HAL
open science

Compilation Modulaire d'un Langage Synchrone

Annie Ressouche, Daniel Gaffé

To cite this version:

Annie Ressouche, Daniel Gaffé. Compilation Modulaire d'un Langage Synchrone. Revue des Sciences et Technologies de l'Information - Série TSI : Technique et Science Informatiques, 2011, Application des méthodes formelles à l'analyse statique et la compilation, 4 (30), pp.441-471. inria-00524499

HAL Id: inria-00524499

https://inria.hal.science/inria-00524499

Submitted on 8 Oct 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Compilation modulaire d'un langage synchrone

Spécification, simulation, implémentation et vérification d'applications synchrones

Daniel Gaffé* — **Annie Ressouche****

* *Laboratoire LEAT, Université de Nice Sophia-Antipolis, CNRS*
250 rue Albert Einstein, 06560 Valbonne France
Daniel.Gaffe@unice.fr

** *INRIA Sophia Antipolis Méditerranée*
2004 route des Lucioles – BP 93, 06902 Sophia Antipolis France
Annie.Ressouche@sophia.inria.fr

RÉSUMÉ. *Dans cet article, nous étudions la compilation modulaire de programmes synchrones impératifs. Nous nous appuyons sur des méthodes formelles pour compiler et valider les applications spécifiées. Nous avons défini et implémenté un langage dédié (*LE*) et sa* `sémantique équationnelle` *qui permet la compilation modulaire des programmes vers différentes cibles logicielles et matérielles (code C, code Vhdl, synthétiseurs fpga, format d'entrée d'outils de vérification, ...). Nous montrons que cette sémantique est correcte et nous introduisons un algorithme pour vérifier la causalité qui respecte notre approche modulaire. En nous appuyant sur cette approche formelle, nous avons réalisé une boîte à outils pour compiler et vérifier des applications réactives synchrones.*

ABSTRACT. *In this paper, we study the modular compilation of imperative synchronous programs. We rely on a formal framework well suited to perform compilation and formal validation of systems. In practice, we design and implement a special purpose language (*LE*) and its* `execution equational semantics` *that allows the modular compilation of programs into software and hardware targets (C code, Vhdl code, FPGA synthesis, Verification tools). We show the correctness of this semantics, and we introduce a new algorithm to check program causality with respect to our modular approach. Relying in this formal approach, we defined a toolkit dedicated to the compilation and the verification of reactive applications.*

MOTS-CLÉS : *langages synchrones, méthodes formelles, compilation séparée, vérification*

KEYWORDS: *synchronous languages, formal methods, separated compilation, verification*

1re soumission à *RSTO - TSI*, le 30/11/2008

1. Introduction

En Sciences, la convergence d'idées émanant de groupes différents issus de cultures scientifiques et techniques elles aussi différentes, est souvent un gage de qualité. La modélisation par systèmes synchrones en est une parfaite illustration : en Électronique Numérique, s'est posé le problème de la synthèse fiable de systèmes séquentiels dès le milieu du 20e siècle (Moore, 1956; Mealy, 1955). Les concepteurs ont rapidement compris qu'il était plus facile de décomposer ce problème en deux parties bien distinctes : (1) la détermination du nouvel état du système à partir d'un état *stable*, connu et clairement identifié sur une échelle discrète du temps (Horloge) : (2) l'étude temporelle de tous les aspects dynamiques du système ; en particulier l'assurance que le système se stabilise effectivement avant la prochaine occurrence de l'horloge et que cette évolution donne toujours la même conclusion dès que le concepteur a garanti la stabilité des entrées. Avec ces deux hypothèses, le temps d'évolution interne ainsi que la manière dont le système évolue localement n'a plus d'importance pour la prévisibilité globale. L'informatique a suivi la même démarche avec les langages synchrones dans les années 80 pour modéliser et implanter les systèmes de commande multitâches : c'est une réponse au problème du non-déterminisme des évolutions concurrentes. Ici "l'horloge" devient "l'instant" et la "stabilité des entrées" se traduit en "atomicité des réactions". Que ce soit en Électronique Numérique ou en Informatique Synchrone, la réaction est vue *à temps nul* car sa durée réelle est *projetée* sur le *cycle d'horloge suivant* qui est *l'instant suivant* du système. On définit ainsi un *temps logique* composé d'une suite de réactions. Cette considération constitue *l'hypothèse synchrone* sur laquelle les langages synchrones (Halbwachs, 1993) tels Esterel et SyncCharts d'une part, et Lustre et Signal d'autre part, s'appuient pour modéliser la dynamique des systèmes.

Le problème majeur rencontré par les langages synchrones est la taille de leur modèles. Bien que ce problème ait fait l'objet de nombreux travaux, il existe toujours un besoin pour une compilation efficace et modulaire. Les premiers compilateurs traduisaient les programmes en automates avec des problèmes d'explosion du nombre d'états. En revanche, le langage Signal (Benveniste *et al.*, 1991) a toujours évité cet écueil car le compilateur construit des arbres d'horloges déduits des contraintes entre signaux. Ensuite, un graphe de dépendances conditionnelles des données, dans lequel les conditions sont les horloges auxquelles ces dépendances sont effectives, est calculé. Les autres langages synchrones ont fait face à ce problème d'explosion du nombre d'états grâce à une compilation en systèmes d'équations booléens qui encodent symboliquement l'automate. En revanche le code généré peut parfois être très lent. Une autre approche consiste à traduire les programmes en graphes d'évènements (Weil *et al.*, 2000) ou en graphes flots de données concurrents (Edwards, 1999; Potop-Butucaru *et al.*, 2004) pour générer du code C efficace.

Toutefois, peu d'approches implémentent les règles d'une sémantique formelle pour compiler séparément les programmes synchrones. Bien sûr, il y a une profonde contradiction dans cette démarche car une telle sémantique doit être *réactive*, *modulaire* et *causale*. Une sémantique est réactive dès que l'on considère un temps logique

et que les évènements d'entrée et de sortie sont simultanés. C'est l'un des fondements de l'hypothèse synchrone. La causalité signifie que pour tout évènement de sortie généré dans une réaction, il y a une chaîne causale qui mène à cette génération ; aucun cycle de causalité ne peut arriver. Une sémantique est modulaire quand la sémantique de la composition de deux programmes peut être déduite des sémantiques respectives de ces programmes. Un autre aspect de la modularité est la vue cohérente de l'état d'un système par tous ses sous-systèmes. Quand un signal est présent, sa présence est instantanément diffusée à chaque partie qui l'écoute. Mais, il existe un théorème (Huizing *et al.*, 1991) montrant qu'une sémantique ne peut pas être à la fois réactive, modulaire et causale. Les sémantiques des langages synchrones sont réactives et modulaires mais la causalité reste un problème et doit être vérifiée globalement sur un programme.

Dans cet article nous décrivons notre travail concernant la compilation modulaire d'un langage synchrone impératif. Notre contribution se résume en trois points : tout d'abord, la définition d'une sémantique équationnelle sur laquelle nous nous appuyons pour compiler les programmes. C'est une sémantique "usuelle" sous la forme de systèmes d'équations mais nous introduisons un cadre mathématique étendu (par rapport aux sémantiques précédentes qui considèrent des systèmes d'équations booléens) et de nouvelles règles sémantiques pour les opérateurs du langage nous permettant effectivement de faire la compilation modulaire. De plus, nous définissons également une sémantique comportementale qui permet d'exprimer formellement les comportements d'un programme dans une structure algébrique autorisant l'application des techniques de vérification formelle.

Ensuite, notre sémantique étant réactive et modulaire, elle ne peut pas être causale. Pour vérifier la causalité des programmes, nous avons défini un algorithme d'ordonnancement pour nos systèmes d'équations qui s'inspire de la méthode PERT (T.I.Kirkpatrick *et al.*, 1966) et qui nous permet de garder toute l'information nécessaire sur l'ordre de l'évaluation des variables d'un système pour assurer sa causalité. En complément, nous avons défini un algorithme de fusion des systèmes d'équations au préalable ordonnées qui respecte notre ordonnancement et nous permet la compilation modulaire.

Finalement, nous avons introduit une opération de finalisation nécessaire à la génération de code, la simulation et le model-checking, qui complète notre approche. En effet, contrairement aux approches booléennes, nous ne décidons jamais qu'un signal non présent est absent au moment de la compilation car ceci empêche toute approche modulaire. Nous déléguons cette décision dans une phase spécifique post compilation que nous appelons "finalisation", préambule à la génération du code cible. Cette approche s'est concrétisée par la création d'une boîte à outils autour du compilateur pour concevoir, simuler, vérifier et générer du code d'applications réactives. Nous illustrons l'utilisation de ces outils sur un exemple simple mais réel.

nothing	ne fait rien
emit S	le signal S est immédiatement présent dans l'environnement
present S { P1} else { P2}	si S est présent $P1$ est exécuté sinon $P2$
$P_1 \gg P_2$	P_1 est exécuté puis P_2
$P_1 \| P_2$	parallèle synchrone : les exécutions de P_1 et P_2 sont commencées simultanément et l'instruction termine quand ces deux exécutions sont terminées
$abort\ P\ when\ S$	P est exécuté normalement jusqu'à l'instant où S est présent
$loop\ \{P\}$	exécute P et recommence dès que cette exécution est terminée
$local\ S\ \{P\}$	la portée de S est restreinte à P
$run\ M$	appel du module M
$pause$	stoppe jusqu'à la prochaine réaction
$wait\ S$	attend la prochaine réaction où S est présent

Tableau 1. *Les opérateurs de* LE

2. Le langage LE

Le langage LE est un langage réactif synchrone. Nous bénéficions de plus de deux décennies d'études et de développement dans ce domaine. LE n'introduit pas de nouveaux concepts pour manipuler le temps logique, mais notre expérience dans la programmation synchrone nous a enseigné qu'il est parfois difficile de décrire naturellement un automate explicite ou implicite par un système d'équations booléens (pour spécifier des contrôleurs par exemple) dans Esterel. LE est un langage synchrone impératif qui permet d'unifier plusieurs styles de spécification pour concevoir des applications :

1) un langage textuel "à la Esterel"(Berry, 2000) permet la conception (de parties) d'applications dirigées par évènements ;

2) un format textuel et graphique d'automates " à la StateChart" permet de spécifier des contrôleurs ;

3) les (parties d') applications flots de données peuvent être décrites avec des systèmes d'équations " à la Lustre" (N. Halbwachs *et al.*, 1992).

LE définit des *modules*. L'interface d'un module déclare les signaux d'entrée qu'il écoute et les signaux de sortie qu'il émet. De plus, il contient aussi la liste des autres modules déjà compilés et appelés dans le module courant. Le corps du module est exprimé à l'aide d'opérateurs. Certains d'entre eux (*wait*, *pause*) sont dédiés à la manipulation du temps logique. Comme dans tout formalisme synchrone, ces opérateurs permettent de définir des applications réactives comme des modules concurrents et communiquants. Un module communique avec son environnement ou avec ses sous

modules par évènements. Des opérateurs permettent de construire des programmes : la table 1 décrit les opérateurs simples du langage. De plus, le langage possède aussi une définition d'automates et de systèmes d'équations. Les automates peuvent être définis syntaxiquement ou bien générés par un éditeur graphique dédié *galaxy* (Gaffé *et al.*, 2008). Les systèmes d'équations calculent la valeur suivante des registres et celle des signaux de sorties en fonction des valeurs des entrées et des registres, ils permettent d'encoder des "Mealy machines synchrones". La grammaire est détaillée dans (Ressouche *et al.*, 2008). Nous voulons juste souligner l'existence d'une instruction *run* qui appelle un module externe, avec renommage possible de ses signaux d'interface. La compilation séparée du langage repose sur cet opérateur. Il permet aussi d'appeler un module dans les états d'un automate.

3. La sémantique équationnelle de LE

Pour réaliser une compilation modulaire des programmes de LE, nous implémentons les règles de sa sémantique *équationnelle*. Cette dernière traduit chaque programme en un système d'équations. Ensuite, nous définissons la sémantique *comportementale* qui permet de donner une interprétation non ambiguë du comportement de chaque programme. Cette dernière calcule l'ensemble des comportements d'un programme, et nous assurons la correction de la sémantique équationnelle en montrant que ces deux sémantiques coïncident. De plus, la sémantique comportementale nous donne le cadre formel nécessaire à l'application des techniques de vérification de "model-checking" (Jr. *et al.*, 2000).

3.1. *Contexte mathématique*

3.1.1. *L'algèbre ξ*

Tout d'abord, nous introduisons le contexte mathématique qui nous permet de définir la sémantique équationnelle de LE. Comme tous les langages synchrones, LE utilise des signaux à diffusion instantanée comme moyen de communication entre sous modules et avec l'environnement. Un programme réagit à des évènements d'entrée en produisant des évènements de sortie. Un *évènement* est un signal qui porte une information sur son état de présence que l'on appelle son "statut". Nous introduisons un ensemble ξ ($\xi = \{\bot, 0, 1, \top\}$) qui nous permet de définir de façon plus fine le statut des signaux qu'une simple considération booléenne. Nous munissons ξ d'un ordre partiel ($\sqsubseteq$) [1] qui lui donne une structure de treillis. Soit S un signal, S^1 (resp S^0) signifie que le signal est présent (resp absent), $S^\bot$ signifie que le statut de S n'est pas déterminé et $S^\top$ correspond à un signal dont on n'a pas pu induire le statut car il possède deux statuts incomparables dans certaines sous-parties du programme.

1. $\bot \sqsubseteq 0 \sqsubseteq \top$; $\bot \sqsubseteq 1 \sqsubseteq \top$; $\bot \sqsubseteq \top$.

$\sqcup$	1	0	$\top$	$\bot$
1	1	$\top$	$\top$	1
0	$\top$	0	$\top$	0
$\top$	$\top$	$\top$	$\top$	$\top$
$\bot$	1	0	$\top$	$\bot$

$\sqcap$	1	0	$\top$	$\bot$
1	1	$\bot$	1	$\bot$
0	$\bot$	0	0	$\bot$
$\top$	1	0	$\top$	$\bot$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$

x	$\neg\, x$
1	0
0	1
$\top$	$\bot$
$\bot$	$\top$

Tableau 2. *Lois internes de ξ*

Nous munissons ξ de trois lois de composition : $\sqcup$, $\sqcap$ et $\neg$. $\sqcup$ est la borne supérieure de ses deux opérandes, $\sqcap$ la borne inférieure et $\neg$ est la loi inverse (voir table 2). ξ muni de ces trois lois est une algèbre de Boole (voir (Ressouche *et al.*, 2008) pour la démonstration). En conséquence, nous appliquerons les théorèmes classiques des algèbres de Boole pour résoudre des systèmes d'équations dont les variables appartiennent à ξ. La sémantique équationnelle calcule les statuts des signaux comme solutions de systèmes d'équations à valeur dans ξ.

3.1.2. *Notion d'environnement*

Un *environnement* est un ensemble d'évènements construit à partir d'un ensemble enumérable de signaux dans lequel chaque signal a un statut unique. Plus précisément, étant donné un ensemble énumérable $\mathcal{S}$ de signaux : $\mathcal{S} = \{S_0, S_1, ..., S_n, ...\}$, on définit une interprétation $\mathcal{I} : \mathcal{S} \longrightarrow \xi$ comme une fonction qui à tout signal $S \in \mathcal{S}$ associe un élément de ξ, son statut. Toute interprétation $\mathcal{I}$ définit un environnement : $E = \{S^x \mid S \in \mathcal{S}, x \in \xi, \mathcal{I}(S) = x\}$. Les lois définies dans ξ s'étendent naturellement aux environnements, soient E et E' deux environnements :

$$\begin{array}{lcl} E \sqcup E' & = & \{S^z | \exists S^x \in E, \exists S^y \in E', z = x \sqcup y\} \\ & & \cup \{S^x | S^x \in E, \nexists y \in \xi, S^y \in E'\} \\ & & \cup \{S^y | S^y \in E', \nexists x \in \xi, S^x \in E\} \\ E \sqcap E' & = & \{S^z | \exists S^x \in E, \exists S^y \in E', z = x \sqcap y\} \\ \neg E & = & \{S^x | S^{\neg\, x} \in E\}. \end{array}$$

Nous définissons un ordre ($\preceq$) sur les environnements comme suit :

$$E \preceq E' \text{ ssi } \forall S^x \in E, \exists S^y \in E' | x \sqsubseteq y$$

L'ensemble des environnements construit à partir d'un ensemble de signaux muni de la relation d'ordre $\preceq$ est un CPO (ordre partiel complet). Les fonctions $\sqcup$ et $\sqcap$ sont monotones pour cet ordre.

C'est dans ce cadre mathématique que nous définissons la sémantique équationnelle de LE.

3.2. *Définition de la sémantique équationnelle*

3.2.1. *Notion de circuit ξ*

Comme nous l'avons évoqué dans l'introduction, la sémantique des langages synchrones est réactive dans l'instant et modulaire. En conséquence, elle ne peut pas être causale. Pour résoudre ce problème spécifique de vérification de la causalité, des sémantiques constructives ont été introduites (Berry, 1996). De telles sémantiques sont l'application de la logique booléenne constructive à la définition d'une sémantique pour les langages synchrones. L'idée majeure des sémantiques constructives est de s'appuyer uniquement sur la propagation de valeurs pour calculer les valeurs des sorties. Un programme est *constructif* si et seulement si la propagation des statuts des signaux d'entrée est suffisante pour déduire le statut de tous les signaux. Un moyen élégant de définir la sémantique constructive est d'associer un circuit constructif à tout programme. Ainsi, tout programme qui ne comporte pas de cycle de dépendance instantanée entre ses signaux peut être traduit en un circuit sans cycle.

La sémantique équationnelle de LE respecte ce principe de constructivité. Elle associe effectivement un circuit à tout programme, mais l'aspect quadri-valué que nous avons introduit permet de ne pas ordonner `présent` et `absent` entre eux et de faire d'`absent` une vraie information (différente de `absent` ou `inconnu` d'Esterel) qui ne peut croître qu'en devenant `erreur` (et non `présent`). En effet, cette sémantique équationnelle interprète tout programme en un circuit ξ séquentiel quadri-valué. Un tel circuit est un système d'équations qui permet de déterminer le statut des signaux et des registres d'un environnement de sortie en fonction d'un environnement d'entrée, conformément à une *loi de propagation constructive*. Dans ce contexte, un environnement E est l'union disjointe d'évènements relatifs à des signaux (d'entrée, de sortie et locaux) et d'évènements relatifs à des registres, valués dans ξ.

Pour traduire un programme en circuit, nous définissons un ensemble fini de fils ($\mathcal{W}$) pour chaque programme. Cette traduction étant réalisée à partir de la structure du programme, en premier, nous associons un circuit à chaque instruction du programme et la connexion des fils entre les circuits des différentes instructions du programme assure la propagation de flots de contrôle. Le circuit associé à une instruction a trois fils spécifiques :

1) un fil (*Set*) qui démarre l'instruction ;

2) un fil (*Reset*) qui stoppe et réinitialise l'instruction ;

3) un fil (*RTL*) (Ready To Leave) qui devient vrai dès que l'instruction peut terminer dans l'instant courant.

De plus, certaines instructions ont besoin de registres quand la valeur d'un de ses fils est nécessaire pour calculer la prochaine réaction.

Pour calculer un environnement de sortie à partir d'un environnement d'entrée et d'un circuit, nous utilisons une loi de propagation constructive. Ainsi nous pouvons

$$E \vdash v \hookrightarrow v \qquad \frac{E(w) = v}{E \vdash w \hookrightarrow v} \qquad \frac{E \vdash e \hookrightarrow \neg v}{E \vdash \neg e \hookrightarrow v}$$

$$\frac{E \vdash e \hookrightarrow \top \text{ ou } E \vdash e' \hookrightarrow \top}{E \vdash e \sqcup e' \hookrightarrow \top} \qquad \frac{E \vdash e \hookrightarrow \bot \text{ ou } E \vdash e' \hookrightarrow \bot}{E \vdash e \sqcap e' \hookrightarrow \bot}$$

$$\frac{(E \vdash e \hookrightarrow 1 \text{ et } E \vdash e' \hookrightarrow 0) \text{ ou } (E \vdash e \hookrightarrow 0 \text{ et } E \vdash e' \hookrightarrow 1)}{E \vdash e \sqcup e' \hookrightarrow \top \text{ et } E \vdash e \sqcap e' \hookrightarrow \bot}$$

$$\frac{(E \vdash e \hookrightarrow 1 \text{ et } E \vdash e' \hookrightarrow \bot) \text{ ou } (E \vdash e \hookrightarrow \bot \text{ et } E \vdash e' \hookrightarrow 1)}{E \vdash e \sqcup e' \hookrightarrow 1 \text{ et } E \vdash e \sqcap e' \hookrightarrow \bot}$$

$$\frac{(E \vdash e \hookrightarrow 0 \text{ et } E \vdash e' \hookrightarrow \bot) \text{ ou } (E \vdash e \hookrightarrow \bot \text{ et } E \vdash e' \hookrightarrow 0)}{E \vdash e \sqcup e' \hookrightarrow 0}$$

$$\frac{(E \vdash e \hookrightarrow 0 \text{ et } E \vdash e' \hookrightarrow \top) \text{ ou } (E \vdash e \hookrightarrow \top \text{ et } E \vdash e' \hookrightarrow 0)}{E \vdash e \sqcap e' \hookrightarrow 0}$$

$$\frac{E \vdash e \hookrightarrow v \text{ et } E \vdash e' \hookrightarrow v}{E \vdash e \sqcup e' \hookrightarrow v \text{ et } E \vdash e \sqcap e' \hookrightarrow v}$$

$$\frac{(E \vdash e \hookrightarrow \top \text{ et } E \vdash e' \hookrightarrow 1) \text{ ou } (E \vdash e \hookrightarrow 1 \text{ et } E \vdash e' \hookrightarrow \top)}{E \vdash e \sqcap e' \hookrightarrow 1}$$

Tableau 3. *Définition de la loi de propagation constructive dans ξ ($\hookrightarrow$). $E(w)$ est la valeur de w dans E, v représente un élément quelconque de ξ .*

assurer que les solutions construites sont logiquement correctes, et elles suffisent pour déterminer les statuts de tous les signaux dans chaque réaction.

Soient $\mathcal{C}$ un circuit et E un environnement d'entrée, la loi de propagation constructive est un jugement de la forme : $E \vdash e \hookrightarrow v$, e étant une ξ expression, v appartenant à ξ. Cette loi signifie qu'à partir des valeurs affectées aux signaux et aux registres dans E, e s'évalue à v. La loi de propagation est détaillée dans la table 3. Nous définissons aussi la loi de propagation constructive pour les équations et les circuit ξ : si $E \vdash e \hookrightarrow v$ alors $E \vdash (w = e) \hookrightarrow v$. Par extension, $E \vdash \mathcal{C} \hookrightarrow E'$ signifie que l'environnement E' est le résultat de l'application de la loi de propagation à chaque équation du circuit $\mathcal{C}$, avec E comme environnement d'entrée.

Par ailleurs, pour exprimer la sémantique équationnelle, nous avons besoin d'une opération de "translation temporelle" sur les environnements que nous appelons $\mathcal{P}re$:

$$\mathcal{P}re(E) = \{S^{\perp} \mid S^x \in E\} \cup \{S^x_{pre} \mid S^x \in E\}$$

Cette opération duplique les évènements de l'environnement, tout évènement S^x est renommé S^x_{pre} et l'évènement $S^{\perp}$ est ajouté. Son statut sera *raffiné* par les opérations sur l'environnement effectuées par la sémantique.

