Janvier 2000 – Journées Francophones des Langages Applicatifs – JFLA2000

Génération automatique de séquences de test à partir de descriptions LUSTRE : GATeL

B. Marre[1] & A. Arnould[2]

1: Laboratoire de Recherche en Informatique,
CNRS UMR 8623, Université de Paris-Sud,
91405 Orsay cedex, France
`marre@lri.fr`
2: Signal, Image, Communication,
SP2MI, Université de Poitiers,
86962 Futuroscope cedex, France
`arnould@sic.sp2mi.univ-poitiers.fr`

Résumé

Nous présentons une méthode et un outil de génération automatique de séquences de tests à partir de descriptions LUSTRE. Le langage LUSTRE est un langage déclaratif à flots de données synchrones. Il est utilisé dans des programmes de contrôle/commande réactifs, principalement pour des applications de production électrique. Le niveau de criticité de ce type de logiciels nécessite un très haut niveau de confiance. Bien qu'il existe des outils de vérification pour ce langage, il persiste un besoin important de techniques de test appropriées. La méthode et l'outil présentés dans cet article sont applicables en phase de test unitaire et d'intégration, selon une méthode de sélection structurelle (test boîte transparente) ou fonctionnelle (test boîte noire). L'outil de génération repose sur une interprétation des constructions du langage LUSTRE par des contraintes sur des variables booléennes ou à valeur dans des intervalles d'entiers. La génération de séquences de test consiste à résoudre ces contraintes à l'aide de techniques de programmation logique contrainte.

1. Introduction

Le langage LUSTRE est un langage déclaratif à flots de données synchrones. Il est utilisé dans des programmes de contrôle/commande réactifs, principalement pour des applications de production électrique. Le niveau de criticité de ce type de logiciels nécessite un très haut niveau de confiance. Les phases de vérification sont donc très développées et leur outillage est indispensable. Les premiers travaux dans ce sens ont portés sur la preuve de propriétés [10]. D'autres travaux se sont ensuite concentrés sur le test. L'outil Lurette [13] permet de construire un programme testeur qui réagit avec le programme sous test (technique des observateurs synchrones). Le testeur engendre aléatoirement et dynamiquement les entrées de test, en vérifiant les sorties conformément à une description de l'environnement et à une propriété invariante. L'outil Lutess [6, 5] permet de construire dynamiquement des entrées de test booléennes en respectant différents critères (distribution des données, propriétés de sûreté, scénarios de test). La méthode développée par le LAAS [14] permet d'engendrer aléatoirement des séquences de test assurant une couverture statistique de la structure du programme LUSTRE (branches de l'automate de contrôle correspondant). Toutes ces méthodes reposent sur des technique de génération *en avant* et ne peuvent tester que des propriétés invariantes. Nous montrerons que notre méthode et l'outil associé permettent de tester toute propriété pouvant être décrite par une observation du passé.

Le langage LUSTRE est utilisé pour décrire de manière homogène une application. Le code final peut être selon les cas, soit engendré automatiquement, soit développé manuellement (pour des raisons d'efficacité, ou parce qu'il est réutilisé d'une application antérieure). Une description LUSTRE doit donc parfois être vue comme une spécification, parfois comme le code source du programme sous test. Notre méthode de test adopte ce double point de vue, puisqu'elle inclue une méthode de sélection adaptée aux approches structurelles (test boîtes transparentes) et fonctionnelles (test boîte noire).

Il n'existe à priori pas de méthode idéale pour caractériser des cas de test pertinents. L'intérêt de ces cas repose en grande partie sur l'expertise du testeur ou sur des connaissances liées au domaine de l'application sous test. Lorsqu'il s'agit d'exercer un programme contre sa spécification (test fonctionnel), les cas de test tendent à couvrir toutes les fonctionnalités décrites dans la spécification (test de conformité) ou seulement certaines d'entre elles (test de propriétés). Certains travaux sur la génération de tests à partir de spécifications formelles proposent de couvrir ces fonctionnalités par des méthodes de couverture structurelle de la spécification (VDM [8], LOFT pour les spécifications axiomatiques [12], CASTING pour le langage B [15] ...). Cependant, les cas de tests obtenus sont souvent trop dépendants de l'expression de la spécification pour assurer une couverture satisfaisante de ses *intentions*. Il est alors nécessaire de préciser des objectifs de test qui définissent les propriétés devant être exercées par le test. Lorsque ces objectifs sont établis, il reste à engendrer des valeurs d'entrée permettant de les atteindre. Dans notre cas, les programmes décrivent des comportements cycliques. Tester de tels comportements nécessite d'engendrer des *séquences d'entrée* qui, partant de l'état initial, atteignent un état satisfaisant l'objectif de test considéré.

L'outil GATEL[1], permet d'engendrer des séquences de tests à partir d'objectifs formulés en LUSTRE étendu (c.f. section 3). Un objectif peut être la traduction d'une propriété à tester, ou bien un sous-domaine du domaine de définition de l'objet sous test caractérisé par un prédicat de chemin du calcul (comme dans les approches structurelles classiques). Dans ces deux cas, la génération d'une séquence de test est traitée comme la résolution d'un système de contraintes sur les flots de données du programme à un instant quelconque. Ces contraintes sont issues de la propriété à tester et/ou d'un prédicat de chemin. Leur résolution construit alors un passé compatible, qui correspond à une exploration *en arrière* du calcul effectué. A certains instants les valeurs des variables d'entrée peuvent ne pas influer sur la satisfaction des contraintes, ces valeurs peuvent donc restées indéterminées en fin de résolution. On opère alors une phase d'instanciation aléatoire pour obtenir des séquences closes.

2. Le langage LUSTRE

LUSTRE est un langage à flots de données synchrones munis d'opérateurs temporels permettant de se référer à des valeurs passées des données. Il a été développé au Laboratoire de Génie Informatique de l'institut IMAG de Grenoble [10]. LUSTRE est utilisé en production industrielle en raisons de ses nombreux avantages. Il possède une syntaxe graphique très proche de celles utilisées dans ce domaine pour décrire les systèmes matériels (diagrammes de blocs, réseaux d'opérateurs, etc.). Il permet d'exprimer : des propriétés de l'environnement, qui sont intégrées au programme LUSTRE et exploitées par le compilateur pour optimiser le code produit, et des propriétés attendues du programme, qui peuvent être vérifiées à l'aide de techniques de "model checking".

1. GATEL est développé dans le cadre d'une collaboration entre le LRI et le CEA/LETI Saclay.

```
const
    tps_cycle = 50 ;

node FDB_CALC_TEMPO(demarrage : bool ; abort : bool ; tempo : int)
returns (fin_tempo: bool) ;
var
compteur : int ;
en_route : bool ;
let
    en_route = demarrage and not(abort)
               ->
               if pre(en_route)
               then not(abort)
               else (demarrage and not(abort)) ;

    compteur = if demarrage and not(abort)
               then tps_cycle
               else
                    (0
                     ->
                     if en_route
                     then pre(compteur) + tps_cycle
                     else pre(compteur)) ;

    fin_tempo = if abort
                then true
                else if en_route
                     then compteur >= tempo
                     else true ;
tel ;
```

FIG. 1 – *Exemple de noeud* LUSTRE

Un programme LUSTRE décrit des comportements cycliques devant respecter l'hypothèse de synchronisme : il réagit continuement à ses entrées en calculant ses sorties dans un temps borné par la durée de cycle d'une horloge globale. Les programmes et sous-programmes sont appelés des *nœuds*. Ils manipulent des variables (d'entrées, de sorties, et locales) qui prennent une nouvelle valeur à chaque instant. Chaque variable de flot (locale ou de sortie) est définie par une unique équation où elle apparaît en membre gauche. Le membre droit contient son expression de définition à chaque cycle. L'expression peut être une fonction de ses valeurs passées et des valeurs présentes ou passées des autres variables.

Prenons l'exemple d'une temporisation dont le source est donné en figure 1. Cet exemple est extrait de l'étude de cas SRIC fournie par "Schneider Electric" lors de l'action de recherche nationale FORMA [4].

Examinons la première équation. Elle définie la variable locale `en_route` à l'aide des opérateurs temporels `->` et `pre` (qui se lisent "init" et "pré"). L'opérateur `pre` permet d'accéder à la valeur d'un flot au cycle précédent. Ainsi `en_route` est définie récursivement sur un temps discret (dont les instants sont appelés *cycles*). L'opérateur `->` permet d'initialiser le flot au premier cycle par la valeur calculée par son premier argument. A tout cycle ultérieur à l'initialisation sa valeur est calculée par son second argument.

Les autres opérateurs LUSTRE, comme le `if then else`, les opérateurs booléens (`and`, `not` ...), les opérateurs arithmétique (`+` ...) et les opérateurs de comparaison (`>=` ...), sont plus classiques. Ils ne font référence qu'aux valeurs de leurs arguments au cycle courant.

cycles	1	2	3	4
`demarrage`	false	true	false	false
`abort`	false	false	false	true
`(1) : not (abort)`	true	true	true	false
`(2) : demarrage and not (abort)`	false	true	false	false
`(3) : pre (en_route)`		false	true	true
`(4) : if (3) then (1) else (2)`		true	true	false
`en_route : (2) -> (4)`	false	true	true	false

FIG. 2 – *Exemple d'exécution de* `en_route`

La sémantique des différents opérateurs est illustrée par l'exemple d'exécution donné par la figure 2 (pour des valeurs données des flots `demarrage` et `abort`).

Outre la définition de programmes, LUSTRE permet d'imposer une propriété à l'aide de la directive `assert` suivie d'une expression booléenne. Cette propriété est (ou doit être) invariante, c'est à dire assurée à tous les cycles. Reprenons le noeud LUSTRE de la figure 1. Il peut être enrichi de l'assertion de la figure 3. Celle-ci précise qu'une temporisation n'est jamais lancée et abandonnée au même cycle. Cette assertion est vérifiée pour les valeurs des flots données figure 2.

```
assert  not (demarrage and abort) ;
```

FIG. 3 – *Exemple d'assertion*

3. Méthode de test

Notre but est d'adapter les méthodes de test classiques aux spécificités du langage LUSTRE. Nous proposons donc une approche en trois phases :

- la *sélection* des entrées de test : ces dernières sont des flots de données de même longueur et de types appropriés ; elles sont choisies selon des objectifs de test et/ou des critères de couverture ;
- la *soumission* des tests : le programme sous test est exécuté sur les entrées précédemment sélectionnées afin d'obtenir les sorties de test (qui sont aussi des flots de données) ;
- l'*oracle* : la décision du succès (respectivement de l'échec) des tests dépend de la nature de la description LUSTRE utilisée pour la sélection :
  - si celle-ci est une spécification de l'objet sous test (test *fonctionnel*), l'oracle calcule les sorties attendues selon la spécification LUSTRE, puis les compare aux sorties de test précédemment obtenues ;
  - si celle-ci est le programme source (test *structurel*), autrement dit si le programme a été généré automatiquement, l'oracle doit s'appuyer sur une spécification, un prototype, ou des propriétés devant être vérifiées (ces dernières pouvant elles même être écrites en LUSTRE et servir d'objectif lors de la sélection).

Description de l'environnement

```
assert env(I, pre(O))
```

Description du programme

```
node ... (I)
returns (O) ;
...
```

Objectif de test

```
assert prop_inv(I, O)
reach prop_inst(I, O)
```

FIG. 4 – *Description de test*

GATeL permet d'outiller la sélection et partiellement l'oracle. Notre approche s'applique au test unitaire comme au test d'intégration. En effet, les objectifs de test, donnent à l'utilisateur un grand choix de sélections possibles. GATeL génère automatiquement des séquences de test (entrées et sorties de test) à partir de la description de test (c.f. figure 4) fournie par l'utilisateur et si nécessaire de critères de couvertures structurels construits interactivement. Dans le cadre de cet article, nous n'illustrerons cette seconde possibilité que dans le but d'assurer une couverture structurelle d'un objectif de test (c.f. section 4.3.2). D'autres renseignements utiles peuvent être trouvés dans [2, 1].

L'utilisateur fournit à l'outil la description LUSTRE du programme sous test. Elle comprend un ou plusieurs noeuds, dont l'interface du plus englobant détermine le nombre et le type des entrées et sorties de test (respectivement notées `I` et `O` sur la figure 4). La description du programme sera utilisée par GATeL pour construire les contraintes liant les sorties aux entrées.

Comme nous l'avons dit dans la section 2, la description LUSTRE du programme peut contenir certaines assertions de description de l'environnement. Le rôle de ces assertions est généralement d'apporter des informations permettant d'optimiser le code généré. Elles ne sont donc pas toujours complètes. Or il est souhaitable de tester un programme sur des séquences appartenant aux domaines d'utilisation possibles. L'utilisateur peut donc compléter la description du programme sous test par une description de l'environnement. Cette dernière contient des assertions portant sur les entrées, et parfois sur le passé des sorties (car le programme sous test est amené à réagir avec son environnement).

Enfin l'utilisateur fournit son objectif de test. Il s'agit de propriétés sur les entrées/sorties (et si nécessaire sur les variables locales) du programme sous test. Les propriétés invariantes sont précédées de la directive `assert`. Celles qui doivent être vérifiées au moins à un cycle (en fait au dernier cycle des séquences construites par GATeL), sont précédées de la directive `reach`. Cette dernière directive n'est pas du LUSTRE. