Universités Paris VI, VII, XI
École Polytechnique - ENST
ENS Ulm - ENS Cachan

DEA INFORMATIQUE : ALGORITHMIQUE

DU CALCUL DES DURÉES AUX AUTOMATES SYMBOLIQUES

Réalisé par
LAURE DANTHONY

Encadré par ***Nicolas HALBWACHS***
Date : Mars - Juin 2003

Table des matières

Introduction

Dans ce rapport de stage de DEA effectué de mars à juin 2003 au laboratoire Vérimag, à Grenoble, nous décrivons les travaux effectués dans le cadre de la vérification de systèmes, sous la direction de Nicolas Halbwachs. Ces travaux portent plus particulièrement sur la méthode de spécification par observateur, méthode que nous utilisons dans le contexte de systèmes décrits dans le langage LUSTRE, langage développé depuis 1984 au sein de l'équipe, et pour des propriétés décrites dans une logique temporelle particulière, QDDC. Nous mettons en œuvre une traduction de propriétés exprimées à l'aide de QDDC vers des automates LUSTRE.

Contexte de l'étude Dans les systèmes actuels (avions, fusées, mais aussi voitures, appareils ménagers), la part du contrôle logiciel est de plus en plus importante par rapport aux aspects mécaniques. De plus, de tels systèmes sont coûteux et une défaillance peut entraîner des coûts financiers voire humains. Il est important de pouvoir vérifier la sûreté aussi bien que la conformité aux spécifications de tels systèmes *critiques*. C'est l'objet des différentes techniques de vérification qui ont été développées ces vingt dernières années. Aujourd'hui, les problèmes de vérification sont de véritables enjeux économiques. Dans ce contexte, les méthodes formelles se sont vite imposées. Elles utilisent une formulation rigoureuse des besoins (spécifications) et une modélisation adéquate du fonctionnement du système. Cependant, la diversité des propriétés et des systèmes pose de gros problèmes d'expressivité car :

- soit les langages d'expression des propriétés et/ou de modélisation des systèmes sont peu expressifs, et on ne peut vérifier que des systèmes "simples" ;
- soit les langages sont suffisamment expressifs pour exprimer toute la complexité du système, et alors le problème devient indécidable.

Diverses méthodes La vérification des programmes est indécidable. Plusieurs propositions ont donc été faites pour remédier à ce problème :

- La *preuve (interactive) de programme*. Les méthodes de preuve sont en général très puissantes car elles peuvent éventuellement prouver des propriétés sur des systèmes ayant un nombre infini d'états, mais on rencontre rapidement leurs limites. De plus, de telles méthodes sont difficiles à mettre en œuvre car elles imposent souvent d'avoir une idée très précise de la preuve "papier", ce qui n'est pas évident.
- On peut aussi se restreindre à des systèmes d'états finis, on utilise alors les méthodes de model-checking (méthodes exactes), qui sont les méthodes les plus utilisées et les plus étudiées. De telles méthodes ont fait leurs preuves dans la vérification de gros systèmes réels, même si elles sont souvent soumises à une explosion combinatoire, ce qui limite leur champ d'action.
- Si les systèmes sont trop gros, on peut aussi utiliser des approximations : on peut approximer les systèmes "par le haut", *i.e.* , en utilisant des abstractions par exemple (pour se ramener à des systèmes d'états finis et utiliser les outils de model-checking existants). Signalons aussi

les méthodes d'analyse statique qui consistent en des approximations successives d'ensembles d'états accessibles, et qui permettent dans certains cas de conclure. Dans les deux cas, on obtient alors des algorithmes de vérification qui sont *conservatifs* (une réponse positive nous indique que la propriété est vraie, une réponse négative ne nous permet pas de conclure).
- Le *test* (méthode d'approximation par le bas) : le but du test contrairement aux autres méthodes est de détecter des mauvais comportements (debogage), et non de prouver qu'il n'y a jamais de mauvais comportements. Le test est une méthode complémentaire car les méthodes précédentes ne peuvent empêcher une mauvaise formulation des propriétés ou bien l'oubli de certaines spécifications du programme. Les jeux de test peuvent être écrits à la main ou bien fournis par un logiciel qui utilise des techniques similaires à celles du model-checking pour générer des séquences de test cohérentes. Le "model-checking borné" est aussi une méthode de recherche de bogues ([CBRZ01]).

Cadre de l'étude Nous nous plaçons dans le cadre de la vérification de formules temporelles. Les propriétés que nous désirons exprimer sont des propriétés relatives aux variations des valeurs des variables du programme dans le temps. De nombreuses logiques temporelles ont déjà été introduites (comme LTL, voir [Eme90], et CTL, voir [EH86]) ainsi que des procédures de décision pour les formules de ces logiques. Ces procédures traduisent en général la propriété à vérifier en un automate fini déterministe qui accepte *exactement tous* les modèles de la formule. Elles fournissent un résultat de décidabilité intéressant, et peuvent être utilisées dans le cadre du model-checking : dans le cas de LTL, on construit l'automate du complémentaire de la formule, on effectue l'intersection avec les traces système à vérifier, ainsi si le résultat est vide, on aura montré qu'aucune exécution du système ne viole la propriété. Cependant, ces procédures sont soumises à une explosion du nombre d'états de l'automate construit, et sont donc inutilisables en pratique dès que l'on veut traiter de gros systèmes. On utilise alors des abstractions des systèmes, et la vérification n'est alors que partielle, un résultat négatif ne permettant pas de conclure. Dès lors, rien n'interdit plus de considérer des systèmes d'états infinis, dont la vérification est théoriquement indécidable, puisque nous produisons désormais des algorithmes approchés (mais *conservatifs*) . Et en pratique, nombre de tels systèmes peuvent être vérifiés pour les propriétés qui nous intéressent. De plus, nous ne désirons pas construire explicitement les automates "vérificateurs", mais plutôt les construire sous forme "symbolique", *i.e.* sous la forme d'ensembles de variables dont on décrit l'état initial ainsi que l'évolution dans le temps. Cette forme compacte est utilisée dans de nombreux outils, et a comme principal avantage de pouvoir être étendue aux systèmes ayant un nombre infini d'états, en particulier aux systèmes traitant de paramètres numériques. Nous utiliserons donc cette représentation pour nos systèmes.

Objet de l'étude Le but de cette étude est de reconnaître des modèles de formules exprimées dans une logique dont le pouvoir d'expression nous semble assez large, à l'aide d'automates symboliques observateurs construits automatiquement, et qui peuvent éventuellement avoir un espace infini d'états. On considérera ici la logique QDDC : "Quantified Discrete Duration Calculus", qui est une logique d'intervalles et qui permet d'exprimer simplement des propriétés complexes. La traduction utilisera le langage LUSTRE, qui permet de décrire simplement des automates symboliques, qu'ils soient d'espace d'états finis ou non. Nous nous restreignons à la vérification de propriétés de sûreté, qui sont les seules intéressantes en pratique (savoir que le train s'arrête inévitablement ne nous intéresse pas, ce qui nous intéresse est de savoir s'il s'arrête avant de toucher quelqu'un !). Nous verrons qu'il est intéressant de séparer la vérification de programmes en deux parties, les *assertions* (qui décrivent les suppositions faites *a priori* sur le système et sur son environnement) et les *propriétés* qui sont les caractéristiques du système que l'on désire vérifier. Nous dégagerons deux fragments différents de la logique QDDC pour les besoins de l'étude, et nous réaliserons leur traduction automatique vers des automates symboliques sous forme de programme LUSTRE.

Plan Ce rapport suivra la structure suivante :
- Dans le premier chapitre, nous présentons LUSTRE, le langage synchrone que nous utilisons pour décrire les systèmes étudiés, ainsi que la méthode de spécification par automate observateur.
- Dans le chapitre 2, nous présentons la logique QDDC, logique que nous voulons vérifier à l'aide d'automates observateurs traduits automatiquement en LUSTRE.
- Dans le chapitre 3, nous introduisons quelques exemples qui montrent la mise en œuvre du procédé, et nous dégageons un fragment déterministe de QDDC.
- Dans le chapitre 4, nous introduisons l'idée d'automate observateur à oracle, et dégageons un nouveau fragment de QDDC qui a une plus grande expressivité. Nous décrivons dans ces deux chapitres la traduction proposée vers le langage LUSTRE, traduction qui a été implémentée (chapitre 5).
- Enfin nous terminons sur des exemples d'utilisation de notre programme au chapitre 6
- Dans la conclusion, nous faisons le point sur ce travail et sur les perspectives de travaux postérieurs.

Chapitre 1

Observateurs en Lustre

Dans ce chapitre nous présentons le contexte de l'étude qui est la programmation synchrone. De nombreux travaux ont déjà été réalisés sur ce sujet, dont nous ferons une rapide synthèse. Nous présentons en particulier le langage LUSTRE, qui est le langage développé par l'équipe synchrone au laboratoire Vérimag, et pour lequel ont été développés plusieurs outils de vérification et de test. Nous présentons aussi dans ce chapitre la méthode de spécification de propriétés par automate observateur, ainsi que les outils de vérification booléens (LÉSAR) et numériques (NBAC) que nous utiliserons dans la suite. Les exemples de cette partie sont fortement inspirés de [HR99].

1.1 Le modèle synchrone

On considère des systèmes réactifs dont la valeur des variables d'entrée varie au cours du temps, et dont les sorties sont calculées à tout instant en fonction de toutes ces valeurs. Si l'on néglige le temps du calcul des sorties, on peut parler de valeur des variables à chaque cycle d'horloge.

Le temps est donc discrétisé en un ensemble de cycles d'horloge, la valeur d'une entrée pouvant prendre une valeur différente à chaque cycle. La valeur d'une sortie à la fin du cycle i dépend des valeurs que prennent les entrées à ce cycle [1] et d'un ensemble *fini* de mémoires.

1.2 Introduction au langage LUSTRE

Nous décrivons ici les principes du langage LUSTRE. Le lecteur pourra trouver plus de détails dans le manuel de référence ([HR02]) et dans l'article fondateur [CPHP87].

Les entrées et les sorties d'un programme LUSTRE sont des *flots de variables*, c'est à dire des variables dont les valeurs peuvent varier au cours du temps [2]. Une variable `x` représente ainsi un flot, c'est-à-dire une suite infinie $(x_0, x_1, \ldots, x_n, \ldots)$ de valeurs. La valeur x_i représente la valeur du flot `x` au i-ème cycle de l'exécution.

Un programme LUSTRE décrit des relations entre les flots de sorties et les flots d'entrées à l'aide d'équations de la forme "`x=e`" qui signifient "$\forall n, x_n = e_n$". Les opérateurs LUSTRE agissent directement sur des flots, ainsi `x+y` dénote le flot $(x_0 + y_0, x_1 + y_1, \ldots x_n + y_n, \ldots)$ (comportement similaire pour $-$, *div*, *mod*), `x or y` (x,y flots booléens) dénote le flot $(x_0 \vee y_0, x_1 \vee y_1, \ldots x_n \vee y_n, \ldots)$

[1] En LUSTRE, on pourra "remonter" de manière bornée dans le passé

[2] Mais une seule fois par cycle !

(comportement similaire pour *or*, *and*, *not*, *xor*, *if* ... *then* ... *else* ...). Les comparaisons d'entiers (=, <>, ...) sont aussi définies "point à point". On définit en plus deux opérateurs temporels :

- L'opérateur "`pre`" ("previous") effectue une opération de retard sur son argument, ainsi `pre(x)` est le flot $(nil, x_0, x_1, \ldots, x_{n-1}, \ldots)$, la valeur *nil* est une valeur indéfinie (qu'il convient donc d'initialiser) ;
- L'opérateur "`->`" ("followed by") est utile justement pour définir des valeurs initiales : le flot `x->y` dénote la suite $(x_0, y_1, y_2, \ldots)$ dont la valeur initiale est la valeur initiale de `x` et qui coïncide ensuite avec `y`.

Enfin, les constantes `true`, `false`, 42 sont des flots constants, ce qui est cohérent avec la définition des opérateurs. Par exemple, $x < 42$ désigne le flot booléen qui à chaque cycle d'horloge indique si la valeur courante du flot x est inférieure strictement à 42.

EXEMPLE 1 *Le programme ci-dessous est un compteur d'"événements". Il prend en entrée deux flots de booléens :* `evt` *(qui est vrai lorsque l'événement à compter a lieu) et* `reset` *(qui est vrai lorsque le compteur doit être réinitialisé), et retourne le nombre d'occurences du signal* `evt` *depuis la dernière occurrence du signal* `reset` *:*

```
node Count(evt, reset : bool) returns (count : int);
let
  count = if (true ->reset) then 0
     else if evt then pre(count) + 1
     else pre(count)
tel
```

Remarquons qu'il n'y a pas de problème d'initialisation pour les deux utilisations de l'opérateur `pre`*, car la première branche du* `if` *est obligatoirement prise au premier cycle de l'exécution (au premier cycle,* `true->reset` *est vrai).*

Les nœuds déclarés peuvent être réutilisés ultérieurement, ce qui permet une programmation modulaire. Lors de la compilation d'un programme LUSTRE, il faut donc préciser le nœud principal.

1.3 Les outils de base autour de LUSTRE

De nombreux outils ont été élaborés autour du langage LUSTRE. Cette section ainsi que la suivante offrent un bref aperçu de ces outils, la page `http://www-verimag.imag.fr/SYNCHRONE/tools.html` comportant tous les liens nécessaires. Signalons enfin que tous les outils présentés ici sont des outils académiques, même si de nombreux outils industriels ont été développés lors de collaborations avec l'industrie.

L'interpréteur Le programme LUSTRE peut être simulé en appelant l'outil Luciole, qui est un outil graphique dans lequel on simule l'exécution du programme en donnant à chaque étape la valeur de toutes les variables d'entrées. On peut visualiser dans le même temps avec le petit programme graphique "Sim2chro" l'évolution des variables d'entrées et de sorties. Par exemple, sur le nœud classique "edge" suivant :

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

on a simulé l'exécution $X = (1,1,1,0,0,1,1,0,0,1,0,0)$, et l'on a eu la sortie Y correspondante :

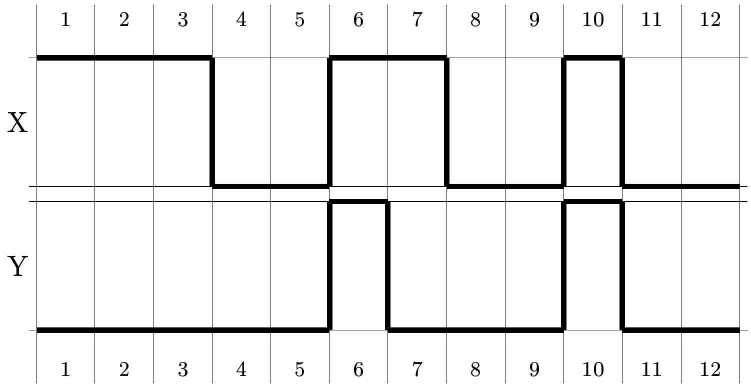


FIG. 1.1 – Une exécution du nœud "edge"

REMARQUE 1
Le nœud "edge" détecte les fronts montants du flot `X`.

Le compilateur Le simulateur graphique est basé sur un interpréteur du langage ; il existe en plus un compilateur de LUSTRE (via un langage intermédiaire OC) vers un programme C qui décrit l'automate du programme. Cet automate (sous la forme OC) peut ensuite par exemple être minimisé en utilisant les méthodes classiques et l'outil OCMIN. Cette séquence pour le nœud "edge" produit ainsi l'automate suivant[3] :

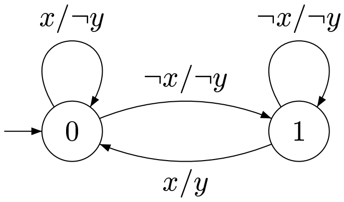


FIG. 1.2 – L'automate minimal du nœud "edge"

Cet automate représente le comportement du programme suivant les valeurs d'entrée. Les transitions représentent les effets d'un cycle sur les valeurs de sorties, suivant les nouvelles valeurs des variables d'entrées. Les états représentent les valuations précédentes des valeurs booléennes (ici, l'état 0 représente `init or pre(x)`, l'état 1 représente `not pre(x)`). Ainsi, sur l'automate précédent, on voit bien que la sortie `y` n'est vraie (`../y`) que lorsque l'on a eu un $\neg$`x` suivi d'un `x`, ce qui est bien le comportement attendu du nœud `edge`.

Ainsi, si l'automate produit n'est pas trop gros, on peut en le visualisant détecter des erreurs du programme, ou alors se convaincre que le comportement est bien le comportement voulu. Cependant, de tels automates sont explicites, et donc sont soumis à l'explosion combinatoire, et il devient rapidement impossible de les visualiser. De plus, si les variations des variables numériques apparaissent dans l'automate (conditions numériques et affectations sur les transitions), elles ne sont pas traitées. D'où l'intérêt de certains outils d'exploration automatique plus sophistiqués, et

[3]Que l'on peut visualiser à l'aide de l'outil ATG grâce à un convertisseur OC2ATG

aussi de méthodes systématiques. Dans la suite, nous proposerons un schéma de vérification qui se veut très général (en particulier, il peut être utilisé dans un langage autre que LUSTRE, voir [PS01] par exemple).

1.4 Les outils de vérification

Dans cette section nous allons présenter certains outils de vérification qui ont été développés autour du langage LUSTRE. Nous verrons quelques premiers exemples simples de vérification par différents outils. Ces exemples utilisent la spécification des propriétés par observateur. Plus de détails pourront être trouvés dans [HR99].

1.4.1 Notion de contrôle

Si le programme à vérifier ne possède que des variables booléennes, il a un nombre fini d'états et peut être totalement vérifié en parcourant cet ensemble et en vérifiant que certains “mauvais” états ne sont jamais atteignables. Ceci peut être réalisé soit de manière énumérative (*i.e.* , en considérant les états un par un, les uns après les autres), soit de manière symbolique (en considérant des ensembles d'états vérifiant une même formule booléenne). Les programmes LUSTRE offrent cependant la possibilité d'utiliser des variables numériques. Ils pourront tout de même être vérifiés en utilisant une approche similaire : si arrive à prouver une propriété concernant le contrôle du système, alors on l'aura prouvé pour le système entier. Dans le cas contraire, on peut éventuellement essayer de réaliser une analyse approchée du comportement des variables numériques.

1.4.2 Vérification à l'aide du seul compilateur

On se restreint dans ce paragraphe à la seule vérification du contrôle. Supposons que nous voulions vérifier une propriété simple sur des variables du programme, une propriété d'invariance, par exemple “toujours, soit la variable p est vraie, soit la variable q est vraie”. On peut alors créer un nouveau nœud LUSTRE qui va “observer” le comportement de ces deux variables, qui peuvent être des entrées, des variables auxiliaires, ou des sorties du programme, et vérifier à tout cycle si la formule $p \vee q$ est vérifiée, et qui stocke cette information dans une variable `ok` :

```
node programme (entree :bool) returns (sortie :bool);
let
    sortie = not(entree);
tel;

node verif (p :bool) returns (ok :bool);
var q : bool;
let
   q = programme(p);
   ok = p or q;
tel;
```

La vérification consiste alors à s'assurer que cette variable `ok` est toujours vraie. Ceci peut être effectué à l'aide du seul compilateur LUSTRE. La compilation du fichier avec comme nœud principal `verif` et la minimisation nous donnent un automate à un seul état et une seule transition `ok`, ce qui signifie que la sortie `ok` est vraie quelle que soit la valeur de p. Nous avons donc bien vérifié la propriété demandée.

REMARQUE 2
Cette méthode peut être utilisée à chaque fois que l'automate avant minimisation n'est pas trop gros. En effet, la compilation du programme LUSTRE *génère l'automate entier (ainsi que son code) avant de le minimiser, ce qui peut rendre la vérification impossible. De plus, la méthode ne peut prendre en compte des variables numériques, que celles-ci soient de valeur fixée ou non.*

1.4.3 Vérification à l'aide de LÉSAR

Le même schéma de vérification peut-être utilisé en vue d'une vérification à l'aide du model-checker LÉSAR (voir l'article [HLR92]). C'est un outil de vérification dédié à LUSTRE qui peut prouver la (non) atteignabilité d'un (ensemble) état(s), de manière symbolique ou bien de manière explicite. Il se restreint cependant à la partie du programme initial qui est importante pour les propriétés exprimées, ce qui est possible car les interdépendances entre les variables LUSTRE sont explicites.

LÉSAR peut réaliser la vérification de programme LUSTRE de deux manières différentes :
- La méthode *énumérative* : elle consiste en un parcours de l'espace des états, mais celle-ci est réalisée “à la volée”, c'est-à-dire qu'à tout instant, seule la partie de cet automate qui nous intéresse est effectivement stockée. On peut aussi prendre en compte *a posteriori* les contraintes numériques, dans ce cas lorsque l'outil détecte un contre-exemple dans le contrôle, il effectue une analyse en arrière pour éliminer ce contre-exemple si une des contraintes numériques portées sur les transitions n'est pas satisfiable.
- La méthode *symbolique* consiste non plus à considérer des états mais des formules satisfaites par les variables booléennes dans les états. On utilise alors des bdds pour améliorer l'efficacité du parcours de l'automate (voir [CBM89, BCM+90]).

1.4.4 Vérification à l'aide de NBAC

L'outil NBAC ([Jea00a]) permet de prouver des propriétés dépendant non seulement du contrôle, mais aussi du comportement dynamique des variables numériques. C'est un analyseur, fondé sur l'interprétation abstraite ([CH78]), qui construit, en chaque point de contrôle, une approximation supérieure des états atteignables des variables numériques, sous forme de polyèdres convexes (systèmes d'inéquations linéaires).

REMARQUE 3 (IMPORTANT)
*Les outils que nous utilisons sont des outils de model-checking qui peuvent éventuellement raisonner par approximations (abstractions). Les résultats que nous obtiendrons seront donc à analyser dans cette perspective : en particulier, une réponse négative tout comme un échec de l'analyse n'apporte aucune information. Seule une réponse positive nous permettra de conclure. Nuançons toutefois cette remarque en signalant que certains outils (*LÉSAR*, par exemple) peuvent en cas d'échec fournir une trace “contre-exemple”, qui est en fait une exécution qui peut (ou non) se révéler être un contre-exemple de la propriété que nous cherchons à vérifier.*

1.5 Schéma de vérification par observateurs synchrones

Nous allons dans cette section décrire plus en détail quel type de spécification/vérification nous voulons réaliser.

L'idée Lorsque l'on veut valider un programme (que ce soit par le test ou par model-checking), on est amené à spécifier :

- des formules qui assurent le bon comportement de l'environnement dans lequel le programme évolue, les "Assertions" (notées A) ;
- des formules qui expriment le comportement attendu du programme sous les suppositions précédentes, les "Propriétés" (notées C pour "conclusion").

En programmation synchrone, il peut être pratique d'utiliser des automates observateurs écrits dans le même langage et dont le but est d'observer que chaque fois que l'environnement vérifie les assertions, l'exécution du programme satisfait la propriété. Plus précisément, les outils de vérification que nous utiliserons procèdent par exploration des états accessibles des automates considérés ; ils vérifieront donc que les propriétés que nous décrirons seront satisfaites par toute exécution qui ne viole *jamais* les assertions. Si on note A les assertions, $Prog$ le programme lui-même, et C ("conclusion") les propriétés, on cherche donc à vérifier :

$$L(A) \cap L(Prog) \cap L(\neg C) = \emptyset,$$

où $L(A)$ décrit le langage de la formule A et $\cap$ désigne le produit synchrone. L'avantage dans notre cas est que ce produit s'exprime facilement dans le cadre de la programmation synchrone, il suffit de composer les codes entre eux.

Formalisation et problèmes Ainsi, si A et P définissent des formules sur les variables du programme $Prog$, on cherche donc à vérifier $\Box A \Rightarrow \Box P$, soit encore $\neg\Box A \vee \Box P$. Le problème est que si l'on se restreint à un parcours des états atteignables, on peut récupérer des diagnostics pessimistes : en effet, sur l'exécution suivante, un parcours en avant donne le diagnostic `false` au deuxième nœud alors que $\Box A \Rightarrow \Box P$ est vraie. En effet, lorsque l'on atteint l'état 2, on constate une violation de la propriété C, alors que l'assertion A n'a jamais été violée. Cependant, cet état ne fait partie d'aucune exécution satisfaisant A en permanence (puisque $\neg A$ est inévitable).

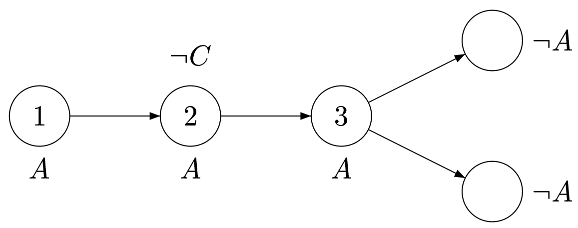


FIG. 1.3 – Un faux négatif

Cela vient du fait que la formule A n'est pas *causale*, *i.e.* , ne vérifie pas $A \Rightarrow EXA$ (à tout instant, si A est vérifiée, il est possible de poursuivre l'exécution de telle sorte qu'elle vérifie A au coup suivant). Ce n'est pas un véritable problème car nous ne considérons que les résultats positifs.

REMARQUE 4
LÉSAR *peut réaliser une vérification du caractère causal lorsqu'ils effectue une analyse en arrière à partir des "mauvais états" qui vérifient A, mais cela induit un surcoût important.*

Dans la suite, par souci de performance, nous préférerons vérifier la formule $\Box\big(\text{after}(\neg A) \vee C\big)$ ($\text{after}(Q)$ est vrai si et seulement si il y a eu un Q dans le passé) à la place de $\Box A \Rightarrow \Box C$. On se ramène ainsi au problème de non-atteignabilité de mauvais état.

Avantages de la méthode Un des nombreux avantages de la méthode est que, étant donné que les observateurs sont décrits dans le même langage que le programme, ils peuvent, tout comme celui-ci, être simulés. Cette possibilité est très importante lorsqu'on connaît la difficulté qu'il y a à exprimer dans des logiques temporelles par exemple la propriété que l'on veut précisément vérifier.

De plus, le même schéma de vérification peut être utilisé pour prouver des tautologies. Il est aussi assez fréquent que les assertions contiennent une spécification complète d'un programme que l'on n'a pas (encore) implémenté.

Mise en œuvre La mise en œuvre du schéma précédent prend en LUSTRE la forme suivante :

- Dans un premier temps, nous créons un nouveau nœud dont le rôle est de décrire les assertions : il prend en entrée les flots du programme qui sont concernés par ces assertions, et calcule un unique flot booléen de sortie *ok*, qui est vrai tant que le programme satisfait les assertions. Nommons `ObsA` ce nœud ;
- Dans un deuxième temps, nous créons de même un observateur des propriétés qui possède une unique sortie booléenne : `ObsC` (pour "conclusion") ;
- Nous créons ensuite un troisième nœud `verif` dont la sortie est égale à $\text{after}(\neg ObsA) \vee ObsC$
- Enfin, nous lançons sur le programme ainsi enrichi (que nous appelons *programme de validation*) un outil de vérification dont le but est de vérifier que *quelles que soient* les entrées du programme, le nœud `verif` ne renvoie jamais 0.

Le schéma suivant résume cette mise en œuvre :

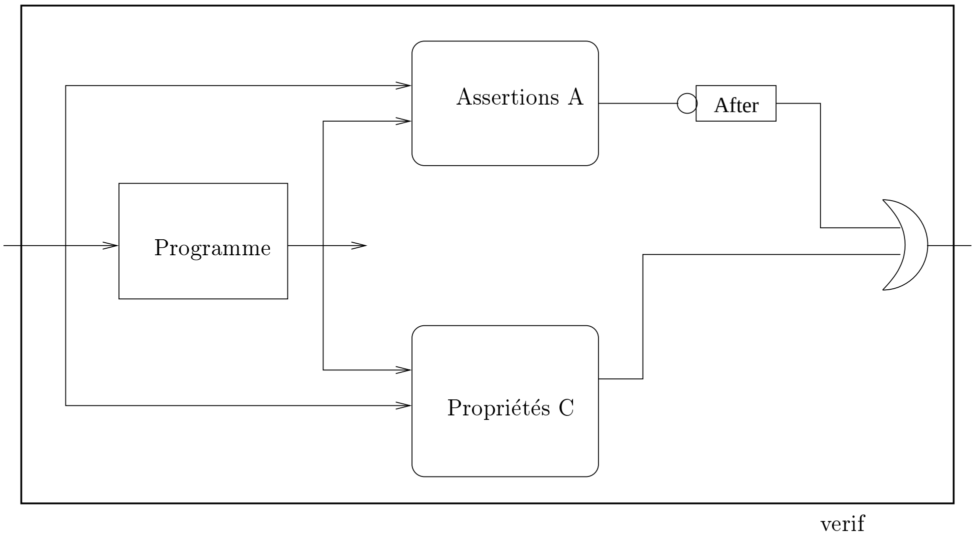


FIG. 1.4 – Spécification par observateurs LUSTRE

REMARQUE 5
Nous avons vu ici que nous cherchons à prouver des propriétés de sûreté, c'est à dire dont on peut détecter la violation en un nombre fini de pas.

1.6 Formalisation de la vérification

Dans la section précédente nous avons vu que nous cherchons à démontrer $\square\big(\text{after}(\neg A) \vee C\big)$. Nous précisons ici le travail que devra effectuer le model-checker utilisé. Cette formalisation sera utilisée dans les chapitres 3 et 4 lors de la preuve de nos différentes traductions.