3.2.2. *Règles de la sémantique équationnelle*

La sémantique équationnelle est structurelle, nous devons la définir sur les instructions du langage avant de l'étendre au programme. La sémantique équationnelle $\mathcal{S}_e$ est une fonction qui calcule un environnement de sortie à partir d'un environnement d'entrée et d'une instruction LE. Soient p une instruction et E un environnement d'entrée, nous notons $\mathcal{C}(p)$ son circuit ξ et $\langle p \rangle_E$ l'environnement de sortie calculé par $\mathcal{S}_e$; celui-ci est défini comme suit : $\mathcal{S}_e(p, E) = \langle p \rangle_E$ ssi $E \vdash \mathcal{C}(p) \hookrightarrow \langle p \rangle_E$. Finalement, soient P un programme et E un environnement global d'entrée (c'est-à-dire, un environnement où les signaux de sortie et locaux ont $\perp$ pour statut et les registres 0), la sémantique équationnelle formalise une réaction de P en fonction de E : $(P, E) \longmapsto E'$ ssi $\mathcal{S}_e(\beta(P), E) = E'$, $\beta(P)$ étant l'instruction LE, corps de P [2].

Le circuit correspondant à une instruction a trois fils spécifiques appartenant à $\mathcal{W}$ (voir section 3.2.1). Pour exprimer le circuit associé à une instruction p, nous utiliserons la convention suivante : RTL_p, Set_p and $Reset_p$ sont respectivement les fils de RTL, Set et $Reset$ du circuit $\mathcal{C}(p)$.

Pour illustrer la construction de l'environnement de sortie, nous ne pouvons pas présenter la sémantique de tous les opérateurs. Nous avons donc choisi de décrire la sémantique équationnelle de deux opérateurs significatifs : *wait* qui manipule le temps synchrone et *parallèle* qui est un opérateur spécifique des langages synchrones.

Considérons l'opérateur $wait\ S$, il ne termine pas dans l'instant courant. Il termine quand le signal S est présent dans l'environnement. Voici la définition de son circuit :

$$\mathcal{C}_{wait\ S} = \left[\begin{array}{lcll} R+ & = & (Set_{wait\ S} \sqcap \neg Reset_{wait\ S}) \sqcup & \\ & & (R \sqcap \neg Reset_{wait\ S} \sqcap \neg S) & (1) \\ RTL_{wait\ S} & = & R \sqcap S & (2) \end{array} \right]$$

Ce circuit a besoin d'un registre (R) pour mémoriser si l'instruction a été activée. Les registres sont des variables particulières car ils propagent une information d'un instant vers l'instant suivant. Dans un système d'équations, on doit noter et conserver leurs valeurs pour l'instant futur. Nous notons $R+$ la valeur de R calculée à l'instant t et utilisée à l'instant $t + 1$ (en tant que R). Les registres sont initialisés à 0. Dans $\mathcal{C}_{wait\ S}$, l'équation (1) met le registre R à vrai (pour l'instant suivant) quand l'instruction n'est pas réinitialisée ($Reset_{wait\ S}$ n'est pas vraie). Dans ce cas précis, soit l'instruction est initialisée ($Set_{wait\ S}$ est vraie), soit elle a déjà été initialisée (R est

2. c'est à dire l'instruction racine de l'arbre syntaxique du programme P.

vrai) et le signal attendu n'est pas présent. L'équation (2) exprime que l'instruction peut terminer si nous ne sommes pas au premier instant et si le signal attendu est présent.

L'environnement de sortie $\langle P_{wait\ S}\rangle_E$ résulte de l'application de $\mathcal{P}re$ à l'environnement provenant de l'application de la loi de propagation au circuit de l'opérateur :

$\langle P_{wait\ S}\rangle_E = \mathcal{P}re(E')$ et $E \vdash \mathcal{C}(P_{wait\ S}) \hookrightarrow E'$ [3]

Maintenant, considérons l'opérateur de mise en parallèle $(P_1 \| P_2)$: l'environnement de sortie $\langle P_1 \| P_2\rangle_E$ contient les évènements dont le statut est la borne supérieure des statuts des signaux communs à P_1 et P_2 : si $S^x \in \langle P_1\rangle_E$ et $S^y \in \langle P_2\rangle_E$ alors $S^{x \sqcup y} \in \langle P_1 \| P_2\rangle_E$. De plus, le parallèle peut terminer quand ses deux arguments le peuvent (comme tout parallèle synchrone). La règle pour construire l'environnement de sortie est la suivante :

$\langle P_1\rangle_E \sqcup \langle P_2\rangle_E \vdash \mathcal{C}(P_1) \cup \mathcal{C}(P_2) \cup \mathcal{C}_{P_1\|P_2} \hookrightarrow \langle P_1 \| P_2\rangle_E$

Et le circuit de l'opérateur est défini par les équations :

$$\mathcal{C}_{P_1\|P_2} = \left[\begin{array}{lcl} Set_{P_1} & = & Set_{P_1\|P_2} \\ Set_{P_2} & = & Set_{P_1\|P_2} \\ Reset_{P_1} & = & Reset_{P_1\|P_2} \\ Reset_{P_2} & = & Reset_{P_1\|P_2} \\ R_1^+ & = & R_1 \sqcap \neg RTL_{P_2} \sqcap \neg Reset_{P_1\|P_2} \\ & & \sqcup \neg R_2 \sqcap RTL_{P_1} \sqcap \neg RTL_{P_2} \sqcap \neg Reset_{P_1\|P_2} \\ R_2^+ & = & R_2 \sqcap \neg RTL_{P_1} \sqcap \neg Reset_{P_1\|P_2} \\ & & \sqcup \neg R_1 \sqcap \neg RTL_{P_1} \sqcap RTL_{P_2} \sqcap \neg Reset_{P_1\|P_2} \\ RTL_{P_1\|P_2} & = & R_1 \sqcap \neg R_2 \sqcap RTL_{P_2} \sqcup \\ & & (\neg R_1 \sqcap RTL_{P_1} \sqcap (R_2 \sqcup RTL_{P_2})) \end{array}\right]$$

Notons que ce circuit a besoin de deux registres R_1 et R_2 pour mémoriser les valeurs respectives des RTL de ses arguments car il peut terminer quand chacun de ses deux argument termine dans l'instant ou a terminé dans un instant antérieur. Ce circuit est complexe, les deux registres permettant d'encoder 4 états internes : un état (1) où P_1 et P_2 ont terminé ($RTL_{P_1\|P_2}$ devient vrai), un état (2) où P_1 a terminé et attend P_2, un état (3) où P_2 a terminé et attend P_1, et un état (4) où ni P_1 et P_2 n'ont terminé.

Les règles de la sémantique équationnelle sont directement opérationnelles. Elles permettent d'associer un ξ système d'équations à tout programme et l'évaluation de ce dernier est faite en appliquant une loi de propagation constructive. Un programme est causal lorsque l'on peut ordonner son système d'équations. Nous savons que cet

3. L'opération $\mathcal{P}re$ sur l'environnement est nécessaire car l'opérateur ne réagit pas dans l'instant. Si les équations de l'opérateur concerne le statut de S à un instant t, les instructions englobantes considèrent le statut de S à un instant $t+1$. Par exemple, dans le circuit ξ de l'instruction : $wait\ S \gg wait\ S$, les équations relatives à chaque $wait\ S$ manipulent deux instances différentes du signal S.

ordre ne peut être que global, notre sémantique étant réactive dans l'instant et modulaire. Toutefois, grâce à l'introduction de l'algèbre ξ pour représenter le statut des signaux, et à la définition d'un nouvel algorithme de tri basé sur la méthode PERT (T.I.Kirkpatrick *et al.*, 1966) nous pouvons ordonner le système d'équation d'un programme en utilisant les systèmes d'équations déjà ordonnés de ses sous modules.

3.3. *Correction de la sémantique équationnelle*

La sémantique équationnelle nous fournit un moyen pour compiler les programmes LE. Dans cette section, nous désirons prouver que cette sémantique est correcte. Pour cela, nous définissons une sémantique *comportementale* de LE qui caractérise formellement le comportement d'un programme. Ensuite, nous prouvons que ces deux sémantiques calculent la même valeur pour les signaux de sorties à chaque réaction.

3.3.1. *Définition de la sémantique comportementale*

Pour définir la sémantique comportementale, nous formalisons la notion de concurrence et nous utilisons un cadre algébrique qui nous permet de décrire les comportements d'un programme de façon formelle. Ce cadre algébrique est totalement détaillé dans (Ressouche *et al.*, 2008). Nous y définissons une algèbre de processus dans laquelle chaque opérateur de LE correspond à un terme de l'algèbre. Les règles de la sémantique comportementale sont définies dans cette algèbre.

La sémantique comportementale formalise une réaction d'un programme P en fonction d'un environnement d'entrée : $(P, E) \longmapsto (P', E')$ signifie que le programme P réagit à l'environnement E en produisant l'environnement E' et en atteignant un nouvel état représenté par P'. Cette sémantique peut être décrite à l'aide d'un ensemble de règles de réécritures qui décrivent le comportement de chaque opérateur du langage. Une règle a la forme : $p \xrightarrow[E]{E', TERM} p'$ où p et p' sont des instructions du langage. E est un environnement qui spécifie le statut des signaux qui sont dans la portée de p et *TERM* est une variable booléenne indiquant si l'instruction termine dans l'instant. Soient P un programme LE et E un environnement d'entrée, $\beta(P)$ l'instruction qui représente son corps, une réaction est calculée comme suit :

$$(P, E) \longmapsto (P', E') \quad \text{ssi} \quad \beta(P) \xrightarrow[E]{E',\, TERM} \beta(P')$$

Nous ne détaillons pas les règles des opérateurs. Elles sont usuelles et décrites complètement dans (Ressouche *et al.*, 2008).

La sémantique comportementale est une "macro" sémantique qui calcule un environnement de sortie comme le plus petit point fixe d'une suite d'applications des règles de réécriture des opérateurs en appliquant l'opérateur $\sqcup$ sur les environnements de sortie. $\sqcup$ est monotone pour l'ordre $\preceq$, donc ce plus petit point fixe existe. En pra-

tique, nous avons : $p \xrightarrow[E]{E',\, TERM} p'$ s'il existe une suite d'approximations successives de l'environnement de sortie de la forme :

$$p \xrightarrow[E]{E_1,\, TERM_1} p_1, p_1 \xrightarrow[E_1]{E_2,\, TERM_2} p_2, \dots p_{n-1} \xrightarrow[E_{n-1}]{E'.\, TERM_n} p'$$

A chaque étape $E_{i+1} = F^i(E_i)$, F étant une combinaison de $\sqcup$ et de l'identité, elle est monotone et continue. Elle admet donc un plus petit point fixe (Tarski, 1955) et ainsi $E' = F^n(E)$.

3.3.2. *Équivalence des deux sémantiques*

La sémantique comportementale décrit comment un programme réagit dans l'instant. Elle est logiquement correcte car elle calcule un unique environnement de sortie pour chaque environnement d'entrée quand le programme est causal.

Pour compléter notre approche, nous montrons que les deux sémantiques coïncident : dans chaque réaction, la sémantique équationnelle et la sémantique comportementale calculent les mêmes valeurs pour les signaux de sortie d'un programme.

Théorème 1 *Soient p une instruction* LE, *O un ensemble des signaux de sortie et $E_{\mathcal{C}}$ un environnement. $\langle p \rangle_{E_{\mathcal{C}}}$ est l'environnement de sortie calculé par la sémantique équationnelle. La propriété suivante est vérifiée :*

$$\exists p' \text{ tel que } p \xrightarrow[E]{E',\, RTL_p} p' \; et \; \langle p \rangle_{E_{\mathcal{C}}} \restriction_O = E' \restriction_O$$

$E \restriction_X$ est la restriction de l'environnement E aux évènements relatifs aux signaux de X : $E \restriction_X = \{S^z | S^z \in E \; and \; S \in X\}$. RTL_P (fil RTL de $\mathcal{C}(p)$) est un fil de contrôle de la sémantique équationnelle. Il ne peut prendre comme valeur que 0 ou 1. On peut donc montrer qu'il est égal au booléen de terminaison de la sémantique comportementale. La preuve du théorème est détaillée dans (Ressouche *et al.*, 2008). On y définit la taille d'une instruction LE et le théorème est prouvé par récurrence sur cette taille.

