Programmation Synchrone et Validation Formelle des Sytèmes Réactifs

-Utilisation de la Technique Formelle LUSTRE-

Alexandre CORTIER
ONERA (*Office National d'Etudes et de Recheche Aéronautique*)
Centre de Toulouse
BP 4025 - 2 Avenue Edouard Belin - 31055 TOULOUSE CEDEX 4
Mél : *Alexandre.Cortier@onecert.fr*
Mél : *Alexandre.Cortier@laposte.net*

09/11/2006

Table des matières

Introduction

L'objectif de ce cours est double : (1) d'une part introduire les concepts et notions liés aux *Techniques Formelles*, (2) d'autre part montrer comment ces techniques peuvent être utilisées dans le cadre de la spécification de *systèmes réactifs* par l'étude du *langage synchrone LUSTRE*.

Une **Technique Formelle** est l'association (1) d'un *langage formel*, c'est-à-dire un langage défini mathématiquement par le biais d'une *syntaxe et d'une sémantique formelle*, (2) et d'un *système de preuve*. Le langage permet l'écriture de programme qui constitue un *modèle formel* du système que nous souhaitons spécifier. Une fois ce modèle obtenu, il est alors possible de prouver que le modèle satisfait les propriétés exigées par le cahier des charges. Pour ce faire, une étape de formalisation des propriétés à vérifier est nécessaire. Suivant la technique adoptée (*Theorem Proving* ou *Model Checking*) ces propriétés sont formalisées en utilisant une logique adéquate comme par exemple *la logique du premier ordre* ou *une logique temporelle*. Des outils informatiques appelés *model-checker* ou *theorem-prover* permettent alors, en utilisant la spécification formelle du système considéré et les propriétés du cahier des charges formalisées, d'assister le concepteur pour vérifier que le modèle satisfait les propriétés requises. Contrairement à la démonstration par preuve ou theorem-proving, le model-checking permet une vérification automatique des propriétés : le travail de l'ingénieur se limite alors à la formalisation des propriétés.

Le chapitre 1 introduira les notions de Technique Formelle. Le chapitre 2 se concentrera sur la technique de preuve appelée Model-Checking. Le principe de vérification d'un programme LUSTRE étant le modèle Model-Checking nous ne présenterons pas la technique du Theorem Proving.

Le chapitre 3 présentera la notion de **systèmes réactifs**. Nous verrons à travers ce chapitre en quoi les *approches synchrones* sont particulièrement adaptée à la spécification des systèmes réactifs.

Enfin, le chapitre 4 présentera le Langage formel LUSTRE. LUSTRE est un langage à flot de données particulièrement adaptée à la spécification de systèmes réactifs, et tout particulièrement à la spécification de système de contrôle-commande. Ce langage a été construit suivant l'approche synchrone, facilitant de ce fait la vérification formelle. La technique du model-checking est exploitable pour vérifier qu'un modèle formel décrit en LUSTRE satisfait des propriétés dites de *sûreté*.

Chapitre 1

Introduction aux Techniques Formelles

Le développement de logiciels de qualité exige l'utilisation de techniques rigoureuses. Ces dernières doivent assurer que les logiciels développé satisfont les propriétés qui traduisent les exigences exprimées dans le cahier des charges. Durant ces vingt dernières années, tant l'utilisation accrue de logiciels que la mise envidence d'anomalies à propos du passage l'an 2000 ou bien le vol 501 d'Ariane ont imposé de faire évoluer les processus de production de logiciels de façon améliorer la fiabilité.

Dans ce but, la définition et l'utilisation de techniques rigoureuses de développement de logiciels apparaît comme une des composantes indispensables des étapes de production de logiciels. Ce type de techniques doit être mis en oeuvre pendant les différentes phases de développement : spécification, conception, vérification, validation, réutilisation, maintenance. Les processus correspondant à ces différentes phases doivent être maîrisés et contrôlés par les développeurs.

Les techniques doivent permettre de raisonner de manière rigoureuse sur le programme de façon à assurer que le produit satisfait certaines propriétés qui traduisent les besoins et les exigences exprimées dans le cahier des charges.

Etablir des propriétés sur les objets programmes nécessite l'utilisation de *Techniques Formelles*.

De nombreuses techniques, qualifiées de formelles car implantant des systèmes formels, ont vu le jour.

Afin d'appréhender la notion de Technique Formelle nous proposons dans cette partie un petit rappel concernant les langages de programmation et la notion de modèle (étroitement liée à celle d'abstraction). Nous définirons la notion de sémantique formelle qui est à la base de la définition d'une Technique Formelle. Nous présenterons ensuite les outils de preuves utilisées pour la vérification formelle.

1.1 Modèles et Langages de programmation

1.1.1 Langages de programmation : du modèle d'exécution au modèle de programmation

Un langage de programmation est une notation systématique pour lequel les processus de calculs sont définis.

Suivant [Teu91], nous pouvons classer les langages de programmation suivant trois g ations :

Première génération de langages. Les ordinateurs opèrent à un niveau binaire (séquence de 0 et 1). Ainsi, au début de la programmation, les codes binaires étaient utilisés pour programmer les ordinateurs. La programmation binaire signifie que le programme reflète directement la structure matériel de l'ordinateur. Les séquences de 0 et de 1 sont responsables des actions de calculs, de contrôle, d'indexation en mémoire... A ce stade, un programme correspond à un *"modèle d'exécution"*, c'est à dire une représentation abstraite directement liée à l'exécution de l'architecture matérielle de l'ordinateur. Cette programmation au plus proche de l'architecture matérielle pose de nombreux problèmes. Par exemple, l'insertion d'une nouvelle instruction - situation classique dans le développement d'un programme - provoque des incorrections au niveau des adresses dans les instructions déja existantes. Pour palier à cette difficulté, il était alors nécessaire d'obtenir

des représentations plus abstraites d'un programme pour pouvoir manipuler plus aisément les données et les opérations de calculs sur ces données.

La seconde génération de langages. Les langages de type assembleurs ont été la première réponse aux difficultés évoquées ci-dessus. Dans ces langages on note l'introduction d'un certain nombre d'abbrévations comme les noms symboliques et l'introduction du concept de commandes et d'opérations. Les langages assembleurs reflètent encore l'architecture matérielle de la machine cible mais à un niveau d'abstraction plus élevé : celui du registre. Nous pouvons encore parlé à ce stade de *"modèle d'exécution"*, bien qu'il s'agisse déja d'une abstraction, c'est à dire d'une représentation abstraite s'éloignant de l'architecture matérielle.

La troisième génération de langages. Les langages assembleurs ont été remplacé par une troisième génération : les "*langages de haut niveau*". Ces langages permettent l'utilisation de structures de contrôle basées sur des données logiques : les variables d'un type spécifique. Ces langages présentent un niveau d'abstraction qui permet la spécification de données, de fonctions ou de procédures, et la spécification de leur contrôle indépendamment de la machine hôte. Un programmeur peut alors se concentrer sur un problème à résoudre sans se soucier de la structure interne de la machine cible. On distingue 4 grands groupes de langages de programmation : les langages de *programmation impérative*, les langages de *programmation fonctionnelle*, les langages de *programmation logique* et enfin les langages de *programmation orientée objets*. A ce stade, un langage de haut-niveau constitue un *modèle de programmation* et le programme écrit dans ce langage un *modèle abstrait* de l'exécution du système. Le lien entre modèle d'exécution et modèle de programmation est effectué par l'étape de compilation du programme source. Cette opération de compilation peut être vue comme une *opération sémantique* : la compilation donne du sens aux opérations utilisées dans un langage de haut niveau afin d'obtenir un modèle d'exécution (un code binaire) exécutable sur la machine cible.

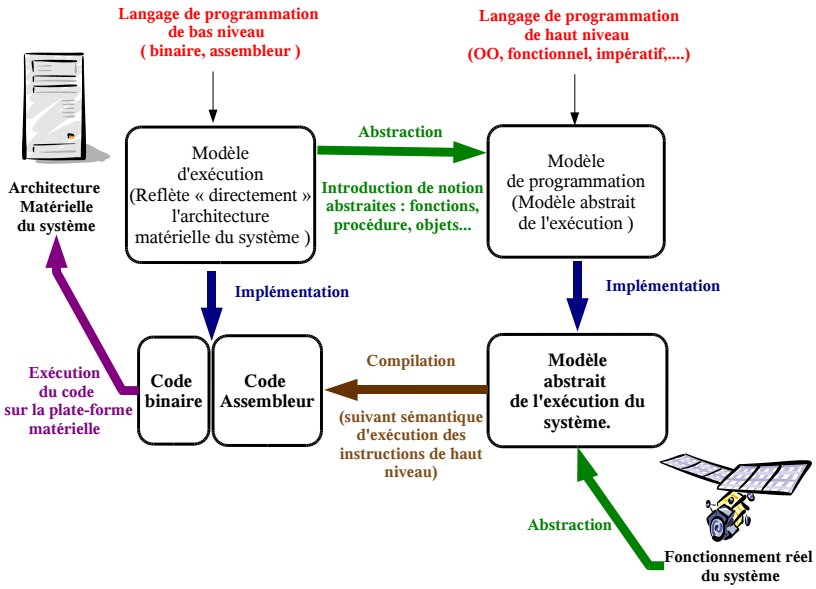


FIG. 1.1 – Notion de modèle et d'abstraction

1.1.2 Le concept d'abstraction et de modèle abstrait

Le plus important concept introduit par les langages de haut niveau est celui d'abstraction, processus permettant d'observer des patterns généraux tout en ignorant les détails non essentiels. Les abstractions possèdent une double relation avec les langages de programmation (cf. figure 1.1) :

– Les langages de programmation sont utilisés pour implémenter des modèles abstraits.

– Les langages de programmation sont des abstractions de l'architecture matérielle, i.e les modèles de programmation sont des abstractions des modèles d'exécution.

1.1.3 Pragmatique, Sémantique et Syntaxe

1.1.4 Définition

Pour les notions de syntaxe, de sémantique et de pragmatique, nous utilisons une définition de Rudolph Carnap [Pia86] :

"Si on se réfère à celui qui parle, ou en termes plus généraux aux usagers du langage, nous attribuons cette investigation à la pragmatique. Si nous faisons abstraction des usagers du langage et si nous analysons seulement les expressions et leur signification, nous nous trouvons dans le domaine de la sémantique. Et si finalement, nous faisons abstraction des significations pour analyser uniquement les relations entre expressions nous entrons dans le domaine de la syntaxe. La totalité de la science du langage se composant de ces trois parties, forme la sémiotique."

Dans la pratique, syntaxe et sémantique sont rattachées à des définitions formelles de type mathématique. Quant à la pragmatique, elle est rattachée à l'usage et aux usagers des techniques et aux jugement et recommandations que l'on peut formuler. Elle ne saurait être formalisée mathématiquement. C'est à ce niveau que des préconisations méthodologiques peuvent s'exprimer.

Ainsi la sémantique est à la base de la définition des techniques formelles. Ces techniques, supportés par des langages, possèdent des sémantiques différentes. Notons que les activités de spécification, de développement, de validation, de maintenance ect., lorsqu'elles sont formalisées, doivent posséder une sémantique formelle.

1.1.4.1 Sémantique(s) formelle(s) d'un programme

Un programme est muni de deux concepts : sa syntaxe (phrase : séquence de mots du langage de programmation) et sa sémantique (son sens, sa signification, ce qu'il fait ou l'action qu'il réalise).

La syntaxe est formellement définie à l'aide d'une grammaire. De nombreux travaux dans ce domaine ont vu le jour et le traitement syntaxique des langages de programmation est un problème assez bien maîtrisé.

La sémantique a fait l'objet de nombreuses études ayant abouti à différentes formalisations. Chacune des ces formalisations a un but précis et sert à réaliser un objectif donné dans la conception d'un programme. On peut penser que la sémantique est une fonction qui associe un sens à un programme :

$$sem : programme \longrightarrow sens$$

C'est la nature des définitions de cette fonction ainsi que de ses constituants qui aboutit à différentes sémantiques. Pour aller un peu plus loin dans notre définition, il est nécessaire de distinguer :

– les données et les structures de données manipulées par les programmes (entiers, flottants, chaînes de caractères, listes, tableaux...). Elles représentent le vecteur d'état d'un programme ou son contexte, son environnement d'exécution, son état mémoire, ect... La formalisation du contexte nécessite l'introduction d'outils formels permettant de représenter aussi bien des entiers que des listes ou des tableaux.
– les différentes opérations permettant de calculer les résultats d'un programme. Ainsi, d'un point de vue sémantique, un programme peut être vu comme une fonction qui à des données en entrée, associe un résultat en sortie.
  Cette vision n'est plus valable pour les programmes dans lesquels la terminaison n'est pas requise (programmes réactifs par exemple), dans ce cas un programme peut être vu comme un calcul représentant la suite des états d'un programme (automates).

Ces deux points montrent qu'en fonction des problèmes à traiter, les données à coder, des langages utilisés, différentes sémantiques peuvent être définies. Entre autres, nous trouvons dans la littérature les sémantiques : dénatotionelle, opérationnelle, fonctionnelle, algébrique, sémantique à états...

Example concret. La syntaxe d'un langage de programmation décrit la *forme* de ce langage. Cette syntaxe est définie par un certain nombre de règles. Ces règles syntaxiques décrivent la manière dont nous devons combiner les éléments basiques (les symboles terminaux) du langage de programmation pour générer des phrases correctes, i.e des phrases qui seront acceptées par la grammaire du langage.

La signification de telles phrases syntaxiquement correctes est définie par les *règles sémantiques*. Par exemple, prenons l'instruction k :=k+1 ; . D'un point de vue syntaxique, "k"," :=","+","1", et ";" sont uniquement des symboles qui correspondent à aucun objet mathématique. Une description sémantique relie "k :=k+1 ;" à l'action d'incrémenter le contenu d'une case mémoire.

1.2 Techniques Formelles

1.2.1 Modèles formels

Un modèle formel d'un système est un programme écrit via l'utilisation d'un langage formel. On parle de langage formel pour désigner un langage de haut-niveau dont les constructions syntaxiques possédent une sémantique formelle.

Suivant la sémantique adoptée, nous pouvons distinguer deux grandes classes de spécifications formelles [Ses02] :

- Les approches constructives (encore appelées opérationnelle ou basée sur état explicite). Ces approches se fondent sur une sémantique à état de type *"automate"*. Dans ce contexte, l'état est modifié par les opérations, opérations qui modélisent l'aspect comportemental du système. Parmis ces approches nous trouvons les *Systèmes de Transitions Etiquetés* (STE), *les réseaux de Petri* (RdP)...
- Les approches algébrique ou fonctionnelle. Suivant cette approche la sémantique est définie par une algèbre. Les variables et les opérations sont donc définies suivant cet algèbre et le système est décrit par un ensemble d'équations qui définissent son aspect comportemental. Parmis les langages formels suivant cette approche : *LOTOS*, *LUSTRE*...

1.2.2 Technique Formelle : Définition

On appelle Technique Formelle, l'association :

1. d'un **langage à sémantique formelle**. Ce langage permet la description abstraite des objets programmes à développer. Un programme écrit dans un langage formel constitue une spécification (ou modèle) formelle du système à concevoir.
2. un **système de preuve** permettant d'exprimer et de prouver des propriétés sur les modèles. Il existe actuellement deux principales techniques de preuve sur modèle : le *Theorem Proving* et le *Model Checking*.

1.2.3 Techniques de preuves sur modèles

Quelques échecs retentissants (comme par exemple la panne du réseau téléphonique aux USA en 1989 ou la déstruction du premier exemplaire de la fusée Ariane 5 en 1996) ont achevé de convaincre de l'impérieuse nécessité de vérifier certains logiciels ou systèmes de contrôle-commande.

Il existe plusieurs techniques pour effectuer de telles vérifications, les principales étant le test, la démonstration automatique (Theorem Proving) et la vérification sur modèle (Model Checking).

Le test est indispensable et permet de découvrir de nombreuses erreurs, mais il ne peut pas être exhaustif et n'apporte donc que des réponses partielles. La démonstration automatique est en principe capable de répondre à toutes les questions de vérification qui se posent en pratique, mais sa mise en oeuvre est souvent lourde et compliquée et les outils actuels sont seulement capables d'assister un ingénieur à qui revient la charge de construire la preuve.

Le Model Checking est en quelque sorte l'intermédiaire entre le test et la démonstration : il s'agit d'une méthode exhaustive et en grande partie automatique. Le travail de l'ingénieur se limite la construction d'un modèle formel du système et à la formalisation des propriétés à vérifier.

Ces méthodes ne sont pas concurrentes mais complémentaires et aucune d'entre elles ne peut prétendre apporter seule une réponse définitive aux problèmes de sureté de fonctionnement. Néanmoins, dans l' état actuel de leurs développements, ces différentes techniques permettent déja de traiter de vrais exemples industriels, et ceci à coût maîrisé. De plus, les erreurs détectées auraient souvent eu des conséquences financières, économiques ou humaines sans communes mesure avec les investissements effectuées.

1.2.4 Mise en garde

Une technique formelle est donc un outil informatique puissant permettant de spécifier le comportement d'un système et de prouver un certain nombre de propriétés sur ce système. Cependant, une mise en garde s'impose : une technique formelle ne prétend pas apporter une réponse définitive aux problèmes de vérification des systèmes. D'une part ces techniques ne s'appliquent bien qu'à un certain type de problèmes. Ensuite, la phase de vérification peut comporter des erreurs (par exemple un oubli). Enfin, il existera toujours dans un système réel des possibilités de pannes, d'erreurs, ect. qui ne sont pas prévues dans le *modèle formel du système* soumis à la vérification. Cela ne signifie pas que le modèle formel est "incorrect", il s'agit plutôt d'une des limites intrinsèques de la démarche de modélisation mathématique. On valide *le modèle du système* mais pas le système lui-même. Nous travaillons sur une abstraction du monde réel et par conséquent cette abstraction ne peut rendre compte de tous les aspects du monde réel.

Chapitre 2

Principes du Model-Checking

Généralement, les sytèmes qui se prêtent le mieux à la vérification par model-checking sont :

- les systèmes *critiques*, où une erreur peut avoir des conséquences catastrophiques. On trouve de nombreux exemples en régulation, dans l'informatique bancaire, les transports, le nucléaire, etc ;
- les systèmes *distribués*, dont le comportement global dépend de l'interaction de différents sous-systèmes évoluant en parallèle : réseaux de communications, bases de données réparties, automates couplés, en fait tous les systèmes de grande taille. Le comportement non séquentiel des systèmes distribués est particulièrement difficile à maîtriser pour un cerveau humain. La meilleure preuve en est qu'on parle toujours d'"erreurs subtiles" pour ces systèmes ;
- les systèmes *réactifs* qui réagissent en permanence à leur environnement et ne peuvent être vus de façon pertinente comme "faisant un calcul" au sens classique. Ils vont des systèmes d'exploitation aux protocoles de communication, en passant par les systèmes de contrôle-commande où les jeux videos, ect.

Pour vérifier automatiquement un système par la méthode du model-checking, il est nécessaire d'en construire une modélisation formelle, par exemple sous la forme d'un automate mais plus généralement comme un réseau de plusieurs automates synchronisés. Pour cela, on utilise comme nous l'avons déja évoqué, un *langage formel de spécification de systèmes*. Il faut ensuite énoncer formellement les propriétés à vérifier. On utilise un *langage de spécification de propriétés*, par exemple une logique temporelle. Enfin, il faut disposer d'un algorithme capable de dire si le système vérifie ou non les propriétés énoncées. Cet algorithme est incarné dans un *model-checker* : un outil informatique pour le model-checking.

La plupart des model-checkers sont capables de fournir un *diagnostic* d'erreur complétant utilement la vérification qu'une propriété n'est pas satisfaite. Par exemple, dans le cas d'une *propriété de sûreté* que le système examiné ne vérifierait pas, le model-checker proposera un exemple d'exécution du système violant cette propriété.

Nous présenterons dans cette section les concepts sous-jacents aux techniques de model-checking. Nous allons présenter successivement : (1) les systèmes d'automates qui sont à la base des modèles opérationnels utilisés pour spécifier le comportement des systèmes que l'on souhaite vérifier ; (2) la logique temporelle et son utilisation pour la spécification de propriétés ; (3) le model-checking basé sur l'énumération explicite. Nous ne parlerons pas du model-checking symbolique basé sur les arbres de décision binaires, ni des automates temporisés et les méthodes qui leur sont associées. De plus amples informations sur ces derniers points sont accessibles dans l'ouvrage [RHR91].

2.1 Automates

2.1.0.1 Modèle de Kripke

Un modèle de Kripke est un couple (S, R) ou S est un ensemble (fini ou infini) appelé ensemble des états et R une relation binaire sur S appelé relation de transition $R \subseteq S \times S$).

2.1.0.2 Automates

Définition 2.1.0.1 *Un automate est un quadruplet $\mathcal{A} = (Q, E, T, q_0)$ où :*
- *Q est un ensemble fini d'états,*
- *E est un ensemble fini d'étiquettes associées aux transitions,*
- *T avec $T \subseteq Q \times E \times Q$ est l'ensemble des transitions. Une transitions définit une relation antre deux états de Q et est étiquetée par un élément de E.*
- *q_0 est l'état initial de l'automate.*

Une transition t est donc un triplet $t = (p, e, q)$. Un automate est *déterministe* si pour tout état q et pour toute étiquette e il existe au plus une transition ayant pour origine p et étiquette e.

Pour la spécification de systèmes de processus, l'ensemble des étiquettes E associé aux transitions T comportera les actions qui peuvent être activées lors du passage d'un état à un autre. Les mots étiquette et actions seront utilisées indiférament dans la suite.

Notons que pour la description de systèmes de processus, on utilise l'appellation *"systèmes de transitions"* où même *"systèmes de transitions étiquetées"* (STE) pour désigner les automates. Ce dernier est un terme issu de la théorie des langages.

2.1.0.3 Automates et propriétés associées aux états

Définition 2.1.0.2 *Soit $P = \{P_1, P_2, \ldots\}$ un ensemble de propositions décrivant des propriétés élémentaires.*

Un automate peut être étendu à un quintuplé $\mathcal{A} = (Q, E, T, q_0, l)$ où :
- *Q est un ensemble fini d'états,*
- *E est un ensemble fini d'étiquettes associées aux transitions,*
- *T avec $T \subseteq Q \times E \times Q$ est l'ensemble des transitions,*
- *q_0 est l'état initial de l'automate,*
- *l est l'application qui associe à tout état de Q l'ensemble fini des propriétés élémentaires vérifiées dans cet état.*

L'application l permet d'obtenir les différentes propriétés vérifiées dans un état donné. Elle permet entre autre de décrire les variables d'états qui caractérisent un système de processus et de les observer lors d'un changement d'état.

2.1.0.4 Définitions complémentaires

Lors de la représentation d'un système de transitions, il est souvent nécessaire de manipuler des variables. En général, ces variables sont des *variables d'état*.

Variables d'états. Les variables d'états caractérisent une propriété de l'état du système représenté. Une variable d'état prend des valeurs dans un ensemble fini ou infini. Les techniques de vérification et de preuve dépendent de la nature de l'ensemble des valeurs des variables d'états.

Les liens entre automates et variables d'état peuvent être de deux types :
- *Affectations* : une transition peut modifier la valeur d'une (ou de plusieurs) variable(s).
- *Gardes* :une transition peut être gardée par une condition sur les variables d'états. Le franchissement de la transition n'est possible que si la condition est vérifiée.

Représentation graphique. Il est possible de représenter un automate en utilisant une représentation graphique. Les états sont représentés par des ronds. L'état initial est distingué par une flêche arrivant sur cet état sans origine. L'état terminal, s'il en existe un, est représenté par deux cercles concentriques. Les transitions sont des arcs orientés dans le sens état de départ vers état d'arrivée. L'arc désignant la transition est annoté par l'étiquette.

Dépliage. Le dépliage d'un automate consiste à produire un automate dans lequel toutes les transitions de l'automate sont présentes.Les états de l'automate déplié sont des états *globaux*. Le dépliage d'un automate est particulièrement utilisé dans le cas de la présence de gardes et de variables d'états. Cela permet de se rapporter à un automate classique.

2.1.0.5 Comportement

Définition 2.1.0.3
- *Un chemin dans un automate $\mathcal{A}$ est la suite σ, finie ou infinie, de transitions $(q_i, e_i, q_i^{'})$ de $\mathcal{A}$ qui s'enchaînent, c'est à dire $q_i^{'} = q_{i+1}$ pour tout i. Un chemin est souvent noté $p_1 \rightarrow_{e_1} p_2 \rightarrow_{e_2} p_3 \rightarrow_{e_3} p_4 \ldots$*
- *La longueur d'un chemin $|\sigma|$ est le nombre de transitions qu'il contient. Cette longueur peut être infinie et $|\sigma| \in N \cup \{w\}$.*

Dans la terminologie des systèmes de processus, un chemin est également désigné par le mot *trace*.

Définition 2.1.0.4
1. *Une exécution partielle est un chemin partant de l'état initial.*
2. *Uneexécution complète est une exécution (partielle) maximale, c'est-à-dire une exécution qui ne peut être prolongée.*

2.1.1 Produit cartésien : synchronisation d'automates

Lors de la description d'un système de processus, on procède souvent par la description des différents processus individuellement. Chaque processus peut être décrit par un STE. Le système global est lui décrit par l'ensemble des processus et donc par l'ensemble des STE. La description du système de transitions global associée au système de processus permet de définir complètement ce système de processus.

Les produit cartésien (produit libre) et produit synchronisé de STE permettent de décrire des systèmes de processus par assemblage (opération de produit) de STE de base.

Ces opérations de produit permettent d'obtenir un système de transitions étiquetées qui décrit globalement le système de processus. En général, le système de transitions obtenu possède un très grand nombre d'états, si bien qu'il est impossible de le construire. On parle ici *d'explosion du nombre d'états*.

2.1.1.1 Produit cartésien ou produit libre

Considérons une famille de n automates $\mathcal{A}_i = (Q_i, E_i, T_i, q_{0,i}, l_i)$, $i \in 1 \ldots n$. Soit $'-'$ une nouvelle étiquette permettant d'exprimer l'action fictive. Cette action fictive servirait à décrire qu'un des sous-automates n'effectue aucune transition dans l'automate global.

Définition 2.1.1.1 *Le produit cartésien $\mathcal{A}_1 \times \mathcal{A}_2 \times \ldots \times \mathcal{A}_n$ de ces automates, est l'automate $\mathcal{A} = (Q, E, T, q_0, l)$ tel que :*
- $Q = Q_1 \times Q_2 \times \ldots \times Q_n$
- $E = \prod_i = 1^n (E_i \cup \{-\})$
- $T = \{((q_1, q_2, \ldots, q_n)(e_1, e_2, \ldots, e_n)(q_1^{'}, q_2^{'}, \ldots, q_n^{'}) | \forall i \in 1..n, (e_i =' -' \text{ et } q_i = q_i^{'}) \text{ ou bien } (e_i \neq' -' \text{ et } (q_i, e_i, q_i^{'}) \in T_i)\}$
- $q_0 = (q_{0,1}, q_{0,2}, \ldots, q_{0,n})$
- $l = \bigcup_{i=1}^{n} (l_i(q_i))$

Dans un produit cartésien, chaque composante locale (ou automate) $\mathcal{A}_i$ peut, lors d'une transition, soit effectuer une transition locale, soit ne rien faire (action fictive). Il n'y a aucune obligation de synchronisation entre les différentes composantes. De plus, le produit cartésien permet des transitions où toutes les composantes ne font rien.

2.1.1.2 Produit synchronisé

Pour synchroniser les différentes composantes d'un produit cartésien d'automates, il est nécessaire de restreindre les transitions possibles dans l'automate résultant du produit cartésien. Seules les transitions correspondant à des transitions de synchronisations acceptées sont conservées.

Considérons un ensemble de synchronisations *Sync* (ou bien un vecteur de synchronisations) défini par :

Définition 2.1.1.2 $Sync \subseteq \prod_{i=1}^{n}(E_i \cup \{-\})$

Sync indique, parmi les étiquettes du produit cartésien, lesquelles correspondent réellement à des synchronisations (transitions groupées autorisées).

On peut également définir le vecteur de synchronisation en utilisant les états du produit cartésien avec $Sync \subset Q_1 \times Q_2 \times \ldots \times Q_n$.

Informellement, le produit synchronisé est un produit cartésien dans lequel seules les transitions prises dans l'ensemble de synchronisation *Sync* sont autorisées.

Définition 2.1.1.3 *Formellement, le produit synchronisé* $\left(\mathcal{A}_1 \parallel \mathcal{A}_2 \parallel \ldots \parallel \mathcal{A}_n\right)_{Sync}$ *des automates* $\mathcal{A}_i$ *est l'automate* $\mathcal{A} = (Q, E, T, q_0, l)$ *tel que :*

- $Q = Q_1 \times Q_2 \times \ldots \times Q_n$
- $E = \prod_i = 1^n(E_i \cup \{-\})$
- $T = \{((q_1, q_2, \ldots, q_n)(e_1, e_2, \ldots, e_n)(q_1^{'}, q_2^{'}, \ldots, q_n^{'}) \mid (e_1, e_2, \ldots, e_n) \in Sync\ \forall i \in 1..n, (e_i =' -'$ *et* $q_i = q_i^{'})$ *ou bien* $(e_i \neq' -'$ *et* $(q_i, e_i, q_i^{'}) \in T_i)\}$
- $q_0 = (q_{0,1}, q_{0,2}, \ldots, q_{0,n})$
- $l = \bigcup_{i=1}^{n}(l_i(q_i))$

La définition précédente est fondée sur l'utilisation d'un vecteur de synchronisation composé de transitions.

2.1.1.3 Cas particulier : produit synchronisé de deux systèmes de transitions

Le produit synchronisé sur le cas particulier de deux systèmes de transitions est souvent utilisé pour définir un pour définir un produit synchronisé entre deux systèmes de transitions. Il fournit une construction inductive et itérative du produit synchronisé de plusieurs systèmes de transitions.

Définition 2.1.1.4 *Soit* $\mathcal{A}_1 = (Q_1, E_1, T_1, q_{0,1}, l_1)$ $\mathcal{A}_2 = (Q_2, E_2, T_2, q_{0,2}, l_2)$ *deux systèmes de transitions étiquetées. Le produit synchronisé* $\mathcal{A} = (\mathcal{A}_1 \parallel \mathcal{A}_2)_{Sync}$ *avec* $Sync \subset (E_1 \cup \{-\}) \times (E_2 \cup \{-\})$ *est défini par la relation de transition :*

1. $(p_1, p_2) \longrightarrow_{\sigma_1,\sigma_2} (q_1, q_2)$ *si et seulement si* $(\sigma_1, \sigma_2) \in Sync$ *et* $p_1 \longrightarrow_{\sigma_1} q_1$ *et* $p_2 \longrightarrow_{\sigma_2} q_2$
2. $(p_1, p_2) \longrightarrow_{\sigma_1,'-'} (q_1, q_2)$ *si et seulement si* $(\sigma_1,' -') \in Sync$ *et* $p_1 \longrightarrow_{\sigma_1} q_1$ *et* $p_2 \longrightarrow_{'-'} q_2$
3. $(p_1, p_2) \longrightarrow_{'-',\sigma_2} (q_1, q_2)$ *si et seulement si* $('-', \sigma_2) \in Sync$ *et* $p_1 \longrightarrow_{'-'} q_1$ *et* $p_2 \longrightarrow_{\sigma_2} q_2$

2.1.1.4 Comportements : Entrelacement, parallélisme,...

Le produit synchronisé permet de représenter différents types de comportement et de mode d'exécution. Cette définition est fondamentale dans la description de systèmes de processus.

Toujours dans le cas du produit synchronisé de deux systèmes de transitions, on peut définir les modes de synchronisations suivants :

- *parallélisme par entrelacement* : ce mode d'exécution permet l'entrelacement de processus **asynchrones**. Seuls les cas 2 et 3 de la définition ci-dessus sont autorisées.
- *parallélisme par entrelacement de processus asynchrones* avec quelques synchronisations par rendez-vous entre actions conjuguées en autorisant les cas 2 et 3 et le cas 1 pour ces seules synchronisations.
- *vrai parallélisme entre* **processus synchrones** si le vecteur de synchronisation *Sync* contient tous les couples possibles d'actions.

2.1.2 Synchronisation par messages

Un cas particulier de produit synchronisé est défini par une synchronisation réalisée par des envois et des réceptions de messages.

Dans ce cas, parmi les étiquettes des systèmes de transitions, on désigne l'étiquette correspondant à l'envoi d'un message m notée $!m$ et l'étiquette correspondant à la réception d'un message m notée $?m$.

De plus, il faudra que le produit synchronisé n'autorise que les transitions qui assurent que toute émission de message est accompagnée de la réception correspondante (et vice versa). Cette contrainte devra être satisfaite par le vecteur de synchronisation *Sync*. Ce type de synchronisation assure une communication **synchrone**.

La contrainte précédente constitue une première propriété à assurer lors de la construction de tout système de processus où les processus sont synchronisés par envois de message.

2.1.3 Messages Asynchrones

Il existe une autre façon d'échanger des messages : la communication **asynchrone**. Ce type de communication est établi lorsque les messages ne sont pas reçus instantanément.

En mode d'exécution par communication asynchrone, on suppose que les messages déjà émis qui ne sont pas encore reçus se trouvent quelque part dans un ou plusieurs canaux, parfois appelés *buffers*. Ces messages sont gérés dans ces canaux suivant une discipline donnée comme FIFO (First In First Out, i.e l'ordre d'émission est respecté).

2.1.4 Lien synchrone-asynchrone

La communication asynchrone au travers de canaux peut se comprendre directement en termes de communications synchrone.

Pour cela il suffit d'introduire un automate i.e. un autre système de transitions (ou une variable) représentant le comportement des canaux. Un envoi asynchrone de A vers A' devient alors un échange synchrone entre A et le canal (représenté par un STE) suivi plus tard d'un échange synchrone entre le canal et A'.

2.1.5 Utilisations synchrone-asynchrone

Le mode de communication asynchrone est bien adapté à la description de protocoles de communication tandis que le mode de communication synchrone est plutôt bien adapté à la description de systèmes de contrôle/commande.

2.1.6 Synchronisation par variables partagées

Un autre moyen de faire communiquer entre elles les différentes composantes d'un système (ensemble d'automates) consiste à leur faire partager un certain nombre de variables.

Ce type de communication peut être également représenté par l'opération de produit synchronisé et les automates avec variables.

2.2 Logique temporelle

Motivations. Prenons l'exemple d'un système de contôle d'un ascenseur, et supposons que son cahier des charges contienne les propriétés suivantes :

- tout appel de l'ascenseur doit finir par être satisfait ;
- l'ascenseur ne traverse jamais un étage pour lequel un appel existe sans le satisfaire.

Ces propriétés parlent du *comportement dynamique* du système. Il serait possible de les formaliser par des notations parlant de la "position au temps t", finalement assez semblables à ce qui est utilisé en mécanique (le fameux $z(t) = (-1/2)gt^2$ qui s'applique temporairement à un ascenseur en chute libre) et cinématique (un point de vue plus descriptif, où les causes des mouvements ne sont pas considérées). En posant par exemple

$H(t)$ pour la position de la cabine à l'instant t, en notant $app(n,t)$ l'existence d'un appel pour l'étage n au temps t, nous pourrions transcrire notre première propriété sous la forme :

$$\forall t, \forall n(app(n,t) \Rightarrow \exists t' \geq t : dess(n,t')) \quad (1)$$

$$\forall t, \forall t', \forall n \left[\begin{array}{c} \Big(app(n,t) \wedge H(t') \neq n \wedge \exists t_{trav} : t \leq t_{trav} \leq t' \wedge H(t_{trav}) = n\Big) \\ \Rightarrow \Big(\exists t_{dess} : t \leq t_{dess} \leq t' \wedge dess(n, t_{dess})\Big) \end{array} \right] \quad (2)$$

Pour les théoriciens, les formules que nous venons d'écrire relèvent de la *logique du premier ordre*. Elles lèvent les ambiguïtés des propriétés écrites en français (on suppose bien sûr que H, app, ect. sont définis par ailleurs). Par exemple, le "doit finir par" est rendu par $\exists t' > t$, qui ne fixe aucune limite à t' (sinon l'interdiction de $t' = t$!). De la même façon, nous n'avons pas exigé que la date t_{dess} de la desserte soit égale à la date t_{trav} de passage : la première est une date dont l'implémentation aura la charge, l'existence de la deuxième est une hypothèse sur le comportement.

Un inconvénient flagrant des notations que nous venons d'utiliser est leur lourdeur. La **logique temporelle** est un autre formalisme, mieux adapté à la situation que nous venons d'illustrer. La logique temporelle est une forme de logique spécialisée dans les enoncés et raisonnements faisant intervenir la notion d'ordonnancement dans le temps. C'est en 1977 qu' A. Pnueli a proposé pour la première fois de l'utiliser pour la spécification formelle des propriétés comportementales des systèmes [Pnu77]. Par rapport aux formules mathématiques que nous venons d'écrire plus haut, les notations de la logique temporelle sont plus claires et plus simples. Par exemple, le paramètre t disparaît totalement. Mais la logique temporelle propose aussi des concepts prêts à l'emploi. Ses opérateurs sont calqués sur des constructions linguistiques (les adverbes "toujours", "tant que", ect., les temps de la conjugaison des verbes, etc.) de sorte que les énoncés en langue naturelle et leur formalisation en logique temporelle sont assez proches. Enfin, la logique temporelle est livrée avec une *sémantique formelle*, équipement indispensable pour un langage de spécification.

Nous allons dans cette section décrire le langage formel qu'est la logique temporelle. Nous verrons ensuite comment des proprétés concrètes sont exprimées. Il nous a fallu choisir un formalisme particulier parmi plusieurs variantes possibles : pour des raisons de généralité nous avons retenu la logique connu sous le nom de CTL^* (pour *Computation Tree Logic*) introduite par Emerson et Halpern [EH86]).

2.2.1 Le langage de la logique temporelle

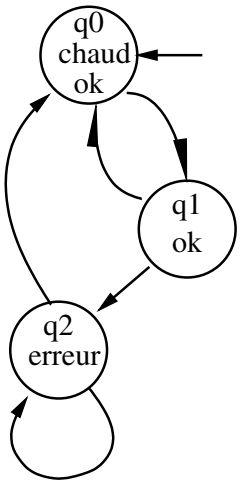
La logique temporelle CTL^*, comme les autres logiques temporelles utilisées dans les outils de model-checking, set à énoncer formellement des propriétés portant sur l'exécution d'un système.

1. Comme nous l'avons vu précédemment, une exécution est une suite d'états. La logique temporelle utilise les *propositions atomiques* pour parler des états. Ces propositions sont des énoncés élémentaires qui, dans un état donné, ont une valeur de vérité bien définie. Par exemple, on considérera que `"beau_temps"`, `"ouverture"`, `"in_phase_1"`, `"x+2=y"` dont des propositions. Rappelons qu'on les regroupe dans un ensemble noté $Prop = \{P_1, P_2, \ldots\}$ et qu'une proposition P est définie comme étant vraie si $P \in l(q)$.

   La figure 2.1 montre un automate $\mathcal{A}$, la façon dont ses états sont étiquetés par des propositions de $Prop$, et suggère quelques-unes de ses exécutions.

2. Les **combinateurs booléens** classiques sont indispensables. Il s'agit des constantes `true` et `false`, de la négation $\neg$, et des opérateurs $\wedge$ (conjonction, "et"), $\vee$ (disjonction, "ou"), $\Longrightarrow$ (implication logique[1] et $\Longleftrightarrow$ (double implication logique, "si et seulement si"). Ils permettent de construire des énoncés complexes reliant différentes sous-formules plus simples.

[1] L'implication logique conduit parfois à des malentendus. Ils disparaissent si l'on prend l'habitude de lire $P \Longrightarrow Q$ comme "si P alors Q" et pas comme "P implique Q". "P implique Q" laisse croire à une relation de cause à effet entre P et Q. "si P alors Q" se contente de constater que P et $\neg Q$ ne peuvent être tous les deux vraies. Le lecteur pourra s'essayer à lire $(1 = 2) \Longrightarrow le\_Pere\_Noel\_existe$ des deux façons et à goûter la différence.



$\sigma_1 : (q_0 : chaud, ok) \rightarrow (q_1 : ok) \rightarrow (q_0 : chaud, ok) \rightarrow (q_1 : ok) \rightarrow (q_0 : chaud, ok) \ldots$
$\sigma_2 : (q_0 : chaud, ok) \rightarrow (q_1 : ok) \rightarrow (q_2 : erreur) \rightarrow (q_0 : chaud, ok) \rightarrow (q_1 : ok) \ldots$
$\sigma_3 : (q_0 : chaud, ok) \rightarrow (q_1 : ok) \rightarrow (q_2 : erreur) \rightarrow (q_2 : erreur) \rightarrow (q_2 : erreur) \ldots$

FIG. 2.1 – Des propositions atomiques sur un automate et ses exécutions

On parle de *formule propositionnelle* quand on a affaire à une combinaison de propositions et de combinateurs booléens. Par exemple `erreur` $\Longrightarrow \neg$ `chaud` qui se lit "si `erreur` alors non `chaud`", est une formule propositionnelle vraie dans tous les états de l'exemple de la figure 2.1.

3. Les **combinateurs temporels** permettent de parler de l'enchaînement des états le long d'une exécution, et non plus d'états considérés individuellement. Les combinateurs les plus simples sont X, F et G.
   - Tandis que P énonce une propriété de l'état courant, XP énonce que l'*état suivant* (X pour "next") vérifie P. Par exemple $P \vee$ XP énonce que P est vérifiée dans l'état courant (maintenant) ou dans l'état suivant (ou les deux). Dans l'exemple de la figure 2.1, les trois exécutions σ_1, σ_2, σ_3 vérifient tous :
   $$\text{XX } \texttt{erreur} \vee \text{XXX}ok.$$
   - FP dit qu'*un état futur* (F pour "futur") vérifie P sans préciser quel état, et GP dit que "*tous les états futurs*" vérifient P. Ces deux combinateurs peuvent se lire informellement "*il y aura P un jour*" (au moins une fois) et "*il y aura toujours P*". On écrira par exmple :
   $$\texttt{alerte} \Longrightarrow \text{F } \texttt{arret}$$
   pour dire que *si* nous sommes dans un état d'alerte (maintenant *alors* nous serons dans un état d'arrêt (*plus tard*). Si nous voulons préciser que cette propriété reste toujours vraie, c'est-à-dire qu'*à tout moment* un état d'alerte sera forcément suivi d'un état d'arrêt *plus tard*, nous écrirons :
   $$\text{G}(\texttt{alerte} \Longrightarrow \text{F}\texttt{arret}).$$
   - Dans l'exemple de la figure 2.1, toute occurence d'un état `chaud`. Donc G(`chaud` $\Longrightarrow$ F$\neg$ `chaud`) est vraie pour toutes les exécutions de $\mathcal{A}$. On peut même affirmer que toutes les exécutions de $\mathcal{A}$ vérifient G(`chaud` $\Longrightarrow$ X$\neg$ `chaud`), c'est à dire qu'*il est toujours vrai que quand il fait* `chaud` *dans l'état courant, alors dans l'état suivant il ne fera pas* `chaud`.
   - G est le *dual* de F : quelle que soit la formule ϕ, si ϕ est toujours satisfaite, alors il n'est pas vrai que $\neg\phi$ soit satisfaite un jour. Donc Gϕ et $\neg$F$\neg\phi$ sont équivalentes [2], ce qu'on note G$\phi \equiv \neg$F$\neg\phi$.

4. C'est la possibilité d'*emboîter de façon arbitraire* les différents combinateur temporels qui donne toute sa force et sa richesse à la logique temporelle : l'exemple G(`alerte` $\Longrightarrow$ F `arret`) utilise un F dans la portée de G. A partir de formules plus simples, les combinateurs temporels construisent de nouvelles formules dont le sens se déduit du sens des composantes (appelées *sous-formules*).
   - L'emboîtement de F et G est très fréquemment utilisé pour exprimer des propriétés de répétition. Ainsi GFϕ qui, littéralement, doit se lire *toujours il y aura un jour un état tel que* ϕ, énonce que

[2] Il s'agit ici de l'équivalence au sens fort de la logique. Deux formules sont *équivalentes* si et seulement si elles ont la même signification, sont donc vraies dans les mêmes modèles, et peuvent être remplacées l'une par l'autre même quand elles sont sous-formules d'une formule plus large.

ϕ est vérifiée un nombre infini de fois le long de l'exécution considérée. Cette construction est si fréquente que l'on utilise l'abréviation F_∞ (lire "infiniment souvent") pour GF.
- Le dual est G_∞, abréviation de FG, qui se lit "tout le temps à partir d'un certain moment", ou bien "à tous les instants, sauf peut-être un nombre fini de fois".
- Si l'on considère une exécution dans l'exemple de la figure 2.1, il y a deux cas possibles : soit on elle visite infiniment souvent l'état `chaud`, soit elle finit par rester indéfiniment souvent dans un état `erreur` . Par conséquent, toutes les exécutions vérifient la formule F_∞ `chaud` $\vee G_\infty$ `erreur` .

5. Le combinateur U (de l'anglais *until*, ne pas confondre avec le symbole de l'union ensembliste !) est plus riche et plus compliqué. $\phi_1 U \phi_2$ énonce que ϕ_1 est vérifiée *jusqu'à* ce que ϕ_2 le soit. Plus précisément : ϕ_2 sera vérifiée un jour, et en attendant ϕ_1 restera vraie. On peut compléter l'exemple précédent en précisant qu'"à partir d'une `alerte`, l'`alarme` est en marche jusqu'à l'`arret` qui suivra forcément" :

$$G(\texttt{alerte} \Longrightarrow (\texttt{alarme U arret})).$$

Le combinateur F est un cas particulier de U dans la mesure où $F\phi$ et `true` Uϕ sont équivalents.

Il existe un "*until* faible", noté W. Dans $\phi_1 W \phi_2$, on exprime encore la notion ϕ_1 *jusqu'à* ϕ_2 mais sans exiger que ϕ_2 finisse par avoir lieu (et si ϕ_2 n'a jamais lieu, alors ϕ_1 reste vraie jusqu'à la fin). On peut lire "ϕ_1 tant que non ϕ_2". Notons que W peut s'exprimer en terme de U :

$$\phi_1 W \phi_2 \equiv (\phi_1 U \phi_2) \vee G\phi_1.$$

Dans l'exemple de la figure 2.1, toutes les exécutions issues de l'état q_0 vérifient `ok` W `erreur` mais il existe une (unique) exécution partant de q_0 qui ne vérifie pas `ok`U`erreur`.

6. Il reste à exprimer le côté arborescent du comportement (plusieurs futurs sont possibles à partir d'une situation donnée). Des quantificateurs spécialisés, A et E, permettent de quantifier sur l'ensemble des exécutions. On les appelle aussi *quantificateurs de chemins*.
   - La formule Aϕ énonce que *toutes les exécutions* partant de l'état courant satisfont la propriété ϕ , tandis que Eϕ énonce qu'à partir de l'état courant, *il existe une exécution satisfaisant* ϕ.
   - Il ne faut pas confondre A et G : Aϕ dit que toutes les exécutions partant de l'état courant satisfont la propriété ϕ , Gϕ dit qu'à tout instant de l'exécution considérée on vérifie ϕ. Plus généralement, A et E quantifient sur les chemins, F et G quantifient sur les positions le long d'un chemin donné.
   - Les combinateurs A et E d'une part, G et F d'autre part, s'utilisent souvent par paire. Par exemple, EFP dit qu'il est possible (en suivant une des exécutions) d'avoir P un jour. AFP dit que l'on aura forcément (quelle que soit l'exécution retenue) P un jour. On exprime là la différence entre le possible et l'inévitable. AGP dit que P est toujours vraie[3] tandis que EGP dit qu'*il existe* une exécution le long de laquelle P reste toujours vraie. La figure 2.2 illustre les quatre combinaisons possibles de E ou A avec F ou G.

Revenons maintenant à l'exemple de la figure 2.1. Ici toutes les exécutions partant de q_0 finissent par passer en q_1. Or en q_1, il est possible d'aller en un coup dans un état vérifiant `erreur`. Donc toute exécution partant de q_0 vérifie FEX `erreur` . Notons que l'usage du quantificateur E est crucial, et qu'il existe une exécution qui ne vérifie pas FX `erreur`.

La terminologie "*logique du temps arborescent*" (*branching time* en anglais) désigne les logiques qui disposent de cette possibilité de quantifier librement sur les chemins possibles.

Le rôle joué par les quantificateurs apparaît clairement dans la différence entre les formules AGFP et AGEFP. La première énonce que *"le long de toutes les exécutions (A), à tout moment (G), on rencontrera inévitablement plus tard (F) un état vérifiant P.* Ainsi P sera inévitablement vérifiée une infinité de fois, comme le dit plus clairement l'écriture équivalente $AF_\infty P$. La seconde formule AGEFP, énonce que *à tout moment de toute exécution, il serait possible d'atteindre P, autrement dit P est toujours potentiellement atteignable.* AGEFP *peut être vérifiée même si, dans une exécution donnée, P n'est jamais réalisée.* Le long de chaque exécution, le deuxième quantificateur (E) permet de parler de l'existence d'exécutions alternatives qui constitueraient des façons différentes de poursuivre le comportement du système.

[3] On dit aussi que P est un *invariant*. Les invariants sont des propriétés qui restent continuellement vraies. Nous les retrouverons lorsque nous parlerons des *propriétés de sûreté*

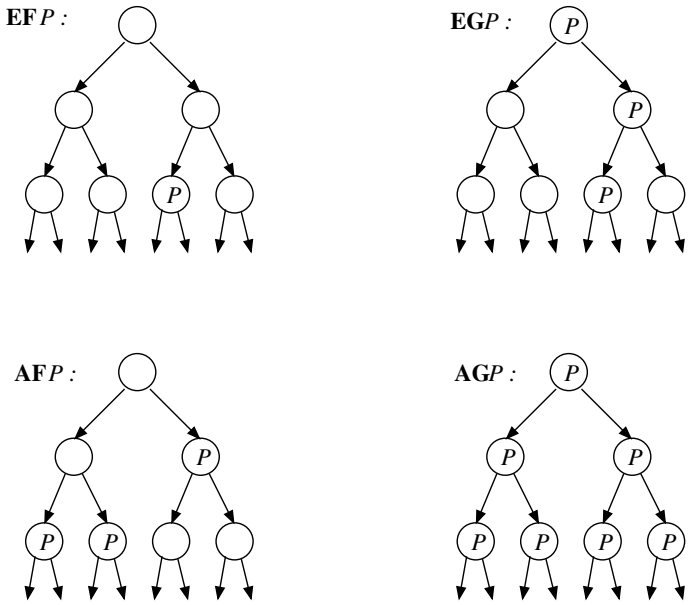


FIG. 2.2 – Quatre manières de combiner E et F

Pour CTL*, A et E sont duals l'un de l'autre, comme c'est habituel pour les quantificateurs universel et existentiels. De fait, si Aϕ n'est pas vérifiée, alors il existe une exécution qui ne vérifie pas ϕ, et donc qui vérifie $\neg\phi$. Ainsi Aϕ et $\neg$E$\neg\phi$ sont équivalents.

2.2.2 La syntaxe formelle de la logique temporelle

Les concepts que nous venons de présenter et d'illustrer conduisent naturellement à la grammaire formelle pour CTL* donnée en figure 2.3 :

ϕ, ψ	::=	$P_1 \mid P_1 \mid \ldots$	(propositions atomiques)
		$\mid \neg\phi \mid \phi \wedge \psi \mid \phi \Longrightarrow \psi \ldots$	(combinateurs booléens)
		$\mid \mathrm{X}\phi \mid \mathrm{F}\phi \mid \mathrm{G}\phi \mid \phi \mathrm{U} \psi$	(combinateurs temporels)
		$\mid \mathrm{E}\phi \mid \mathrm{A}\phi$	(quantificateurs de chemin)

FIG. 2.3 – Grammaire formelle de la logique temporelle CTL_*

Il s'agit ici d'une grammaire abstraite. En pratique, chaque outil manipulant des formules temporelles va autoriser l'emploi de parenthèses, et aura ses conventions sur la priorité des opérateurs. De même sera fixé un jeu particulier de propositions atomiques et de combinateurs. Surtout, un model-checker se restreindra souvent à un fragment de cette logique, le plus souvent CTL et PLTL que nous ne présenterons pas dans ce cours (voir [SBB+99]). Nous exposerons la logique SL (pour *Safety Logic*) utilisée pour la vérification formelle de spécification Lustre en section **??**.

2.2.3 La sémantique de la logique temporelle

Quels modèles ? Les modèles de la logique temporelle sont appelés des *structures de Kripke* (cf. section 2.1.0.1). Pour nous, ce n'est qu'un autre nom pour les automates, avec une nuance toutefois : les propositions qui étiquettent les états des automates jouent un rôle fondamental pour une logique *basée sur états*[4] telle que CTL*, et les actions qui étiquettent les transitions ont moins d'importance.

[4] Il existe bien sûr des varaintes de CTL* qui sont adaptée à des automates où ce sont surtout les étiquettes des transitions qui sont pertinentes. On parle de logique *basée sur les actions*. Ces deux points de vue sont très semblables [NV90] et on adopte l'un et l'autre en fonction des modèles avec lesquels on travaille.

Les étiquettes des transitions jouent un rôle fondamentale puisqu'elles permettent de construire un modèle en synchronisant plusieurs sous-systèmes (voir section 2.1.1.1). Ici, nous oublierons ces étiquettes de transitions et considérerons des automates $\mathcal{A} = (Q, T, q_0, l)$ ou $T \subseteq Q \times Q$. A l'inverse nous utiliserons beaucoup l'étiquetage l qui à chaque état $q \in Q$ associe l'ensemble $l(q)$ des propositions atomiques vérifiées par q. Rappelons qu'il s'agit d'une partie essentielle de la modélisation fournie par un automate : la structure de l'automate et les propositions qui étiquettent ses états sont élaborées simultanément, dans la même démarche de modélisation.

Satisfaction. Nous allons maintenant définir formellement la notion de "*formule satisfaite dans une situation donnée*". Les discussions et exemples de la section précédente montrent qu'une formule CTL* se réfère à un moment donné d'une exécution d'un automate donné.

On écrira :

Définition 2.2.3.1 *$\mathcal{A}, \sigma, i \models \phi$, et on lira "au temps i de l'exécution σ, ϕ est vraie", cela en parlant d'exécutions de $\mathcal{A}$ dont on n'exige pas qu'elles soient issues de l'état initial.*

Le *contexte* $\mathcal{A}$ est très souvent laissé implicite et on l'omet dans les écritures. On écrit $\sigma, i \not\models \phi$ pour dire que ϕ *n'est pas satisfaite* au temps i de σ.

La définition de $\sigma, i \models \phi$, se fait par induction sur la structure de ϕ. C'est-à -dire que la valeur de vérité d'une formule composée est donnée à partir des valeurs de vérité de ses sous-formules.

$\sigma, i \models \mathrm{P},$	ssi	$P \in l(\sigma(i))$
$\sigma, i \models \neg\phi,$	ssi	il n'est pas vrai que $\sigma, i \models \phi$
$\sigma, i \models \phi \wedge \psi,$	ssi	$\sigma, i \models \phi$ et $\sigma, i \models \psi$
	ssi	
$\sigma, i \models \mathrm{X}\phi,$	ssi	$i < \|sigma\|$ et $\sigma, i+1 \models \phi$
$\sigma, i \models \mathrm{F}\phi,$	ssi	il existe j tel que $i \leq j \leq \|\sigma\|$ et $\sigma, j \models \phi$
$\sigma, i \models \mathrm{G}\phi,$	ssi	pour tout j tel que $i \leq j \leq \|\sigma\|$, on a $\sigma, j \models \phi$
	ssi	
$\sigma, i \models \phi \mathrm{U} \psi,$	ssi	il existe $j, i \leq j \leq \|\sigma\|$ tel que $\sigma, j \models \psi$ et
	ssi	pour tout k tel que $i \leq k < j$, on a $\sigma, k \models \phi$
	ssi	
$\sigma, i \models \mathrm{E}\phi,$	ssi	il existe σ' tel que $\sigma(0) \ldots \sigma(i) = \sigma'(0) \ldots \sigma'(i)$ et $\sigma', i \models \phi$
$\sigma, i \models \mathrm{A}\phi,$	ssi	pour tout σ' tel que $\sigma(0) \ldots \sigma(i) = \sigma'(0) \ldots \sigma'(i)$, on a $\sigma', i \models \phi$

FIG. 2.4 – La sémantique de CTL_*

La figure 2.4 présente neuf clauses de définition correspondant à neuf façons différentes de construire une formule temporelle à partir de sous-formules. (Rappelons que $\sigma(i)$ est le i-ème état de σ et que $|\sigma|$ est la longueur de σ.) Les clauses pour les opérateurs dérivés ($\Longrightarrow, \vee, \mathrm{F}_\infty, \mathrm{W}$, etc.) s'en déduisent et ne sont pas explicitées, certaines des clauses retenues (celles pour F, G et A) sont redondantes et pourraient être déduites des autres.

On peut alors introduire une notion dérivée, "l'automate $\mathcal{A}$ satisfait ϕ", notée $\mathcal{A} \models \phi$, et définie par :

Définition 2.2.3.2 *$\mathcal{A} \models \phi$ ssi $\sigma, 0 \models \Phi$ pour toute exécution σ de $\mathcal{A}$.*

C'est une notion bien commande pour parler de la correction d'un modèle. Mais elle n'est pas élémentaire au sens où elle regroupe la correction de toutes les exécutions (issues de q_0) d'un modèle. Ainsi, $\mathcal{A} \not\models \phi$ n'implique pas nécessairement $\mathcal{A} \models \neg\phi$ (alors que $\sigma, i \not\models \Phi$) équivalent à $\sigma, i \models \neg\Phi$).

La nature du temps. On retrouve dans les définitions de la figure 2.4 la lourdeur des formules du premier ordre vues en introduction 2.2. Le "il existe j tel que $i \leq j \leq |\sigma| \ldots$" de la clause pour F évoque un $\exists t' \geq t$. De fait, dans un énoncé de la forme à $\sigma, i \models \Phi$, le paramètre i représente bien le temps qui s'écoule le long de σ. Néanmoins, il existe une différence importante entre les deux cadres. La sémantique de CTL*

précise quelle est la nature du temps : les instants sont les points le long des exécutions. Les formules du premier ordre laissent cette question implicite. Quand on écrit $\exists t' > t$, où se trouve t' ? Plus tard dans la même exécution ou bien plus tard dans une autre exécution ? Et d'abord qu'est-ce que t ? Si l'on souhaitait formaliser un cahier des charges en utilisant les formules du premier ordre, il serait nécessaire de répondre à toutes ces questions, c'est-à-dire de choisir un modèle du temps.

En CTL*, le temps *discret*, contrairement au temps *continu* ou *dense* tel qu'il est définit en physique. En CTL* il n'y a rien entre les instants i et $i+1$. La logique temporelle rend le paramètre temps implicite : tout énoncé fait implicitement référence à un instant courant. Le choix de combinateurs fixe une bonne fois pour toutes quelles constructions peuvent être utilisées. Les propriétés qu'il serait plus simple d'exprimer directement en logique du premier ordre existent mais elles sont rares. On peut dire que la logique temporelle, par rapport à la logique du premier ordre, est comme un langage de haut-niveau qui se compilerait en langage machine.

2.3 Model-Checking

Nous allons décrire très brièvement dans cette section les principes sous-jacents des *algorithmes* utilisés pour le model-checking, c'est-à-dire les algorithmes qui permettent de savoir si un automate donné vérifie une formule temporelle donnée.

2.3.1 Model-Checking de CTL

L'algorithme de model-checking pour CTL est dû essentiellement à Queille, Sifakis, Clarke, Emerson et Sisla [QS82, CES86] et a été amélioré par la suite (cf. par exemple [CGL94]).

Cet algorithme fondamental tient une place très importante dans le domaine de la vérification. Ceci vient en partie du fait qu'il ne demande qu'un temps linéaire en chacune de ses composantes (l'automate d'une part, et la formule CTL de l'autre). Il utilise le fait que CTL ne permet d'exprimer que des formules d'état. En effet, cette particularité de CTL permet de raisonner en termes de quels états vérifient quelles formules, plutôt que de considérer les exécutions qui sont les vrais objets auxquels nous nous intéressons.

Principe de base. La composante fondamentale de l'algorithme de model-checking pour CTL est une procédure `marquage` qui travaille sur un automate $\mathcal{A}$ et qui, à partir d'une formule CTL ϕ, va marquer, pour chaque état q de l'automate et pour chaque sous-formule ψ de ϕ, si ψ est satisfaite dans l'état q. A la fin, pour chaque état et chaque sous-formule, `q.psi` vaut `true` si $q \models \psi$, `false` sinon.

On emploie le terme de "marquage" pour signifier que la valeur de `q.psi` est calculée puis mémorisée. La mémorisation est importante car le marquage de `q.phi` utilise les valeurs `q'.psi` pour des sous-formules `psi` de `phi` et des états `q'` atteignables à partir de `q`. Qaund le marquage pour `phi` est achevé, il est facile de dire si $\mathcal{A} \models \phi$ en consultant la valeur de `q0.phi` pour l'état initial q_0 de $\mathcal{A}$. Voici le corps de l'algorithme :

```
procedure marquage(phi)

cas 1 : phi = P
  pour tout q dans Q, si P dans l(q) alors faire q.phi := true
                      sinon faire q.phi := false

cas 2 : phi = not psi
  faire marquage(psi);
  pour tout q dans Q, faire q.phi := not(q.psi).

cas 3 : phi = psi1 /\ psi2
  faire marquage(psi1) ; marquage(psi2) ;
  pour tout q dans Q, faire q.phi := et(q.psi1, q.psi2).

cas4 : phi = EX psi
  faire marquage(psi);
  pour tout q dans Q, faire q.phi := false /* initialisation */
```

```
  pour tout (q,q') dans T, si q'.psi = true alors faire q.phi := true.

cas 5 : phi = E psi1 U psi2
  faire marquage(psi1); marquage(psi2);
  pour tout q dans Q,
    q.phi := false; q.dejavu := false;    /* Initialisation*/
  L := {};
  pour tout q dans Q, qi q.psi2 = true alors faire L := L + {q};
  tant que L non vide {
    prendre un q dans L;
    L := L - {q};
    q.phi := true;
    pour tout (q',q) dans T {
      si q'.dejavu = false alors faire {
        q'.dejavu := true;
        si q'.psi1 = true alors faire L:= L + {q'};
        }
      }
    }

cas 6 : phi = A psi1 U psi 2
  faire marquage(psi1); marquage(psi2);
  L := {}
  pour tout q dans Q,
    q.nb := degre(q); q.phi := false;
  pour tout q dans Q, si q.psi2 = true alors faire L:= L + {q};
  tant que L non vide {
    prendre un q dans L;
    L := L - {q};
    q.phi := true;
    pour tout (q',q) dans T {
      q'.nb := q'.nb - 1;
      si (q'.nb=0) et (q'.psi1 = true) et (q'.phi = false)
        alors faire L := L + {q'};
    }
  }
```

On voit comment le marquage est simple à faire quand ϕ est une proposition atomique (cas 1), une négation (cas 2) ou une conjonction (cas 3). Dans chacun de ces trois cas, le marquage pour ϕ n'a besoin que d'un parcours de Q - donc d'un temps en $O(|Q|)$ - en plus du travail demandé par le marquage pour les sous-formules de ϕ.

Quand ϕ est de la forme $\mathrm{EX}\psi$ (cas 4), le marquage ne demande qu'un parcours de T (l'ensemble des transitions de l'automate). Ainsi cette étape ne nécessite pas plus d'un temps $O(|T|)$, ceci en plus de l'initialisation et du marquage pour ψ. Le cas $\mathrm{AX}\psi$ n'a pas été explicité : il est équivalent à $\neg\mathrm{EX}\neg\psi$.

Quand ϕ est de la forme $\mathrm{E}\psi_1\mathrm{U}\psi_2$ (cas 5), le marquage pour ϕ à partir des marquages pour ψ_1 et ψ_2 utilise un algorithme standard pour l'atteignabilité contrôlée dans un graphe (à ceci près que les transitions sont parcourues en arrière). Nous avons choisi de donner une description détaillée d'un tel algorithme de façon à bien souligner comment il est possible d'implémenter cette étape en ne visitant chaque transition $(q, q') \in T$ qu'une fois au plus, de sorte que le calcul peut se faire en temps $O(|Q| + |T|)$.

L'algorithme de marquage pour ϕ de la forme $\mathrm{A}\psi_1\mathrm{U}\psi_2$ (cas 6) est un peu plus compliqué. Il se base sur la constatation qu'un état q vérifie $\mathrm{A}\psi_1\mathrm{U}\psi_2$ si et seulement si soit **(a)** q satisfait ψ_2, soit **(b.1)** q satisfait ψ_1, **(b.2)** q a au moins un état successeur, et **(b.3)** tous ses successeurs satisfont $\mathrm{A}\psi_1\mathrm{U}\psi_2$. L'algorithme de marquage va maintenir un compteur `nb` associé à chaque état. Au départ, `q.nb` vaut `degré(q)`, c'est-à-dire le nombre de successeurs de q dans le graphe de l'automate. Par la suite, chaque fois qu'un successeur de q est marqué comme vérifiant $\mathrm{A}\psi_1\mathrm{U}\psi_2$, le compteur de q est décrémenté. Quand après décrémentation `q.nb` atteint la valeur 0, on sait que tous les successeurs de q vérifient $\mathrm{A}\psi_1\mathrm{U}\psi_2$. Si de plus q vérifie ψ_1, on sait

alors qu'il vérifie ϕ.

2.3.2 Le problème de l'explosion du nombre d'états

Le principal obstacle que rencontrent les algorithmes de model-checking est le problème dit "d'explosion du nombre d'états" (en anglais, *state-explosion problem*).

En effet, les algorithmes reposent sur une construction explicite de l'automate $\mathcal{A}$ à vérifier : il est nécessaire de disposer de $\mathcal{A}$ pour le parcourir et l'étiqueter.

En pratique, le nombre d'états de $\mathcal{A}$ est vite très élevé. Quand on construit $\mathcal{A}$ par synchronisation de composants $\mathcal{A}_1, \ldots, \mathcal{A}_n$ (cf. section 2.1.1.1), le résultat aura une taille de l'ordre de $|\mathcal{A}_1| \times |\mathcal{A}_2| \times \ldots \times |\mathcal{A}_n|$, c'est-à -dire potentiellement exponentielle par rapport à la description du système.

Encore plus fréquemment, on rencontre des situations d'explosion chaque fois que l'on utilise des automates interprétés, par exemple des automates travaillant sur quelques variables d'état. Dans la mesure où le comportement (et peut-être certaines propositions atomiques) dépendent de la valeur des variables, l'automate $\mathcal{A}$ qui sera soumis au model-checker doit être l'automate des configurations. Par exemple, pour un automate à $m = |Q|$ états de contrôle et n variables d'état simplement booléennes, le model-checker devra examiner un automate à $m.2^n$ états.

Quand le système examiné requiert la mémorisation dans un état global de valeurs non bornées (des entiers, une file d'attente, ect.) le système donne en fait lieu à un automate à nombre inifini d'états et les méthodes classique comme celle présentée ne s'applique plus.

2.3.3 Model-Checking symbolique

De façon générale, le terme de *model-checking symbolique* s'applique à toute méthode de model-checking qui chercherait à représenter de façon symbolique (par opposition à "de façon explicite") les états et les transitions d'un automate à vérifier. Par ailleurs, on utilise souvent ce terme pour désigner une méthode symbolique particulière où l'on utilise des *diagrammes de décision binaires* pour représenter les ensembles d'états.

Comme nous l'avons vu précédemment le principal problème des algorithmes de model-checking est l'explosion du nombre d'états. Cette explosion se produit chaque fois que l'on décide d'énumérer et de représenter explicitement en mémoire tous les états de l'automate examiné.

L'idée sous-jacente aux méthodes symboliques est de pouvoir représenter de façon concise des ensembles très grands d'états et de les manipuler en quelque sorte par paquets. Remarquons qu'une telle approche devient moins sensible à la finitude du nombre total d'états et qu'elle peut s'appliquer également à des systèmes à infinité d'états (par exemple disposant de canaux non bornés, de variables entières non bornées ou d'horloges à valeur dans $\mathbb{R}$, utilisant un parallélisme dynamique, etc.)

Vous trouverez de plus amples renseignements sur ce sujet dans [SBB+99].

Chapitre 3

La programmation réactive

Ce chapitre est en grande partie une traduction d'un chapitre du livre "*Synchronous Programming of Reactive Systems*" de N. Halbwachs [Hal93].

Suivant l'historique présenté par Halbwachs [Hal93] le terme de "programmation réactive" a été introduit dans le but d'éviter toute ambiguité avec celui de la "programmation temps réel", terme plus connu mais possédant tant d'acceptions qu'il en devient largement galvaudé et mal compris.

D'un point de vue historique, l'étude de la gestion du temps au niveau des systèmes s'est relevé tardive et laissée de coté par la recherche. Jusqu'au début des années 1980, les problèmes liés au temps n'ont été considérés qu'en terme d'évaluation de performances, d'ingénierie industriel ou, au mieux, en terme de systèmes d'exploitation.

Par contraste, le courant des années 1980 a connu un développement de la recherche très important concernant les sytèmes liés au temps. La gestion du temps est soudainement devenu un objectif fondamental pour la plupart des modèles de concurrence. En particulier, les travaux d'avant-garde de Robin Milner sur les algèbres de processus synchrones ont donner naissance à une école de pensée qui adopte le point de vue abstrait suivant :

Dès que nous admettons que le système peut réagir instantanément aux évènements, i.e, si le temps d'exécution de la machine est considéré comme négligeable vis à vis des délais de réponse de son environement, le comportement temporel d'un système peut être formalisé d'une manière simple et élégante.

Ce point de vue synchrone, de manière surprenante, a été appliqué pour la programmation quasi exclusivement au sein de projets français. Trois projets, débutés au début des années 1980 de manière indépendantes ont mis au jour 3 langages de programmation synchrone : ESTEREL (ENSMP[1] & INRIA[2]), SIGNAL (INRIA/IRISA[3]), et LUSTRE (IMAG[4]).

D'autres langages comme SML, STATECHARTS, ou L.0 ont été développé dans d'autres pays, adoptant certains aspect du paradigme synchrone ; cependant, d'une part ces langages n'utilisent pas complètement le modèle synchrone, et d'autres part, ces langages n'étaient pas destinés à être utilisés pour la programmation (SML est un langage de description d'architecture matérielle, STATECHARTS a été concu comme un langage de specification et L.0 est un langage pour spécifier des protocoles de communication).

Les trois groupes français ont rapidement remarqué que leurs langages étaient basé sur le même modèle. Une coopération s'est alors mise en place visant principalement l'étude des méthodes de compilation et la diffusion du point de vue synchrone dans le milieu industriel. A cette communauté s'est joint un autre projet, concernant le langage ARGOS (IMAG), une variante purement synchrone du formalisme STATECHARTS.

[1] Ecole des Mines de Paris
[2] Institut National de Recherche en Informatique et Automatique
[3] Institut de Recherche en Informatique et Systèmes Aléatoires
[4] Institut d'Informatique et Mathématiques Appliqués de Grenoble

3.1 Systèmes réactifs

Nous appelons **systèmes réactifs**, les systèmes logiciels qui réagissent de manière continue à leur environnement, et à une vitesse déterminée par cet environnement. Cette classification a été introduite par [HP85] [Ber89] afin de différencier ces systèmes des systèmes transformationnels d'une part (i.e les systèmes classiques dont les entrées sont disponibles en début d'exécution et qui fournissent des sorties quand ils se terminent) et des systèmes interactifs d'autres part (ie. systèmes qui interagissent de manière continue avec leur environnement mais à leur vitesse propre, comme les systèmes d'exploitation par exemple).

La plupart des systèmes industriels "temps-réel" sont des systèmes réactifs. Nous pouvons également citer comme autres exemples les protocoles de communication ou les interfaces homme-machine.

3.1.1 Caractéristiques des systèmes réactifs

Les particularités des systèmes réactifs sont les suivantes :

- **Concurrence** : Tout au moins, la concurrence entre le système et son environnement doit être pris en compte. De plus, il est souvent utile et naturel de considérer un tel système comme un ensemble de composants (i.e processus ou tâches) parallèles qui coopèrent dans le but d'atteindre le comportement désiré. Enfin, ces systèmes sont parfois (et c'est le cas pour les systèmes avioniques) implémentés sur des architectures distribuées afin d'accroître leurs performances et leur fiabilité.
- Ils sont soumis à des **contraintes de temps strictes** : Ces contraintes concernent le temps de réponse du système à une sollicitation en entrée. Ces contraintes doivent être exprimées dans les spécifications du système (cahier des charges), doivent être prises en compte à la conception et doivent être satisfaites sur l'implémentation. Le respect des ces contraintes de temps nécessitent une implémentation efficace, et plus spécifiquement une évaluation précise des temps d'exécution.
- Ils sont en générale **déterministes** : Les sorties des systèmes réactifs sont entièrement déterminées par les valeurs et les occurrences dans le temps des entrées . Cet aspect déterministe distingue les systèmes réactifs des systèmes interactifs : la majorité des systèmes interactifs sont intrinsèquement non-déterministes. Un système d'exploitation, par exemple, dispose d'un ordonnanceur qui active et désactive dynamiquement les processus suivant certains critères (charge du CPU, disponibilité des ressources, priorité des tâches,...). Le résultat d'un appel système dépend généralement de ces paramètres. La conception, l'analyse et le débogage d'un système déterministe sont bien plus aisés. Par conséquent, le déterminisme des spécifications de systèmes réactifs se doit d'être préservé lors de leur implémentation.
- **Leur fiabilité est un point crucial** : Il est commun de dire que les erreurs dans un système réactif peut avoir des conséquences dramatiques : ils impliquent des vies humaines et un important coût financier. Les conséquences économiques et humaine liées à une erreur dans un logiciel de commande d'un satellite ou d'une centrale nucléaire est bien évidemment inestimable. Par conséquent, ces systèmes nécessitent des méthodes de conception particulièrement rigoureuses et constituent un champ ou les *Techniques formelles de vérification* doivent être considérées.

3.1.2 Approches classiques pour la conception des systèmes parallèles

Les systèmes réactifs ont pendant longtemps été implémenté par des dispositifs matériels : machines analogiques, systèmes de switch, circuits... L'implémentation logicielle de tels systèmes est souvent programmée en langage assembleur pour des questions d'efficacité.

A un plus haut niveau, des langages parallèles (i.e permettant la programmation de systèmes concurrents) sont utilisés, permettant de modéliser le système. Principalement, les modèles utilisés sont : les automates, les réseaux de Petri et les modèles de processus communiquants.

Automates déterministes. Les automates sont souvent utilisés pour implémenter le coeur (partie contrôle de l'application) des systèmes réactifs. A partir d'un ensemble de valeurs d'entrée (évènements), l'automate selectionne une transition à partir de son état courant, appelle la tâche séquentielle correspondante (réaction), et change son état en prévision de sa prochaine réaction. Une telle approche ammène généralement à de très bonnes performances qu'il est possible d'analyser formellement ; une réaction est un morceau de

code "linéaire" (ni boucle ni récursion, pas d'interruptions...), dont le temps d'exécution maximal peut être déterminé en moyenne.

De plus, les automates sont des objets mathématiques bien connus pour lesquels les techniques de vérification formelle sont disponibles. (cf. Chapitre II)

Cependant, les automates sont des objets "plats" ne disposant ni de mécanismes hiérachisants ni de structures permettant d'exprimer le parallélisme de processus concurrents. En conséquence, ces modèles sont difficilement exploitables pour concevoir des systèmes complexes. Ecrire un automate avec seulement une dixaine d'états est une tâche ardue et sujette aux erreurs. En outre la moindre modification dans la spécification du système va nécessiter une réécriture complète de l'automate.

Modèles basés sur les Réseaux de Petri (RdP). Ces modèles sont principalement utilisés pour les programmes industriels de contrôle-commandes. La possibilité d'exprimer la concurrence dans ces modèles réduit considérablement la complexité de la description du système. Cependant, à cause de leur manque de capacité structurante, ces modèles sont difficilement exploitables pour des systèmes imposants. En outre, leurs sémantiques, et tout particulièrement en ce qui concerne les aspects temporels, est souvent mal défini.

Modèles à base de tâches. (Langage classique + OS Temps Réel : VxWorks, PSOS, OS9, QNX) Ici, l'approche consiste à concevoir le système comme composé d'un ensemble de tâches séquentielles, activées et controllées par un système d'exploitation temps-réel . Le système est décomposé en un ensemble de tâches qui communiquent ensemble par le biais d'une mémoire partagée. La principale faiblesse de cette approche est de ne pas prendre en compte les contraintes temporelles dans la description du système. Ces contraintes temporelles sont uniquement prise en compte au niveau de la politique d'ordonnancement du système d'exploitation (interruption, priorités,...). L'analyse du système est rendu difficile du fait du non-déterminisme introduit par une telle représentation et du manque de vue globale sur le système. Les performances peuvent être déteriorées du fait de l'ordonnancement dynamique.

Processus communiquants. Les langages parallèles tels ADA, OCCAM ou JAVA sont d'un plus haut niveau que les modèles à base de tâches. Ces langages offrent des primitives de haut-niveau pour structurer le programme et ses données. Les mécanismes de communication et de synchronisation (rendez-vous, file d'attente fifo,...) sont plus propres que l'utilisation d'une mémoire partagée. Ces langages ont été développés dans le but d'accroître la portabilité des programmes.

Cependant, cette portabilité est atteinte au prix du non-déterminisme du système. Afin de rendre indépendant le comportement du programme vis à vis de l'architecture cible (mono ou multi-processeurs), seules quelques hypothèses sont faites concernant la synchronisation inter-processus. Bien que ces langages fournissent des primitives "temps-réel", la sémantique de ces constructions reste vague. Nous pouvons illustrer ces problèmes par l'example ADA suivant, ou une tâche A signale `MINUTE` à une tâche B en comptant les "secondes" :

```
loop
delay 60; B.MINUTE
end
```

Ce programme ne possède pas le comportement attendu : pour que l'évènement $MINUTE$ soit reçu par B, A doit avoir attendu 60 secondes, mais B doit également lire le message reçu et en outre, le rendez-vous doit avoir eu lieu - or la date de ce rendez-vous n'est pas spécifié dans la sémantique du langage. Le délai séparant deux réceptions successives du message `MINUTE` est donc d'*au moins 60 secondes*.

De plus, un signal ne peut pas être "broadcaster", c'est à dire diffusé en même temps à plusieurs tâches. Si A doit envoyer le message `MINUTE` à une troisième tâche C, A devra alors exécuter `C.MINUTE`. Par conséquent B et C ne recevrons jamais le message `C.MINUTE` au même moment.

Dans un tel langage, les différents processus ne possèdent jamais la même vue de l'état global du programme.

Pour conclure avec ce bref aperçu des principaux outils de conception des systèmes réactifs, il est important de noter que le programmeur doit choisir entre *déterminisme* et *concurrence*. Tous les langages parallèles sont basés sur un schéma d'exécution *asynchrone* où les processus sont en compétition les uns avec les autres pour utiliser les ressources et où cette compétition ne peut être résolue de manière déterministe.

Les langages synchrones peuvent être vus comme une tentative de réconciliation entre concurrence et déterminisme.

3.2 L'approche synchrone

Les langages synchrones ont été conçus pour rendre la tâche du programmeur plus facile, en lui fournissant les primitives "idéales" qui permettront de considérer le programme comme réagissant *instantanément* aux évènements extérieurs. Les évènements internes et les évènements de sortie sont datés précisément en respect avec le flux des évènements d'entrée. Le comportement d'un programme est entièrement déterministe, aussi bien du point de vue fonctionnel que du point de vue de la gestion du temps.

En réalité, la notion de temps physique (ou chronomètrique) est remplacé par la notion d'ordre sur les évènements : en effet les seules notions intéressantes sont la simultanéité et la précédence entre les évènements. Le temps physique ne joue aucun rôle (comme c'est le cas pour Ada) ; le temps sera pris en compte comme un évènement extérieur, exactement comme les évènements en provenance de l'environnement du programme. On parle de "*notion de temps multi-forme*".

Quand nous parlerons d'*instants*, cette notion doit être comprise comme "instant logique" : l'histoire d'un système est une séquence d'instants logiques ; à chacun de ces instants, zéro, un ou plusieurs évènements interviennent. Les évènements intervenant au même instant logique sont considérés comme simultanés ; ceux qui interviennent à des instants différents sont ordonnés suivant les instances de leur occurence. Mise à part en ces instants logiques, aucun évènement et aucune réaction du système n'a lieu. Par conséquent, tous les processus du système ont la même connaissance des évènements ayant lieu à un même instant.

En pratique, **l'approche synchrone est une hypothèse selon laquelle le programme s'exécute assez rapidement pour percevoir tous les évènements externes**. Autrement dit, l'éxécution du programme est assez rapide pour se terminer avant le prochain instant logique. Si cette hypothèse est satisfaite - et plus précisément si cette hypothèse peut être vérifiée - l'hypothèse synchrone est alors tout aussi réaliste que celle considérant qu'une machine travaille avec des entiers ou des nombres réels.

De plus, nous verrons que les langages synchrones (Lustre et Esterel en particulier) peuvent être implémentés d'une façon efficace et mesurable. Le code objet issu de la compilation d'un programme synchrone est structuré comme un automate fini : chaque transition correpond à une réaction du programme. Le code correspondant à une telle transition est linéaire (sans boucle), et son temps maximal d'exécution peut être déterminé sur une machine donnée. C'est pourquoi la validité de l'hypothèse synchrone peut être vérifiée.

3.3 Systèmes complexes

Cependant, les langages synchrones ne prétendent pas résoudre tous les problèmes intervenant dans la conception des systèmes temps-réel. Un système temps-réel réel implique généralement la coopération de trois types de programmes : par exemple, un programmeur réalisant une interface réactive (clavier, souris, menus graphiques, widgets...) appelle les services interactifs du système d'exploitation et active des tâches transformationnelles.

3.4 Les langages Synchrones

Historiquement, le premier langage Synchrone est Esterel et a été développé au Centre de Mathématiques Appliquées (CMA) de l' École des Mines de Paris à Sophia-Antipolis. Peu de temps après l'INRIA à rejoins le développement de ce langage. Esterel est un langage impératif qui fut initialement inspiré de CCS et SCCS. Esterel introduit des constructions telles que la préemption et la communication par diffusion synchrone de messages. Esterel est un langage dédié à la programmation de systèmes à évènements discrets. L'entreprise Esterel Technologies propose une version industrielle du compilateur Esterel.

Il existe beaucoup d'autres langages synchrones. En voici une liste non exhaustive, présenté par ordre chronologique :

- Comme nous le verrons au Chapitre 4, **Lustre** est un langage à flot de données. Ce langage déclaratif et fonctionnel est inspiré de Lucid. L'outil **Scade**, initialement développé par Verilog et l'Aérospatiale est basé sur le langage Lustre. Scade est aujourd'hui mis sur le marché par Esterel Technologies.
- **Signal** est également un langage à base de flots de données déclaratif mais contrairement à Lustre n'est pas un langage fonctionnel mais plutôt relationnel.
- **Argos** est une pure version du formalisme Statecharts. Argos dispose d'une sémantique compositionnelle. **SyncCharts** et **Mode Automata** sont tous deux inspiré d'Argos.
- **SL**, pour Synchronous Language, est une variante d'**Esterel** pour laquelle l'hypothèse concernant la présence ou l'absence de signal n'est pas autorisée. Le fait qu'un signal soit présent ou absent ne peux être décidé qu'à la fin de l'instant synchrone, par conséquent la réaction à un signal est reporté jusqu'à l'instant suivant. Le principal avantage est d'éviter le problème de causalité. SL est à l'origine de nombreux autres langages synchrones tels que **Sugar Cubes**, **Junior**...

Alors qu'Esterel, Argos et SL conviennent mieux à la modélisation de systèmes à événements discrets, Lustre et Signal sont plus proches des formalismes de spécification utilisés par les ingénieur automaticiens : diagrammes de blocs, équations différentielles, réseaux de flots de données, automates, ect...

3.5 Développements industriels

Les langages synchrones connaissent depuis quelques années déjà une belle percée dans le monde industriel, notamment au sein des entreprises développant des logiciels de contrôle-commandes pour des applications critiques telles que Schneider, Dassault, Airbus, Snecma, Cadence, Thomson...

Par exemple, Lustre a été utilisé pour les commandes de vol de l'A380 chez Airbus, pour des contrôleur de centrales nucléaires (C03N4), pour un logiciel de signalisation du métro de Hong-Kong, chez Eurocopter pour le développement du pilotage automatique...

D'un point de vue général [BG], nous pouvons distinguer trois parties dans un système temps-réel complexe :

- *Une interface interactive* avec l'environnement qui réalise l'acquisition des entrées et calcule les sorties. Ce niveau prend en compte la gestion des interuptions, la lecture des capteurs, la conversion entre des signaux logiques et analogiques des entrées-sorties.
- *Un ou plusieurs noyaux réactifs*. Un tel noyau calcule les sorties à partir des données logiques en entrée, en selectionnant la réaction appropriée (calculs et émissions des signaux de sortie) aux évènements d'entrée.
- Un niveau de *gestion des données*, qui effectue des tâches transformationnelles sous le controle du noyau réactif.

Ce cours s'interessera principalement à la conception du noyau réactif qui est sans doute la partie la plus spécifique et la plus difficile au niveau de la conception des systèmes temps-réel. Cependant, il faut garder à l'esprit que ces noyaux ont pour vocation d'être intégrés dans des systèmes plus complexes. Par conséquent, les langages synchrones ne sont pas des langages complets. En particuliers, ils n'offrent pas les primitives pour définir et manipuler des structures de données complexes qui sont réservées aux langages classiques (langage hôte).

De plus, les compilateurs de langages synchrones produise le code source de l'application dans le langage hôte, code qui devra être intégré dans un programme plus important.

Chapitre 4

Le langage synchrone LUSTRE

Le langage LUSTRE, présenté dans ce chapitre est un langage de haut niveau permettant d'implémenter un modèle formel du système. On parle de langage de spécification formelle.

Comment nous l'avons vu dans les Chapitres 1 et 2, l'utilité de construire une spécification formelle du système est de pouvoir prouver que le système satisfait un certain nombre de propriétés exprimées dans le cahier des charges du système. Ces propriétés sont formalisées de différentes manières suivant la technique de preuve que nous souhaitons utiliser. En l'occurence, nous pourrons formaliser ces propriétés dans la logique temporelle SF (pour *Safety Logic*) pour ensuite les vérifier en utilisant un model-checker appelé LESAR. L'originalité de la méthode Lustre, comme nous le verrons au cours de ce chapitre, est de pouvoir exprimer les propriétés à valider sans avoir recours à un nouveau formalisme de spécification. En effet, la logique temporelle SF, permettant la preuve de *propriétés de sûreté* est exprimable dans le langage Lustre lui-même ! Une propriété sera donc exprimé par un programme Lustre...

Dans ce chapitre nous présenterons en premier lieu les concepts de base du langage Lustre. Nous détaillerons ensuite les constructions de ce langage. Enfin nous présenterons la technique de preuve utilisé.

4.1 Introduction

Les systèmes réactifs appartiennent plus au champ d'activité des automaticiens et des électroniciens que des informaticiens. Par conséquent les outils de description du domaine sont similaires des outils traditionnels de la *théorie du contrôle* (Automatique) : ces outils consistent, à un haut niveau, en des formalismes équationnels (différentiels, équations booléennes, etc...), et à un bas niveau, de nombreux formalismes graphiques pour décrire des réseaux d'opérateurs (diagrammes de blocs, schémas analogiques, diagrammes d'états, graphcet etc...). Tous ces formalismes apprtiennent au modèle "**flots de données**". Dans ce modèle, le système est perçu comme un réseau d'opérateurs interconnectés dont l'exécution est parallèle et qui sont activé par l'arrivée des entrées (cf. figure 4.1).

Ce modèle a initialement été proposé pour la programmation en général. Cependant, celui-ci n'a pas connu un réel succès dans ce contexte, d'une part car ce modèle est éloigné des concepts généralement utilisés par les informaticiens et d'autres part parce que aucune implémentation efficace a été proposée pour les langages à flots de données.

Aujourd'hui, bien que ce modèle soit éloigné des habitudes des informaticiens, il est très naturel pour les automaticiens, qui doivent par conséquent traduire leur point de vue "flot de données" dans un modèle de programmation impérative classique. Même du point de vue des informaticiens, le modèle flot de données possède plusieurs avantages :

- C'est un modèle parallèle (ide. permettant l'expression de processus concurrents) de granularité fine. Conceptuellement, tout comme une puce électronique dans un circuit électrique, dès qu'un opérateur dispose de ses entrées il peut calculer ses sorties. Ainsi, les seules contraintes de synchronisation viennent des relations de dépendance entre les données. Lorsque un informaticien cherche des modèles et des langages qui tirent avantage du parallélisme des ordinateurs, il semble paradoxale que certains utilisateurs pour qui le parallélisme est un point de vue naturel se doivent de traduire leurs modèles

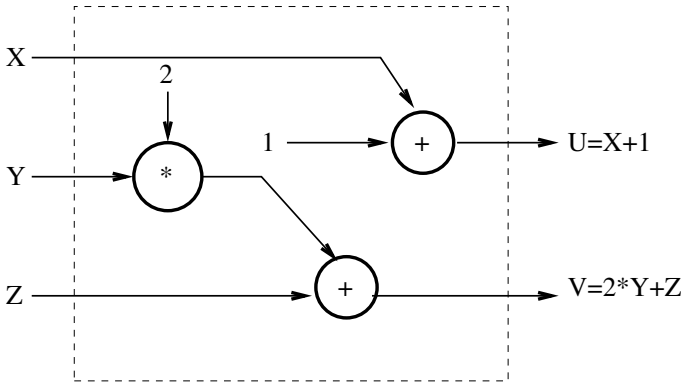


FIG. 4.1 – Descriptions graphique et équationnelle d'un système flot de donnés.

dans des formalismes séquentiels. Une description parallèle de plus ou moins faible granularité permet une grande variété d'implémentations possibles. Il est en effet bien plus difficile de paralléliser un programme séquentiel que de séquentialiser un programme parallèle.

- Généralement, les formalismes *flots de données* sont mathématiquement plus propres que les formalismes impératifs, dans lequel les notions telles que la mémoire et l'affectation impliquent des effets de bord complexes. La bonne définition mathématique rend plus facile l'utilisation des méthodes formelles pour l'analyse des programmes, le design et la vérification.
- Un réseau d'operateurs fournit directement une *représentation graphique* des programmes. En outre, cette représentation facilite la décomposition hiérarchique : un sous-noeud (ou sous-réseau) peut être encapsulé dans un opérateur. L'existence d'un formalisme textuel (*formalisme equationnel*) équivalent au formalisme graphique permet de combiner les avantages des deux approches. Alors que la description graphique est très utile un niveau macroscopique (haut degré d'abstraction) il devient rapidement très complexe lorsque l'on ajoute les détails du système.
- L'importance de l'existence d'implémentations logicielles a déjà été mentionnée. Un autre avantage d'une description en terme de réseaux d'opérateurs est qu'il mène simplement à de telles implémentations.

L'approche flot de données est par conséquent très utilisée dans le domaine de la programmation réactive. Cependant, la majorité des langages flots de données sont asynchrones. Une voie naturelle pour introduire la notion de temps dans des modèles flots de données est de lier le temps au taux d'arriver des données. Les variables considérées peuvent être interprétées comme des fonctions dépendant du temps. Par exemple, la description donnée en figure 4.1 exprime les relations suivantes :

$$\forall t, X(t) = 2Y(t) + Z(t) \text{ et } W(t) = X(t) + 1$$

Une telle interprétation temporelle des réseaux flots de données nécessite quelques restrictions sémantiques. Le temps de réaction maximal d'un programme doit être mesurable, ce qui interdit entre autres la création dynamique de processus (ce qui est permis dans les réseaux de Kahn). Plus généralement, un réseau à flots de données synchrones doit être implémentable par *un automate fini possédant une mémoire bornée.* En Lustre, ces restrictions consistent en des *contraintes d'horloges.*

4.2 Aspects fondamentaux du Langage Lustre

Lustre est un langage textuel à flot de données conçu à l'Institut d'Informatique et Mathématiques Appliquées de Grenoble (IMAG).

4.2.1 Lustre : langage synchrone

Lustre est fondé sur l'hypothèse de synchronisme qui considère qu'un système réagit instantanément tout stimuli ou sollicitation issus de son environnement.

Définition 4.2.1.1 *L'hypothèse du synchronisme fort revient considérer que :*

1. *les temps d'exécution et de communication sont nuls ;*
2. *les actions d'un système de processus sont instantannées ;*
3. *les sorties sont synchrones avec les entrées et se déroulent aux instants où elles ont sollicitées ;*
4. *le système est inactif entre deux sollicitations ;*

Les propriétés précédentes font que l'indéterminisme lié à l'entrelacement des actions possibles des différents processus est levé et que les actions peuvent s'exécuter simultanément. Lustre est un langage linéaire et permet le vrai parallélisme (par opposition au parallisme d'entrelacement) (cf. section 2.1.1.3).

Rappelons que l'hypothèse de synchronisme nécessite que le temps de réaction du système (à travers l'exécution des processus) soit plus petit que l'intervalle de temps qui sépare l'envoi de deux sollicitations ou stimuli.

4.2.2 Lustre : Langage flots de données

L'approche dite flots de données permet de représenter les programmes à un haut niveau d'abstraction. Cette approche consiste représenter un programme par un graphe où les noeuds sont des opérateurs qui transforment des flots de données. Cette approche possède plusieurs avantages :

- c'est une **approche fonctionnelle** qui possède toutes les propriétés mathématiques associées et en particulier l'absence d'effets de bord, et donc l'absence d'ordre dans l'écriture des instructions. Cela rend l'approche flots de données adaptée à la vérification et à la transformation de programme comme nous l'avons évoqué en introduction.
- c'est une approche à **parallélisme vrai** où les contraintes de séquencement et de synchronisation ne proviennent que des flots de données, i.e ce sont les flots de données qui déterminent les séquences d'exécution et de synchronisation. Cela facilite la synthèse de programmes parallèles.

Concrètement : un flot de données X est une suite de valeurs X_n avec $n \geq 0$. X_i est la valeur du flot X l'instant i. Tous les flots sont rythmés ou cadencés par la même horloge (sauf si ceux-ci possèdent une autre horloge : dans ce cas cette horloge est nommée et construite à partir de l'horloge de base). Ainsi deux flots X et Y ont les valeurs X_i et Y_i l'instant i.

Un flot de données est *défini par une une équation* $O = F(I, J, ...,)$ qui indique que $O_n = F(I_i, J_i, ...,)$.

Un programme Lustre sera un ensemble d'équations qui représentent un processus. A chaque top d'horloge, le processus calcule instantanément les flots de sortie en fonction des flots d'entrée.

4.2.3 Horloges et Flots

Définition 4.2.3.1 *En Lustre, chaque variable exprime un flot, c'est dire une paire composée :*

1. *d'une séquence, qui peut être infinie, de valeurs d'un type donné ;*
2. *et d'une horloge repésentant une séquence du temps.*

Un flot prend la *n-ième* valeur de sa séquence à l'instant n. Des flots ralentis peuvent être définis grâce des flots à valeurs booléennes. Un flot défini par un flot à valeurs booléennes (ce flot de booléens est alors l'horloge du flot) prend sa valeur lorsque le flot de booléen correspondant possède une valeur *true*.

A titre d'exemple, la figure suivante montre un flot C défini sur l'*horloge de base* (ide. l'horloge défini par un flot de booléens dont les valeurs sont toujours *true*), et un flot C' défini à partir des valeurs de C.

basic_time	1	2	3	4	5	6	7
C	True	False	True	True	False	True	False
C basic_time	1		2	3		4	
C'	False		True	False		True	
Z = current (Y)			1			2	

Il faut noter que la notion d'horloge n'est pas nécessairement reliée au temps physique (ou temps réel). En effet, des horloges logiques peuvent être définies par des flots de booléens.

4.2.4 Variables, Expressions et Assertions

Les variables et les expressions sont combinées pour décrire des équations. Les assertions permettent d'exprimer des propriétés logiques.

4.2.4.1 Types

Les types de base du langage LUSTRE sont : les booléens (`bool`), les entiers (`int`) et les réels (`real`). Il existe un seul constructeur de types : le constructeur `tuple` qui permet de décrire des n-uplets. Des types complexes peuvent être importés à partir d'autres langages hôtes. Ils sont souvent représentés par des types abstraits de données dans ces langages hôtes.

4.2.4.2 Variables

Les variables sont déclarées et typées. Les variables qui n'apparaissent pas dans les entrées doivent être définies une et une seule fois dans le programme.

4.2.4.3 Equations

Les équations permettent de définir les variables et de les relier. Une équation $x = E$ définie la variable x comme étant l'expression E. Dans ce cas, x et E ont la même valeur de flot (valeurs et horloge). Une équation de la forme :

$$X = Y + Z \ , \ Z = U$$

signifie que pour tout instant n, $X_n = Y_n + Z_n$ et $Z_n = U_n$.
Cela nous conduit au *principe de substitution*. Lorsqu'une équation $x = E$ apparaît dans un programme, alors x peut être substitué par E partout où x apparaît et réciproquement. En conséquence, les équations peuvent être décrites dans n'importe quel ordre, et des variables intermédiaires peuvent être créées pour capturer des sous-expressions, sans changer le sens du programme. Par exemple, les trois équations suivantes sont equivalentes :

$$\begin{array}{ccccc} X = Y + Z & & \text{X=Y+U} & & \text{Z=U} \\ & \Longleftrightarrow & & \Longleftrightarrow & \\ \text{Z=U} & & \text{Z=U} & & \text{X=Y+Z} \end{array}$$

4.2.4.4 Expressions

Les constantes du langage sont les constantes classiques associées aux types définis. Des constantes associées aux types abstraits importés peuvent également être définies.

Les constantes et les variables sont combinées à l'aide d'opérateurs pour former des expressions. Deux types d'opérateurs sont présents dans Lustre.

a- Les opérateurs de données. Les opérateurs de données Lustre sont :
- Les opérateurs arithmétiques : `+`, `-`, `*`, `/`, `div`, `mod`
- les opérateurs booléens : `and`, or ?, not ?
- les opérateurs de comparaison : `=`, `>`, `<`, $\leq$, $\geq$
- la conditionnelle : `if then else`.

Les opérateurs de données agissent sur les opérandes qui partagent la *même horloges*. Ils opèrent sur les valeurs des flots aux mêmes instants d'horloge.

Pa exemple si $X = (x_1, x_2, x_3, ..., x_n, ...)$ et $Y = (y_1, y_2, y_3, ..., y_n, ...)$ alors, l'instant n, l'expression $if\ X > 0\ then\ Y + 1\ else\ 0$ est équivalente $if\ x_n > 0\ then\ y_n + 1\ else\ 0$.

b- Les opérateurs temporels. En plus des opérateurs de données, des opérateurs temporels sont définis. Ils agissent directement sur les flots. Ces opérateurs sont au nombre de quatre :`pre`, `->`, `when` et`current`.

1. L'opérateur `pre` pour *previous* permet la mémorisation. Si $(e_1, e_2, e_3, ..., e_n, ...)$ est la séquence de valeurs de l'expression E, alors $pre(E)$ possède la même horloge que E et a pour séquence de valeurs $(nil, e_1, e_2, e_3, ..., e_n, ...)$. *nil* représente la valeur indéfinie.
2. L'opérateur `->` pour *followed by* détermine le suivant dans une séquence. Si E et F sont des expressions avec la même horloge, et $(e_1, e_2, e_3, ..., e_n, ...)$ et $(f_1, f_2, f_3, ..., f_n, ...)$ sont les deux séquences de valeurs des flots E et F alors le flot E`->`F pour séquence de valeurs $(e_1, f_2, f_3, ..., f_n, ...)$. En d'autres termes, E`->`F est toujours égale à F excepté pour le premier top d'horloge où il vaut E.
3. L'opérateur `when` permet de ralentir une horloge. Si E est une expression et B une expression booléenne avec la même horloge, alors E `when` B est une expression dont l'horloge est définie par B et dont la séquence de valeurs est extraite de celle de E en ne conservant que les valeurs qui correspondent à une valeur vraie de B i.e les séquences de valeurs lorsque B est vraie.
4. L'opérateur `current` permet d'accélérer une horloge. Soit E une expression dont l'horloge n'est pas l'horloge de base (`basic_time`) et soit B l'expression qui dinit cette horloge. `current(`E`)` possède la même horloge C que B. A tout instant, la valeur de cette horloge C est la valeur de l'expression E au dernier top d'horloge où B était vraie.

Le tableau suivant montre un exemple des deux derniers opérateurs temporels.

basic_time	1	2	3	4	5	6	7
B1	False	True	False	True	False	False	True
B2	False	True	True	False	True	True	True
H1=B1 when B2		True		False			True
X	X_1	X_2	X_3	X_4	X_5	X_6	X_7
Y= X when B1		X_2		X_4			X_7
Y1= X when H1		X_2					X_7
Z = current (Y)	nil	X_2	X_2	X_4	X_4	X_4	X_7
Z1= current(Y1)		X_2	X_2		X_2	X_2	X_7
W= current(Z1)	nil	X_2	X_2	X_2	X_2	X_2	X_7

4.2.4.5 Assertions

En plus de contenir des équations, le corps des programmes Lustre contient des assertions. Les assertions généralisent les équations et consistent en des expressions booléennes qui sont toujours vraies. Pratiquement, les assertions expriment les hypothèses sur l'environnement ou sur le programme lui-même. Les assertions permettent d'optimiser la compilation et de vérifier des propriétés sous conditions. Elles simplifient la conception des programmes.

Une assertion est exprimée par le mot clef `assert`. Par exemple, si on sait que deux entrées, représentées par les variables booléennes x et y, n'apparaissent pas en même temps on écrira :

```
assert(not(x and y))
```

De la même façon, l'assertion :

```
assert(true -> not(x and pre(x)))
```

exprime que x n'apparaît jamais deux fois consécutivement dans une séquence.

4.3 Structure d'un programme Lustre

Il est bien connu que la structure de graphe avec des opérateurs sur les noeuds est utilisée pour représenter les langages à flots de données.

A l'image de ces langages, un programme Lustre est un ensemble de noeuds qui définissent des opérateurs sur des flots de données.

Un programme Lustre respecte la structure syntaxique suivante :

```
[déclaration de types et de fonctions externes]

node nom (déclaration des flots d'entrée)
          returns (déclaration des flots de sortie)

var [déclaration des flots locaux]

let

[assertions]

système d'équations définissant une
et une seule fois les flots locaux
et de sortie en fonction d'eux-même
 et des flots d'entrée.

tel

[autres noeuds]
```

Les crochets [expression] indiquent que la présence de "expression" est optionnelle.

Un noeud déclare ses flots d'entrée et de sortie par le mot clef **node**. La clause **var** permet de déclarer les variables qui correspondent des flots de données locaux au noeud. Enfin, **let** est la clause qui permet de définir les flots de sorties en utilisant les opérateurs de données et les opérateurs temporels.

4.4 Causalité en Lustre

Le problème de causalité apparaît en Lustre lors de définitions cycliques : une variable ne doit pas dépendre instantanément d'elle-même. Ainsi, le compilateur ne donnera pas de sens à la construction `X=3*X+1`. Une telle définition est similaire à un *deadlock*. Ces deadlocks sont détectés par une simple analyse statique des dépendences. Lustre interdit généralement les "*faux deadlocks*" tels que :

$$\begin{cases} \texttt{X=if C then Y else Z;} \\ \texttt{Y= if C then Z else X;} \end{cases}$$ puisqu'il s'agit ici d'un problème indécidable.

4.5 Exemple de programmes

Le langage Lustre permet l'écriture d'une grande variété de programmes. Les programmes suivant en sont des exemple simples.

4.5.1 Détection de fronts montants

Y vaut vrai lorsque X change de valeur depuis `false` à `true`.

```
node EDGE(X:bool) returns(Y:bool);
let
    Y = X -> (X and not(pre(X)));
tel
```

4.5.2 Calcul d'une intégrale

Nous proposons ici un exemple calculant le résultat d'une intégrale par la méthode des triangles.

$$\int f(x)dx \rightsquigarrow \sum_{i=1}^{n} \left(\frac{\left(f(x_i)+f(x_{i+1})\right)(x_{i+1}-x_i)}{2} \right)$$

Pour calculer le résultat de l'intégrale nous pouvons donc utiliser réursivement l'équation :

$$Y_{n+1} = Y_n + (F_n + F_{n+1}) \times \left(\frac{Step_{n-1}}{2}\right)$$

avec : F le flot réel des valeurs de la fonction $f(x)$, et $Step$ le flot des pas d'échantillonage.
Nous obtenons le noeud suivant :

```
node INTEGRALE(F, STEP : real) returns (Y:real)
let
  Y= 0. -> pre(Y) + (F + pre(F))*STEP / 2.0 ;
tel
```

4.5.3 Dection des fronts descendants

Cet exemple montre une composition, c'est dire un noeud appelé par un autre noeud.

```
node FALLING_EDGE(X:bool) returns(Y:bool);
let
    Y = EDGE(not(X));
tel
```

4.5.4 Compteur

Un compteur renvoie la somme des pas précédents tant que le flot `reset` n'a pas été réinitialisé.

```
node COUNTER(val_init, val_incr : int ; reset :bool) returns(n: int);
let
   n= val_init -> if reset then val_init else pre(n) + val_incr;
tel
```

Ce noeud peut être instancié dans une autre expression par :

```
even = COUNTER(0,2,FALSE);
modulo5=COUNTER(0,1,pre(modulo5)=4);
```

Ici les flots `even` et `modulo5` définissent sur l'horloge de base respectivement une séquence de nombres pairs et la séquence cyclique des nombres modulo 5. Ces deux flots montrent un exemple de composition et de récursivité dans l'utilisation des variables.

4.5.5 Chiens de garde : Watchdogs

Nous présentons ici trois versions d'un "watchdog". Un watchog est très utilisé en programmation temps réel pour réveiller des processus qui risqueraient de ne pas se déclencher, (suite à une erreur ou à un mauvais ordonnancement) et par conséquent qui risqueraient de ne pas respecter leur "deadlines". La première version (cf. figure 4.2) reçoit trois flots booléens en entrée : deux commandes `set` et `reset` pour lancer et arrêter le chien de garde et un flot `deadline`. Le chien de garde doit émettre une `alarm` à chaque fois que la `deadline` est `true` et que le chien de garde est activé (état "on"). A l'état initial, le chien de garde est dans l'état "off".

```
node WATCHDOGS1 (set, reset, deadline : bool) ·
    returns (alarm : bool);
var watchdog_is_on : bool;
let
  assert not(set and reset);

  alarm = deadline and watchdog_is_on
  watchdog_is_on = false -> if set then true
                            else if reset then false
                            else pre(watchdog_is_on);
tel
```

FIG. 4.2 – Première version du watchdog.

Considérons maintenant une seconde version (figure **??**)dans laquelle le chien de garde possèdent toujours en entrées les commandes `set` et `reset`, mais doit émettre une `alarm` quand `set` est true pendant un certain delai, délai compté en nombre de cycles de l'horloge de base.

```
node WATCHDOGS2 (set, reset : bool ; delay : int) ·
    returns (alarm : bool);
var remaining_delay : int; deadline : bool;
let
  alarm = watchdog_is_on and deadline
  deadline = (remaing_delay=0) and pre(remaining_delay)>0
  watchdog_is_on = false -> if set then true
                            else if reset then false
                            else pre(watchdog_is_on);
  remaining_delay = 0 -> if set then delay
                         else if pre(remaining_delay)>0
                         then pre(remaining_delay)-1
                         else pre(remaining_delay)
tel
```

FIG. 4.3 – Seconde version du watchdog.

Suppposons maintenant que nous souhaitons un chien de garde tel que le précédent, mais que le délai doit être compté suivant une autre unité de temps : un nombre d'occurence d'une évènement `time_unit`. Nous avons juste à faire appel au `WATCHDOG2` en utilisant une horloge bien choisie :

```
node WATCHDOGS3 (set, reset,time_unit : bool ; delay : int) ·
    returns (alarm : bool);
var clock : bool;
let
  alarm = current(WATCHDOG2((set,reset,delay) when clock);
 clock = true -> set or reset or time_unit;
tel
```

FIG. 4.4 – Troisième version du watchdog.

4.6 Séquencement

L'approche flots de données considère les exécutions comme instantannées. Ces exécutions se déroulent pendant un cycle d'horloge.

Il peut arriver que les opérateurs associés à un noeud aient une durée d'exécution supérieure à un cycle d'horloge. Dans ce cas notre hypothèse de synchronisme fort n'est plus vérifiée, et par conséquent notre modèle ne peut pas être considéré comme reflétant correctement le système réel. Une solution pour remédier à ce problème consiste décomposer l'opération d'un noeud et exécuter les résultats de la décomposition en séquence.

Le séquencement est donc l'opération qui permet d'exécuter les opérations d'un noeud en une séquence d'opérations issues de la décomposition.

4.6.1 L'opérateur condact

L'opérateur `condact` est l'opérateur d'activation conditionnelle. On écrit : `condact(B,F(X,...),Init)` où :

- `F` est un opérateur (un noeud) de signature $F : T \rightarrow T'$,
- `B` est un flot de booléens
- `Init` est un flot de type T'.

L'équation `Y=condact(B,F(X,Init))` définit un flot Y de type T' égal à :

- `Init` tant que `B` n'a jamais été vrai,
- `F(X)` quand `B` est vrai,
- la valeur qu'avait `Y` lors du dernier instant ou `B` ait vrai quand `B` est faux.

Formellement on obtient :

$$Y_n = \begin{cases} Init_n \text{ si } \forall i < n,\ B_i = false \\ F(X_n) \text{ si } B_n = true) \\ Y_{n-1} \text{ si } B_n = false \text{ et si } \exists i < n \text{ tel que } B_i = true \end{cases}$$

Autrement dit $F(X)$ n'est exécuté que lorsque `B` est vrai et la valeur de `Y` est misejour lorsque `F(X)` est exécuté. `condact` permet d'activer une opération lorsqu'une condition est vérifiée. Elle permet une forme de synchronisation entre différentes opérations.

4.6.2 Séquencement d'opérations

Si des fonctions apparaissant dans un noeud prennent un temps d'exécution trop grand par rapport au cycle d'horloge, il est possible d'alterner l'exécution des fonctions, une fois sur deux.

Considérons le noeud suivant (`F1` et `F2` sont deux fonctions ou noeuds préalablement définis) :

```
node P(X:int) returns (Z:int);

var Y : int;

let
    Y=F1(X) ;
    Z=F2(Y) ;
tel
```

Supposons que `F1` et `F2` aient un temps d'exécution supérieur au top de l'horloge associé au noeud `P`. Une solution serait d'exécuter les fonctions `F1` et `F2` en alternance. On écrit :

```
node P(X:int) returns (Z:int);

var Y : int; B : bool ;

let
    B=true -> not(pre(B))
    Y=condact(B,F1(X),0) ;
    Z=condact(not(B),F2(Y),0) ;
tel
```

Ici les fonctions `F1` et `F2` s'exécutent une fois sur deux grâce au flot de booléens défini par `B`. Il est possible de définir des séquencement plus compliqué avec d'autres cadences en utilisant la même technique.

4.7 Programme Lustre et Automates

4.8 Génération de code séquentiel : le compilateur LUSTRE

(Voir Hawlbachs)

Le compilateur Lustre génère un code purement séquentiel. Il est maintenant admis qu'un code séquentiel ne peut généralement pas être généré à partir d'un programme concurrent sous une forme modulaire : on ne peut pas séquentialiser un sous-programme concurrent indépendamment de son contexte d'utilisation. Un programme Lustre illustre ce problème. Considérons le noeud suivant :

```
node two_copies (a,b:int) returns (x,y : int);
let
  x=a ;
  y=b
tel
```

Il existe deux implémentation possible séquentiel possible de ce "programme" : soit " `x:=0 ; y:=b`" ou " `y:=b ; x:=a`". Le problème intervenant est que le choix entre ces deux codes possibles peux dépendre de la manière dont le noeud est appelé dans un autre noeud. Par exemple, considérons l'appel : `(x,y)=two_copies(a,x);` qui correspond à la Figure 4.5.

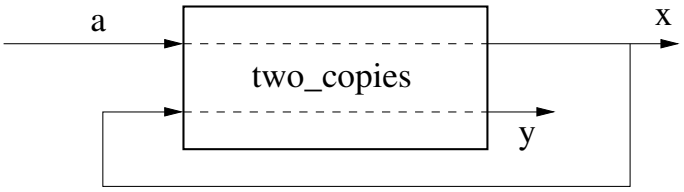


FIG. 4.5 – Un appel récursif.

Dans cette situation seule la première implémentation `x:=0 ; y:=b` est correcte. Ainsi, avant la génération de code, le compilateur doit d'abord déplier récursivement chaque noeud appelé dans le programme principal, i.e remplacer chaque noeud appelé par le corps du noeud, après avoir renommé les paramètres, les variables locales et les horloges. Ainsi, la génération de code commence à partir d'un code "aplatit", ou il n'existe plus aucun appel de noeud.

4.8.1 Loops

La manière la plus simple pour traduire un code LUSTRE en un code impératif est de construite une boucle infinie dont le corps réalise les cycles basiques du programme. Pour obtenir ce corps, nous devons tout d'abord choisir les variables du code généré (les variables de sorie et le plus petit nombre de variables locales qui implémentent soit les mémoires soit des buffers temporaires), pour construire les opérations qui mettent à jour les variables, et pour placer ces actions dans un ordre correct, suivant les dépendances entre les variables introduit par la structure en réseaux des noeuds. Donnons une illustration de cette technique : considérons la version "aplatit" du programme `WATCHDOG3` présenté à la section 4.5.5 :

```
node WATCHDOGS4 (set, reset,time_unit : bool ; delay : int) ·
    returns (alarm : bool);
let
  assert not(set and reset);

  alarm = watchdog_is_on and (remaining_delay = 0) and
                      pre(remaining_delay)>0;
  watchdog_is_on = false -> if set then true
                            else if reset then false
                            else pre(watchdog_is_on);
  remaining_delay = 0 ->  if set then delay else
                          else if time_unit and pre(remaining_delay)>0
                                  then  pre(remaining_delay)-1
                           else pre(remainig_delay);
tel;
```

La boucle infinie pourrait être la suivante :

```
 _init := true
while true do
  read(set,reset,time_unit, delay);
  if _init then   % first cycle%
     watchdog_is_on := false; remaining_delay := delay ;
     alarm := false; _init := false;
  else  % other cycles%
     if set then
        watchdog_is_on := true; remaining_delay := delay
     else
        if reset then watchdog_is_on := false endif;
        if time_unit and (_pre_remaining_delay>0) then
            remainig_delay := _pre_remaining_delay - 1;
        endif;
     endif;
     alarm := watchdog_is_on and (remainig_delay=0) and
                  (_pre_remaining_delay>0);
  endif
 write(alarm); _pre_remaining_delay := remaining_delay;
endwhile;
```

Remarques :

- Pour générer ce code, le compilateur a dû introduire des variables auxiliaires (celles dont les identifiants commencent par un caractère "underscore") : la variable `_init` - la valeur qui est vraie seulement au premier cycle de la boucle, et qui est utilisée pour implémenter l'opérateur "`->`" - et la variable `_pre_remaining_delay` - qui conserve la valeur précédente de `remaining_delay`. Il est à noter que l'expression "`pre(watchdog_is_on)`" ne nécessite pas l'utilisation d'une variable de mémoire auxiliaire puisque le compilateur a trouvé un moyen de l'éviter.
- Alors qu'il est relativement simple de trouver un ordre d'exécution qui est compatible avec les relations de dépendances entre les différentes variables (l'analyse statique de causalité assure qu'il existe un tel ordre), choisir un "bon" ordre est difficile. En particulier, l'ordre choisi au niveau des instructions conditionnelles est critique du point de vue de la longueur du code.
- La vitesse d'exécution du code généré pourrait être améliorée. La preuve d'inefficacité la plus évidente apparaît du fait que la variable `_init` est vérifiée à chaque cycle d'horloge. Une solution consiste à utiliser des structures de contrôle un peu plus complexes que la simple boucle infinie.

4.8.2 Compiler un programme Lustre sous forme d'automate

En utilisant certaines options, le compilateur LUSTRE est capable d'améliorer les performances du code en synthétisant plus ou moins certaines structures de contrôle. Cette opération de synthèse est basée sur les remarques suivantes :

1. Dans un langage déclaratif comme LUSTRE, les structures de contrôle, qui sont accessibles dans un langage impératif, sont remplacées par des opérations sur des expressions booléennes (conditionnelle, changement d'horloges...)
2. Evidemment, si une condition ou une horloge dépend des valeurs d'une variable booléenne calculée au cycle précédent - c'est-à-dire une expression de la forme `pre(B)` ou bien `current(B)` - le code du cycle courant peut être simplifié si cette valeur est connue. En d'autres termes, le code à exécuter au prochain cycle peut être selectionné suivant la valeur courante de `B`.

La synthèse des structures de contrôle consiste à choisir un ensemble de *variables d'état* (des expressions booléennes), et de simuler lors de la phase de compilation le comportement de ces variables. Il existe plusieurs choix possibles parmis les variables d'état :

1. les expressions booléennes retournées par les opérateurs `pre` et `current` ; et

2. les variables auxiliaire `_init_Ck` qui représente, pour chaque horloge `Ck` présente dans le programme, l'expression `(true when Ck) -> (false when Ck)` ; ces variables, dont la valeur indique le cycle courant est le premier pour l'horloge `Ck`, sont utilisées pour implémenter l'opérateur "`->`".

En partant de la configuration initiale des variables d'états, et pour chaque configuration atteinte pendant l'exécution, la simulation consiste à construire un code différent pour la suite du programme. Le résultat est un **automate à états fini**, pour lequel les transitions sont associées au code correspondant à la réaction du programme. Illustrons cette méthode sur l'exemple du `WATCHDOG4` : Nous choisissons `pre(watchdog_is_on)` et `_init` comme variables d'état.

1. Au premier cycle : "`pre(watchdog_is_on)=nil`" et "`_init=true`". Soit S_0 l'état initial. Puisque "`_init=true`" dans cet état, tous les opérateurs `->` sont évalués comme étant égal à leur première opérande. Ainsi, "`watchdog_is_on=false`" et "`remaining_delay=0`". Un calcul booléen élémentaire nous mène à "`alarm=false`". De plus, comme `watchdog_is_on` est évalué à `false`, `false` sera la valeur de `pre(watchdog_is_on)` au prochain cycle. L'état suivant S_1, correspond donc à "`pre(watchdog_is_on)=false`" et "`_init=false`". Le code correspondant à l'état S_0 ressemble à :

```
S0 : remaining_delay := 0;
     alarm := false ;
     _pre_remaining_delay := remaining_delay;
     goto S1;
```

2. Dans l'état S_1, `pre(watchdog_is_on)` est `false` et `watchdog_is_on` est évalué à `true` si et seulement si `set` est `true`. Soit S_2 l'état où `pre(watchdog_is_on)` est `true` et `_init` est `false`. Le code pour S_1 est alors :

```
S1 : if set then
        remaining_delay := delay;
        alarm := (remaining_delay = 0) and
                    (_pre_remaining_delay > 0) ;
        _pre_remaining_delay := remaining_delay;
        goto S2;
     else
        remaining_delay :=
           if time_unit and _pre_remaining_delay > 0
           then  _pre_remaining_delay - 1
           else  _pre_remaining_delay ;
           alarm := false;
           _pre_remaining_delay  := remaining_delay;
           goto S1;
     endif;
```

3. Le code de l'état S_2, où `pre(watchdog_is_on)` est considéré à `true` et `_init` à `false` est comme suivant :

```
S1 : if set then
        remaining_delay := delay;
        alarm := (remaining_delay = 0) and
                    (_pre_remaining_delay > 0) ;
        _pre_remaining_delay := remaining_delay;
        goto S2;
     else
        remaining_delay :=
           if time_unit and _pre_remaining_delay > 0
           then  _pre_remaining_delay - 1
           else  _pre_remaining_delay ;
           if reset then
```

```
            alarm := false;
            _pre_remaining_delay  := remaining_delay;
            goto S1;
        else
            alarm := (remaining_delay = 0) and
                 (_pre_remaining_delay > 0) ;
            _pre_remaining_delay  := remaining_delay;
            goto S2;
        endif;
    endif;
```

A ce stade tous les états ont été traité. La génération de code est donc terminée. La figure 4.6 représente l'automate produit.

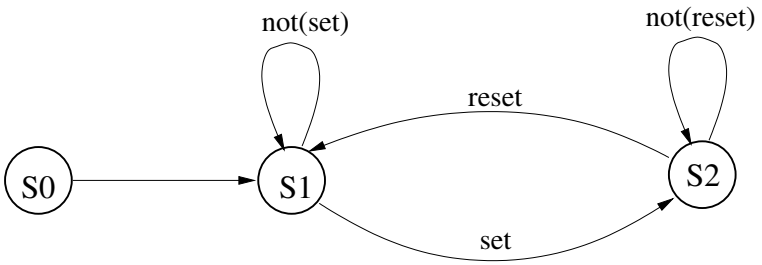


FIG. 4.6 – Automate représentant l'exécution du watchdog.

4.8.3 The OC code and associated tools

Les automates générés par le compilateur LUSTRE sont encodés dans un format intermediaire appelés OC (pour "*Object code*") [PS87] comme nous le montre la figure 4.7. Ce format est partagé par plusieurs compilateurs, notamment le compilateur des langagse synchrones ESTEREL et SIGNAL.

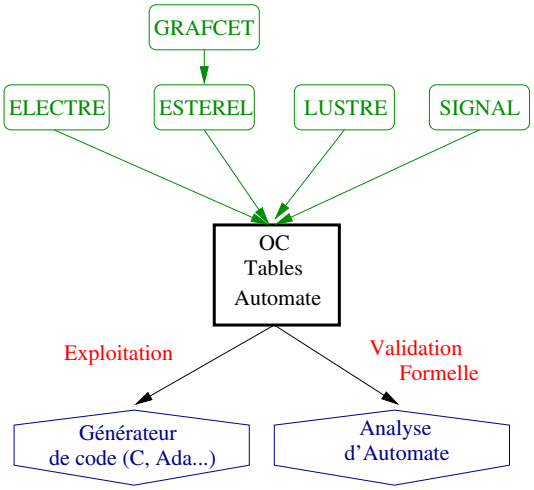


FIG. 4.7 – Outils utilisés avec LUSTRE.

Générations de code. Des traducteurs en langages de haut niveau (ADA, C,...) sont disponibles. Ils génèrent une procédure dont l'appel réalise la réaction encoder par l'automate. Pour activer cette procédure, il faut écrire un programme principal qui récupère les entrées physiques du programme et qui écrit les sorties :

– Pour chaque signal d'entrée `X`, le générateur de code fournit une procédure `I_X`, qui doit être utilisée - avec la valeur du signal en paramètre - pour signaler la présence de `X` à l'automate.

– Pour chaque sortie `Y`, une procédure `O_Y` est fournie qui est appelée par l'automate quand le signal `Y` est émis.

La structure du programme principal doit être la suivante :

```
Initializations
Infinite loop
     Input handling
     Call of the selected I_... procedures
     Call of the automaton
       (which call some O_... procedure by itself)
end loop;
```

ce qui peut être représenter par le schéma présenté en figure 4.8.

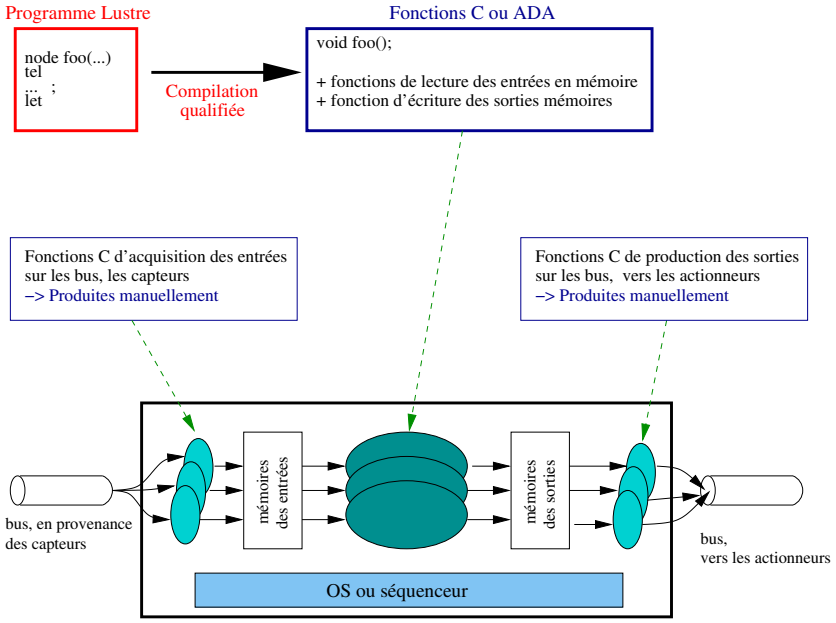


FIG. 4.8 – Utilisation de code généré par le compilateur LUSTRE dans un programme principal.

Minimisation de l'automate. Il existe également des outils permettant de minimiser l'automate.

4.9 Vérification de programme Lustre : l'outil Lesar

Comme nous en avons déjà amplement parlé, les systèmes réactifs concernent souvent des applications "critiques", et par conséquent il est souvent nécessaire de vérifier ces programmes. Cependant, nombreux sont les développeurs de systèmes critiques qui sont sceptiques concernant l'utilisation des méthodes formelles. Il convient de donner des arguments convaincants pour supporter le point de vue suivant lequel ces méthodes sont d'un intérêt particulier.

La recherche sur la vérification de programmes, qui a debuté au début des années 1970, avait pour objectif de fournir des preuves complètes du bon fonctionnement de programmes généraux et par conséquent très complexes. Ces travaux ont ammené de très importantes contributions concernant les techniques de programmation et la conception de langage mais nous devons bien dire que ces techniques sont peu utilisées en pratique.

Cependant, le but concernant la programmation des systèmes réactifs peut être moins ambitieux. La plupart du temps, *la sûreté d'applications critiques ne dépend pas de la correction totale de son programme* de

contrôle, mais plutôt d'un petit ensemble de propriétés que le programme doit satisfaire. Par exemple, l'occurrence d'une situation critique devrait déclencher une alarme en respectant un certain délai. La preuve de ce type de propriétés peut la plupart du temps être prise en charge en utilisant des théories décidables simples puisque ces propriétés sont rarement dépendantes des relations numériques et des traitements effectués par le programme.

De plus, la plupart des propriétés que nous souhaitons vérifier sont des "**propriétés de sûreté**" (*Safety Properties* en anglais), qui stipulent qu'une situation donnée (un état critique) ne doit jamais être observé durant l'exécution du programme, ou bien qu'une propriété doit toujours être satisfaite (invariant du système). A contrario les "**propriétés de vivacité**" (*Liveness Properties* en anglais) qui stipulent qu'une situation donné pourra éventuellement avoir lieu dans le future. Par exemple, une question relevante n'est pas qu'un train s'arrêtera éventuellement, mais plutôt qu'il ne franchira jamais un feu rouge. C'est un point très important car les techniques de preuves pour les propriétés de sûreté sont connus comme étant beaucoup plus simple que celles pour les propriétés de vivacité :

1. *Une propriété de sûreté peut être vérifiée sur une abstraction du programme concerné.* Informellement, si une propriété de sûreté est satisfaite pour un programme, elle sera aussi satisfaite pour des programmes dont les comportements sont des sous-ensembles du comportement du programme initial. Ainsi, il est possible d'abstraire des programmes en ignorant certains détails, par exemple les calculs numériques ;
2. *Une propriété de sûreté peut être vérifiée en vérifiant simplement les propriétés des états atteignables.* Ceci permet d'utiliser des méthodes très efficace basée sur l'atteignabilité [Hol87] ;
3. *Les propriétés de sûreté peuvent être vérifiées modulairement.* Les propriétés des sous-modules peuvent être combinées pour dériver une propriété du module complet. Ceci permet de réduire la complexité de la preuve, grâce à la décomposition modulaire, décomposition suivant l'architecture du programme.

Nous proposons ici des méthodes pour spécifier et vérifier des propriétés sur un programme LUSTRE.

4.9.1 Spécification des propriétés de sûreté

Il existe une multitude de formalismes qui ont été proposés afin d'exprimer les propriétés de systèmes temps-réel. On peut distinguer deux approches : celles basées sur les logiques temporelles (cf. section 2.2) et celles basées sur la théorie des automates (Petri nets, Statecharts, timed graphs and process calculi). Par sa nature declarative, LUSTRE est un bon langage pour exprimer des propriétés de programme lUSTRE [HPOG90, RHR91]. Cette affiramation est supportée par les arguments suivants :

- LUSTRE peut être considéré comme un sous-ensemble d'une logique temporelle [PH88]. Le propos est alors d'exprimer une propriété de sûreté P par une expression booléenne B, de telle manière que P soit satisfaite si et seulement si B est `true` pendant n'importe quelle exécution du programme. Toutes les propriétés de sûreté peuvent être exprimées de cette manière.
- De plus le mécanisme d'assertions proposé par LUSTRE permet d'exprimer des propriétés sur l'environnement du programme.
- L'utilisation d'un unique langage de programmation pour exprimer à la fois le programme et ses propriétés est intéressant puisque toute les facilités offertes par le langage sont accessibles rendant la spécification de propriétés plus lisible et plus expressive. Par exemple nous verrons qu'il possible de définir ses propres opérateurs temporels.

Voyons comment il est possible d'exprimer des opérateurs temporels non triviaux en utilisant des noeuds LUSTRE. Considérons la propriétés suivante :

"Chaque occurence d'une situation critique doit être suivie par une alarme avant un délai de 5 secondes."

Une telle propriété fait référence à trois évènements : l'occurence d'une situation critique, l'alarme et la deadline. Cette propriété peux facilement être exprimée en LUSTRE. Un pattern général de cette propriété est le suivant :

"Chaque occurence d'un évènement A est suivi par l'occurence d'un évènement B avant la prochaine occurence de l'évènement C."

Cependant, cette formulation n'est pas directement traduisible en LUSTRE puisqu'elle fait référence à quelque chose qui se passe dans le future. LUSTRE permet seulement des références au passé vis à vis de l'instant courant. Nous devons donc traduire la propriété précédente dans une forme équivalente faisant explicitement référence au passé :

"A chaque occurence de C, soit A ne s'est jamais produit, soit B s'est produit depuis la dernière occurence de A."

Définissons un noeud, prenant en entrée trois flots booléens `A`, `B` et `C`, et retournant en sortie un booléen `X` tel que `X` est toujours `true` si et seulement si la propriété est vérifiée :

```
node onceBfromAtoC(A,B,C :bool) returns (X:bool);
let
  X=implies(C, never(A) or since(B,A));
tel
```

L'équation définissant X utilise la définition de trois noeuds auxiliaires :

1. Le noeuds `implies` implémente l'implication logique ordinaire :

```
node implies(A,B:bool) returns (AimpliesB : bool);
let
AimpliesB = not A or B;
tel
```

2. Le noeud `never` retourne `true` aussi longtemps que son entrée n'a jamais été égale à `true`. Si l'entrée a été égale à `true` alors la sortie restera `false` pour toujours.

```
node never(B:bool) returns (neverB : bool);
let
 neverB=(not B) -> (not B and pre(neverB));
tel
```

3. Enfin, le noeud `since` à deux entrées, et retourne `true` si et seulement si : (1) soit la seconde entrée n'a pas encore été `true`, (2) ou bien la première entrée a été `true` au moins une fois depuis la dernière valeur `true` de la seconde entrée :

```
node since(X,Y : bool) returns (XsinceY : bool);
let
 XsinceY= if Y then X else (true -> X or pre(XsinceY))
tel
```

4.9.2 Verification

La méthode de vérification est très similaire au model-checking : tout d'abord, le graphe d'état du programme est construit, et ensuite chaque propriété est vérifiée sur ce graphe d'état. Le problème de cette approche, comme nous l'avons déja évoqué dans le chapitre concernant le model-checking, est le nombre d'état, qui peut être très important dans le cas d'un programme réaliste. Nous allons voir que la restriction aux propriétés de sûreté, et l'expression de ces propriétés dans le même langage que le programme à vérifier peut aidé à résoudre ce problème.

Dans le cas de LUSTRE, le graphe d'état existe déjà et correspond à l'automate construit par le compilateur. Ce graphe est une abstraction du graphe à base d'état actuel puisqu'il exprime uniquement les variables de contrôle et ignore les détails concernant les variables non-booléennes et booléennes qui n'influence pas le contrôle de l'application.

Une importante observation pour diminuer la taille du graphe consiste à prendre en compte la propriété à vérifiée lors de la construction de l'automate. Dans le cas de LUSTRE ceci peut se faire facilement puisque nous utilisons le même langage pour les propriétés et le programme : afin de prouver qu'une expression Phi est un invariant du programme Pg, nous construisons un nouveau programme Pg_phi composé du corps de Pg et du système d'équations définissant Phi, dont la seule sortie est est un flot booléen B (voir la figure **??**).Vérifié la propriété revient alors à vérifier qu'aucun des états de l'automate résultant de la compilation du noeud Pg_phi ne réalise l'affectation à `false` de la sortie booléenne B.

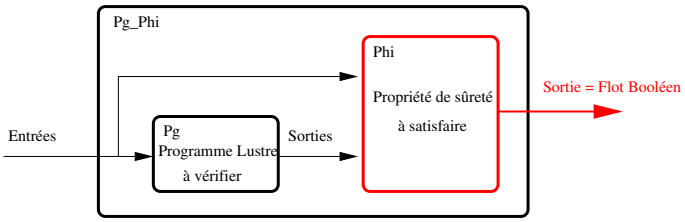


FIG. 4.9 – Vérification d'un noeud lustre Pg : notion d'observateur.

Pour réduire la taille de l'automate obtenu nous pouvons utiliser le mécanisme d'assertions pour exprimer des hypothèses sous lesquelles la propriété doit être vérifiée. Ces assertions sont alors un moyen très utile pour exprimer des propriétés sur les nombres qui seraient autrement ignoré par le compilateur. Par exemple, si un programme utilise des tests numériques tels que `X<=Z` et `Y<=Z`, l'assertion :

```
assert implies(X<=Y and Y<=Z, X<=Z);
```

force le compilateur à ne pas générer les états satisfaisant `Z<X<=Y<=Z`, état qui bien sûre ne serait pas atteignable par le programme concerné.

Comme example, considérons le noeud suivant qui représente un interrupteur : la valeur de sa sortie alterne de `true` à `false` suivant la valeurs de ses flots d'entrée `ON` et `OFF`; une troisième entrée définie la valeur initiale de l'interrupteur. Une première version de ce noeud pourrait être :

```
node SWITCH_1(ON, OFF, INIT : bool) returns (STATE : bool);
let
  STATE= INIT -> if ON then true
                 else if OFF then false
                 else pre(STATE);
tel.
```

Cependant, cette version a une faille. Considérons l'appel suivant :

```
state= SWITCH_1(button, button, init)
```

Dans cet appel la sortie ne change pas à chaque fois que le bouton est pressé comme nous l'aurions attendu. Ainsi, une version plus générale doit prendre cet aspect en considération en prenant en compte l'état précédent du système :

```
node SWITCH_2(ON, OFF, INIT : bool) returns (STATE : bool);
let
  STATE= INIT -> if ON and not pre(STATE) then true
                 else if OFF and pre(STATE) then false
                 else pre(STATE);
tel.
```

Nous pourrions souhaiter vérifier que cette généralisation est correcte, dans le sens où les deux versions se comporte de la même manière tant que les entrées `ON` et `OFF` ne sont jamais `true` en même temps. Ceci

peut être effectué en construisant un noeud de comparaison qui appelle les deux noeuds précédents avec les mêmes entrées et et en comparant leurs sorties, sous l'hypothèse que les entrées `ON` et `OFF` sont exclusives (cf. figure 4.10) :

```
node COMPARE(ON, OFF, INIT : bool) returns (OK : bool);
var state, state_1 : bool;
let
  state = SWITCH(ON, OFF, INIT);
  state_1 = SWITCH_1(ON, OFF, INIT);
  OK = (state = state_1);

  assert not(ON and OFF);
tel.
```

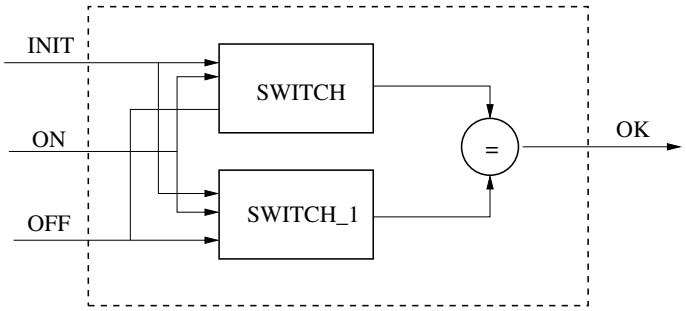


FIG. 4.10 – Equivalence sous hypothèses de programmes.

La compilation de ce noeud produit un automate possédant 5 états, chaque transition affectant la valeur `true` à la sortie `OK`.

Le dernier moyen pour contrer le problème d'explosion du nombre d'état est la *vérification modulaire*. Si nous avons à prouver qu'une expression `B` est toujours vraie pendant l'exécution d'un programme `P` faisant appel au noeud `Q` (cf. figure 4.11(a)), l'idée est de décomposer la preuve deux sous-preuves : l'une concernant `Q` et l'autre concernant `P` sans `Q` :

- Trouvons (par intuition) une propriété de `Q`, i.e une expression `C` sur les paramètres d'entrées/sorties de `Q`, et prouvons que `C` est toujours vraie pendant l'exécution de de `Q`.
- maintenant, considérons `Q` comme faisant partie de l'environnement de `P`, i.e, replaçons dans `P` l'appel du noeud `Q` par une assertion `assert C`. Alors, nous prouvons l'invariance de `B` sur le programme modifié. (cf. figure 4.11(b))

FIG. 4.11 – Vérification modulaire.

Un outil de vérification prototype appelé LESAR a éta implémenté : étant donné un programme possédant une seule sortie booléenne, LESAR vérifie que les états du système n'assignent jamais la sortie à `false`. Si une

telle situation se produit, un diagnostic est fournit par le model-checker (un exemple d'exécution ammenant la sortie à la valeur `false` est exhibé). Sinon, LESAR conclue que la propriété est satisfaite. En fait, "deux outils" de vérification sont disponibles :

- Le premier outil enumère explicitement l'ensemble des états atteignables, comme c'est le cas pour les models-checkers standards. La limitation de cette approche est bien entendu le nombre d'états considéré. La version actuelle de l'outil est capable de gérer des programmes d'environ 1 million d'états en un temps raisonnable (moins d'une heure de calcul).
- Le deuxième outil procède de façon symbolique : partant d'une formule booléenne F_0, caractérisant l'ensemble des état où la sortie est `true`, il calcul itérativement la séquence $F_0, F_1, \ldots, F_n$ des formules où F_{i+1} caractérise l'ensemble des états, appartenant à F_i et menant nécessairement (en un pas d'exécution) à F_i. Dès que l'état initial ne satisfait pas F_i, nous pouvons alors conclure qu'il existe un chemin menant à un état où la sortie prend la valeur `false`. Sinon, puisque l'ensemble des états est fini, la séquence des formules converge après un nombre fini de pas. Cet outil réalise des calculs symboliques sur les formules en utilisant des *diagrammes de décision binaire*. Cette approche est parfois appelée *"model-checking symbolique"*.

Les deux approches sont complémentaires : dans certains cas, la méthode énumérative est plus efficace que la symbolique, et vice-versa.

Bien entendu, la validité de la preuve tient en la satisfaction de l'hypothèse synchrone : toutes les preuves sont éffectuée au sein du modèle synchrone, et n'a rien à voir avec une analyse de performance. Comme nous l'avons déjà mentionné, vérifier la validité de l'hypothèse synchrone revient à évaluer le temps maximum de réaction du programme sur une plateforme logicielle donnée.

4.10 Scade : un outil industriel pour Lustre

4.10.1 Historique de Scade

1986 : Merlin Gerin maintenant Schneider Electric utilise les concepts de Lustre pour développer un outil appelé Saga afin de spécifier le logiciel de monitoring et contrôle pour une centrale nucléaire. Le projet fut un succès et Merlin Gerin décida de créer une start-up appelée Verilog afin de commercialiser Saga .

Les ingénieurs de Verilog ont réalisé que l'aérospatiale (maintenant Airbus) avait développé indépendament de son un outil similaire appelé SAO . Cet outil fût utilisé pour spécifier le logiciel des commandes de vol de l'airbus A320.

Avec ces deux technologies en main, Verilog décida de créer, en partenariat avec Merlin Gerin et l'Aerospatiale, un produit intégrant Saga et SAO .Ce nouveau produit s'apelle Scade .

1999 : Verilog est racheté par Telelogic et Scade prolonge son succès dans le domaine de la spécification de logiciels critiques.

2001 : Telelogic est racheté par Esterel Technologies.

2006 : Scade Suite^{TM} est une référence dans la spécification de logiciel embarqué critique devant satisfaire les normes de sécurité avioniques DO-178B et ED-12B. Scade Suite^{TM} est un concentré d'outils comportant :

- un environnement de travail : Scade Editor,
- un simulateur : Scade Simulator,
- un outil de vérification formelle : Scade Prover,
- un générateur de documentation : Scade Reporter,
- un vérificateur de cohérence : Checker,
- un générateur de code C, ada, Kcg.

4.10.2 L'interface graphique de Scade Suite^{TM}

Aujourd'hui, scade est urtilisé pour spécifier les commandes de vol de l'A380 et A400M, les derniers avions d'Airbus.

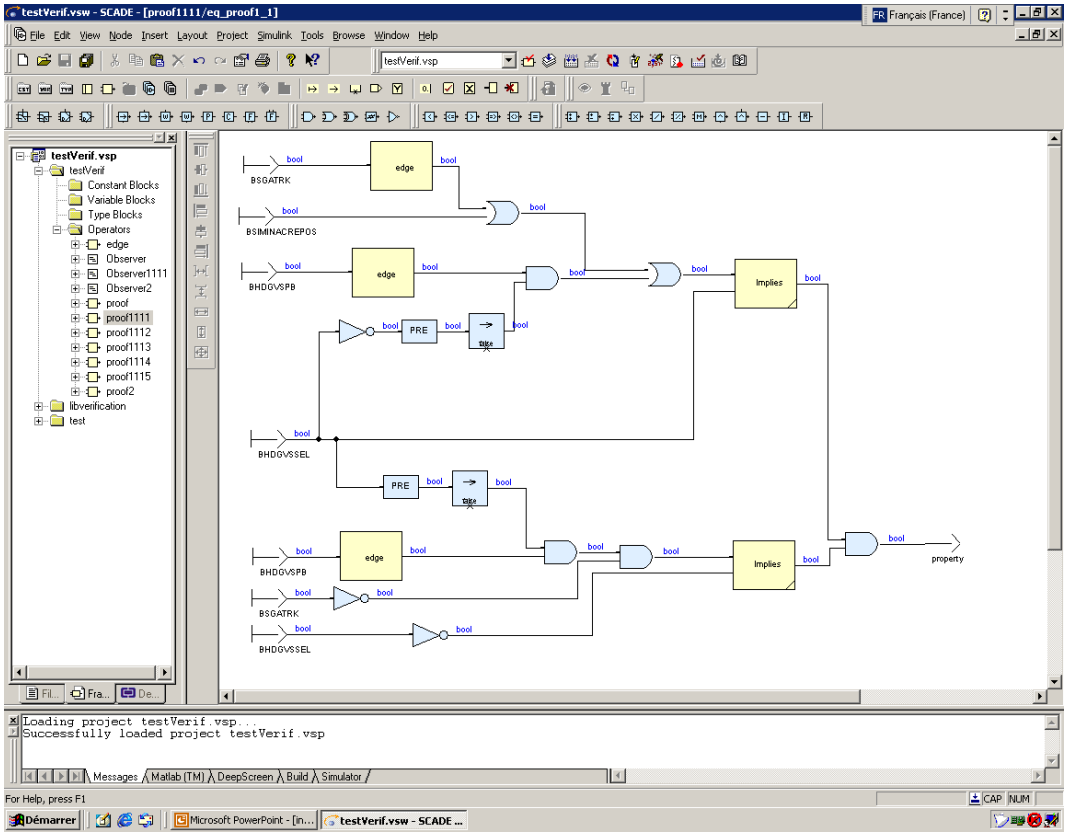


FIG. 4.12 – Interface de développement de Scade Suite^{TM}.

A partir de ces spécifications, du code C certifié est généré automatiquement. Le code généré est certifié DO 178B level A, c'est à dire qu'il ne comporte pas de bug du point de vu structurel. Cependant, les activités de tests restent toujours nécessaires pour vérifier le produit final d'un point de vue "fonctionnel". Les activités de preuves formelles pour la correction fonctionnelle du système sont à l'heure actuelle à l'état de recherche chez Airbus. Par exemple, certains travaux s'intéressent à la génération de séquences de tests fonctionnels à partir des spécifications formelles du systèmes et des propriétés à vérifier. L'avantage d'une telle technique est de s'assurer que l'activité de test sera exhaustif.

La figure 4.12 présente l'interface graphique de Scade. Cette interface graphique permet la représentation de noeuds LUSTRE sous la forme d'un réseau d'opérateurs. De nombreux opérateurs prédéfinis permettent de spécifier rapidement un système. Tout comme LUSTRE, il est possible de définir de nouveaux opérateurs par la suite utilisables dans les futurs développement.

L'environnement de vérification de Scade est présenté en figure 4.13. A partir de la représentation graphique du système, un code LUSTRE est généré. Les principes du model-checking et de model-checking symbolique sont alors utilisés pour prouver la correction du système.

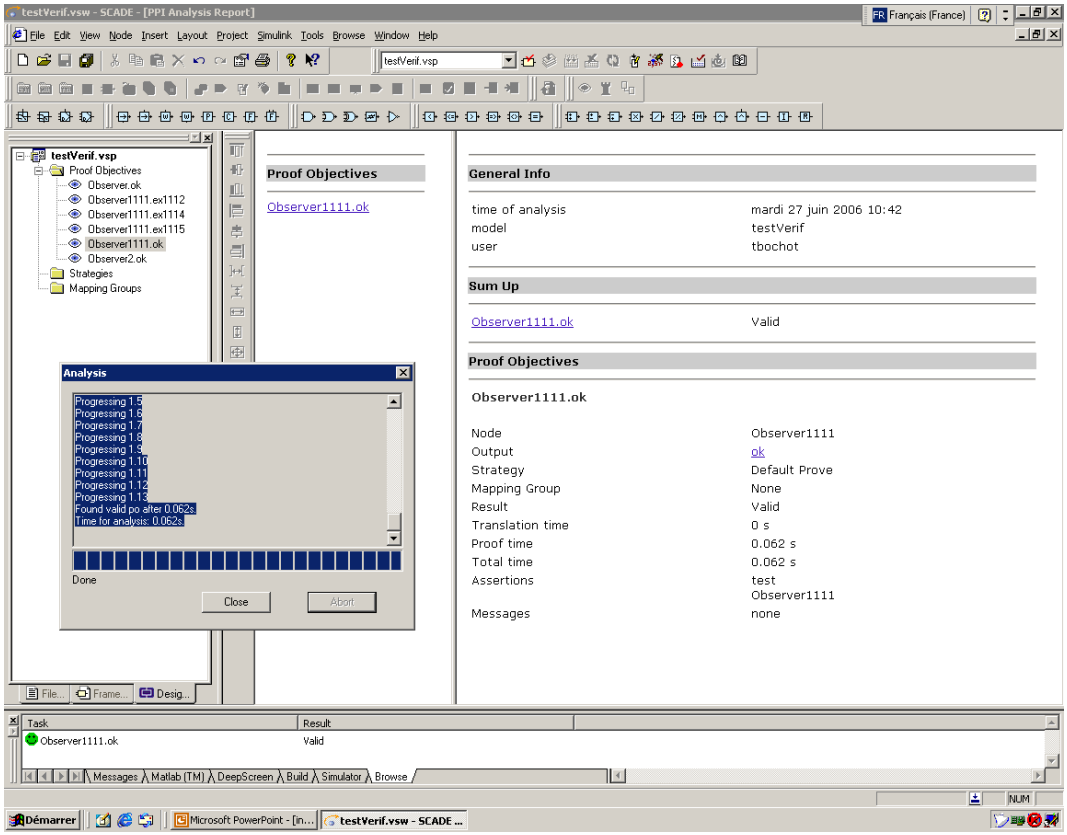


FIG. 4.13 – Interface du prover de Scade Suite^{TM}

Table des figures

Bibliographie

[Ber89] G. Berry. Real time programming : Special purpose or general purpose languages. In *IFIP World Computer Congress*, 1989.

[BG] G. Berry and G. Gonthier. The synchronous programming language esterel, design, semantics, implementation. Technical report, INRIA.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2) :244–263, 1986.

[CGL94] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5) :1512–1542, September 1994.

[EH86] E. Allen Emerson and Joseph Y. Halpern. "sometimes" and "not never" revisited : on branching versus linear time temporal logic. *J. ACM*, 33(1) :151–178, 1986.

[Hal93] Nicolas Halbwachs. *Synchronous Programming of Reactive Systems.* Kluwer Academic Publishers, 1993.

[Hol87] G.J. Holzmann. On limits and possibilities of automated protocol analysis. In H. Rudin and C. West, editors, *Proc. 6th Int. Conf on Protocol Specification, Testing, and Verification, INWG/IFIP*, Zurich, Sw., June 1987.

[HP85] D. Harel and A. Pnueli. On the development of reactive systems. *Logic and Models of Concurrent Systems, NATO Adavnced Study Institute on Logics and Models for Verification and Specification of Concurrent Systems.*, Springer Verlag,1985.

[HPOG90] N. Halbwachs, D. Pilaud, F. Ouabdesselam, and A-C. Glory. Specifying, programming and verifying real-time systems using a synchronous declarative language. In *Proceedings of the international workshop on Automatic verification methods for finite state systems*, pages 213–231, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

[NV90] Rocco De Nicola and Frits Vaandrager. Action versus state based logics for transition systems. In *Proceedings of the LITP spring school on theoretical computer science on Semantics of systems of concurrent processes*, pages 407–419, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

[PH88] Daniel Pilaud and Nicolas Halbwachs. From a synchronous declarative language to a temporal logic dealing with multiform time. In *Proceedings of a Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 99–110, New York, NY, USA, 1988. Springer-Verlag New York, Inc.

[Pia86] J. Piaget. *Logique et connaissance scientifique.* EncyclopÃ©die : La PlÃ©iade, 1986.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57, 1977.

[PS87] J.A Plaice and J.B Saint. The lustre-esterel portable format. *Unpublished report*, no volume(no number) :no pages, 1987.

[QS82] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in cesar. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, pages 337–351, London, UK, 1982. Springer-Verlag.

[RHR91] Christophe Ratel, Nicolas Halbwachs, and Pascal Raymond. Programming and verifying critical systems by means of the synchronous data-flow language lustre. In *SIGSOFT '91 : Proceedings of the conference on Software for citical systems*, pages 112–119, New York, NY, USA, 1991. ACM Press.

[SBB+99] Philippe Schnoebelen, Béatrice Bérard, Michel Bidoit, François Laroussinie, and Antoine Petit. *Vérification de logiciels : techniques et outils du model-checking*. Vuibert, April 1999.

[Ses02] Irina Sessitskaïa. *Apport des Techniques d'abstraction pour la vérification des Interfaces Homme Machine*. PhD thesis, Ecole Nationale Supérieure de l'Aéronautique et de l'Espace, 2002.

[Teu91] Bernd Teufel. *Organization of Programming Languages*. Springer-Verlag, Wien New York, 1991.