le formalisme RTC, et ce avec les mêmes éléments d'analyse que ceux des sous-systèmes. Un résultat non intuitif et très intéressant est le suivant :

Théorème A.19. ***Optimalité de l'étude globale*** *: Soit deux systèmes en série décrits par leurs courbes de services* (β_u^1, β_l^1) *et* (β_u^2, β_l^2)*, et un flot d'entrée décrit par sa courbe d'arrivée* (α_u, α_l)*. Soient* D_1 *et* D_2 *les délais maximaux calculables pour les deux systèmes. Soit* D *le délai maximal calculable pour le système global formé des deux systèmes précédents. La relation suivante est vraie : * $D \leq D_1 + D_2$

En d'autres termes, l'analyse globale se révèle être plus précise que la concaténation des analyses locales. Cela vient du fait que des couplages peuvent avoir lieu entre les deux systèmes, et que ces couplages ne sont pas pris en compte lorsqu'on concatène des analyses locales.

A.4 Exemple : preuve du bon fonctionnement d'un scheduler

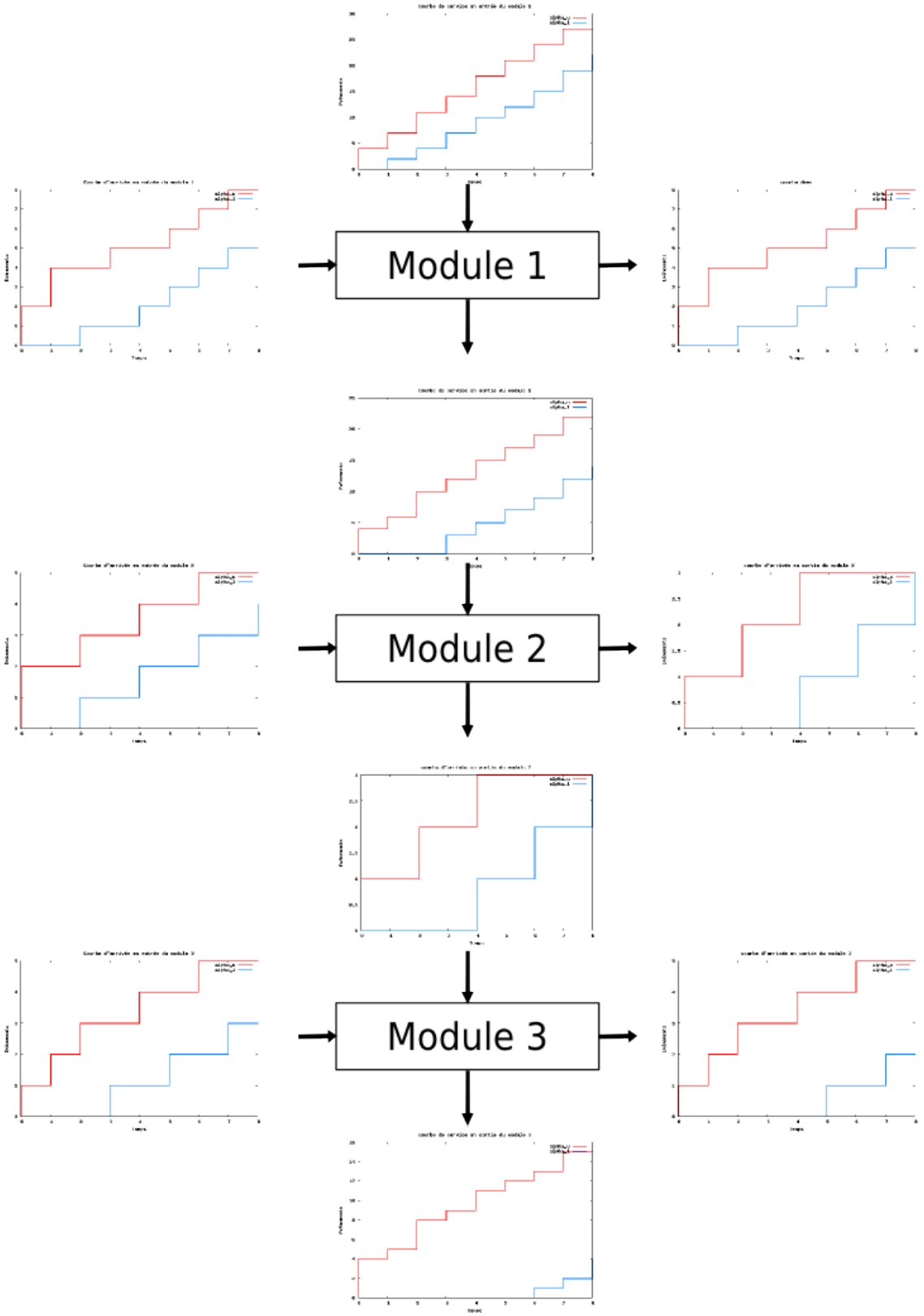
On considère un scheduler qui supervise plusieurs processus ayant des priorités différentes. Plusieurs processus traitant des flots de données vont devoir se partager des ressources allouées globalement au scheduler. Une politique simple est d'allouer toutes les ressources disponibles aux processus de priorité élevées et de donner les ressources inutilisées aux processus de priorité moindre. On souhaiterait prouver qu'aucune situation de famine, où un processus restera bloqué indéfiniement ne peut arriver.

Dans cet exemple on traitera un scheduler avec trois processus, dont on spécifiera les courbes d'arrivée. L'analyse du système est montrée sur la figure A.1. On peut voir que la courbe de sortie de chaque module que la courbe basse n'est pas nulle. Cela prouve donc formellement qu'il ne peut pas y avoir de famine. On peut également dimensionner la file d'attente de chaque module pour être sur qu'il n'y aura pas de dépassement de capacité.

On peut faire quelques remarques sur cette étude :

- La courbe basse de service se décale vers la droite. Cela donne une idée du temps maximal où un module peut rester en attente.
- On a supposé que tous les modules fonctionnent de la même façon, c'est à dire qu'une unité de resoources est utilisée de la même manière par chaque module. Dans une étude plus réaliste, c'est bien évidemment faux. Cependant on peut se ramener à ce cas en normalisant le flot d'entrée. On se réfèrera à [CKT03] pour plus de détails.

FIG. A.1 – Analyse d'un scheduler



Annexe B

Documentation technique sur ROBERT

B.1 Utilisation des différents outils de *ROBERT*

B.1.1 Utilisation de l'opérateur de produit synchrone implicite

L'opérateur de produit synchrone (éventuellement projeté) est implémenté directement à partir des fichiers au format NBac. Étant donné deux fichiers décrivant des automates, cet opérateur retourne un fichier qui correspond au produit synchrone des deux automates, éventuellement projeté sur l'un d'eux. Une première remarque est que le format NBac n'est pas conçu pour être directement utilisé, car il est difficile à manipuler à la main. Habituellement il est automatiquement généré à partir de code Lustre ou AutoC. Nous avons donc dans nos expérimentations utilisé des fichiers au format NBac générés à partir de fichiers en AutoC, suivant la procédure suivante :

1. On décrit l'automate en AutoC sans utiliser de canaux (ils sont inutiles pour faire des produits synchrones)
2. L'outil *autoc2auto* génère un fichier Auto que l'on transforme directement en fichier au format *NBac* par l'outil *auto2nbac*.
3. Le fichier NBac doit être modifié. Il y a une variable à retirer : une variable de type channel_t qui se nomme en général "tau". Toutes les conditions du type $channel = tau$ sont remplaçables par $true$. On peut également, si l'automate présenté est déterministe, retirer une variable d'entrée appelée $aux0$ qui ne sert que si l'automate manipulé est non-déterministe.

Il est important de savoir que cet opérateur ne fait pas d'analyse (au sens contextuel) des deux automates dont il fait le produit. En particulier, il ne détecte pas les variables non déclarées, ou bien les variables déclarées deux fois. Cela a des avantages et des inconvénients. Par exemple, on peut faire partager des variables entre plusieurs automates par un simple renommage. Dans l'automate où on aura renommé la variable, ROBERT ne détectera pas de variable non déclarée, et fera quand même le produit.

Pour connecter deux automates analysables par NBac, les fichiers doivent être traités au préalable à la main (cela pourra faire l'objet d'une implémentation future) de la manière

suivante :

1. Il faut "connecter" les automates, c'est à dire leur faire correctement partager les variables. Il faut faire cela par un renommage d'une variable dans un automate (et retirer la ligne qui déclare l'autre variable).
2. Il faut identifier les variables d'entrée, qui ont été déclarées en variable d'état, et qui sont vues par NBac comme des constantes (des variables d'état dont la transition est du type $v' = v$).

A l'heure actuelle, l'option "project" dans le produit n'a pas encore été bien testée, elle est adaptée pour les automates exprimés au format NBac générés à partir d'un automate exprimé en AutoC.

B.1.2 L'opérateur de projection sur des variables

Actuellement, l'opérateur de projection sur des variables prend en entrée un fichier sur lequel l'utilisateur spécifie les variables à garder.

L'opérateur est basé sur les fonctions de Apron. La majeure partie de cet opérateur consiste à exprimer les invariants des états donnés dans la structure de données adaptée à ROBERT sous forme de domaines abstraits manipulés par Apron, et inversement. Grâce aux fonctionnalités proposées par Apron, on peut suivant les cas choisir de raisonner sur des polyhèdres, des octogones, des intervalles, etc...

L'opérateur de projection sur des variables crée une abstraction du système que l'on étudie. Des extensions possibles de cet opérateur seraient de la génération de code dans divers formats, la seule sortie étant pour le moment graphique (au format dot).

B.2 Compilation et différentes options

Apron est disponible sur le lien `http://apron.cri.ensmp.fr/library/`

Pour fonctionner, Apron a besoin des librairies suivantes qui ne sont pas installées à Verimag :

- Parma Programmation Library (PPL) disponible au lien : `http://www.cs.unipr.it/ppl/`
- mpfr disponible au lien `http://www.mpfr.org/`

ROBERT utilise bison++ v 1.21-8, qui n'est pas installé à Verimag et qui est disponible au lien suivant : `http://www.mario-konrad.ch/index.php?page=20024`

ROBERT utilise également flex++ v 2.5.33, qui est installé à Verimag. Une évolution possible serait de faire passer la génération de parseur sous flex et bison, qui sont plus documentés, et qui permettent actuellement de générer du code c++.

Il faut rajouter la ligne suivante dans son bashrc (en remplaçant /home/xjean/ par le dossier correspondant) :

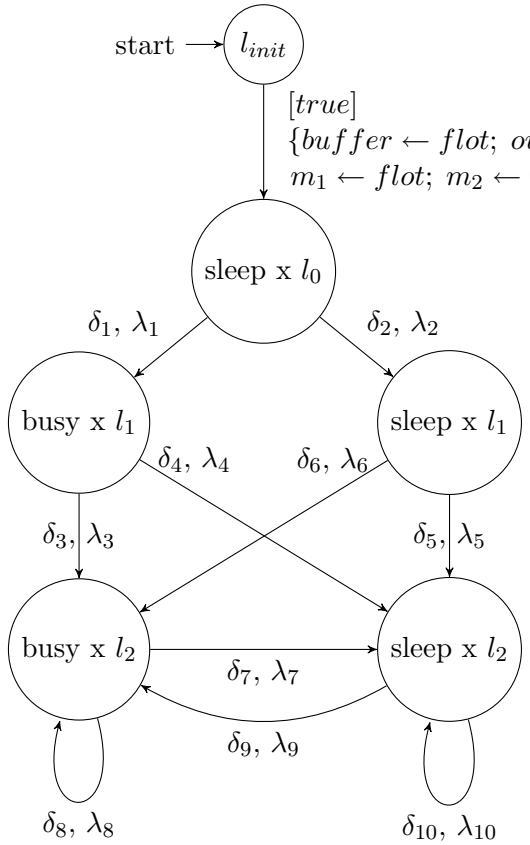
```
export LD_LIBRARY_PATH=/home/xjean/ppl/lib/ :/home/xjean/apron/lib/ :
/home/xjean/mpfr/lib/ :\$LD_LIBRARY_PATH
```

Le makefile suivant montre l'ensemble des librairies à inclure, ainsi que l'ensemble des liens à préciser.

```
APRON_PREFIX = /home/xjean/apron
MLGMPIDL_PREFIX = /home/xjean/apron/lib
GMP_PREFIX = /usr
MPFR_PREFIX = /home/xjean/mpfr
PPL_PREFIX = /home/xjean/ppl

COMPILE = -c
CXXFLAGS = -Wall -Wextra -g
CXX = g++
EXEC = main.exe
LIB =   -lap_ppl_debug -lppl -lgmpxx -lpolka_debug
        -loctMPQ_debug -lboxMPQ_debug -lapron_debug
        -litvMPQ_debug -litv_debug -lmpfr -lgmp

ICFLAGS = -I$(GMP_PREFIX)/include -I$(MPFR_PREFIX)/include
          -I$(APRON_PREFIX)/include
LCFLAGS = -L$(GMP_PREFIX)/lib -L$(MPFR_PREFIX)/lib
          -L$(APRON_PREFIX)/lib -L$(PPL_PREFIX)/lib

all :   utils.o flags.o automaton_representation.o converter.o
        NBac_representation.o automaton_product.o
        Parseur_NBac.o lex.yy.o Main.o

        ${CXX} ${CXXFLAGS} $(ICFLAGS) $(LCFLAGS) $^ -o ${EXEC} ${LIB}

%.o :%.cpp
        ${CXX} ${CXXFLAGS} ${COMPILE} $^ -o $@

automaton_representation.o  : automaton_representation.cpp
        ${CXX} ${CXXFLAGS} $(ICFLAGS) $(LCFLAGS) ${COMPILE} $^ -o $@

converter.o  : converter.cpp
        ${CXX} ${CXXFLAGS} $(ICFLAGS) $(LCFLAGS) ${COMPILE} $^ -o $@

Main.o  : Main.cpp
        ${CXX} ${CXXFLAGS} $(ICFLAGS) $(LCFLAGS) ${COMPILE} $^ -o $@

Parseur_NBac.o : parseur_NBac.l parseur_NBac.y Scanner.h
        /home/xjean/bison++/bin/bison++ -o Parseur_NBac.cpp -h
        Parseur_NBac.h -d parseur_NBac.y
        flex++ parseur_NBac.l
        g++ ${COMPILE} ${CXXFLAGS} lex.yy.cc
        g++ ${COMPILE} ${CXXFLAGS} Parseur_NBac.cpp

clean :
        rm -f *~ *.o *.gch Parseur_NBac.h Parseur_NBac.cpp
        lex.yy.cc ${EXEC}
```

B.3 Etapes intermédiaires dans l'analyse de système

On s'intéresse ici à l'étude complète du power manager introduit dans le chapitre 9. On rappelle que l'on fait la composition du système représenté sur la figure 9.5 page 57, et de l'automate à compteur représenté sur la figure 9.4 page 56. Le produit synchrone explicite des deux automates est donné sur la figure B.1. Ce produit analysé par *NBac* est donné sur la figure B.2. L'abstraction finale est donnée sur la figure 9.8 page 60. Sur l'abstraction finale, on a projeté l'abstraction déterminée par *NBac* sur la seule variable de sortie *output*, et le compteur n_1.



$$
\begin{array}{lcl}
\delta_1 & = & buffer \geq 5 \text{ and } 0 \leq m_1 \leq 5 \\
\delta_2 & = & buffer \leq 4 \text{ and } 0 \leq m_1 \leq 5 \\
\delta_3 & = & buffer \geq 1 \text{ and } 0 \leq m_1 \leq 5 \\
 & & \text{and } 1 \leq m_2 \leq 9 \\
\delta_4 & = & buffer = 0 \text{ and } 0 \leq m_1 \leq 5 \\
 & & \text{and } 1 \leq m_2 \leq 9 \\
\delta_5 & = & buffer \leq 4 \text{ and } 0 \leq m_1 \leq 5 \\
 & & \text{and } 1 \leq m_2 \leq 9 \\
\delta_6 & = & buffer \geq 5 \text{ and } 0 \leq m_1 \leq 5 \\
 & & \text{and } 1 \leq m_2 \leq 9 \\
\delta_7 & = & buffer = 0 \text{ and } 0 \leq m_1 \leq 5 \\
 & & \text{and } 1 \leq m_2 \leq 9 \text{ and } 2 \leq m_3 \leq 14 \\
\delta_8 & = & buffer \geq 1 \text{ and } 0 \leq m_1 \leq 5 \\
 & & \text{and } 1 \leq m_2 \leq 9 \text{ and } 2 \leq m_3 \leq 14 \\
\delta_9 & = & buffer \geq 5 \text{ and } 0 \leq m_1 \leq 5 \\
 & & \text{and } 1 \leq m_2 \leq 9 \text{ and } 2 \leq m_3 \leq 14 \\
\delta_{10} & = & buffer \leq 4 \text{ and } 0 \leq m_1 \leq 5 \\
 & & \text{and } 1 \leq m_2 \leq 9 \text{ and } 2 \leq m_3 \leq 14
\end{array}
$$