4. Compilation modulaire

Pour effectivement compiler les programmes LE, nous utilisons les règles de la sémantique équationnelle (voir section 3.2) pour générer le ξ système d'équations de chaque programme. Mais utiliser un système d'équations pour générer du code, simuler ou faire une intégration dans du code externe nécessite de trouver un ordre, valide à tous les instants, pour évaluer les équations. Habituellement, dans l'approche synchrone, cet ordre est déterminé statiquement. Cette approche interdit toute compilation séparée comme l'illustre la figure 1. Deux scénarios de compilation du module

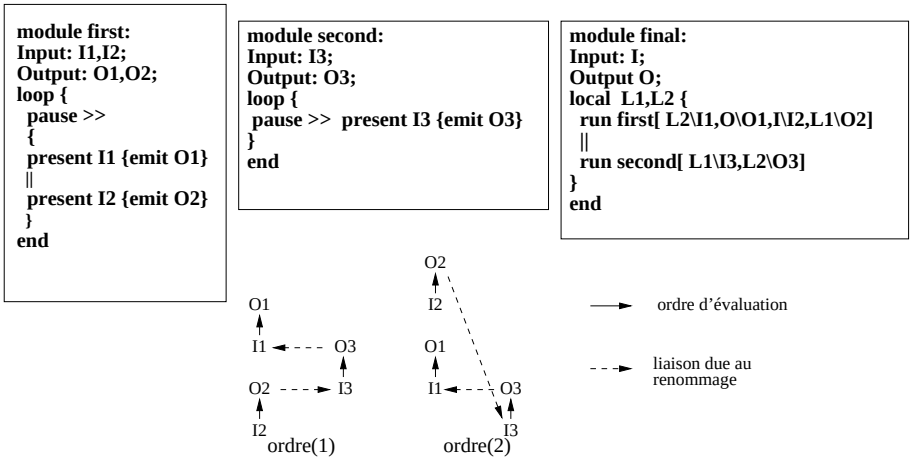

Figure 1. *Cycle de causalité :les signaux O1, O2 et O3 sont indépendants. Mais, le choix d'un ordre total peut introduire un cycle de causalité. Si l'ordre (1) est choisi, dans le module final, après renommage, nous obtenons le système d'équations : { L1 = I, L2 = L1, O = L2 } qui est correctement ordonné. En revanche, si l'ordre (2) est choisi, dans le module final nous obtenons : { L2 = L1, O = L2,L1 = I } qui a un cycle de causalité. Notons que le renommage en* LE *est fait par association des noms des signaux et indépendemment de l'ordre des déclarations*

final sont montrés. Le premier permet d'obtenir un système d'équations bien ordonné, le second introduit un cycle de causalité qui empêche toute génération de code.

Pour éviter ces situations, les signaux indépendants ne doivent pas être reliés par un ordre arbitraire : nous désirons construire un ordre partiel incrémental. Pour cela, nous gardons suffisamment d'information sur la causalité des signaux (grâce au ξ statut attribué à chaque signal). Pratiquement, à chaque variable du système d'équations, nous associons deux variables entières (*CanDate*, *MustDate*) qui indiquent la date à laquelle la variable *peut* et *doit* être évaluée. En fait, la *MustDate* est l'ultime date après laquelle l'évaluation du reste du système va prendre du retard. Une date représente un niveau d'évaluation. La date 0 caractérise les variables évaluées en premier parce qu'elles ne dépendent d'aucune autre variable. Une date $n + 1$ caractérise les variables qui dépendent de variables de dates inférieures pour être évaluées. Les variables de même date sont indépendantes et peuvent être évaluées dans n'importe quel ordre (entre elles). *CanDate* caractérise la date à laquelle une variable est évaluée par rapport aux entrées du système. *MustDate* caractérise la date à laquelle une variable est évaluée par rapport aux sorties du système.

Cette technique s'inspire de la méthode PERT (T.I.Kirkpatrick *et al.*, 1966), connue depuis plusieurs décennies dans la gestion de la production industrielle. Cette

Equations1 :	$can(x) = \{a, b\}$	$must(x) = \emptyset$
$a = x \sqcup y$	$can(y) = \{a, b\}$	$must(y) = \emptyset$
$b = x \sqcup not\ y$	$can(t) = \{c, e\}$	$must(t) = \emptyset$
$c = a \sqcup t$	$can(a) = \{c, e, d\}$	$must(a) = \{x, y\}$
$d = a \sqcup c$	$can(b) = \emptyset$	$must(b) = \{x, y\}$
$e = a \sqcap t$	$can(c) = \{d\}$	$must(c) = \{a, t\}$
	$can(d) = \emptyset$	$must(d) = \{a, c\}$
	$can(e) = \emptyset$	$must(e) = \{a, t\}$

Tableau 4. *exemple du calcul des graphes de dépendances can et must pour le système Equations1.*

approche nous permet de construire des ordres partiels et de ne choisir un ordre total que pour une utilisation future (simulation, génération de code,...).

4.1. *Calcul des ordres partiels*

Notre algorithme comporte deux phases : la première construit un ordre partiel initial qui représente les dépendances de variables du système d'équations. C'est un ensemble de dépendances partielles. La seconde phase propage récursivement les *CanDate* et *MustDate* des variables, en suivant l'ordre partiel préalablement construit. Si durant cette phase de propagation un cycle est trouvé [4], il y a un réel cycle de causalité dans le programme. Quand il n'y a pas de cycle, la propagation se termine car il y a un nombre fini de variables dans un système d'équations.

4.1.1. *Principe de l'algorithme*

Plus précisément, la première phase de l'algorithme parcourt le système d'équations et construit deux graphes de dépendances : Γ_{can} et Γ_{must}. Dans ces graphes de dépendances, on relie une variable X à toute variable X' dont elle dépend pour être évaluée. En fait, $\Gamma_{can} = \{V_S, \stackrel{can}{\rightarrow}\}$ et $\Gamma_{must} = \{V_S, \stackrel{must}{\rightarrow}\}$. V_S est l'ensemble (fini) des variables du système d'équations S. On le considère comme l'ensemble des noeuds des graphes Γ_{can} et Γ_{must}. Ensuite, $\stackrel{can}{\rightarrow}$ (resp. $\stackrel{must}{\rightarrow}$) est une relation binaire qui relie deux noeuds du graphe : $(X, X') \in \stackrel{can}{\rightarrow}$ (notée $X \stackrel{can}{\rightarrow} X'$) si et seulement si X' est nécessaire au calcul de X (c'est-à-dire s'il existe dans S une équation de la forme $X = f(X')$). De façon duale, $X \stackrel{must}{\rightarrow} X'$ si et seulement si X est nécessaire au calcul de X' (c'est-à-dire si il existe dans S une équation de la forme $X' = f(X)$). Les variables d'entrée du système d'équations sont des feuilles de Γ_{can} (car elles n'ont

4. Dès qu'une *CanDate* ou une *MustDate* est supérieure au nombre de variables N du système. En pratique, la divergence est poussée jusqu'à $2 \times N$ pour identifier toutes les variables impliquées dans l'éventuel cycle de causalité.

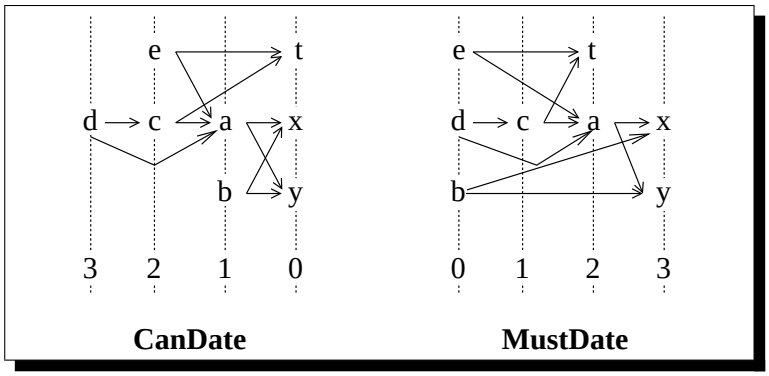

Figure 2. *CanDate et MustDate pour les variables de Equations1*

besoin d'aucune variable pour être calculée) tandis que les variables de sortie sont des feuilles de Γ_{must}. À partir de ces graphes, à chaque variable X on associe *can(X)* l'ensemble de ses dépendances can, et *must(X)*, l'ensemble de ses dépendances must : $can(X) = \{X' \mid X' \stackrel{can}{\rightarrow} X \in \Gamma_{can}\}$ et $must(X) = \{X' \mid X \stackrel{must}{\rightarrow} X' \in \Gamma_{must}\}$. Le tableau 4 illustre le calcul de ces dépendances pour le système *Equations1*. Toutefois, notons que can et $must$ ne sont pas des relations inverses, leur composition ne donnant pas toujours l'identité. Par exemple, considérons la variable a du système $Equations1$: $must(can(a)) = \{a, t, c\}$ et $can(must(a)) = \{a, b\}$.

Ensuite, la propagation des *CanDate* et des *MustDate* est effectuée. Au départ, toutes les variables sont considérées comme indépendantes et leur *CanDate* ainsi que leur *MustDate* sont initialisées à 0. Les dépendances *can* servent à calculer les *CanDate* des variables. Pour chaque variable, on incrémente de 1 la *CanDate* des variables de son ensemble de dépendances *can*. De façon duale, les dépendances *must* servent à calculer les *MustDate* des variables. Pour chaque variable, on incrémente de 1 la *MustDate* des variables dans son ensemble de dépendances *must*. La figure 2 illustre le résultat obtenu pour le système $Equations1$. L'algorithme mis en jeu calcule les ensembles de dépendances *can* et *must* "à la volée" et les *CanDate* et *MustDate* des variables simultanément avec une complexité que nous estimons de *n log n*.

4.1.2. *Description de l'algorithme*

L'algorithme *tri_equations* (voir Algorithme 1) comporte une phase d'initialisation qui met toutes les *CanDate* et *MustDate* des variables à 0. Pour tout système d'équations S, on définit une variable globale $borne_S$ qui est la date maximale atteinte. Ensuite l'algorithme appelle deux fonctions récursives : *propagation_CanDate* (voir Algorithme2) et *propagation_MustDate* (voir Algorithme3) sur toutes les variables de S à partir de la date 0. Les algorithmes sont décrits en pseudo code avec les conventions suivantes : V_S est l'ensemble des variables de S, $CanDate_S(v)$ est

la *CanDate* de v dans S et $MustDate_S(v)$ sa MustDate. En revanche, nous ne détaillons pas le calcul des graphes $\Gamma_{can}(S)$ et $\Gamma_{must}(S)$ qui résultent d'un parcours classique du système d'équations S. Dans la description de l'algorithme, nous notons can_S (resp. $must_S$) le tableau des dépendences can (resp. must) des variables de S.

Algorithme 1 tri_equations

Entrées: S {le système d'équations}
Entrées: $\Gamma_{can}(S)$ {graphe des dépendances can de S}
Entrées: $\Gamma_{must}(S)$ {graphe des dépendances $must$ de S}
  $can_S = \emptyset$
  $must_S = \emptyset$
  pour $v \in V_S$ **faire**
    $can_S[v] = \{v' \mid v' \xrightarrow{can} v \in \Gamma_{can}(S)\}$
    $must_S[v] = \{v' \mid v \xrightarrow{must} v' \in \Gamma_{must}(S)\}$
  fin pour
  pour $v \in V_S$ **faire**
    $CanDate_S(v) = 0$
    $MustDate_S(v) = 0$
  fin pour
  $borne_S = 0$
  pour $v \in V_S$ **faire**
    propagation_CanDate(can_S, v, 0)
    propagation_MustDate($must_S$, v, 0)
  fin pour

Algorithme 2 propagation_CanDate

Entrées: can_S {le tableau des dépendances can des variables}
Entrées: v {la variable concernée par la propagation}
Entrées: $date$ {la date propagée}
  si $CanDate_S(v) \leq date$ **alors**
    $CanDate_S(v) = date$
    $borne_S = max(borne_S, date + 1)$
    si $borne_S \leq 2 * card(V)$ **alors**
      pour $x \in can_S[v]$ **faire**
        propagation_CanDate(can_S, x, $date + 1$)
      fin pour
    sinon
      ERREUR("Cycle de causalité")
    finsi
  finsi

4.1.3. *Calcul effectif des ordres partiels*

Le calcul des *CanDate* et *MustDate* des variables d'un système nous permet de calculer les ordres partiels valides de celui-ci. En effet, la *CanDate* d'une variable est

Algorithme 3 propagation_MustDate

Entrées: $must_S$ {le tableau des dépendances $must$ des variables}
Entrées: v {la variable concernée par la propagation}
Entrées: $date$ {la date propagée}
  si $MustDate_S(v) \leq date$ **alors**
    $MustDate_S(v) = date$
    $borne_S = max(borne_S, date + 1)$
    si $borne_S \leq 2 * card(V_S)$ **alors**
      pour $x \in must_S[v]$ **faire**
        propagation_MustDate($must_S$, x, $date + 1$)
      fin pour
    sinon
      ERREUR(“Cycle de causalité”)
    finsi
  finsi

la date au plus tôt où elle peut être évaluée. La *MustDate* représente la date d'évaluation au plus tard d'une variable dans un repère où les sorties ont 0 pour date. Ceci afin de calculer les deux dates des variables de manière symétrique et d'éviter de calculer la date maximale au préalable. Pour trouver la réelle date au plus tard d'évaluation, il faut considérer que les sorties ont pour date au plus tard N (la date maximale du système qui correspond à la longueur du plus grand chemin qui relie une variable d'entrée à une variable de sortie dans Γ_{can} et Γ_{must} ; appelé aussi chemin critique). En conséquence, la date au plus tôt d'une variable est sa *CanDate*. Pour calculer sa date au plus tard, il faut effectuer une symétrie de sa *MustDate* qui change 0 en N et N en 0. Ainsi, si $MustDate_S(x) = n$ alors x devra être évaluée au plus tard à la date $N - n$. Sur l'exemple $Equations1$, la date maximale est 3 et les dates au plus tôt et au plus tard des variables est décrite dans le tableau 5. Toute variable x peut être évaluée à toute date comprise entre sa date au plus tôt et sa date au plus tard. Elle possède donc un certain nombre de dates valides auxquelles elle peut être évaluée : $N - MustDate_S(x) - CanDate_S(x) + 1$. Appelons $\delta_S(x)$ le nombre de dates valides de x dans S. Le nombre d'ordonnancements valides de S est : $\prod_{x \in V_S} \delta_S(x)$. Sur l'exemple $Equations1$, les nombres de dates des variables sont spécifiés dans le tableau 5. Toutes les variables ont pour nombre de dates 1 sauf b(3), e(2) et t(2). Le nombre d'ordonnancements valides pour $Equations1$ est donc 12. Notons que les variables qui n'ont qu'une seule date possible, caractérisent le chemin critique du système d'équations. Ce sont les variables “pivots” que l'on doit évaluer à une date bien précise. Toutefois, à une date donnée, les variables sont indépendantes entre elles et peuvent être évaluées quelque soit l'ordonnancement choisi entre elles. Le nombre d'ordres totaux calculés précédemment ne tient pas compte des permutations possibles à une date donnée.

	a	b	c	d	e	x	y	t
date au plus tôt	1	1	2	3	2	0	0	0
date au plus tard	1	3	2	3	3	0	0	1
nombre de dates	1	3	1	1	2	1	1	2

Tableau 5. *Dates au plus tôt et au plus tard d'évaluation des variables de Equations1. La date maximale est 3 la date au plu tôt d'une variable x est $CanDate_{Equations1}(x)$ et sa date au plus tard est $3 - MustDate_{Equations1}(x)$.*

4.2. *Fusion de deux ordres partiels*

Notre approche permet d'intégrer un système d'équations trié dans un autre système d'équations trié lui aussi sans recalculer les ordres partiels depuis le stade initial. Ce qui nous permet de faire une compilation modulaire. Considérons deux systèmes d'équations A et B déjà triés avec l'algorithme *tri_equations*. Pour appliquer l'algorithme de fusion (voir algorithme 4), nous considérons qu'une variable a une seule définition dans $A \cup B$ [5]. Comme nous utilisons l'algorithme de fusion pour intégrer le code d'un module appelé dans celui d'un module appelant, nous n'avons jamais besoin de redéfinir des variables. Même si les signaux d'interface du module appelé portent le même nom que dans le module appelant, les variables qui les représentent dans le système d'équations sont différentes. Cette considération n'est donc pas une restriction.

Pour fusionner les systèmes A et B et obtenir un système d'équations global trié, nous ne propageons que les *CanDate* et *MustDate* des variables communes à A et B. Si la *CanDate* de x dans A est inférieure à sa *CanDate* dans B, on propage récursivement $CanDate_B(x)$ dans les dépendances can de x dans $A \cup B$ [6]. Si au contraire la *CanDate* de x dans A est supérieure à sa *CanDate* dans B, on propage $CanDate_A(x)$. Si les deux *CanDate* sont égales, on ne fait rien car les dépendances sont déjà correctement pris en compte dans A et B. Ensuite, les *MustDate* sont propagées d'une façon analogue. L'algorithme 4 décrit notre technique de fusion de deux ordres partiels en pseudo code

Pour illustrer le fonctionnement de cet algorithme, nous supposons que l'on veut intégrer le système d'équations (*Equations2*) :

$$\begin{array}{lcl} y & = & m \\ z & = & d \\ v & = & w \end{array}$$

5. $A \cup B$ est le système constitué par l'union des équations de A et de B.
6. $can_{A\cup B}(x) = \{x' \mid x' \xrightarrow{can} x \in \Gamma_{can}(A \cup B)\}$; chaque variable a une définition unique dans $A \cup B$ donc $can_{A\cup B}(x) = can_A(x) \cup can_B(x)$, par construction. On déduit ainsi les dépendances can de x dans $A \cup B$ des dépendances respectives de x dans A et dans B. On peut soit recalculer ces dépendances au moment de la fusion, soit les mémoriser pendant le calcul des CanDate et MustDate initiales.

Algorithme 4 fusion_dates

Entrées: can_A, $must_A$ {les tableaux des dépendences can et $must$ des variables de A}
Entrées: can_B, $must_B$ {les tableaux des dépendences can et $must$ des variables de B}
Entrées: $CanDate_A$ {le tableau des CanDate des variables de A}
Entrées: $MustDate_A$ {le tableau des MustDate des variables de A}
Entrées: $CanDate_B$ {le tableaux des CanDate des variables de B}
Entrées: $MustDate_B$ {le tableau des MustDate des variables de B}
  pour $v \in V_A \cup V_B$ **faire**
    $can_{A \cup B}[v] = can_A[v] \cup can_B[v]$
    $must_{A \cup B}[v] = can_A[v] \cup can_B[v]$
  fin pour
  pour $v \in V_A \cap V_B$ **faire**
    propagation_CanDate$(can_{A \cup B}, v, max(CanDate_A[v], CanDate_B[v]))$
    propagation_MustDate$(must_{A \cup B}, v, max(MustDate_A[v], MustDate_B[v]))$
  fin pour

	$Equations1$	$Equations2$	$can_{\mathcal{E}}$	$must_{\mathcal{E}}$	**d**	**y**
a	$(1,2)$	$-$	$\{c,e,d\}$	$\{x,y\}$	$(1,\mathbf{3})$	$(\mathbf{2},3)$
b	$(1,0)$	$-$	$\emptyset$	$\{x,y\}$	$(1,0)$	$(\mathbf{2},0)$
c	$(2,1)$	$-$	$\{d\}$	$\{a,t\}$	$(2,\mathbf{2})$	$(\mathbf{3},2)$
d	$(3,0)$	$(0,1)$	(z)	$\{a,c\}$	$(3,\mathbf{1})$	$(\mathbf{4},1)$
e	$(2,0)$	$-$	$\emptyset$	$\{a,t\}$	$(2,0)$	$(\mathbf{3},0)$
x	$(0,3)$	$-$	$\{a,b\}$	$\emptyset$	$(0,\mathbf{4})$	$(0,4)$
y	$(0,3)$	$(1,0)$	$\{a,b\}$	$\{m\}$	$(0,\mathbf{4})$	$(\mathbf{1},4)$
t	$(0,2)$	$-$	$\{c,e\}$	$\emptyset$	$(0,\mathbf{3})$	$(0,3)$
m	$-$	$(0,1)$	$\{y\}$	$\emptyset$	$(0,1)$	$(0,\mathbf{5})$
v	$-$	$(1,0)$	$\emptyset$	$\{w\}$	$(1,0)$	$(1,0)$
w	$-$	$(0,1)$	$\{v\}$	$\emptyset$	$(0,1)$	$(0,1)$
z	$-$	$(1,0)$	$\emptyset$	$\{d\}$	$(\mathbf{4},0)$	$(\mathbf{5},0)$

Tableau 6. *Calcul du système Equations1-2. A chaque variable, on associe la paire (CanDate, MustDate). La colonne $Equations1$ (resp. $Equations2$) représente les dates calculées pour les variables de ce système avec l'algorithme de tri. La colonne $can_{\mathcal{E}}$ (resp. $must_{\mathcal{E}}$) represente l'ensemble des dépendances can (resp. $must$) de la variable, obtenu comme union des ensembles can (resp. $must$) de cette variable dans $Equations1$ et $Equations2$. Les colonnes **d** et **y** représentent la propagations des $CanDate$ et $MustDate$ de d et y.*

dans le système *Equations1*. Nous appelons le système résultant de cette fusion *Equations1-2*. Le tableau 6 montre comment on propage les réajustements des variables d et y communes à *Equations1* et *Equations2*. Nous définissons $\mathcal{E} = Equations1 \cup Equations2$. Considérons la variable commune d, $CanDate_{Equations1}(d) = 3$ et $CanDate_{Equations2}(d) = 0$, on appelle donc **propagation_CanDate (**$can_{\mathcal{E}}$**,** d**,** 3**)**. On propage donc la date 3 dans les dépendances $can_{\mathcal{E}}[d]$. Ensuite nous considérons la *MustDate* de d. Le maximum des *MustDate* respectives de d dans les deux systèmes d'équations est 1, on propage donc cette valeur dans $must_{\mathcal{E}}[d]$. Ces propagations sont décrites dans la colonne **d**. Ensuite nous propageons les dates de y de la même manière. Le résultat du calcul des ordres partiels du système Equations1-2 est donné dans la dernière colonne. Pour comparer, la figure 3 montre les graphes Γ_{can} et Γ_{must} de $\mathcal{E}$ et sur ces derniers on voit que l'on obtiendrait le même résultat en appliquant directement **tri_equations** à $\mathcal{E}$.

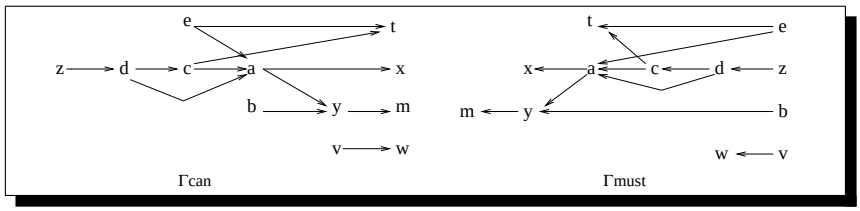

Figure 3. *Graphes* Γ_{can} *et* Γ_{must} *pour le système* $Equations1 \cup Equations2$

5. Finalisation

Notre approche de compilation des programmes LE en ξ systèmes d'équations nécessite toutefois d'être complétée par une phase de *finalisation* pour générer du code causal. Cette dernière est spécifique à notre approche car la compilation séparée implique de garder toutes les informations relatives aux statuts des signaux afin de mémoriser tous les ordres partiels valides et ne pas choisir un ordre total. Ainsi il existe des signaux dont le statut est resté à $\perp$ après propagation (constructive) des informations relatives aux statuts des signaux. Contrairement aux autres approches synchrones (Berry, 1996), nous ne décidons pas au moment de la compilation que les signaux dont le statut est $\perp$ sont absents. Cette affectation est appelée "réaction à l'absence". En effet cette opération rend effectivement la sémantique de comportement causale mais a le défaut irrévocable d'empêcher toute connexion instantanée avec d'autres modules synchrones susceptibles d'émettre justement ces signaux donc d'empêcher toute compilation modulaire. Une solution aurait été de rechanger le statut des signaux, d'oublier toutes les implications comportementales associées ; mais alors la sémantique n'aurait plus été constructive ...

Dans notre approche, nous déléguons donc la réaction à l'absence dans une phase spécifique que nous appelons *finalisation* et que nous appliquons séparément et après

la phase de compilation. Ainsi, la finalisation consiste à substituer tous les $\perp$ par 0 (absent) dans les systèmes d'équations quadri-valuées et à propager cette information au niveau du comportement du programme. Comme aucun signal n'est en `erreur` (la compilation aurait échouée précédemment), le système d'équations obtenu devient booléen et peut dès lors être réécrit dans du code cible ou servir de point d'entrée pour des outils de vérification symbolique.

6. Application

D'un point de vue applicatif, nous avons implémenté un compilateur pour le langage LE à partir de cette approche théorique, et nous avons aussi développé un ensemble de logiciels (la "Clem Toolkit") afin de spécifier, compiler, simuler et vérifier des applications réactives synchrones.

6.1. *Compilation*

Pour compiler les programmes LE nous implémentons les règles de la sémantique équationnelle. Ainsi nous associons un ξ système d'équations à chaque programme (en calculant le circuit associé à chaque noeud de l'arbre syntaxique du programme). Ensuite nous transformons ce système d'équations en un système d'équations booléennes grâce à une bijection qui encode les éléments de ξ par une paire de booléens : $\mathcal{B} : \xi \rightarrow \mathbb{B} \times \mathbb{B}$, avec $\mathcal{B}(\perp) = (0,0)$, $\mathcal{B}(0) = (1,0)$, $\mathcal{B}(1) = (1,1)$, $\mathcal{B}(\top) = (0,1)$. On étend cette fonction aux expressions de ξ ($x \sqcup y$, $x \sqcap y$ et $\neg x$). La définition complète est détaillée dans (Ressouche *et al.*, 2008). Cet encodage nous permet de transformer chaque ξ équation en deux équations booléennes. Finalement, le compilateur est l'implémentation de la loi $\hookrightarrow$ de propagation constructive (voir section 3.2) qui permet de calculer la valeur des sorties et des registres à chaque instant. Pratiquement, cet encodage permet de représenter les systèmes d'équations de façon symbolique avec des BDD (Binary Decision Diagram). L'implémentation du compilateur est de ce fait efficace, de plus, les opérations de constructions des graphes de dépendances *can* et *must* sont immédiates dans ce cadre.

Nous avons appelé CLEM (Compilation de LE Modules) le compilateur du langage LE. Pour réaliser une compilation séparée, nous avons défini un format interne (LEC) qui s'inspire très fortement du langage BLIF [7] (Berkeley Logic Interchange Format). Celui-ci est un format très compact qui permet de représenter des équations booléennes et nous lui avons ajouté une syntaxe pour représenter les *CanDate* et *MustDate* de chaque variable. En pratique, le compilateur CLEM, parmi tous ses codes de sorties génère ce format afin de pouvoir réutiliser efficacement des modules déjà compilés et appelés dans un *run* module, grâce à notre méthode pour trier les systèmes d'équations et fusionner des ordres partiels (voir section 4).

7. `http ://embedded.eecs.berkeley.edu/Research/vis`

6.2. *Processus de Finalisation*

Comme nous l'avons écrit dans le chapitre 5, ce processus consiste à projeter l'espace des signaux quadri-valués dans un espace de signaux binaires `absent`, `présent` en affirmant que tous les signaux $\perp$ sont finalement absents. Par notre codage (S_{stat}, S_{val}) des signaux quadri-valués S (voir section 6.1), cette finalisation revient très simplement à forcer à $vrai$ tous les S_{stat} dans les membres droits des équations et à "oublier" toutes les équations où S_{stat} est membre gauche !

Cette finalisation est donc irréversible. Elle n'intervient bien-sûr qu'en phase finale du processus de compilation au moment de la génération réelle des codes de sortie. La phase de finalisation est implémentée dans un outil spécifique (CLEF) dont le rôle est également de générer le code cible.

6.3. *La boîte à outils CLEM*

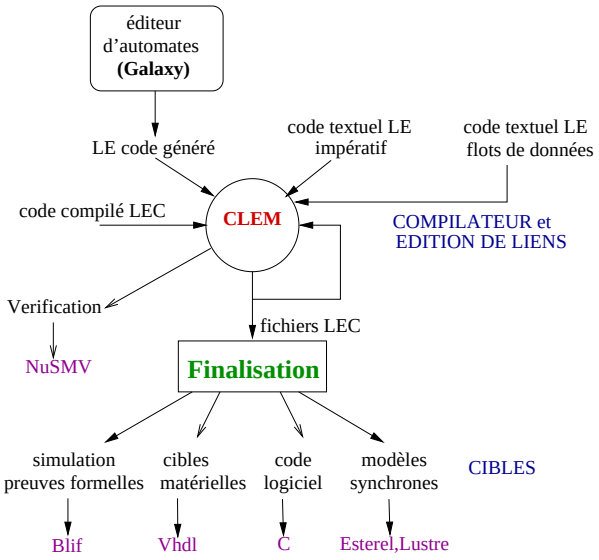

Figure 4. *Description de la boîte à outils CLEM*

Autour de CLEM, nous avons défini une boîte à outils pour spécifier, compiler, simuler et vérifier les programmes LE. Le langage textuel permet de spécifier et les automates peuvent être dessinés avec un éditeur graphique (*galaxy*). Chaque module est compilé dans le format LEC et peut inclure des références à d'autres modules déjà compilés en LEC. Quand la compilation est terminée, la *finalisation* simplifie le système d'équations et divers générateurs fournissent du code pour différentes cibles : simulation, dispositifs matériels ou logiciels. De plus, nous générons également du code SMV pour interfacer l'outil de "model-checking" NuSMV (Cimatti *et al.*, 2002)

et vérifier des propriétés en utilisant soit la technique du "model-checking" symbolique ou bien celle du "model-checking" borné qui utilise des méthodes de SAT-solver. Ces différentes fonctionnalités de notre boîte à outils [8] sont résumées dans la figure 4.

7. Exemple

Nous illustrons l'utilisation de notre boîte à outils sur un exemple industriel qui concerne la conception du contrôleur d'un système mécatronique : un préhenseur pneumatique. Nous présentons sa spécification en LE, sa compilation modulaire et sa vérification.

7.1. *Description du préhenseur pneumatique*

Le préhenseur pneumatique prend des pignons dans un tiroir et les monte sur un axe. Le système physique est essentiellement composé de deux vérins pneumatiques double effet et d'une ventouse. Cet exemple, volontairement simplifié, a été proposé par le groupe COSED [9] du club EEA [10], afin d'expérimenter de nouvelles techniques de conception et d'analyse des systèmes à évènements discrets. La cinématique du système ("cycle en U") est décrite dans la figure 5. Le mouvement horizontal se fait toujours en position haute.

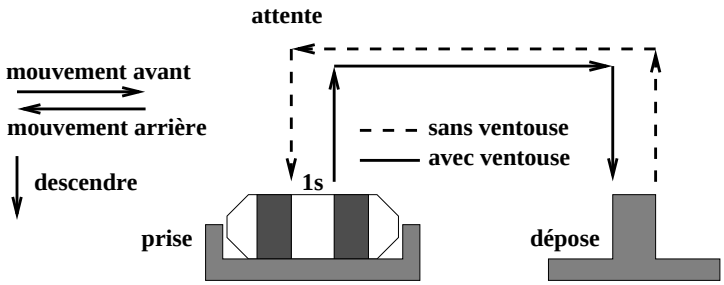

Figure 5. *Le préhenseur pneumatique*

Les commandes de mouvement horizontal sont `MoveFor` et `MoveBack`. La commande de mouvement vertical est `MoveDown`. Le bras remonte automatiquement quand il n'est pas activé (ce qui explique l'absence de commande `MoveUp`). Des capteurs fournissent la position du bras quand il arrive en butée : `backward`, `upward`, `downward` et `forward`. La ventouse est activée par la commande `SuckUp`.

8. CLEM est disponible à l'adresse suivante : "http ://www.inria.fr/sophia/pulsar/Projects/Clem"
9. `http ://www.lurpa.ens-cachan.fr/cosed`
10. `http ://www.clubeea.org`

```
module Control:

Input:  init, forward, backward, upward, downward, StartCycle;
Output: MoveFor, MoveBack, MoveDown, SuckUp, EndCycle, BeltOn;

Run: "./mecatronics/" : Temporisation;
    ".mecatronics/" : NormalCycle;
    "./mecatronics/" : Sustain;
local start_tempo, end_tempo {

    { wait init >> wait upward
      >> run Sustain[backward\terminate, MoveBack\action]
      >> run NormalCycle
    }
  ||
    run Temporisation
}
end

module NormalCycle :

Input: StartCycle, downward, upward,
       backward, end_tempo, forward;
Output: start_tempo, MoveDown,MoveBack, EndCycle,
        MoveFor, SuckUp, BeltOn;
Run:  "./mecatronics/" : Sustain;
wait StartCycle >>
{
  loop {
    run Sustain[downward\terminate, MoveDown\action]
    >>  { emit start_tempo || run Transport }
    >> wait upward
    >> {run Sustain [backward\terminate, MoveBack\action]
        ||
        run Sustain [backward\terminate, BeltOn\action]}
    >> pause >> emitEndCycle
  }
}
end
```

```
module Temporisation :
Input:  start_tempo;
Output: end_tempo;
loop {
   wait start_tempo
   >> pause >> pause
   >>  emit end_tempo
}
end
```

```
module Transport :
Input: end_tempo, upward, forward,
       downward;
Output: MoveDown,MoveFor, SuckUp;
Run:  "./mecatronics/" : Sustain;

local exitTransport {
     { run Sustain[end_tempo\terminate,
                MoveDown\action]
       >> wait upward
       >> run Sustain [forward\terminate,
                  MoveFor\action]
       >>  run Sustain[downward\terminate,
                  MoveDown\action]
       >>  pause >>  emit exitTransport
     }
    ||
    run Sustain[exitTransport\terminate,
              SuckUp\action]
}
end
```

Figure 6. *Code* LE *du système mécatronique*

7.2. *Implémentation du préhenseur pneumatique*

Nous avons programmé le contrôleur du préhenseur en LE. Son interface est composée des capteurs de butée : `backward`, `upward`, `forward` et `downward` et d'un signal de démarrage `init`. Il émet les commandes pour bouger les vérins : `MoveBack`, `MoveDown` et `MoveFor`, une commande de succion pour la ventouse : `SuckUp` ainsi qu'une commande qui démarre le tapis : `BeltOn`. Pour faire cette implémentation, nous avons suivi la philosophie synchrone et défini le contrôleur comme le parallèle de deux modules : (i) un module d'initialisation qui précède le lancement du cycle normal (*NormalCycle*) ; (ii) un module (*Temporisation*) qui gère le temps et les attentes. Les signaux locaux `start_tempo` et `end_tempo` servent aux deux branches du parallèle à communiquer. Le module *Sustain* émet le signal `action` tant que le signal `terminate` n'est pas là. Ce module est souvent utilisé dans l'exemple, il sert à maintenir une commande tant que le capteur de fin de butée n'est pas présent.

Le second niveau de l'implémentation spécifie les modules, *Temporisation* et *NormalCycle*. Ces modules sont définis dans des fichiers externes. La temporisation attend un signal `start_tempo`, attend deux ticks et émet le signal `end_tempo`. Le cycle normal est une boucle qui attend le signal `StartCycle` pour démarrer et dont le corps est l'implémentation d'un cycle aller-retour des vérins et il comporte un troisième niveau de spécification (module *Transport*) qui correspond au transport du pignon quand la ventouse est active. Le code est détaillé dans la figure 6.

Pour compiler cette application, nous avons compilé séparément les modules *Sustain*, *Temporisation* et *NormalCycle* et leur code a été sauvegardé dans des fichiers au format `lec`. Ensuite nous avons compilé le module *Control* en fusionnant les compilations de *Temporisation* et de *NormalCycle* dans celle de *Control* conformément à la technique décrite section 4.

7.3. *Vérification du préhenseur pneumatique*

Pour vérifier le comportement du contrôleur ainsi implémenté, nous l'avons dans un premier temps simulé avec le simulateur fourni dans la boîte à outils. Ce dernier est un outil graphique qui interprète le format `blif` généré à partir du code compilé et finalisé. Ensuite nous avons utilisé la fonctionnalité de "model-checking" de la boîte à outils pour prouver formellement des propriétés de bon comportement.

7.3.1. *Vérification formelle dans Clem*

Comme nous l'avons mentionné dans l'introduction, une conséquence intéressante de notre approche est la possibilité de faire des preuves exhaustives du comportement des programmes LE. Dans cette technique, on vérifie le comportement exigé en prouvant qu'une structure représentant tous les comportements du programme satisfait la propriété représentant le comportement exigé (ou ne satisfait jamais une propriété représentant un comportement prohibé). La sémantique comportementale construit l'ensemble des comportements d'un programme LE sous la forme d'un système de transitions étiquetées. Beaucoup d'outils de "model-checking", et particulièrement NuSMV, utilisent un tel modèle pour faire leurs preuves.

Dans les outils de "model-checking", on peut essentiellement vérifier des propriétés de vivacité et de sûreté de fonctionnement. Les propriétés que l'on peut prouver sur un programme s'expriment comme des formules de logique temporelle (Clarke *et al.*, 1994). Toutefois, nous préférerons nous appuyer sur la technique dîte "d'observateur" (Halbwachs *et al.*, 1993) qui a le mérite d'être plus homogène avec la représentation des machines d'états. Dans cette technique, pour prouver une propriété sur un programme P, on définit un autre programme Ω qui observe de façon non intrusive P et émet un signal d'alarme quand la propriété est violée. Dans la pratique, Ω est un programme qui a pour signaux d'entrée les signaux d'entrée et de sortie de P. Pour prouver la propriété, il suffit de prouver à l'aide d'un outil de vérification que le programme $P \parallel \Omega$ n'émet jamais le signal d'alarme.

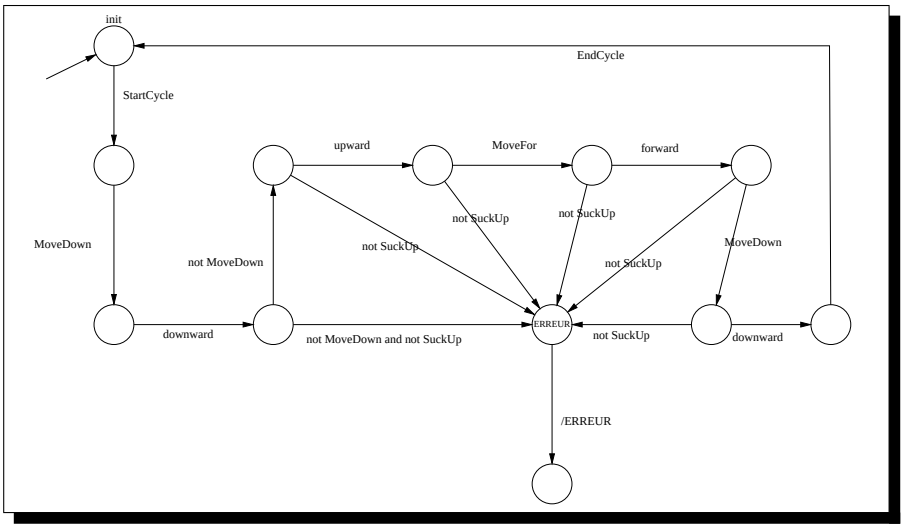

Figure 7. *Obervateur du préhenseur pneumatique*

7.3.2. *Application au préhenseur*

La boîte à outils `Clem` offre la possibilité de générer du code `smv` afin d'entrer dans NuSMV. Pour prouver le bon fonctionnement du préhenseur, nous avons généré un fichier d'entrée `Control.smv` et nous avons utilisé NuSMV pour prouver (1) que le contrôleur est vivant et (2) que deux ordres contradictoires ne peuvent pas être émis simultanément. Ainsi, nous avons prouvé que le signal `EndCycle` pouvait être émis (EF(`EndCycle`)) et que les signaux `MoveFor` et `MoveDown` sont exclusifs (AG !(`MoveFor` and `MoveDown`)).

Ensuite, nous avons prouvé que la ventouse fonctionne correctement. Pour cela nous avons utilisé un observateur qui regarde toutes les séquences d'entrée qui mènent à un état où `SuckUp` doit être émis et si ce n'est pas le cas un signal `ERREUR` est émis. Cet observateur est naturellement un automate et nous avons donc utilisé *galaxy* pour l'implémenter. Il est décrit dans la figure 7. Ensuite nous avons généré le code `smv` du programme qui réalise la composition parallèle du module `Control` et du module de l'observateur et nous avons prouvé avec NuSMV que `ERREUR` n'est jamais émis (AG(!`ERREUR`)) dans les comportements de ce programme.

8. Travaux Apparentés

Le problème de la modularité dans les langages synchrones a été beaucoup étudié. Une première étude (Maraninchi, 1992) sur ce sujet concerne le langage graphique synchrone *Argos*. Ce dernier est un langage graphique voisin des StateChart avec une sémantique synchrone compositionnelle mais il n'a pas de compilation ef-

fective modulaire. Par ailleurs, un certain nombre de travaux concernent la compilation séparée des langages synchrones. Dès 1988, P Raymond a étudié la compilation séparée du langage Lustre (Raymond, 1988). Sa préoccupation est orthogonale à la nôtre car il détermine des ensembles indépendants de variables dépendantes pour les évaluer séparément. En revanche, il caractérise les ensembles de variables en considérant des dépendances au plus tôt et au plus tard d'une façon similaire à nôtre approche (et à la méthode Pert). Beaucoup plus tard, S Edwards and E Lee (Edwards *et al.*, 2003) ont introduit un langage de blocs de diagrammes pour assembler différents types de logiciels synchrones. Ils définissent une sémantique de points fixes similaire à la sémantique constructive d'Esterel qui compile les programmes en systèmes d'équations booléennes. Ils restent déterministes et peuvent considérer des blocs de diagrammes avec rebouclage instantané. Ils s'intéressent principalement au problème de trouver un ordre d'évaluation et proposent un algorithme exact et des heuristiques pour résoudre ce problème. Leur langage est un langage d'assemblage sans hiérarchie tandis que (Lublinerman *et al.*, 2008) considèrent des blocs de diagrammes hiérarchiques. Le sujet de leur travail est de générer du code pour des "macros" (blocs de diagrammes composés de sous blocs connectés entre eux) ayant des fonctions d'interface qui permettent la connexion avec un contexte externe. Dans une étude plus récente (Lublinerman *et al.*, 2009), ces auteurs cherchent à réduire le nombre de fonctions d'interface créées. Pour résoudre ce problème ils montrent que la solution optimale correspond à la satisfaction d'une formule de logique propositionnelle qu'ils résolvent à l'aide d'un SAT-solver. Dans la même voie, Signal construit des graphes de dépendances conditionnelles qui permet la compilation séparée. Pour faire cette compilation séparée de façon consistante, (Nowak *et al.*, 1997) définissent un système de type qui permet d'associer une signature à chaque processus Signal. Ils s'appuient sur un système d'inférence de types "à la ML" pour prouver que le processus est bien utilisé conformément à sa signature. Toutes ces approches sont pertinentes mais diffèrent de notre problématique qui est d'assembler des composants non homogènes déjà compilés et de les recharger dynamiquement.

Récemment, (Biernacki *et al.*, 2008) ont étudié et formalisé la compilation modulaire des langages synchrones flots de données. Leur but est de prouver avec un démonstrateur de théorème la compilation modulaire du langage Lustre à des fins de certification. Par ailleurs, un certain nombre de travaux sur la modularité des langages synchrones concernent la distribution du code généré. Les approches plus anciennes pour générer du code efficace proposées dans (Weil *et al.*, 2000; Edwards, 1999) permettaient aussi une compilation distribuée. Dans (Zeng1 *et al.*, 2005), les auteurs considèrent une évaluation partielle des programmes Esterel : ils génèrent du code distribué qui évalue autant de sorties possiblement calculables quand certaines entrées restent inconnues. Dans (Schneider *et al.*, 2006), un langage synchrone *Quartz* est défini ainsi que sa compilation séparée. Cette dernière est réalisée en générant des processus séquentiels correspondant aux points de contrôle du programme compilé.

9. Conclusion

Les travaux décrits dans cet article font suite à notre participation à la conception du langage synchrone *Esterel*. Ce dernier offre des solutions efficaces pour concevoir des systèmes critiques validés. Toutefois, son manque de modularité au niveau de la compilation empêche parfois de faire face aux problèmes de taille du code compilé et prévient aussi l'utilisation de librairies compilées et validées dans la conception d'applications. Compiler un langage synchrone impératif comme Esterel est complexe car on doit vérifier qu'un programme est causal et faire face au problème de *schyzophénie*. En effet, cette dernière résulte d'emboitements de déclarations de signaux locaux dans l'opérateur de boucle (*loop*) qui se traduisent par plusieurs exécutions d'un même code dans le même instant et différents statuts pour un même signal. Ceci est bien sûr contraire à l'hypothèse synchrone. La solution adoptée par Esterel est de séparer le corps des boucles en deux parties : la *surface* est la partie du corps de la boucle exécutée au premier instant de son activation ; la *profondeur* est la partie activée aux autres instants. Ce qui permet de distinguer deux instances des signaux schizophrènes. De plus, Esterel possède comme opérateur noyau un opérateur d'exception très puissant (*trap exit*) qui permet de sortir d'un nombre quelconque d'imbrications d'opérateurs et qui peut imposer des duplications de la surface des boucles pour traiter la réincarnation multiple des signaux. En conséquence, le traitement de la causalité et de la schizophénie rend la modularité difficile dans la compilation d'Esterel.

Dans notre approche, nous avons privilégié la modularité. Nous nous appuyons sur la non résolution du statut $\perp$ des signaux à chaque instant (déléguée à la finalisation, voir section 5) pour assurer cette modularité. Par ailleurs nous définissons l' opération $\mathcal{P}re$ de "décalage" de l'environnement (voir section 3.1.2) qui duplique les signaux locaux lors de la compilation des opérateurs qui ne réagissent pas dans l'instant. Les boucles n'étant pas instantanées, on sépare ainsi surface et profondeur. De plus, nous ne considérons pas l'opérateur trap-exit mais nous le remplaçons par un opérateur *abort* [11] pour éviter de dupliquer un certain nombre de fois les surfaces des boucles ; ce nombre est certes borné mais dépend du contexte englobant. Toutefois, nous n'évitons pas les problèmes de schyzophrénie qui ne se résolvent pas par simple distinction de surface et profondeur des boucles. Dans ce cas, nous avons choisi de ne pas dupliquer le code mais de provoquer un cycle de causalité. Cette solution évite des réplications systématiques et parfois inutiles mais onéreuses (on peut passer d'un code compilé linéaire dans la taille du programme à un code quadratique) et de plus l'utilisateur prend conscience d'un problème peut être ignoré (et doit le résoudre). D'autre part, notre expérience relative à Esterel a permis de mettre en évidence le manque de formalisme pour décrire des automates explicites ou implicites sous forme d' équations booléennes. Ces différents points nous ont amenés à considérer un langage avec des opérateurs noyau différents de ceux d'Esterel et justifient la définition de LE. De plus, une autre syntaxe nous permet dêtre plus en adéquation avec les langages de programmation généraux usuels et familiers.

---

11. Ce qui nous parait réaliste en vue de l'utilisation de l'opérateur *abort* d'Esterel par rapport au *trap-exit*.

Pour permettre la modularité, dans LE, nous avons introduit une technique de compilation séparée des programmes. Nous avons défini une sémantique équationnelle sur laquelle repose la compilation modulaire du langage. Nous avons également défini une nouvelle méthode pour vérifier la causalité qui respecte cette compilation modulaire. C'est en nous appuyant sur cette démarche formelle que nous avons implémenté une boîte à outils qui permet de spécifier, implémenter, simuler et vérifier des applications synchrones de différents types. Ce travail a deux buts essentiels : tout d'abord offrir une solution à l'implémentation d'applications conséquentes avec des outils issus des méthodes formelles, et à l'élaboration de sous-systèmes "clé en main" dont les qualités sont connues et prouvées. Par ailleurs notre approche n'est pas orthogonale mais complémentaire aux nombreuses techniques étudiées pour générer du code séquentiel efficace. Ces dernières peuvent être appliquées sur le code finalisé. La seconde motivation est d'avoir à disposition des outils pour l'enseignement de l'approche synchrone qui permettent de comprendre, en expérimentant, la théorie sous jacente. Notre compilateur est l'implémentation directe de la sémantique équationnelle constructive, ainsi il permet d'étudier pratiquement ses règles.

Dans le futur, nous projetons d'améliorer nos travaux dans deux directions : la première est l'extension du langage aux données. Nous prévoyons d'étendre la syntaxe d'une part, et d'utiliser des techniques d'Interprétation Abstraite (Cousot *et al.*, 2002) pour tenir compte des contraintes dues aux données dans le contrôle quand cela est nécessaire et ainsi rester dans le cadre où nos résultats s'appliquent. De plus, nous voulons intégrer notre travail dans le monde du traitement du signal et de l'automatique en générant des représentations des programmes LE en Matlab/Simulink, par exemple.

La seconde amélioration que nous voulons mener concerne la vérification. Notre approche synchrone nous ouvre la voie pour appliquer les techniques de "model-checking". Nous désirons tout d'abord intégrer la définition d'observateurs dans la boîte à outils et directement appeler NuSMV sans intervention des utilisateurs. Ensuite, nous pensons montrer que les techniques de model-checking modulaire peuvent s'appliquer dans notre formalisme.

10. Bibliographie

Benveniste A., Guernic P. L., Jacquemot C., « Synchronous programming with events and relations : the SIGNAL language and its semantics », *Science of computer programming*, vol. 16, n° 2, p. 103-149, 1991.

Berry G., *The Constructive Semantics of Pure Esterel*, Draft Book, available at : http ://www.esterel-technologies.com, 1996.

Berry G., « The Foundations of Esterel », *in* G. Plotkin, C. Stearling, M. Tofte (eds), *Proof, Language, and Interaction, Essays in Honor of Robin Milner*, MIT Press, 2000.

Biernacki D., Colaço J., Hamon G., Pouzet M., « Clock-directed modular code generation for synchronous data-flow languages », *LCTES '08 : Proceedings of the 2008 ACM SIGPLAN-*

SIGBED conference on Languages, compilers, and tools for embedded systems, ACM, New York, NY, USA, p. 121-130, 2008.

Cimatti A., Clarke E., Giunchiglia E., Giunchiglia F., Pistore M., Roveri M., Sebastiani R., Tacchella A., « NuSMV 2 : an OpenSource Tool for Symbolic Model Checking », *in* E. Brinksma, K. G. Larsen (eds), *Proceeeding CAV*, n° 2404 in *LNCS*, Springer-Verlag, Copenhagen, Danmark, p. 359-364, July, 2002.

Clarke E., Grumber O., Long D., « Verification tools for finite-state concurrent systems », *LNCS : A Decade of Concurrency, Proc REX School/Symp*, vol. 803, p. 124-175, 1994.

Cousot P., Cousot R., « On Abstraction in Software Verification », *in* E. Brinksma, K. G. Larsen (eds), *Proceeeding CAV*, n° 2404 in *LNCS*, Springer-Verlag, Copenhagen, Danmark, p. 37,56, July, 2002.

Edwards S., « Compiling Esterel into Sequential Code », *Proceedings of the 7th International Workshop on Hardware/Software Codesign (CODES 99)*, Rome, Italy, p. 147-151, May, 1999.

Edwards S. A., Lee E. A., « The Semantics and Execution of a Synchronous Block-Diagram Language », *Science of Computer Programming*, vol. 48, n° 1, p. 21-42, July, 2003.

Gaffé D., Ressouche A., « The Clem Toolkit », *Proceedings of 23rd IEEE/ACM International Conference on Automated Software Engineering(ASE 2008)*, L'aquila, Italy, September, 2008.

Halbwachs N., *Synchronous Programming of Reactive Systems*, Kluwer Academic, 1993.

Halbwachs N., Lagnier F., Raymond P., « Synchronous observers and the verification of reactive systems », *in* M. Nivat, C. Rattray, T. Rus, G. Scollo (eds), *Third Int. Conf. on Algebraic Methodology and Software Technology, AMAST'93*, Workshops in Computing, Springer Verlag, Twente, June, 1993.

Huizing C., Gerth R., « Semantics of Reactive Systems in Abstract Time », *Real Time : Theory in Practice, Proc of REX workshop*, W.P. de Roever and G. Rozenberg Eds,LNCS, p. 291-314, June, 1991.

Jr. E. M. C., Grumberg O., Peled D., *Model Checking*, MIT Press, 2000.

Lublinerman R., Szegedy C., Tripakis S., « Modular Code Generation from Synchronous Block Diagrams : Modularity vs. Reusability », *ACM Symposium on Principles of Programming Languages(POPL'09)*, Savannah, GA, USA, 2009.

Lublinerman R., Tripakis S., « Modularity vs. Reusability : Code Generation from Synchronous Block Diagrams », *Design, Automation and Test in Europe (DATE08)*, 2008.

Maraninchi F., « Operational and Compositional Semantics of Synchronous Automaton Compositions », *CONCUR*, Springer Verlag, LNCS 630, August, 1992.

Mealy G. H., « A method for synthesizing sequential circuits », *Bell Sys. Tech. Journal*, vol. 34, p. 1045-1080, September, 1955.

Moore E. F., « Gedanken-experiments on sequential machines », *Automata Studies*, Princeton University Press, Princeton N.J. USA, 1956.

N. Halbwachs F. L., Ratel C., « Programming and verifying critical systems by means of the synchronous data-flow programming language Lustre », *Special Issue on the Specification and Analysis of Real-Time Systems*, IEEE Transactions on Software Engineering, 1992.

Nowak D., Talpin J., Gautier T., Guernic P. L., « An ML-like module system for the synchronous language Signa », *European Conference on Parallel Processing (Euro-Par'97)*, vol. LNCS 1300, Springer-Verlag, p. 1244-1252, August, 1997.

Potop-Butucaru D., Simone R. D., *Formal Methods and Models for System Design*, Gupta, P. LeGuernic, S. Shukla, and J.-P. Talpin, Eds ,Kluwer, chapter Optimizations for Faster Execution of Esterel Programs, 2004.

Raymond P., *« Compilation Séparée de Programmes Lustre »*, Master's thesis, IMAG, 1988.

Ressouche A., Gaffé D., Roy V., Modular Compilation of a Synchronous Language, Research Report n° 6424, INRIA, 01, 2008. https ://hal.inria.fr/inria-00213472.

Schneider K., Brand J., Vecchié E., « Modular Compilation of Synchronous Programs », *From Model-Driven Design to Resource Management for Distributed Embedded Systems*, vol. 225 of *IFIP International Federation for Information Processing*, Springer Boston, p. 75-84, January, 2006.

Tarski A., « A lattice-theoretical fixpoint theorem and its applications », *Pacific Journal of Mathematics*, vol. 5, n° 2, p. 285-309, 1955.

T.I.Kirkpatrick, Clark N., « PERT as and an Aid to Logic Design », *IBM Journal of Research and Develepment*, vol. 10, p. 135-141, March, 1966.

Weil D., Bertin V., Closse E., Poize M., Venier P., Pulou J., « Efficient compilation of ESTEREL for real-time embedded systems », *Proceedings of the 2000 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*,, San Jose, California, United States, p. 2-8, November, 2000.

Zeng1 J., Edwards S. A., « Separate Compilation for Synchronous Modules », *Embedded Software and Systems*, vol. 3820 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, p. 129-140, November, 2005.

Daniel Gaffé *est maître de conférences en Génie informatique, automatique et traitement du signal à la faculté des sciences de l'Université de Nice Sophia-Antipolis. Il est membre de l'équipe* MCSOC *(Modélisation Conception Systèmes d'Objets Communiquants) du laboratoire d'électronique LEAT. Ses activités de recherche portent essentiellement sur les langages synchrones et la vérification formelle de systèmes de contrôle. Il est également intéressé par la génération de code cible efficace, logiciel ou matériel à partir des modèles synchrones.*

Annie Ressouche *est chargée de recherche à l'Inria Sophia-Antipolis Méditerranée. Ses travaux de recherche portent sur les langages synchrones et de la vérification formelle . Elle a participé à l'étude et à l'implémentation d'outils de vérification par model-checking . Elle a étudié la sémantique des langages synchrones et participé activement au développement du langage Esterel. Actuellement, elle utilise le modèle synchrone présenté dans cet article pour valider la composition de composants synchrones dans les middlewares réactifs et adaptatifs et aussi pour concevoir des moteurs de reconnaissance d'activités.*

SERVICE ÉDITORIAL – HERMES-LAVOISIER
14 rue de Provigny, F-94236 Cachan cedex
Tél. : 01-47-40-67-67
E-mail : revues@lavoisier.fr
Serveur web : http://www.revuesonline.com

ANNEXE POUR LE SERVICE FABRICATION
A FOURNIR PAR LES AUTEURS AVEC UN EXEMPLAIRE PAPIER
DE LEUR ARTICLE ET LE COPYRIGHT SIGNE PAR COURRIER
LE FICHIER PDF CORRESPONDANT SERA ENVOYE PAR E-MAIL

1. ARTICLE POUR LA REVUE :
   RSTO - TSI

2. AUTEURS :
   Daniel Gaffé[*] *— Annie Ressouche*[**]

3. TITRE DE L'ARTICLE :
   Compilation modulaire d'un langage synchrone

4. TITRE ABRÉGÉ POUR LE HAUT DE PAGE MOINS DE 40 SIGNES :
   CLEM

5. DATE DE CETTE VERSION :
   24 mars 2010

6. COORDONNÉES DES AUTEURS :

– adresse postale :

[*] Laboratoire LEAT, Université de Nice Sophia-Antipolis, CNRS
250 rue Albert Einstein, 06560 Valbonne France

Daniel.Gaffe@unice.fr

[**] INRIA Sophia Antipolis Méditerranée
2004 route des Lucioles – BP 93, 06902 Sophia Antipolis France

Annie.Ressouche@sophia.inria.fr

– téléphone : 04 92 38 79 44

– télécopie : 04 92 38 79 39

– e-mail : Annie.Ressouche@sophia.inria.fr

7. LOGICIEL UTILISÉ POUR LA PRÉPARATION DE CET ARTICLE :
   LaTeX, avec le fichier de style `article-hermes2.cls`,
   version 1.23 du 17/11/2005.

8. FORMULAIRE DE COPYRIGHT :
   Retourner le formulaire de copyright signé par les auteurs, téléchargé sur :
   `http://www.revuesonline.com`

SERVICE ÉDITORIAL – HERMES-LAVOISIER
14 rue de Provigny, F-94236 Cachan cedex
Tél. : 01-47-40-67-67
E-mail : revues@lavoisier.fr
Serveur web : http://www.revuesonline.com