Notons Π le programme d'étude, sans perdre de généralité nous pouvons considérer ce programme comme une relation entre entrées et sorties (en particulier, on peut considérer que pour un vecteur d'entrées fixé, plusieurs vecteurs de sortie sont possibles). Donc si e désigne le vecteur des variables d'entrées et s les variables de sortie, on peut considérer trois relations : $\Pi(e, s)$, $A(e, s)$ et $C(e, s)$, et nous cherchons donc à prouver :

$$\forall e, \forall s \text{ tel que } \Pi(e, s), \square\big(\text{after}(\neg A(e, s)) \vee C(e, s)\big),$$

soit encore, comme e et s sont en fait des flots de valeurs ($f[i]$ désigne la valeur du flot f au top i) :

$$\forall e, \forall s \text{ tel que } \Pi(e, s), \forall i, \text{after}(\neg(A(e, s)[i]) \vee C(e, s)[i].$$

REMARQUE 6 (IMPORTANT)
Nous avons vu que les outils utilisés parcourent complètement (ou sur-approximent) l'espace des états. En particulier ici ils font exactement le travail que nous cherchons à prouver, i.e. ils regardent toutes les entrées possibles. Cela impliquera plus tard lorsque nous rajouterons de nouvelles entrées aux automates de nos formules (chapitre 4), la vérification devra s'écrire via un quantificateur universel (surtout pas de quantificateur existentiel !). Dès lors, les parties A et C de nos formules ne jouent pas le même rôle, puisque A est niée et pas C. En particulier, le fait de faire des approximations supprime la symétrie de la vérification, dont nous ne gardons que le sens "positif".

Chapitre 2

La logique QDDC

Le Calcul des Durées en temps discret, avec quantificateurs (en anglais, "Quantified Discrete-time Duration Calculus"), que nous appellerons dans la suite plus simplement QDDC, est une logique temporelle pour la spécification et la vérification de systèmes réactifs. En effet, cette logique permet de rendre compte de l'évolution des variables du systèmes sur certains intervalles d'exécution, contrairement aux logiques temporelles classiques qui ne peuvent pas parler par exemple de longueur d'intervalle. L'introduction faite ici reprend largement pour les notations et les exemples celle de [Pan00].

Dans ce chapitre, nous faisons une description complète de la logique, du point de vue syntaxe et sémantique, puis nous montrons quelques exemples simples de formules "classiques" qui peuvent être exprimées. Enfin, nous faisons un bilan des principaux résultats du domaine.

2.1 Une première idée de la logique

QDDC est une logique *temporelle* dont les formules expriment des propriétés d'intervalle et de compteurs. On peut exprimer que sur un intervalle de taille 5 il ne peut pas avoir plus de 2 occurrences du signal p.

La richesse de QDDC provient du fait que l'on peut exprimer la séquentialité de deux formules sur un intervalle à l'aide de l'opérateur "chop" qui est très similaire au "point" des expressions rationnelles : $x \in E.F$ signifie qu'*il existe un découpage* de x en deux mots u, v tel que $u \in E$ et $v \in F$. Ici, un programme satisfera $\phi_1 \frown \phi_2$ sur l'intervalle I si il existe un découpage de I en deux intervalles I_1 et I_2 tel que le programme satisfait ϕ_1 sur I_1 et ϕ_2 sur I_2.

Les modèles de ces formules sont des vecteurs de flots de variables booléennes, c'est-à-dire qu'à chaque top, chacune des variables booléennes a une valeur de vérité. Ces variables peuvent être vues comme des propositions atomiques exprimant des conditions simples sur les variables, par exemple $p = (a + b < 6)$ fournit un flot booléen p dont la valeur à tout instant dépend des deux valeurs numériques a et b. Un tel vecteur de flots booléens est appelé *comportement* du programme.

2.2 Les modèles de QDDC

Soit PVar un ensemble fini de variables propositionnelles qui représentent des valeurs prises durant l'exécution d'un programme. On note VAL(PVar) l'ensemble des valuations sur PVar, c'est-à-dire l'ensemble des fonctions qui assignent une valeur de vérité (0 ou 1) à chaque variable :

$$\mathrm{VAL}(\mathrm{PVar}) \stackrel{\mathrm{def}}{\equiv} \mathrm{PVar} \to \{0,1\}$$

Les *comportements* d'un programme seront alors modélisés par des suites finies de valuations sur PVar, c'est-à-dire par des mots appartenant à $\mathrm{VAL}(\mathrm{PVar})^+$ (suites finies non vides).

Exemple 2 *Voici un comportement σ sur les variables p et q :*

```
p 1 0 1 1 0
q 0 0 0 0 1
```

Définition 1
On note $\sigma[i]$ la i-ème "lettre" (qui peut être vue comme un vecteur) du "mot" σ (en commençant par 0).

Exemple 3 *Sur l'exemple précédent, $\sigma[4]$ désigne la fonction vérifiant $\sigma[4](p) = 0$ et $\sigma[4](q) = 1$.*

Remarque 7
Au lieu de considérer σ comme une suite de fonctions, on peut considérer de façon duale σ comme un n-uplet de mots sur $\{0,1\}^*$ qui ont tous la même longueur, en notant $\sigma(p) = \sigma_0(p)\sigma_1(p)\ldots$. Ainsi on récupère $\sigma[i](p) = \sigma(p)[i]$.

Finissons la section par une définition qui permet de considérer des exécutions "en parallèle" :

Définition 2
Soient $\sigma \in VAL(PVar_1)$ et $\sigma' \in VAL(PVar_2)$ avec $PVar_1 \cap PVar_2 = \emptyset$ et $|\sigma| = |\sigma'|$. Alors on note $\sigma' \odot \sigma$ le nouveau comportement sur les variables de $PVar_1 \cup PVar_2$ défini pour $0 \leq i \leq |\sigma| - 1$ de la façon suivante :

$$(\sigma \odot \sigma')(p)[i] = \begin{cases} \sigma(p)[i] & si\ p \in PVar_1 \\ \sigma'(p)[i] & si\ p \in PVar_2 \end{cases}$$

2.3 Syntaxe et Sémantique de QDDC

2.3.1 Les formules propositionnelles

Syntaxe L'ensemble $\mathbb{P}$ des formules propositionnelles est défini de la façon suivante ($p \in \mathrm{PVar}$) :

$$P ::= 0 \mid 1 \mid p \mid \neg P \mid P \wedge P \mid\ -P$$

Sémantique On définit la satisfaction d'une formule propositionnelle, étant donnée une exécution σ, et un instant i, par induction sur la structure de cette formule :
- $\sigma, i \models 1$;
- $\sigma, i \not\models 0$;
- $\sigma, i \models p$ ssi $\sigma(p)[i] = 1$;
- $\sigma, i \models P \wedge Q$ ssi $\sigma, i \models P$ et $\sigma, i \models Q$;

- $\sigma, i \models \neg P$ ssi $\sigma, i \not\models P$;
- $\sigma, i \models -P$ ssi $i > 0$ et $\sigma, i-1 \models P$.

Remarque 8
Dans la suite, nous qualifierons souvent ces formules de "formules instantanées" car ces formules ne décrivent que des propriétés d'instant. En effet, les formules qui ne contiennent pas le connecteur "–" rendent compte de l'instant i considéré, tandis que ce connecteur permet de rendre compte d'un instant antérieur, mais bien connu.

Exemple 4 *Dans l'exemple 2, on a :* $\sigma, 3 \models p \wedge \neg(- - p)$ *(car* $\sigma(p)[1] = 0$ *et* $\sigma(p)[3] = 1$*).*

2.3.2 La logique Qddc

Syntaxe

$$D ::= \lceil P \rceil^0 \mid \lceil\lceil P \rceil \mid \eta \; op \; c \mid \Sigma P \; op \; c \mid D_1 \wedge D_2 \mid \neg D \mid \exists p D \mid D_1 \frown D_2$$

où $P \in \mathbb{P}$, $op \in \{\leq, <, =, >, \geq\}$ et $c \in \mathbb{N}^*$

Sémantique On définit la satisfaction des formules de Qddc pour un comportement σ sur des intervalles de la forme $[b, e]$, où b et e sont des entiers vérifiant $b \leq e$. La sémantique est définie par induction de la façon suivante [1] :

$\boxed{\lceil P \rceil^0}$ $\sigma, [b, e] \models \lceil P \rceil^0$ ssi $b = e$ et $\sigma, e \models P$: cette formule exprime une propriété instantanée, elle exprime que $P \in \mathbb{P}$ a lieu au top $b = e$ (Figure 2.1).

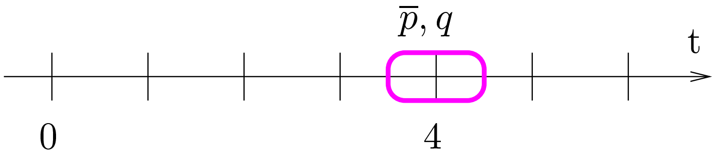


Fig. 2.1 – $\sigma, [4, 4] \models \lceil q \wedge \neg p \rceil^0$

$\boxed{\lceil\lceil P \rceil}$ $\sigma, [b, e] \models \lceil\lceil P \rceil$ ssi $b < e$ et pour tout k tel que $b \leq k \leq e, \sigma, k \models P$: la proposition P est satisfaite tout au long de l'intervalle entier entre b (compris) et e (non compris) (Figure 2.2).

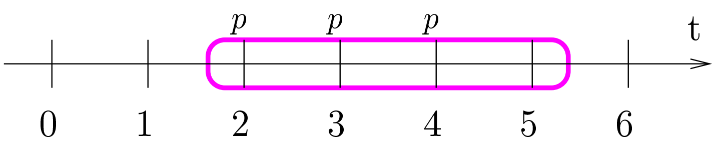


Fig. 2.2 – $\sigma, [2, 5] \models \lceil\lceil p \rceil$

Remarque 9
La formule $\lceil\lceil P \rceil$ *impose à l'intervalle considéré d'être au moins de taille 1.*

[1] dans les dessins, on note $\overline{p}$ au top i si $\sigma_i(p) = 0$, et p sinon.

$\boxed{\eta \text{ } op \text{ } c}$ $\sigma, [b, e] \models \eta \ op \ c$ ssi $e - b \ op \ c$: par exemple, $\sigma, [2, 5] \models \eta > 1$. Remarquons que ces formules ne dépendent pas de σ, mais de la longueur de l'intervalle considéré.

$\boxed{\Sigma P \text{ } op \text{ } c}$ $\sigma, [b, e] \models \Sigma P \ op \ c$ ssi $\left|\left\{k \text{ tel que } b \leq k < e \text{ et } \sigma, k \models P\right\}\right| \ op \ c$, autrement dit le nombre d'occurences de la formule propositionnelle P entre b (compris) et e (non compris) est plus petit (ou plus grand, ou égal) à une certaine constante fixée c (Figure 2.3).

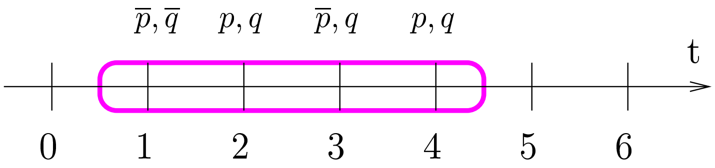


FIG. 2.3 – $\sigma, [1, 4] \models \Sigma(p \wedge q) = 1$

$\boxed{D_1 \wedge D_2}$ $\sigma, [b, e] \models D_1 \wedge D_2$ ssi $\sigma, [b, e] \models D_1$ et $\sigma, [b, e] \models D_2$.

$\boxed{\neg D}$ $\sigma, [b, e] \models \neg D$ ssi $\sigma, [b, e] \not\models D$.

$\boxed{\exists p.D}$ $\sigma, [b, e] \models \exists p.D$ ssi $\sigma', [b, e] \models D$, avec σ' coïncidant avec σ pour toutes les variables booléennes sauf pour p.

$\boxed{D_1 \frown D_2}$ $\sigma, [b, e] \models D_1 \frown D_2$ ssi $\exists m, b \leq m \leq e, \sigma, [b, m] \models D_1$ et $\sigma, [m, e] \models D_2$: cette formule ("chop") exprime l'existence d'un découpage de l'intervalle considéré en deux parties, la première vérifiant la formule D_1, la deuxième vérifiant la formule D_2 (Figure 2.4). Remarquons que le point "frontière" m fait partie des deux intervalles considérés.

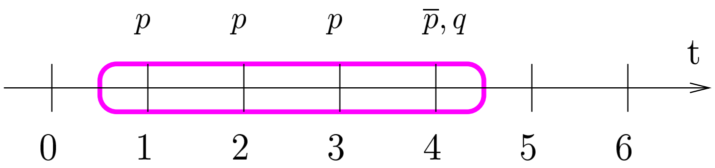


FIG. 2.4 – $\sigma, [1, 4] \models \lceil\lceil p \rceil \frown \lceil q \rceil^0$

La satisfaction d'une formule par une exécution est ensuite définie de la façon suivante :

DÉFINITION 3
Enfin, $\sigma \models D$ *ssi* $\sigma, [0, |\sigma| - 1] \models D$, *où* $|\sigma|$ *désigne la taille de l'exécution* σ *(le nombre de "lettres").*

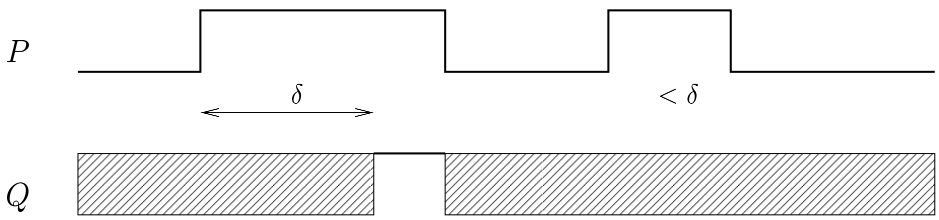
EXEMPLE 5 *Reprenons* σ *tel qu'à l'exemple 2. On vérifie aisément* $\sigma \models \lceil p \rceil^0 \frown \lceil\lceil \neg q \rceil \frown \lceil q \rceil^0$, *autrement dit, p est vrai initialement, q est vrai à l'instant final mais n'est jamais vrai auparavant.*

REMARQUE 10
On voit bien comment les connecteurs "chop", $\lceil\lceil P \rceil$, $\eta \ op \ c$ *et* $\Sigma P \ op \ c$ *permettent d'exprimer des propriétés sur des intervalles (et même, dans le cas du "chop", non complètement spécifiés). C'est ce qui fait la richesse de cette logique.*

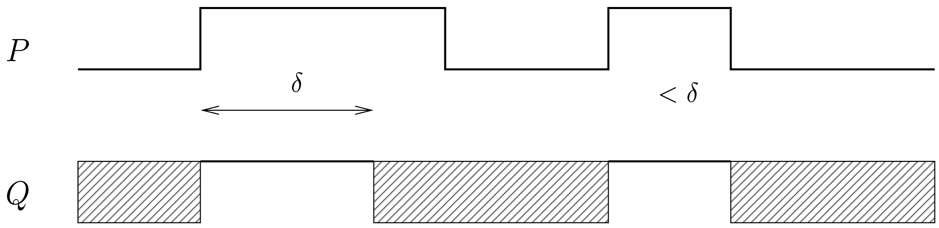
2.3.3 Constructeurs dérivés

Pour faciliter l'expression des propriétés, on définit quelques opérateurs dérivés :

- Les combinateurs booléens $\vee$, $\Rightarrow$, $\Leftrightarrow$ sont définis de façon habituelle avec $\neg$ et $\wedge$;
- $\lceil\lceil P\rceil\rceil \overset{\text{def}}{\equiv} \lceil\lceil P\rceil \frown \lceil P\rceil^0$ exprime que P est invariant sur l'intervalle $[\![b,e]\!]$ fermé à droite ;
- $\lceil\lceil P\rceil\rceil^+ = \lceil\lceil P\rceil\rceil \vee \lceil P\rceil^0$ exprime que la proposition P est vraie sur l'intervalle $[\![b,e]\!]$, *même si* $b=e$. On définit de façon similaire $\lceil\lceil P\rceil^+$;
- $\Diamond D \overset{\text{def}}{\equiv} true \frown D \frown true$ exprime que D est vrai sur un certain sous-intervalle ;
- $\Box D \overset{\text{def}}{\equiv} \neg\Diamond\neg D$ exprime que D est vrai quel que soit le sous-intervalle considéré ;
- Pour $P, Q \in Prop$, $P \overset{\delta}{\rightarrow} Q \overset{\text{def}}{\equiv} \neg\Diamond\Big((\lceil\lceil P\rceil\rceil^+ \wedge \eta \geq \delta) \frown \lceil\neg Q\rceil^0\Big)$ exprime que si P dure continûment depuis δ tops, alors Q doit être vrai (les hachures indiquent que le comportement n'est pas spécifié) :



- Pour $P, Q \in Prop$, $P \overset{\delta}{\hookleftarrow} Q \overset{\text{def}}{\equiv} \neg\Diamond\Big(\lceil\neg -P\rceil^0 \frown (\lceil\lceil P\rceil\rceil^+ \wedge \eta < \delta) \frown \lceil\neg Q\rceil^0\Big)$ exprime que dès que P devient vrai ($\lceil\neg -P\rceil^0$ exprime un front montant), Q devient vrai et le demeure au moins tant que P est vraie et pendant δ tops :



2.4 Quelques exemples

Voici maintenant quelques exemples de formules classiques qui peuvent être facilement exprimées dans la logique QDDC, ces exemples sont inspirés de [Jea00b, Pan00] :

- Le métro 1 précède toujours le métro 2 d'au moins 40 secondes :
$$\Box\big(\lceil metro1\rceil^0 \frown \eta \leq 40 \Rightarrow \lceil\lceil \neg metro2\rceil\rceil\big).$$
- Si le processeur A demande la libération du bus et que celui-ci est libre, alors la requête de A sera acceptée dans les 5 secondes, à moins que un des processeurs B ou C ne demande le bus entre-temps ($\eta = 4$ provient d'un effet de bord) :
$$\Box\Big((\lceil Areq \wedge Busfree\rceil^0 \frown (\eta = 4 \wedge \lceil\lceil \neg Breq \neg CReq\rceil\rceil)) \Rightarrow \Diamond\lceil Ahold\rceil^0\Big).$$
- Quand l'alarme sonne, elle sonne au moins pendant 20 secondes :
$$\Box\Big((\lceil\neg -Alarm\rceil^0 \wedge \lceil Alarm\rceil^0) \frown (\eta < 20) \Rightarrow \lceil\lceil Alarm\rceil\rceil\Big).$$

2.5 Contexte et résultats

Cette logique est une restriction du "Full Duration Calculus" introduit pour la première fois sous cette forme dans [CHR91]. Les modèles du "Full Duration Calculus" sont des "séquences temporisées d'état", autrement dit des suites finies de couples (t_i, s_i) où t_i représente la date d'entrée dans l'état s_i. Les formules d'état sont évalués à une "position" i de façon similaire à QDDC, et on introduit de nouveaux opérateurs qui mesurent le temps écoulé dans un certain intervalle, et le temps réel d'exécution d'une proposition P dans un intervalle. Les opérateurs ΣP et η demeurent et comptent quant à eux le nombre de "pas" effectués pendant l'intervalle.

Le Full Duration Calculus a été montré indécidable dans [CHS93] (par réduction à l'arrêt d'une machine à deux compteurs). Dans [Pan02], on exhibe un fragment décidable qui possède encore des contraintes de temps continu. Enfin, dans [Pan00] on montre la décidabilité de QDDC (temps discret, cette fois), par réduction de cette logique à la logique monadique du second ordre. Cette réduction a été implémentée dans un outil de model-checking nommé DCVALID [2] mais la procédure de décision fait intervenir des automates *explicites* dont la taille peut exploser.

L'objectif est donc de pouvoir spécifier des propriétés QDDC sous la forme d'automate observateur LUSTRE de manière à pouvoir effectuer une vérification *symbolique*. Nous voulons dans la suite pouvoir traduire automatiquement en LUSTRE des assertions et des propriétés écrites dans cette logique (éventuellement en rajoutant des paramètres numériques inconnus). Comme assertions et propriétés jouent un rôle différent dans la vérification (voir le chapitre 1), les fragments effectivement utilisés vont être distincts. Nous verrons dans la suite la construction d'un fragment déterministe (au sens où la vérification se fait uniquement à partir des entrées/sorties du programme), et un fragment indéterministe (dans lequel la traduction fera intervenir d'autres entrées)

[2] Voir `http://www.tcs.tifr.res.in/~pandya/dcvalid.html`

Chapitre 3

Un fragment déterministe et sa traduction

Dans ce chapitre nous réalisons tout d'abord quelques exemples simples de traduction de formules de la logique QDDC vers des automates observateurs écrits en LUSTRE. Nous verrons en particulier les problèmes engendrés par le connecteur "chop", ces problèmes impliquent entre autres le fait que tout QDDC ne sera pas directement traduisible en LUSTRE. Nous introduirons alors un premier fragment, dit "déterministe", et nous effectuerons sa traduction en LUSTRE.

3.1 Premiers exemples

Étant donnée une formule de QDDC, nous voulons construire un automate observateur associé, dont les entrées sont les variables atomiques de la formule, ainsi qu'une entrée additionnelle, b, qui donne le signal de début de(s) l'intervalle(s) considéré(s). La sortie de l'automate doit satisfaire :
- tant qu'il n'y a pas eu le premier b, la sortie est vraie ;
- ensuite, la sortie est vraie si et seulement si l'évolution des variables d'entrées est un modèle de la formule pour l'intervalle défini par le dernier b et le top considéré.

Une formule simple Considérons la formule $\lceil\lceil p \rceil\rceil$. La sortie de l'observateur doit être vraie jusqu'à la première occurrence de b, prendre la valeur de p lorsque b est vraie, et rester vraie tant que p est vrai :

```
node prop1(b,p :bool) returns (out :bool);
let
    out = not after(b) or (if b then p else (p and pre(out)));
tel;
```

Ainsi, seule l'apparition d'un nouveau b peut permettre à `prop1` de redevenir vraie après un $\neg p$.

Une formule à compteur Considérons la formule $p \xrightarrow{\delta} q$ qui exprime que si p a été vrai (continûment) durant d tops, alors q est forcément vrai. L'automate observateur doit alors compter le nombre de p qui se suivent dans le passé, tout en mettant à zéro ce compteur lorsque b est vrai. La sortie devient fausse quand le compteur dépasse d et q est faux, et ne peut redevenir vraie que si un nouvel intervalle est considéré :

```
node prop2(b,p,q :bool;d :int) returns (out :bool);
var reset : bool;
let
    out = not after(b) or b or (pre(reset) and (x¡d or q));
    x = if p then (if reset then 1 else pre(x)+1) else 0;
tel;
```

Ainsi certaines formules de QDDC sont exprimables sous la forme d'automates écrits dans le langage LUSTRE. Dès lors notre objectif est de savoir si toute la logique QDDC est traduisible en observateur LUSTRE, et si ce n'est pas le cas, d'identifier un fragment le plus gros possible que nous pourrons traduire.

3.2 Vers un fragment déterministe

Une formule qui pose problème : Considérons la formule $\Box\big((\eta > c) \Rightarrow (\Sigma P \geq d)\big)$ qui dit que dans tout intervalle de longueur strictement plus grande que c, p a lieu au moins d fois. On peut comme dans l'exemple précédent utiliser des compteurs pour mesurer la longueur d'un intervalle et compter le nombre d'occurences de p. Le problème est que l'on doit déclencher un nouveau compteur à chaque nouvelle occurrence de p (il est aisé de voir que chaque intervalle dont le premier instant vérifie p peut être un "pire cas" pour la formule), et ces compteurs doivent au pire être maintenus jusqu'à ce qu'ils atteignent la valeur d. Dès lors, si l'on utilise l'approche classique, on doit utiliser d paires de compteurs, et si d est un paramètre symbolique, on ne peut pas construire l'observateur.

Le problème de la formule précédente vient du fait que l'opérateur "chop" (qui est caché derrière l'utilisation du connecteur $\Box$), introduit un choix *a priori* quelconque. En effet, considérons la formule $E \frown F$: pour que cette formule soit satisfaite, il faut qu'*il existe* un découpage de l'intervalle en deux morceaux, E étant vrai sur le morceau de gauche, et F sur le morceau de droite. Mais l'exemple précédent nous montre que dans certains cas, il est impossible de prévoir ledit découpage. Cet exemple montre bien l'impossibilité de traduire entièrement la logique QDDC. Nous allons donc essayer d'extraire un fragment aussi grand que possible mais qui est traduisible.

La première idée est celle de s'affranchir du connecteur "chop" et du connecteur existentiel, mais alors la logique perd beaucoup de son expressivité. Cependant, on s'aperçoit assez facilement en traitant des exemples que certaines propriétés exprimées à l'aide de l'opérateur "chop" sont en fait vérifiables par automates déterministes, par exemple la flèche $p \xrightarrow{\delta} q$ qui, bien que définie à l'aide du connecteur chop, a été exprimé sans problème dans notre exemple plus haut.

Pour pouvoir vérifier des formules qui s'apparentent au "chop", l'idée est de pouvoir détecter à quel moment la formule de gauche devient fausse. Dans le cas général, ceci n'est pas possible car la formule de gauche peut devenir fausse, puis redevenir vraie. Dans le cas présent, nous allons donc forcer les formules de gauche à être d'un certain type, c'est-à-dire qu'elles ne pourront que devenir fausses. Il faut aussi modifier un peu la définition du "chop", pour forcer la vérification de la partie droite à démarrer "au plus tard", c'est-à-dire juste au moment où la partie gauche devient fausse. On va donc définir un nouvel opérateur : " then ".

3.3 Quelques définitions utiles

Notion de starter QDDC étant une logique d'intervalle, la vérification va faire intervenir des notions de début et de fin d'intervalle d'étude. L'astuce en LUSTRE va être d'introduire une nouvelle entrée, qui donnera le signal de début d'observation de l'exécution. La fin de l'intervalle sera l'instant courant. La nouvelle entrée LUSTRE qui marque le début de l'observation devra être un *starter*, *i.e.* n'avoir lieu qu'une seule fois :

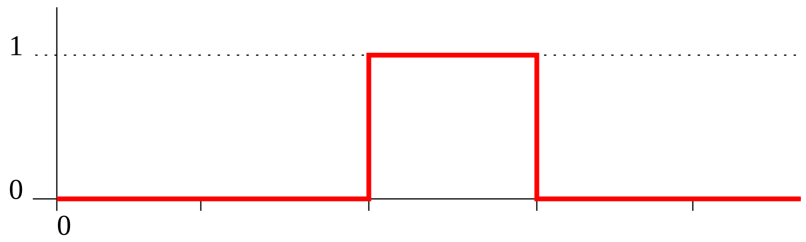


FIG. 3.1 – Un starter

Dans la suite, nous supposerons donc que l'entrée b qui détermine le début de l'intervalle d'étude est elle-même un starter.

De nouveaux connecteurs Dans le but d'introduire un fragment déterministe directement traduisible en LUSTRE, nous définissons de nouvelles formules, étant donné $P \in \mathbb{P}$:

DÉFINITION 4 (AGE)
Age(P) exprime le nombre de P qui ont eu lieu continûment dans le passé, autrement dit, si on note :

$$\mathrm{age}(P)_{[b,e]} = \begin{cases} e - \max\{i : b \leq i \leq e \textit{ et } \sigma, i \models \neg P\} & \textit{si l'événement } \neg P \textit{ a eu lieu depuis le dernier } b, \\ e - b + 1 & \textit{sinon} \end{cases}$$

alors $\sigma, [b, e] \models Age(P)$ *op* c *ssi* $\mathrm{age}(P)_{[b,e]}$ *op* c.

REMARQUE 11
La fonction age peut aisément être écrite en LUSTRE *:*

```
node age (p : bool) returns (age_of_p : int);
let
   age_of_p = 0 ->pre(if p then age_of_p + 1 else 0);
tel
```

DÉFINITION 5 (DÉBUT, FIN)
On définit les deux formules suivantes :
- $\sigma, [b, e] \models begin(P)$ *ssi* $\sigma, b \models P$
- $\sigma, [b, e] \models end(P)$ *ssi* $\sigma, e \models P$

3.4 Un fragment déterministe de QDDC

Le sous-fragment $\mathbb{G}$ On désigne par $\mathbb{G}$ le sous-fragment suivant, dont la principale caractéristique est que, si on allonge l'intervalle d'étude par la droite, on ne peut passer que de vrai à faux :

$$G ::= \text{begin}(P) \mid \lceil\lceil P\rceil \mid \eta \leq c \mid \Sigma P \leq c \mid \text{Age}(P) \leq c \mid G \wedge G \mid G \vee G$$

On définit ensuite le nouvel opérateur suivant :

DÉFINITION 6 (THEN)

$$\sigma, [b,e] \models G \text{ then } F \text{ ssi } \Big(\exists m, \sigma, [b,m] \not\models G \text{ et } \forall i < m, \sigma, [b,i] \models G \text{ et } \sigma, [m,e] \models F\Big)$$

Ainsi "E then F" exprime une succession des formules E et F, tout comme $E \frown F$*. Remarquons toutefois que* E *then* $F \Rightarrow \neg E \frown F$*, car* $\sigma, [b,m] \models \neg E$ *et* $\sigma, [m,e] \models F$*.*

Définition de QDDC-DET Finalement, on note $\mathbb{F}$ les formules :

$$F ::= G \mid \text{end}(P) \mid G \text{ then } F \mid F_1 \wedge F_2 \mid F_1 \vee F_2 \mid \neg F$$

avec $op \in \{\leq, <, =, >, \geq\}$, $c \in \mathbb{N}^*$.

3.5 Traduction en automate LUSTRE

3.5.1 But de la traduction

Le but est de construire, étant donnée une formule $F \in \mathbb{F}$, un circuit $\mathcal{D}_F$ dont les entrées sont σ (mélange de flots d'entrées et de sortie du programme) et un starter b, et à une seule sortie, de sorte que :

$$\forall i, \Big(\sigma, [b,i] \models F \text{ ssi } \mathcal{D}_F(\widetilde{b}, \sigma)[i] = 1\Big)$$

(où $\widetilde{b}$ désigne le flot vrai uniquement à l'instant b)

3.5.2 Construction de $\mathcal{O}_P^{\mathbb{P}}$

Idée de la construction, notations : Soit $P \in \mathbb{P}$ et n le nombre de ses variables : $\text{PVar}(P) = \{p_1, \ldots, p_n\}$. On définit par induction sur la structure de P l'observateur $\mathcal{O}_P^{\mathbb{P}}$ qui a pour entrées un nombre n de variables (sous forme de flots booléens) et pour sortie un unique flot booléen qui indique à l'instant i si les variables qui ont été données en entrée vérifient P à cet instant.

Par souci de rigueur, nous avons dans cette construction (ainsi que les suivantes) changé le nom des paramètres d'entrée du programme, afin que l'on ne confonde pas les *paramètres* de l'observateur et les *variables* du programme Π, qui sont destinées à être utilisées comme entrées de l'observateur. Les paramètres de l'observateur seront notés e_i dans la suite, et les variables du programme Π seront les p_i (p_i va être donné en place du paramètre e_i lors de la vérification).

Remarquons que dans la construction le nombre de paramètres d'entrées est variable (ce qui favorise la construction par induction). On aurait pu faire le choix de mettre en entrées le nombre de paramètres de la formule entière, même si ils n'apparaissent pas tous dans chacune des sous-formules.

Construction de $\mathcal{O}^{\mathbb{P}}_P$, par induction

- Si $P = 1$ (true) alors $\mathcal{O}^{\mathbb{P}}_P() = \texttt{true}$
- Si $P = 0$ (false) alors $\mathcal{O}^{\mathbb{P}}_P() = \texttt{false}$
- Si $P = p$ (variable) alors $\mathcal{O}^{\mathbb{P}}_p(e) = e$.
- Si $P = \neg P_1$ alors $\mathcal{O}^{\mathbb{P}}_{\neg P_1}(e_1, \ldots, e_n) = \neg \mathcal{O}^{\mathbb{P}}_{P_1}(e_1, \ldots, e_n)$.
- Si $P = P_1 \wedge P_2$, alors si $Var(P_1) = \{p_{i_1}, \ldots p_{i_k}\}$ et $Var(P_2) = \{p_{j_1}, \ldots p_{j_r}\}$, (par construction $\{i_1, \ldots, i_k, j_1, \ldots j_r\} = \{1, \ldots n\}$), on construit $\mathcal{O}^{\mathbb{P}}_P(e_1, \ldots e_n) = \mathcal{O}^{\mathbb{P}}_{P_1}(e_{i_1}, \ldots, e_{i_k}) \wedge \mathcal{O}^{\mathbb{P}}_{P_2}(e_{j_1}, \ldots, e_{j_r})$.
- Si $P = P_1 \wedge P_2$, alors la construction est similaire : $\mathcal{O}^{\mathbb{P}}_P(e_1, \ldots e_n) = \mathcal{O}^{\mathbb{P}}_{P_1}(e_{i_1}, \ldots, e_{i_k}) \vee \mathcal{O}^{\mathbb{P}}_{P_2}(e_{j_1}, \ldots, e_{j_r})$.
- Si $P = \text{–}P_1$, on construit $\mathcal{O}^{\mathbb{P}}_P(e_1, \ldots e_n) = false \to pre(\mathcal{O}^{\mathbb{P}}_{P_1}(e_1, \ldots e_n))$.

LEMME 1 *Soit $P \in \mathbb{P}$. On a alors :*

$$\forall i, \Big(\sigma, i \models P \ ssi \ \mathcal{O}^{\mathbb{P}}_P\big(\sigma(p_1), \sigma(p_2), \ldots, \sigma(p_n)\big)\big)[i] = 1\Big)$$

Autrement dit, si sur les flots d'entrées l'observateur $\mathcal{O}^{\mathbb{P}}_P$ renvoie 1 à l'instant i, alors ces flots vérifient la proposition P à cet instant ; et réciproquement.

PREUVE : On va montrer ce lemme par induction sur la structure de la formule $P \in \mathbb{P}$:

- Si $P = 1$ (true) ou $P = 0$ (false) c'est évident.
- Si $P = p$ (variable). Fixons i. Alors par définition $\sigma, i \models P$ ssi $\sigma_i(p) = 1$. Or $\mathcal{O}^{\mathbb{P}}_p(\sigma(p))[i]$, donc on a bien l'équivalence voulue.
- Si $P = \neg P_1$. Fixons i. Par définition $\sigma, i \models \neg P_1$ ssi $\sigma, i \not\models P_1$ ssi (par hypothèse de récurrence) $\mathcal{O}^{\mathbb{P}}_{P_1}(e_1, \ldots, e_n)[i] = 0$ ssi (par construction) $\mathcal{O}^{\mathbb{P}}_P(e_1, \ldots, e_n)[i] = 1$.
- Si $P = P_1 \wedge P_2$, ou $P = P_1 \wedge P_2$, l'équivalence est immédiate.
- Si $P = \text{–}P_1$. Si $i = 0$, alors $\sigma, 0 \not\models P$ et $\mathcal{O}^{\mathbb{P}}_P[0] = 0$, et on a bien l'équivalence. Si $i > 0$, par définition $\sigma, i \models P$ ssi $\sigma, i-1 \models P_1$ ssi (par hypothèse de récurrence) $\mathcal{O}^{\mathbb{P}}_{P_1}(e_1, \ldots, e_n)[i-1] = 1$ ssi (par définition de pre) $\mathcal{O}^{\mathbb{P}}_P(e_1, \ldots, e_n)[i] = 1$. D'où l'équivalence pour tout i.

■

Ces programmes vont nous servir de "briques de base" pour les constructions ultérieures ; elles permettront de tester des sous-formules de type $\lceil\lceil P \rceil$ et $\lceil P \rceil^0$, pour $P \in \mathbb{P}$.

3.6 Traduction des formules de $\mathbb{G}$

Nous allons utiliser les observateurs $\mathcal{O}^{\mathbb{P}}_P$ pour construire par induction les automates $\mathcal{D}^{\mathbb{G}}_G$.

Formules de base Dans la suite, nous allons utiliser les constructeurs de base suivants (voir leur définition LUSTRE dans l'annexe A) :

- $\texttt{since}(P, Q)$ est un opérateur qui exprime que P a toujours été vrai depuis la dernière fois que l'on a eu Q.
- $\texttt{age}(b, P)$ compte le nombre de P qui ont eu lieu continuement dans le passé (strict), depuis b ;
- $\texttt{after\_strict}(P)$ indique si il y a eu une occurrence de P dans le passé *strict.*
- $\texttt{occ\_since}(P, Q)$ compte le nombre de fois que l'on a eu P depuis la dernière occurrence de Q (compris)
- $\texttt{firstfalse}(b, P)$ exprime que P est pour la première fois faux depuis b.

La construction Soit $G \in \mathbb{G}$ avec $\mathrm{PVar}(G) = \{p_1, \ldots, p_n\}$. L'observateur construit par induction aura $n+1$ entrées, une entrée par variable et une entrée pour le starter b. On utilise les constructions $\mathcal{O}_P^{\mathbb{P}}$ de la section précédente :

- Si $G = \mathrm{begin}(P)$: $\mathcal{D}_G^{\mathbb{G}}(b, e_1, \ldots, e_n) = \mathtt{after}\big(b \wedge \mathcal{O}_P(e_1, \ldots, e_n)\big)$
- Si $G = \lceil\lceil P \rceil$: $\mathcal{D}_G^{\mathbb{G}}(b, e_1, \ldots, e_n) = \mathtt{after\_strict}(b) \wedge \mathtt{since}(b, \mathcal{O}_P(e_1, \ldots e_n)$.
- Si $G = \eta \leq c$: $\mathcal{D}_G^{\mathbb{G}}(b) = \mathtt{occ\_since}(b, true) \leq c$.
- Si $G = \Sigma P \leq c$: $\mathcal{D}_G^{\mathbb{G}}(b, e_1, \ldots, e_n) = \mathtt{occ\_since}\big(b, \mathcal{O}_P(e_1, \ldots, e_n)\big) \leq c$.
- Si $G = \mathrm{Age}(P) \leq c$: $\mathcal{D}_G^{\mathbb{G}}(b, e_1, \ldots, e_n) = \mathtt{age}\big(\mathcal{O}_P(b, e_1, \ldots, e_n)\big) \leq c$
- La construction pour $\wedge$ et $\vee$ est évidente.

Validation On a donc le théorème :

THÉORÈME 1 *Soit $G \in \mathbb{G}$ et $b \in \mathbb{N}$. Alors :*

$$\forall i, \Big(\sigma, [b, i] \models G \text{ ssi } \mathcal{D}_G^{\mathbb{G}}(\widetilde{b}, \sigma(p_1), \ldots, \sigma(p_n))[i] = 1\Big)$$

PREUVE : c'est construit pour ! ■

3.7 Traduction des formules de $\mathbb{F}$

On construit maintenant les observateurs $\mathcal{D}_F^{\mathbb{F}}$ pour $F \in \mathbb{F}$, la construction étant très similaire à la précédente. Le nombre d'entrées sera toujours égal à $n+1$, où n est le nombre de variables de la formule G. On note encore $\mathrm{PVar}(G) = \{p_1, \ldots p_n\}$.

Construction

- Si $F = \mathrm{end}(P)$, $\mathcal{D}_F^{\mathbb{F}}(b, e_1, \ldots e_n) = \mathtt{after}(b) \wedge \mathcal{D}_P^{\mathbb{P}}(e_1, \ldots, e_n)$.
- Si $F = G$, $\mathcal{D}_F^{\mathbb{F}} = \mathcal{D}_G^{\mathbb{G}}$.
- Si $F = F_1 \wedge F_2$, $F = F_1 \vee F_2$, $F = \neg F_1$, la construction est évidente.
- Si $F = G$ then F_1, $\mathcal{D}_F^{\mathbb{F}}(b, e_1, \ldots, e_n) = \mathcal{D}_{F_1}\big(\mathtt{firstfalse}(b, \mathcal{D}_G^{\mathbb{G}}(s, e_1, \ldots e_n), e_1, \ldots, e_n)\big)$.

Preuve succincte La preuve de la correction de la construction précédente s'effectue par induction sur la structure de la formule F. Pour les formules de type G then F, la correction vient du fait que les formules de E ne peuvent redevenir vraies après avoir été fausses, d'où le fait que le seul déclenchement possible pour la formule F est détectée par `firstfalse`.

3.8 Exemples

Nous donnons ici quelques exemples simples de formules qui sont exprimables avec QDDC-DET. Nous verrons plus tard (dans le chapitre 6) l'utilisation de certaines de ces formules dans un plus gros exemple.

Les flèches Remarquons tout d'abord que malgré la présence de l'opérateur "chop" dans leur définition, les deux flèches définies à la fin du chapitre 2 peuvent être spécifiés à l'aide de notre fragment déterministe. En effet :

– $P \xrightarrow{\delta} Q \equiv \Box\Big(\big(\mathrm{Age}P < \delta\big) \vee \mathrm{end}(Q)\Big)$. En effet, cette formule exprime le fait que si P a lieu continûment pendant δ unités de temps, alors Q est vraie à la fin de l'intervalle considéré. Formellement :

$$
\begin{aligned}
P \xrightarrow{\delta} Q &\stackrel{\text{def}}{\equiv} \neg\Diamond\Big(\big(\lceil\lceil P\rceil\rceil^{+} \wedge \eta \geq \delta\big) \frown \lceil\neg Q\rceil^{0}\Big) \\
&\equiv \Box\Big(\big(\lceil\lceil P\rceil\rceil^{+} \wedge \eta \geq \delta\big) \Rightarrow \mathrm{end}(Q)\Big) \\
&\equiv \Box\Big(\mathrm{Age}(P) < \delta \vee \mathrm{end}(Q)\Big)
\end{aligned}
$$

– $P \stackrel{\delta}{\hookleftarrow} Q \equiv \Box\Big(\big(\mathrm{Age}(P) \leq \delta\big) \Rightarrow \mathrm{end}Q\Big)$: la preuve se fait de façon similaire.

Ainsi, lorsque nous voudrons exprimer $P \xrightarrow{\delta} Q$, nous pourrons écrire $\big(\mathrm{Age}P < \delta\big) \vee \mathrm{end}(Q)$, le "toujours" étant introduit un cran plus haut lors de la vérification (ce sont nos outils qui vérifieront ce "toujours").

Remarque 12
En fait, nous sommes partis de ces formules pour identifier un fragment déterministe : les formules contenant un opérateur "chop" sont souvent utilisées alors que le découpage de l'intervalle est tout-à-fait connu (ici, on découpe l'intervalle avec $m = e$, et le connecteur "chop" est en fait un "et"). C'est cette remarque qui a permis l'authentification du fragment utilisé pour les assertions, et tout particulièrement la construction du connecteur then *comme restriction déterministe du constructeur "chop".*

Un exemple avec compteur Soit $c \in \mathbb{N}$ et la formule :

$$\big(\Sigma P \leq c\big) \text{ then } \big(\Sigma P \leq d\big) \Rightarrow \big(\Sigma P \leq c + d + 1\big),$$

qui est une tautologie [1]. En suivant la traduction, nous écrivons donc le nœud Lustre suivant :

```
node verif(P : bool ; c0 , d0 : int) returns (ok : bool) ;
var left : bool ;c : int ;
let
   assert (c ≥ 0 and d ≥ 0) ;
   c = c0 ->pre(c) ;
   d = d0 ->pre(d) ;
   left = occ_since(firstfalse(occ_since(true->false, P) ≤ c), P) ≤ d ;
   ok = not(left) or occ_since(true->false, P) ≤ (c+d+1) ;
tel
```

Quelques remarques :
– la ligne `c = c0->pre(c);` est une astuce pour rendre c constant. (idem pour d)
– la variable `left` stocke la partie gauche de l'implication.
– la variable `ok` réalise la vérification de la propriété.
– la première ligne du code du nœud est une *assertion*. Lustre permet ainsi d'exprimer des contraintes *a priori* sur les valeurs des variables, ce qui permet aux models-checkers de ne pas considérer des branches inutiles.

L'appel à NBac renvoie alors un diagnostic positif. On a donc bien prouvé la tautologie.

[1] Attention aux bornes de l'intervalle

Chapitre 4

Un fragment indéterministe

Dans ce chapitre nous décrivons le deuxième fragment de QDDC construit. Dans un premier temps, nous exposerons notre idée d'automates *à oracle*, idée qui nous amène à construire ce fragment. Nous décrivons ensuite la mise en œuvre de la traduction de ce fragment en observateurs LUSTRE. Après la preuve de cette traduction, nous terminerons par un exemple de vérification d'un programme simple qui utilise un tel oracle.

4.1 Vers un fragment indéterministe

L'idée : Considérons la formule $\varphi = \Box\big((\eta > c) \Rightarrow (\Sigma P \geq d)\big)$ qui était à l'origine de l'introduction du connecteur " then " dans le chapitre précédent. Pour résoudre le problème posé par ce genre de formules, nous allons introduire des automates observateurs avec oracle. Au lieu d'utiliser un nombre potentiellement non borné de compteurs, un tel accepteur pourra choisir à quel moment démarrer un intervalle de vérification. La formule sera alors satisfaite si l'observateur accepte les entrées *quels que soient* les choix non-déterministes effectués. C'est l'objet de la section suivante.