$$
\begin{array}{lcl}
\lambda_1 & = & output \leftarrow 5;\ buffer \leftarrow buffer - 5 + flot;\ m_1 \leftarrow flot;\ m_2 \leftarrow m_1 + flot \\
\lambda_2 & = & output \leftarrow 0;\ buffer \leftarrow buffer + flot;\ m_1 \leftarrow flot;\ m_2 \leftarrow m_1 + flot \\
\lambda_3 & = & output \leftarrow min(5, buffer);\ buffer \leftarrow buffer - min(5, buffer) + flot; \\
 & & m_1 \leftarrow flot;\ m_2 \leftarrow m_1 + flot;\ m_3 \leftarrow m_2 + flot \\
\lambda_4 & = & output \leftarrow 0;\ buffer \leftarrow flot;\ m_1 \leftarrow flot;\ m_2 \leftarrow m_1 + flot;\ m_3 \leftarrow m_2 + flot \\
\lambda_5 & = & output \leftarrow 0;\ buffer \leftarrow buffer + flot;\ m_1 \leftarrow flot; \\
 & & m_2 \leftarrow m_1 + flot;\ m_3 \leftarrow m_2 + flot \\
\lambda_6 & = & output \leftarrow 5;\ buffer \leftarrow buffer - 5 + flot;\ m_1 \leftarrow flot; \\
 & & m_2 \leftarrow m_1 + flot;\ m_3 \leftarrow m_2 + flot \\
\lambda_7 & = & output \leftarrow 0;\ buffer \leftarrow flot;\ m_1 \leftarrow flot;\ m_2 \leftarrow m_1 + flot;\ m_3 \leftarrow m_2 + flot \\
\lambda_8 & = & output \leftarrow min(5, buffer);\ buffer \leftarrow buffer - min(5, buffer) + flot; \\
 & & m_1 \leftarrow flot;\ m_2 \leftarrow m_1 + flot;\ m_3 \leftarrow m_2 + flot \\
\lambda_9 & = & output \leftarrow 5;\ buffer \leftarrow buffer - 5 + flot;\ m_1 \leftarrow flot;\ m_2 \leftarrow m_1 + flot; \\
 & & m_3 \leftarrow m_2 + flot \\
\lambda_{10} & = & output \leftarrow 0;\ buffer \leftarrow buffer + flot;\ m_1 \leftarrow flot;\ m_2 \leftarrow m_1 + flot; \\
 & & m_3 \leftarrow m_2 + flot
\end{array}
$$

FIG. B.1 – Calcul exact de la représentation du flot de sortie du power manager

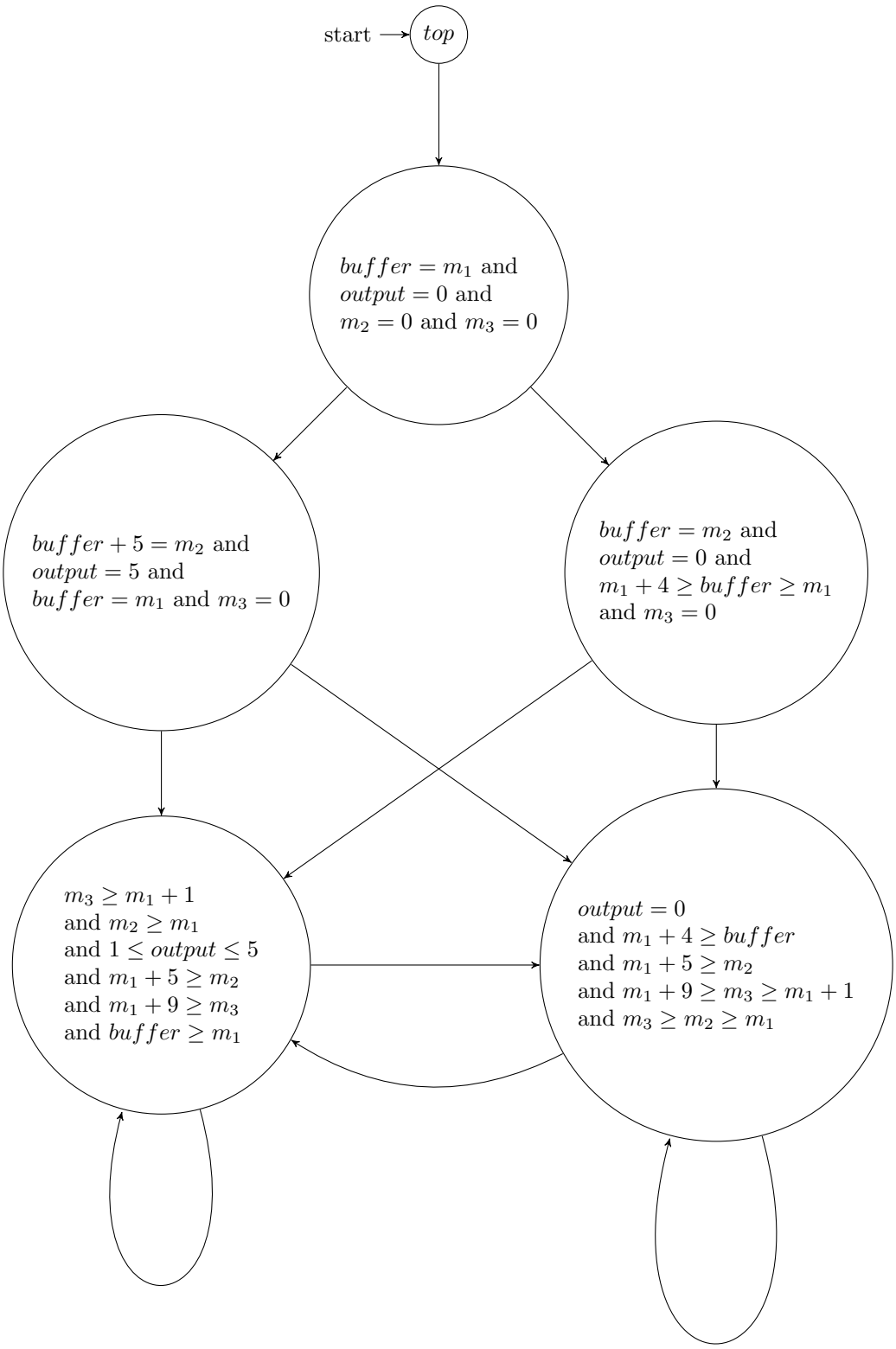


FIG. B.2 – Analyse par *NBac* de la représentation du flot de sortie du power manager (les transitions ne sont pas explicitées pour permettre une meilleure lisibilité)

Annexe C

Documentation technique sur le langage AutoC/Auto

C.1 Code source des programmes présentés dans la partie 8

```
channel start();
channel inp(int);
channel outp(int);

process delayer {
sync start, inp, outp;
var sortie : int; memoire :int;
init pc = linit;
loc linit :
  when true sync start() goto l1 assign {memoire :=0; sortie :=0};
loc l1 :
  when true sync inp(x) goto l2 assign {memoire :=x};
loc l2 :
  when sortie = x sync outp(x) goto l1 assign {sortie :=memoire};
}

system delayer;
explicit delayer;
```

FIG. C.1 – Code source d'un delayer

```
channel start();
channel inp(int);
channel outp(int);

process power_manager {
sync start, inp, outp;
var sortie : int; memoire :int; veille : bool;
init pc = linit;
loc linit :
  when true sync start() goto l1
    assign {memoire :=0; sortie :=0; veille := true};
loc l1 :
  when memoire <= 5 and veille sync inp(x) goto l2
    assign {memoire := memoire + x};
  when (memoire >= 6 and veille) or not veille
    sync inp(x) goto l3
    assign {memoire := memoire + x; veille := false};

-- Cet etat correspond a l'etat de veille
loc l2 :
  when x=0 sync outp(x) goto l1;
loc l3 :
  when memoire >= 5 and x=5 sync outp(x) goto l2
    assign {memoire := memoire - x; sortie := x};
  when memoire =x and x <=4 sync outp(x) goto l2
    assign {memoire := 0; sortie := memoire; veille := true};
}

system power_manager;
explicit power_manager;
```

FIG. C.2 – Code source d'un power manager

Bibliographie

[Bou05] Anne Bouillard. *Optimisation et analyse probabiliste de systèmes à événements discrets*. PhD thesis, ENS Lyon, 2005.

[Bry86] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, 35(8) :677–691, 1986.

[CE82] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, 1982. Springer-Verlag.

[CGP00] Edmund Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 2000.

[CH78] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the Fifth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 84–97, Tucson, Arizona, 1978. ACM Press, New York, NY.

[CKT03] Samarjit Chakraborty, Simon Kunzli, and Lothar Thiele. A general framework for analysing system properties in platform-based embedded system designs. In *DATE '03 : Proceedings of the conference on Design, Automation and Test in Europe*, page 10190, Washington, DC, USA, 2003. IEEE Computer Society.

[CPHP87] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. Lustre : a declarative language for real-time programming. In *POPL '87 : Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 178–188, New York, NY, USA, 1987. ACM.

[CPT05] Samarjit Chakraborty, Linh T. X. Phan, and P. S. Thiagarajan. Event count automata : A state-based model for stream processing systems. In *RTSS '05 : Proceedings of the 26th IEEE International Real-Time Systems Symposium*, pages 87–98, Washington, DC, USA, 2005. IEEE Computer Society.

[Del] Charles Delbé. Modélisation des apprentissages. `http://leadserv.u-bourgogne.fr/IMG/pdf/1_Introduction-2.pdf`.

[Hal93] N. Halbwachs. A tutorial of lustre. 1993.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language lustre. *Proceedings of the IEEE*, 79(9) :1305–1320, September 1991.

[Jea] Bertrand Jeannet. Nbac format. `http://pop-art.inrialpes.fr/people/bjeannet/nbac/index_4.html`.

[Jea00] Bertrand Jeannet. *Partitionnement dynamique dans l'Analyse de Relations Linéaires et Application à la Vérification de programmes Synchrones.* PhD thesis, INPG, 2000.

[JM09] B. Jeannet and A. Miné. Apron : A library of numerical abstract domains for static analysis. In *Proc. of the 21th Int. Conf. on Computer Aided Verification (CAV 2009)*, volume 5643 of *Lecture Notes in Computer Science*, pages 661–667. Springer, Grenoble, France, June 2009. `http://www.di.ens.fr/~mine/publi/article-mine-jeannet-cav09.pdf`.

[LBT04] Jean-Yves Le Boudec and Patrick Thiran. *Network Calculus : A Theory of Deterministic Queuing Systems for the Internet (Lecture Notes in Computer Science).* Springer, 1 edition, August 2004.

[MA08] Matthieu Moy and Karine Altisen. Connecting real-time calculus to the synchronous programming language lustre. 2008.

[MBBS] Oded Maler, Marius Bozga, and Ramzi Ben Salah. Compositional timing analysis. submitted (2009), available at `http://www-verimag.imag.fr/~maler/Papers/tabst-new.pdf`.

[PCT08] Linh T. X. Phan, Samarjit Chakraborty, and P. S. Thiagarajan. A multi-mode real-time calculus. In *RTSS '08 : Proceedings of the 2008 Real-Time Systems Symposium*, pages 59–69, Washington, DC, USA, 2008. IEEE Computer Society.

[PCTT07] Linh T. X. Phan, Samarjit Chakraborty, P. S. Thiagarajan, and Lothar Thiele. Composing functional and state-based performance models for analyzing heterogeneous real-time systems. In *RTSS '07 : Proceedings of the 28th IEEE International Real-Time Systems Symposium*, pages 343–352, Washington, DC, USA, 2007. IEEE Computer Society.

[PLW] Krčál Pavel, Mokrushin Leonid, and Yi Wang. A tool for compositional analysis of timed systems by abstraction. `https://www.it.uu.se/research/group/darts/papers/texts/kmy-nwpt07.pdf`.

[QS82] J. P. Queille and J. Sifakis. A temporal logic to deal with fairness in transition systems. In *SFCS '82 : Proceedings of the 23rd Annual Symposium on Foundations of Computer Science*, pages 217–225, Washington, DC, USA, 1982. IEEE Computer Society.

[RE02] K. Richter and R. Ernst. Event model interfaces for heterogeneous system analysis. In *DATE '02 : Proceedings of the conference on Design, automation and test in Europe*, page 506, Washington, DC, USA, 2002. IEEE Computer Society.

[Zür] ETH Zürich. Modular performance analysis of distributed embedded real-time systems. `https://www.rdb.ethz.ch/projects/project.php?proj_id=17830`.