Automates observateurs à oracles Informellement, un automate observateur à oracle est un automate observateur rendu non déterministe par l'ajout de nouvelles entrées. L'ensemble des entrées est alors séparé en deux parties, une partie contenant des entrées "classiques", qu'il faudra instancier lors de l'appel de l'automate, et une autre contenant de nouvelles entrées, quelconques, que nous nommerons "oracles". L'outil de vérification utilisé répondra alors oui si quel que soit le choix des oracles, l'automate répond "vrai" à la fin de la séquence étudiée.

EXEMPLE 6 *Soit k un oracle. Alors un automate observateur non déterministe pour la formule φ peut être :*

```
node ex3(p , k : bool) returns (out : bool);
var length, nb_p : bool
let
   assert (unique (k));
   length = if k then 1 else (0 ->pre(length)+1);
   nb_p = if k then (if p then 1 else 0) else
     if p then (pre(nb_p)+1) else pre(nb_p);
   out = not after(k) or (true ->pre(out) and (length ≤ c or nb_p ≥ d));
tel
```

La variable k (oracle) est un starter (l'assertion l'oblige à n'avoir lieu qu'une fois) qui détermine le début de l'intervalle d'étude. La variable `length` *détermine la taille de l'intervalle,* `nb_p` *compte le nombre d'occurrences de la variable p pendant ce temps. La sortie est fausse ssi k a eu lieu et que l'intervalle est de taille supérieure strictement à c et que le nombre de p est plus petit (strictement) que d. D'où la variable* `out`*. Pour que la propriété φ soit vraie, il faut et il suffit que quel que soit la variable k, la sortie* `out` *soit toujours vraie.*

L'utilisation des automates non déterministes pour la vérification des propriétés est requise si l'on veut vérifier des propriétés du type $\neg(E \frown F)$, car cette propriété exprime un découpage (qui est quelconque a priori) de l'intervalle d'étude. Au lieu de lancer une vérification par découpage possible, on va utiliser un oracle m qui sera destiné à prendre toutes les valeurs possibles pour le découpage. A un m fixé sera associé un découpage de l'intervalle considéré, les outils feront le reste (*i.e.* prouveront $\forall m, \ldots$) ! De même, nous allons pouvoir facilement vérifier des formules du type $\forall p.D$, il suffira juste d'abstraire la variable p lors de la vérification de la formule D (en fait, on construira un reconnaisseur de la partie D, et on enlèvera la variable p des entrées du programme), et on parcourra toutes les valeurs possibles pour la variable p.

REMARQUE 13
Ceci dit, on voit que les oracles utilisés pour la vérification d'un "non chop" et la vérification d'un quantificateur universel sont de nature très différentes : un oracle pour vérifier un quantificateur universel est une entrée quelconque, il n'y a aucune contrainte supplémentaire sur son comportement, alors que l'oracle pour le "non chop" doit être un starter. *En effet, la vérification de la partie droite du "chop" ne doit être déclenchée qu'une seule fois par starter, ce que nous verrons plus loin dans la traduction.*

Déterminisme vs non déterminisme Ce non-déterminisme n'apporte bien sûr aucun pouvoir d'expression supplémentaire à nos automates, si nos constantes ont des valeurs fixées (notre méthode est tout de même intéressante car la complexité de l'observateur est alors indépendante de la taille des constantes). Cependant, si nous voulons garder nos constantes symboliques, il est possible de le faire avec nos automates non-déterministes (de nouvelles entrées ...), et impossible autrement.

REMARQUE 14 (FORALL AUTOMATA)
Nous verrons dans la suite qu'une séquence d'entrées/sorties sera acceptée si et seulement si pour toute *nouvelle entrée (oracle), l'observateur renvoie* `true`*. On pourrait donc voir nos automates comme des $\forall$-automata (voir [MP87]) dans lesquels une séquence est acceptée si et seulement si* tous *les runs de la séquence sont acceptés.*

4.2 Le fragment choisi

Soit toujours :

$$P ::= 0 \mid 1 \mid p \mid \neg P \mid P \wedge P \mid\ -P.$$

On définit alors le sous-fragment $\mathbb{E}$ suivant :

$$E ::= \lceil P \rceil^0 \mid \lceil\lceil P \rceil \mid \eta\ op\ c \mid \Sigma P\ op\ c \mid \neg E,$$

avec $op \in \{\leq, <, =, >, \geq\}$, et enfin on définit le fragment $\mathbb{D}$ de QDDC avec la grammaire :

$$D ::= E \mid D_1 \wedge D_2 \mid D_1 \vee D_2 \mid \exists p D \mid D_1 \frown D_2$$

REMARQUE 15
Remarquons que ce fragment est n'est pas clos par la négation. Les sous-formules qui peuvent être niées sont les formules de $\mathbb{E}$, qui sont aisément vérifiables sans oracle.

REMARQUE 16 (NOM DU FRAGMENT)
Vu que nous ne pouvons vérifier que des "quels que soient", *notre fragment indéterministe sera défini par* QDDC-INDET::= $\neg\mathbb{D}$. *En fait, il n'y a qu'une seule négation possible, celle qui s'effectue au niveau le plus haut, et qui permet de transformer les existences d'oracles en "pour tout oracle".*

4.3 But de la construction

Le but est de construire, étant donnée une formule $D \in \mathbb{D}$, un circuit $\mathcal{I}_D$ dont les entrées sont σ (en fait, un sous-ensemble des variables de σ), b (un starter) et un oracle (Δ, Υ) pour D, et à une seule sortie (nommée $\mathcal{I}_D(b, \sigma, \Delta, \Upsilon)$) de sorte que :

$$\forall i, \Big(\sigma, [b, i] \models D \text{ ssi } \exists \Delta, \Upsilon, \big(\mathcal{I}_D(\widetilde{b}, \sigma, \Delta, \Upsilon)\big)[i] = 1\Big).$$

(où $\widetilde{b}$ désigne le flot uniquement vrai à l'instant b)

Autrement dit, les entrées σ vérifient la formule D sur l'intervalle $[\![b, i]\!]$ si et seulement si on peut deviner des oracles qui permettent de faire répondre 1 à l'instant i à l'observateur.

Nous allons donc construire le circuit $\mathcal{I}_D$ par induction sur la structure de la formule $D \in \mathbb{D}$.

4.4 Traduction des formules de $\mathbb{E}$

La construction des observateurs $\mathcal{O}_P^{\mathbb{P}}$ est exactement la même que celle du chapitre précédent. On construit à partir d'eux les observateurs des formules de $\mathbb{E}$, puis de $\mathbb{D}$.

La construction On vérifie les formules de $\mathbb{E}$ à partir de b, flot initial donné en paramètre et les flots des variables d'entrée. Soit $E \in \mathbb{E}$ avec $\mathrm{PVar}(E) = \{p_1, \ldots p_n\}$, l'observateur construit par induction aura $n+1$ entrées : une entrée par variable et une entrée pour le starter b. On ne garantit la correction que lorsque l'on a réellement un starter comme première entrée.

- Si $E = \lceil P \rceil^0$: $\mathcal{O}_E^{\mathbb{E}}(b, e_1, \ldots, e_n) \stackrel{\text{def}}{\equiv} \mathcal{O}_P^{\mathbb{P}}(e_1, \ldots e_n) \wedge b$;
- Si $E = \lceil\lceil P \rceil\rceil$: $\mathcal{O}_E^{\mathbb{E}}(b, e_1, \ldots, e_n) \stackrel{\text{def}}{\equiv} \texttt{since}(b, \mathcal{O}_P^{\mathbb{P}}(e_1, \ldots, e_n)) \wedge \texttt{after\_strict}(b)$ [1] ;
- Si $E = \eta \ op \ c$: $\mathcal{O}_E^{\mathbb{E}}(b) \stackrel{\text{def}}{\equiv} \texttt{occ\_since}(b, \mathit{true}) \ op \ c$;
- Si $E = \Sigma P \ op \ c$: $\mathcal{O}_E^{\mathbb{E}}(b, e_1, \ldots, e_n) \stackrel{\text{def}}{\equiv} \texttt{occ\_since}(b, \mathcal{O}_P^{\mathbb{P}}(e_1, \ldots, e_n)) \ op \ c$;
- Si $E = \neg E_1$: $\mathcal{O}_E^{\mathbb{E}}(b, e_1, \ldots, e_n) \stackrel{\text{def}}{\equiv} \neg \mathcal{O}_{E_1}^{\mathbb{E}}(b, e_1, \ldots, e_n)$.

Validation de la construction On obtient alors le théorème :

THÉORÈME 2 *Soit $E \in \mathbb{E}$ et $b \in \mathbb{N}$. Alors :*

$$\forall i, \Big(\sigma, [b, i] \models E \ \textit{ssi} \ \mathcal{O}_E^{\mathbb{E}}(\widetilde{b}, \sigma(p_1), \ldots \sigma(p_n))[i] = 1\Big)$$

PREUVE : c'est construit pour ! ■

[1] On s'assure bien que l'intervalle considéré est de taille au moins 1.

4.5 Traduction des formules de $\mathbb{D}$

On construit maintenant les observateurs $\mathcal{I}_D$ pour $D \in \mathbb{D}$. Les entrées d'un tel observateur seront b (qui sera un starter), Σ (qui sera un sous-ensemble des variables de σ), et deux ensembles (Δ et Υ) de variables nouvelles (oracles). Par abus de notation, le paramètre d'entrée Δ (ou Υ) désigne l'entrée formée de l'énumération des éléments de l'ensemble Δ (idem pour Υ).

Soit $D \in \mathbb{D}$, avec $\mathrm{PVar}(D) = \{p_1, \ldots, p_n\}$, l'entrée Σ de $\mathcal{I}_D$ désignera un sous-ensemble de l'ensemble $\{e_1, \ldots e_n\}$, correspondant aux variables de D qui ne sont pas dans la portée d'un quantificateur existentiel. De même que précédemment, l'entrée correspondant à la variable p_i sera notée e_i.

▶**Cas de base** Si $D \in \mathbb{E}$, on construit l'observateur :

$$\mathcal{I}_D(b, e_1, \ldots, e_n) = \mathcal{O}_D^{\mathbb{E}}(b, e_1, \ldots, e_n)$$

▶**Construction inductive** On suppose que l'on dispose des programmes $\mathcal{I}_{D_1}$ et $\mathcal{I}_{D_2}$, qui servent à vérifier D_1 (resp. D_2) à l'aide des ensembles d'oracles Δ_1 et Υ_1 (resp. Δ_2 et Υ_2) et d'entrées Σ_1 et Σ_2. On suppose aussi que les éléments de Δ_1 et Δ_2 sont disjoints, et aussi les éléments de Υ_1 et Υ_2. Enfin, par abus de notation l'union de deux ensembles ordonnés Σ_1 et Σ_2 sera l'ensemble ordonné "union".

- Si $D = D_1 \wedge D_2$:

$$\mathcal{I}_D(b, \Sigma_1 \cup \Sigma_2, \Delta_1 \cup \Delta_2, \Upsilon_1 \cup \Upsilon_2) = \mathcal{I}_{D_1}(b, \Sigma_1, \Delta_1, \Upsilon_2) \wedge \mathcal{I}_{D_2}(b, \Sigma_2, \Delta_2, \Upsilon_2),$$

- Si $D = D_1 \vee D_2$:

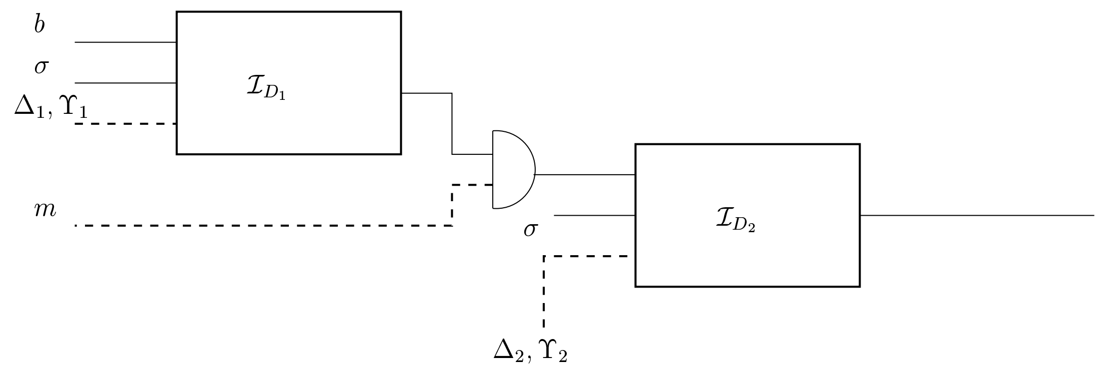
$$\mathcal{I}_D(b, \Sigma_1 \cup \Sigma_2, \Delta_1 \cup \Delta_2, \Upsilon_1 \cup \Upsilon_2) = \mathcal{I}_{D_1}(b, \Sigma_1, \Delta_1, \Upsilon_2) \vee \mathcal{I}_{D_2}(b, \Sigma_2, \Delta_2, \Upsilon_2),$$

- Si $D = \exists p_i \ D_1$:
$$\mathcal{I}_D(b, \Sigma_1 \setminus \{e_i\}, \Delta_1, \Upsilon_1 \cup \{e_i\}) = \mathcal{I}_{D_1}(b, \sigma, \Delta_1, \Upsilon_1)$$

- Si $D = D_1 \frown D_2$, soit m une variable "fraîche " (elle n'apparaît ni dans Δ_1, ni dans Δ_2) [2], alors :

$$\mathcal{I}_D(b, \Sigma_1 \cup \Sigma_2, \Delta_1 \cup \Delta_2 \cup \{m\}, \Upsilon_1 \cup \Upsilon_2) = \mathcal{I}_{D_2}\Big(\mathcal{I}_{D_1}(b, \Sigma_1, \Delta_1, \Upsilon_1) \wedge m, \Sigma_2, \Delta_2, \Gamma_2\Big),$$

ce qui est résumé par le circuit suivant (où l'on a représenté en pointillés les oracles) :



[2] m devra en fait être un starter, ce que l'on vérifiera par ailleurs, et apparaître après b.

REMARQUE 17 (EXPLICATION DE LA CONSTRUCTION "CHOP")
Informellement, pour s'assurer qu'une séquence vérifie $(D_1 \frown D_2)$, *il faut vérifier qu'il existe un découpage de l'intervalle induit par l'oracle* m, *où l'intervalle de gauche vérifie* D_1 *et la partie de droite vérifie* D_2. *L'entrée* m *échantillonne la sortie de* I_{D_1} *et déclenche l'automate observateur* $\mathcal{I}_{D_2}$ *si l'observateur de gauche a répondu* 1.

REMARQUE 18
Lors de la traduction en LUSTRE, *il s'ajoute quelques détails d'ordre technique :*
- *Pour s'assurer que* m *est bien un* starter, *on peut par exemple construire un nœud* LUSTRE `unique(B:bool)` *qui répond* 1 *à l'instant* i *si il n'y a eu qu'une seule instance de* B *dans le passé. Une autre possibilité est de prendre un flot quelconque et de le transformer en* starter *en détectant par exemple le premier instant où il est vrai.*
- *De plus, il faut s'assurer que* $b < i$, *i.e. , que l'intervalle considéré en est bien un ! Ceci peut se faire en rajoutant* `after_strict`(b) *dans (chacun de) nos automates. Même remarque pour* $b \leq m \leq i$, *par exemple en utilisant (comme dans l'exemple 6)* `once_since(m,b)`

EXEMPLE 7 *Pour* $D = \lceil\lceil p \rceil\rceil \frown \lceil\lceil q \rceil\rceil$ *on obtient le programme :*

$$\begin{aligned}
\mathcal{I}_D(b, e_1, e_2, m) &= \mathcal{I}_{\lceil\lceil q \rceil\rceil}\big(\mathcal{I}_{\lceil\lceil p \rceil\rceil}(b, e_1) \wedge m, e_2\big) \\
&= \mathcal{I}_{\lceil\lceil q \rceil\rceil}\big(\texttt{after_strict}(b) \wedge \texttt{since}(b, e_1) \wedge m, e_2\big) \\
&= \texttt{after_strict}\big(\texttt{after_strict}(b) \wedge \texttt{since}(b, e_1) \wedge m\big) \\
&\quad \wedge \texttt{since}\big(\texttt{after_strict(b)} \wedge \texttt{since}(b, e_1) \wedge m, e_2\big)
\end{aligned},$$

sous l'hypothèse que m *est bien un starter*

4.6 Preuve de la traduction

THÉORÈME 3 *Soit* $D \in \mathbb{D}$. *Alors :*

$$\forall b, \forall i, \textit{tels que } b < i, \Big(\sigma, [b, i] \models D \textit{ ssi } \exists \Delta, \Upsilon, \big(\mathcal{I}_D(\widetilde{b}, \sigma(\Sigma), \Delta, \Upsilon)\big)[i] = 1 \wedge \bigwedge_{m \in \Delta} \textit{unique}(m)\Big).$$

REMARQUE 19
$\sigma(\Sigma)$ *désigne par abus de notation les valeurs données aux entrées* Σ *du programme par la suite finie de valuation* σ *considérée. Précisons aussi que* $\forall\Delta$ *signifie : "quelles que soient les valeurs de chacune des variables de* Δ, *le programme répond [une certaine valeur]".*

PREUVE : Fixons i et prouvons le théorème par induction sur la structure de D :
- Si $D \in \mathbb{E}$, alors d'après le Théorème 2, $\sigma, [b, i] \models D$ ssi $\mathcal{I}_D(\widetilde{b}, \sigma(e_1), \ldots, \sigma(\mathrm{e}_n))[i] = 1$. Par construction, comme $\mathcal{I}_D = \mathcal{O}_D^{\mathbb{E}}$, *i.e.* les ensembles Δ et Υ sont vides, on a trivialement le résultat dans les deux sens.
- Pour $D_1 \wedge D_2$, $D_1 \vee D_2$, pas de problème.
- Si $D = \exists p.D_1$, par définition, $\sigma, [b, i] \models \exists p.D_1$ ssi il existe σ' p-variante de σ, $\sigma', [b, i] \models D_1$ (σ' coïncide avec σ sur toutes les variables sauf sur p). Par induction, il existe $\sigma', \Delta, \Upsilon$ tels que $\mathcal{I}_{D_1}(\widetilde{b}, \sigma'(\Sigma), \Delta, \Upsilon)[i] = 1$, autrement dit il existe $\sigma, \Delta, \Upsilon, \mathcal{I}_{D_1}(\widetilde{b}, \sigma(\Sigma \setminus \{e_i\}), \Delta, \Upsilon \cup \{p_i\})[i] = 1$, soit encore il existe σ, Δ, Υ, $\mathcal{I}_D(\widetilde{b}, \sigma(\Sigma), \Delta, \Upsilon)[i] = 1$.
- Si $D = D_1 \frown D_2$, alors par définition, si $\sigma, [b, i] \models D$, alors il existe $m \in [b, e[$, tel que $\sigma, [b, m] \models D_1$ et $\sigma, [m, i] \models D_2$, donc en utilisant l'induction sur la structure de D, on a l'existence de valeurs pour Δ_1, Δ_2, Υ_1,Υ_2 tels que $\mathcal{I}_{D_1}(\widetilde{b}, \Sigma_1, \Delta_1, \Upsilon_1)[m] = 1$ et

$\mathcal{I}_{D_2}(\widetilde{m}, \Sigma_2, \Delta_2, \Upsilon_2)[i] = 1$, ce qui nous assure, en prenant $\Delta = \Delta_1 \cup \Delta_2 \cup \{m\}$ et $\Upsilon = \Upsilon_1 \cup \Upsilon_2$, l'existence de valeurs pour Δ (avec m *starter*) et Υ tels que $\mathcal{I}_D(\widetilde{b}, \Sigma, \Delta, \Upsilon)[i] = 1$. La réciproque se fait de manière similaire.

■

4.7 Être ou ne pas être déterministe

Nous avons construit des observateurs qui induisent des quantificateurs existentiels. Dès lors, il convient de faire attention à la place des négations. Seules les propriétés du type $\neg D$, *i.e.* les formules de QDDC-INDET, pourrons être vérifiées de façon indéterministe. Les assertions et les propriétés ne jouent donc pas le même rôle : les assertions sont niées dans notre schéma (voir le chapitre 1), au contraire des propriétés.

4.8 Un exemple d'utilisation des oracles

Dans cette section nous donnons un exemple plus conséquent de traduction vers un automate à oracle. Il illustre en particulier la traduction du connecteur "chop".

Dans un premier temps, nous allons reprendre l'exemple 7, la traduction nous donne le nœud LUSTRE suivant :

```
node chop(b, p, q, m :bool) returns (ok : bool);
var aux, out_1 :bool;
let
   assert unique(m);
   assert unique(b);
   out_1 = (since(b,p) and after_strict(b));
   ok = (since(aux,q) and after_strict(aux));
   aux = m and out_1;
tel
```

On désire alors vérifier que le programme suivant vérifie $\neg[\![p]\!] \frown [\![q]\!]$

```
node prog(a, b : bool) returns (p, q : bool);
var tops : int;
let
   tops = 1 ->pre(tops)+1;
   q=false;
   p = (tops > 2);
tel
```

Remarquons que la propriété est triviale car on a $\neg p$ au top 0.
Nous écrivons donc le nœud suivant qui écrit la négation de la formule :

```
node Verif(a, b, m : bool) returns (ok : bool);
var p, q : bool;
let
    (p, q) = prog(a, b);
    ok = not (chop(true ->false, p, q, m))
tel;
```

Le diagnostic positif de NBAC nous prouve alors que le programme vérifie toujours $\neg(\lceil\lceil p\rceil\rceil \frown \lceil\lceil q\rceil\rceil)$ (pour des intervalles qui commencent au top 0), car *quel que soit* l'oracle m considéré, le découpage ne convient pas.

Remarque 20
La traduction que nous avons faite ici utilise $b =$ `true->false`. Suivant le contexte, nous pourrons faire varier la valeur de la date de déclenchement de la vérification. Ce choix a été fait car QDDC *peut servir à exprimer des propriétés du type "quel que soit l'intervalle", et ces intervalles peuvent ne pas commencer au premier top.*

Chapitre 5

L'implémentation

Dans ce chapitre nous décrivons sommairement les deux traducteurs implémentés. Il s'agit de compilateurs écrits dans le langage OCAML [1] et qui réutilisent en grand partie un code écrit par Pascal Raymond [2]. Le choix des mots-clefs et des symboles pour les fichiers d'entrée a été fortement inspirée de celle de P. Pandya pour son vérificateur DCVALID(voir `http://www.tcs.tifr.res.in/~pandya/dccheck/demo/overview.html`).

5.1 Le fragment indéterministe : QDDC-INDET

Voici le tableau de correspondance entre les formules du fragment QDDC-INDET et leur écriture concrète ASCII :

QDDC-INDET	ASCII
p	p
$\neg P$	!P
$P \wedge Q$	P && Q
$P \vee Q$	P || Q
1(true)	true
0(false)	false
$\lceil P \rceil^0$	<P>
$\lceil\lceil P \rceil\rceil$	[P]
$\eta\ op\ c$	slen op c
$\Sigma P\ op\ c$	sdur P op c
$\leq$	<=
$<$	<
$=$	=
$>$	>
$\geq$	>=
$D_1 \wedge D_2$	D_1 && D_2
$D_1 \vee D_2$	D_1 || D_2
$D_1 \frown D_2$	D_1 ^ D_2
$\exists p D$	ex p D

[1] Voir `pauillac.inria.fr/caml`

[2] `http://www-verimag.imag.fr/~raymond/`

Le programme écrit effectue une traduction d'UNE formule de `D` donnée sur l'entrée standard ou bien sous la forme d'un fichier. Il renvoie un fichier LUSTRE sur la sortie standard, programme qui ne comporte qu'un seul nœud (il comporte en plus un "include" des nœuds annexes utilisés, comme `after`, *etc.*). Les entrées de ce nœud comportent les variables libres de la formule de départ, et éventuellement les entrées supplémentaires correspondant aux oracles. La traduction suit le schéma du chapitre 4, en utilisant l'induction sur la structure de la formule D.

REMARQUE 21 (IMPORTANTE)
Nos observateurs à oracles ont de nouvelles entrées qui sont quelconques. En conséquence, ils ne peuvent plus être simulés "à la main", ce qui est un inconvénient par rapport aux observateurs déterministes.

5.2 Le fragment déterministe : QDDC-DET

La grammaire concrète pour le fragment déterministe décrit au chapitre 3 est très voisine de celle de celle du fragment indéterministe (on enlève les connecteurs "chop" et "forall"), et l'on rajoute en plus les traductions ASCII suivantes :

QDDC-DET	ASCII
Age(P)	`age(P)`
begin(P)	`begin(P)`
end(P)	`end(P)`

Là aussi, la traduction suit exactement le schéma décrit au chapitre 3. Par contre, il n'y a pas apparition de nouvelles variables, le nombre de variables d'entrées est égal aux nombres de variables de la formule plus une, qui est la variable de début de vérification.

5.3 Schéma de compilation et détails d'implémentation

Le schéma de compilation utilisé est classique et est résumé par la figure 5.1. Quelques remarques :

- Les analyses lexicales et syntaxiques ont été réalisées à partir des outils lex/yacc d'OCAML. On a rajouté un *front-end* qui permet de récupérer la position des éventuelles erreurs de syntaxe (ligne/caractères). Les lexèmes par exemple contiennent l'information de ligne et de colonne. Les erreurs sont gérées par un fichier séparé et permettent d'écrire des messages cohérents sur la sortie d'erreur.
- A la suite de l'analyse syntaxique, le fichier d'entrée est transformé en un objet interne qui contient les informations sur la structure de la formule (arbre syntaxique).
- Cet arbre syntaxique est compilé selon les schémas de traduction des chapitres 3 et 4 en une structure interne qui représente un nœud LUSTRE.
- La dernière partie du travail consiste à imprimer ce nœud LUSTRE selon la syntaxe LUSTRE actuelle, sur la sortie standard.

REMARQUE 22
Signalons enfin que des options de l'exécutable permettent de visualiser les résultats de l'analyse lexicale (liste de "tokens") et de l'analyse syntaxique (pretty-print de l'arbre syntaxique, avec ajout de parenthèses qui rendent la sortie plus lisible). Cette partie a été réalisée à l'aide du module `Arg` *d'OCAML.*

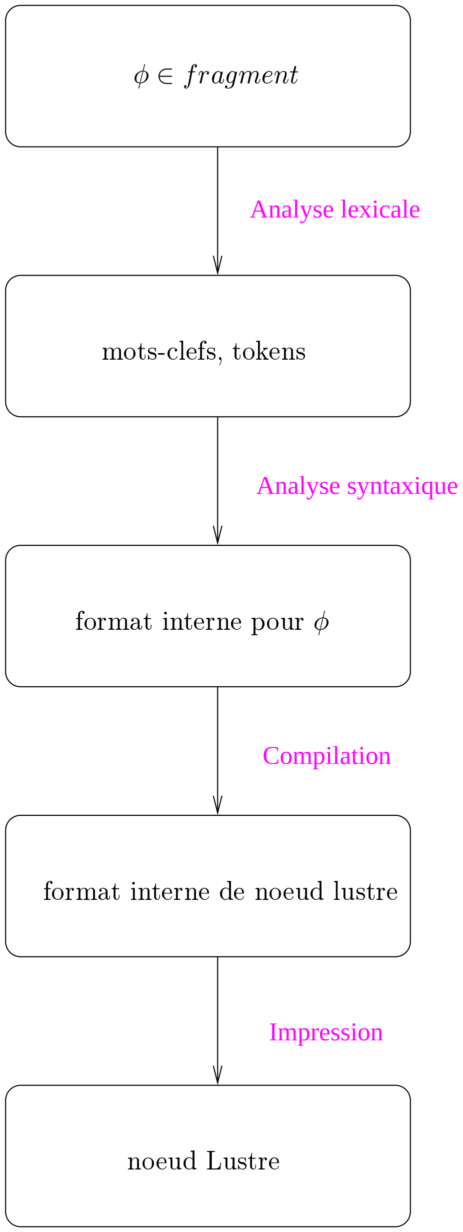


FIG. 5.1 – Schéma de compilation des fragments

Chapitre 6

Études de cas

Dans ce chapitre nous montrons quelques exemples de traductions réalisées automatiquement par notre outil. Dans un premier temps, nous vérifions que ces traductions font bien le travail demandé. Ensuite, nous nous servons de traductions de quelques propriétés pour vérifier un algorithme de "Mine Pump" ([Pan00]). Nous verrons que dans ce dernier cas NBAC a des difficultés à prouver le résultat voulu.

6.1 Exemples simples de traduction

Dans les traductions données on utilise des nœuds LUSTRE prédéfinis, dont les codes sont fournis en annexe. Voyons ce que donne notre implémentation sur des exemples simples :

Un exemple simple La traduction de la formule $\lceil\lceil p \rceil\rceil$ nous donne :

```
node node_1(_p,start_1 :bool) returns (out_1 :bool);
let
   out_1= (since(start_1,(not _p)) and after_strict(start_1));
tel
```

REMARQUE 23
Remarquons que les variables de la formule sont transformées en leur équivalent avec underscore. Ceci est une astuce pour éviter les conflits de nom entre les variables de la formule et les variables générées par le programme.

Un exemple avec compteur Une formule un peu plus compliquée : $\Sigma P \geq 42 \vee \lceil q \rceil^0$:

```
node node_1(_p,start_1,_q :bool) returns (out_3 :bool);
var out_2,out_1 :bool;
let
   out_3= (out_1 or out_2);
   out_1= (occ_since(start_1,_p)) ≥ 42;
   out_2= (start_1 and _q);
tel
```

Un exemple avec oracle Un exemple avec l'opérateur chop, encore la formule $\lceil\lceil p \rceil\rceil \frown \lceil\lceil q \rceil\rceil$:

```
node node_1(var_1,start_1,_p,_q :bool) returns (out_3 :bool);
var var_2,out_2,out_1 :bool;
let
   assert unique(var_1);
   out_3= out_2;
   out_1= (since(start_1,_p) and after_strict(start_1));
   out_2= (since(var_2,_q) and after_strict(var_2));
   var_2= var_1 and out_1;
tel
```

Ici on a donc généré l'oracle `var_1`, qui est une nouvelle entrée (et qui joue le rôle de l'oracle m de l'exemple 7).

6.2 L'exemple du Mine Pump

Dans [Pan00] est décrit un système de pompe qui sécurise une mine à charbon. La mine est soumise à des infiltrations d'eau qui peuvent être régulées en utilisant une pompe qui doit se déclencher lorsque le niveau d'eau devient trop haut. Si le niveau d'eau est dangereux, une alarme doit être déclenchée (dans un laps de temps donné). Il faut aussi détecter des éventuelles poches de méthane à l'aide d'un capteur :si il y a trop de méthane, l'alarme doit aussi se déclencher (mais pas la pompe, à cause du risque d'explosion). Les spécifications de la pompe sont écrits dans l'article selon la logique QDDC, et nous allons essayer de spécifier/vérifier le système avec notre méthode et nos outils. Le code LUSTRE de cette partie pourra être trouvé dans l'annexe B.

6.2.1 Le code

Contrairement à l'article qui ne fait que spécifier les comportements voulus pour la pompe et pour l'alarme, nous avons écrit des codes LUSTRE correspondant à un comportement "au pire" (dans la spécification de l'article) de ces deux éléments (nœuds `alarm` et `pump`) :

- La pompe ne peut se déclencher que lorsque il n'y a pas de méthane ¬`HCH4` et que le niveau d'eau est haut (`HH2O`). Elle se déclenche en δ secondes et s'éteint lorsqu'il s'est écoulé δ secondes d'absence de méthane et de niveau d'eau faible.
- L'alarme se déclenche δ secondes lorsqu'il y a présence de méthane ou que le niveau d'eau est dangereux (`DH2O`) .

Les assertions portent alors uniquement sur l'environnement du système, elles sont décrites dans le nœud `assertion` :

- Un niveau d'eau dangereux implique un niveau d'eau élevé (`ws1`) : $DH20 \Rightarrow HH20$;
- Cela prend au moins w secondes pour passer du niveau d'eau élevé au niveau dangereux (`ws2`) : $HH20 \overset{\delta}{\hookleftarrow} DH20$;
- Cela prend au plus ε secondes à la pompe une fois en marche pour faire descendre l'au à un niveau bas (`pc`) : $PumpOn \overset{\delta}{\rightarrow} HH20$;
- Il se passe au moins ζ secondes entre deux instants où le méthane est haut (`mr1`) : $\Box\Big(\lceil HCH4\rceil^0 \wedge \eta \leq \zeta \Rightarrow \lceil\lceil HCH4\rceil\rceil\Big)$;
- Le méthane ne peut pas rester à un haut niveau plus de κ secondes (`mr2`) : $\lceil\lceil HCH4\rceil\rceil \Rightarrow \eta < \kappa$.

6.2.2 La vérification

Nous voulons montrer que sous certaines hypothèses de contraintes sur δ, ε, *etc.*, le système est sûr, *i.e.* le niveau d'eau ne sera jamais dangereux, ce qui s'écrit :
`ok = true->pre(ok and (after(not A) or not DH2O)),`
avec A désignant une contrainte sur les variables et la conjonction des assertions données plus haut.

Avec des valeurs Si l'on ajoute des valeurs particulières pour les constantes au code (en éliminant les lignes qui parlent des contraintes), la vérification est plus facile. Ainsi, pour $\delta = 2$, $w = 21$, $\varepsilon = 7$, $\zeta = 25$, $\kappa = 2$ (mêmes valeurs que P. Pandya), NBAC renvoie un diagnostic positif (en plus d'une minute-user sur Pentium III 1Ghz, ram 1Ghz) : on a donc trouvé une combinaison des paramètres qui rend le système sûr. Notons que pour d'autres valeurs, l'outil ne répond pas `false`, mais `echec`, ce qui n'est pas très satisfaisant.

Avec des variables symboliques Si on laisse les constantes symboliques tout en mettant des contraintes sur ces constantes, par exemple des contraintes dont on est certain qu'elles font planter le programme, NBAC échoue sur tous les exemples que nous avons essayés. Ceci n'est donc pas satisfaisant puisque l'on n'arrive pas à vérifier l'exemple en symbolique, ce qui était un des objectifs de la traduction en automate LUSTRE.

Signalons qu'un outil actuellement développé au laboratoire Vérimag a pour objectif de pouvoir trouver des contraintes sur des variables symboliques, ce qui est encore plus fort, et donc pourrait nous être utile. En effet, les contraintes avancées par Pandya dans son article ne sont pas suffisantes dans le cas de notre implémentation, puisque nous avons trouvé certains contre-exemples.

Conclusion et Perspectives

Ainsi si notre objectif de réaliser une traduction de QDDC en automate symbolique est atteint, les résultats pratiques ne sont pas satisfaisants dans la mesure où nous n'arrivons pas à traiter effectivement de gros exemples. Cependant, la méthode utilisée est intéressante car elle permet de traiter le cas de variables numériques symboliques, ce qui n'est (évidemment pas) réalisé par les model-checkers de QDDC existants.

Le principal aspect à développer est l'aspect indéterministe de nos automates à oracles. L'utilisation de nouvelles entrées pour spécifier un automate vérificateur nous semble intéressante dans la mesure où elle permet parfois de s'affranchir d'un certain nombre d'états inutiles. Cette méthode pourrait être appliquée à d'autres logiques. De plus, nous ne savons pas si les fragments indéterministes et déterministes construits sont maximaux au sens de l'expressivité, et nous pourrions continuer sur cette voie.

Les principales difficultés rencontrées viennent des variables numériques. Lorsque ces variables sont fixées, les models-checkers classiques les traitent généralement en énumérant leurs valeurs, ce qui n'est pas satisfaisant car d'une part la complexité explose, et d'autre part on n'exploite pas les particularités de ces variables (contraintes insatisfaites, *etc.*). NBAC essaie de traiter ces variables numériques, mais nos exemples ont montré ses limites. De nombreux progrès restent donc à faire dans ce domaine. En particulier, chercher des contraintes que doivent satisfaire des variables numériques pour que le programme marche reste difficile. Les outils que nous avons utilisés se révèlent incapables de faire ce travail. Ceci ouvre donc des perspectives de recherche intéressantes.

Annexe A

Routines Lustre Utilisées

Voici le fichier `routines.lus` qui est inclus dans les fichiers LUSTRE générés par les deux programmes écrits.

```
node edge(x : bool) returns (ok : bool);
--front montant
let
    ok = false ->x and not(pre(x));
tel

node unique(event : bool) returns (unique : bool);
--verification du starter
var already : bool;
let
    already = after_or_at(event);
    unique = not(event and (false ->pre(already)));
tel

node firstfalse(b,P) returns (ok :bool);
--premier non p depuis b
var fst : bool;
let
    ok = after(b) and (not P) and false ->pre(since(b,P));
tel

node after(event : bool) returns (after_event : bool);
--l'evenement event a bien eu lieu
let
    after_event = event ->event or pre(after_event);
tel
```

```
node after_strict(event : bool) returns (after_event : bool) ;
– event a eu lieu dans le passe strict
let
   after_event = false ->pre(event) or pre(after_event) ;
tel

node since(instant,cond :bool) returns (cond_since_instant :bool) ;
– depuis instant, on a toujours eu cond
let
   cond_since_instant = if instant then cond
      else cond and (false ->pre(cond_since_instant)) ;
tel

node notyet(d : bool) returns (res : bool) ;
– on n'a pas encore vu le signal d
let
   res = not(d) ->not(d) and pre(res) ;
tel

node occ_since (d, sig : bool) returns (nb_occ : int) ;
– nb d'occurrences de sig depuis d, sans compter now
let
   nb_occ = 0 ->if (notyet(d) or d) then 0 else if pre(sig) then pre(nb_occ)+1
        else pre(nb_occ) ;
tel

node TWO_STATES (init, up, down : bool) returns (state : bool) ;
– interrupteur
let
   state = (init or up) ->if up and not pre(state) then true
       else if down and pre(state) then false
       else pre(state) ;
tel

node age (p : bool) returns (age_of_p : int) ;
–nombre de p continument dans le passe
let
   age_of_p = 0 ->pre(if p then age_of_p + 1 else 0) ;
tel
```

Annexe B

Code du Mine Pump

Voici les codes de l'exemple du "Mine Pump", détaillé au chapitre 6.

```
node pump(HH2O, HCH4 : bool ; delta : int) returns (PumpOn : bool) ;
var SafePump : bool ;
let
   SafePump = HH2O and not HCH4 ;
   PumpOn = TWO_STATES(false, (age(SafePump) = delta),
      (age(not SafePump) = delta)) ;
tel

node alarm(DH2O, HCH4 : bool ; delta : int) returns (alarm : bool) ;
var ah2o, ach4, na : bool ;
let
   ah2o = (age(DH2O) = delta) ;
   ach4 = (age(HCH4) = delta) ;
   na = (age(not DH2O and not HCH4) = delta) ;
   alarm = TWO_STATES(false, ah2o or ach4, na) ;
tel

node assertion(DH2O, HH2O, HCH4 , PumpOn : bool ; w, eps, zeta, kappa : int)
returns (A : bool) ;
var ws1 , ws2 , pc , mr1 , mr2 : bool ;
let
– Water Seepage Assumptions
   ws1 = (DH2O ⇒ HH2O) ;
   ws2 = edge(DH2O) ⇒ (age(HH2O) ≥ w) ;
– Pump Capacity
   pc = (HH2O ⇒ age(PumpOn) ≤ eps) ;
– Methane Release Assumptions
   mr1 = (edge(HCH4) and strictafter(HCH4)) ⇒ (pre(age(not HCH4)) > zeta) ;
   mr2 = (age(HCH4) < kappa) ;
   A = ws1 and ws2 and pc and mr1 and mr2 ;
tel
```

```
node verif(DH2O, HH2O, HCH4 : bool ;delta0, w0, eps0, zeta0, kappa0 : int)
returns (ok : bool) ;
var PumpOn,A, aconst : bool ;
 delta, w, eps, zeta, kappa : int ;
let
   ok = true->pre(ok and (after(not A) or not DH2O)) ;
   A = assertion(DH2O, HH2O, HCH4 , PumpOn, w, eps, zeta, kappa) and aconst ;
   PumpOn = pump(HH2O, HCH4, delta) ;
   (delta, w, eps, zeta, kappa) =
      constantes (delta0, w0, eps0, zeta0, kappa0) ;
   aconst = contraintes (delta0, w0, eps0, zeta0, kappa0) ;
tel

node constantes (delta0, w0, eps0, zeta0, kappa0 : int)
returns (delta, w, eps, zeta, kappa : int) ;
let
   delta =delta0 ->pre(delta) ;
   w = w0 ->pre (w) ;
   eps = eps0 ->pre(eps) ;
   zeta = zeta0 ->pre(zeta) ;
   kappa = kappa0 ->pre(kappa) ;
tel

node contraintes (delta0, w0, eps0, zeta0, kappa0 : int)
returns (ok : bool) ;
   ok = ((delta0+eps0 + 2*kappa0) < w0) and
      (w0 < 2*kappa0 + zeta0) and
      (delta0 + eps0 < zeta) ;
tel
```

Bibliographie

[BCM+90] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and J. Hwang. Symbolic model checking : 10^{20} states and beyond. In *Fifth IEEE Symposium on Logic in Computer Science, Philadelphia*, 1990.

[CBM89] O. Coudert, C. Berthet, and J. C. Madre. Verification of synchronous sequential machines based on symbolic execution. In *International Workshop on Automatic Verification Methods for Finite State Systems, Grenoble*. LNCS 407, Springer Verlag, 1989.

[CBRZ01] E. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design, volume 19 issue 1*, July 2001.

[CH78] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *5th ACM Symposium on Principles of Programming Languages*, Tucson (Arizona), January 1978.

[CHR91] Zhou Chaochen, C.A.R. Hoare, and A.P. Ravn. A calculus of durations. In *Information Processing Letter, 40, 5, pp. 269-276*, 1991.

[CHS93] Zhou Chaochen, M.R. Hansen, and P. Sestoft. Decidability and undecidability results for duration calculus. In *Proceedings of STACS '93. 10th Symposium on Theoretical Aspects of Computer Science, Wurzburg, LNCS 665*, 1993.

[CPHP87] P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice. LUSTRE : a declarative language for programming synchronous systems. In *14th ACM Conf. on Principles of Programming Languages*, Munich, January 1987.

[EH86] E. A. Emerson and J. Y. Halpern. "Sometimes" and "Not Never" revisited : on branching versus linear time temporal logic. *JACM*, 33(1) :151–178, January 1986.

[Eme90] E. A. Emerson. Handbook of theoretical computer science, volume b : Formal models and semantics, 1990.

[HLR92] N. Halbwachs, F. Lagnier, and C. Ratel. Programming and verifying critical systems by means of the synchronous data-flow programming language LUSTRE. *IEEE Transactions on Software Ingeneering, Special Issue on the Specification and Analysis of Real-Time Systems*, September 1992.

[HR99] N. Halbwachs and P. Raymond. Validation of synchronous reactive systems : from formal verification to automatic testing. In *ASIAN'99, Asian Computing Science Conference*, Phuket (Thailand), December 1999. LNCS 1742, Springer Verlag.

[HR02] Nicolas Halbwachs and Pascal Raymond. A tutorial of LUSTRE, 2002.

[Jea00a] B. Jeannet. Partitionnement dynamique dans l'analyse de relations linéaires et application à la vérification de programmes synchrones. Thèse, Institut National Polytechnique, Grenoble, September 2000.

[Jea00b] Bertrand Jeannet. Partitionnement dynamique dans l'analyse de relations linéaires et application à la vérification de programmes synchrones. Thèse, INPG, 2000.

[MP87] Z. Manna and A. Pnueli. Specification and verification of concurrent programs by ∀-autmata. In *14th ACM Symposium on Principles of Programming Languages, POPL'87*, Munchen, January 1987.

[Pan00] Paritosh K. Pandya. Specifying and deciding quantified discrete-time duration calculus formulae using dcvalid. Technical report, Tata Institute of Fundamental Research, TCS00-PKP-1, 2000.

[Pan02] Paritosh K. Pandya. Interval duration logic : Expressiveness and decidability. In *Electronic Notes in Theoritical Computer Science 65 No. 6*, 2002.

[PS01] Charles Pecheur and Reid G. Simmons. From livingstone to smv. 103-113. In *Formal Approaches to Agent-Based Systems, First International Workshop, FAABS 2000 Greenbelt, MD, USA, April 5-7, 2000, Revised Papers*, volume 1871 of *Lecture Notes in Computer Science*. Springer, 2001.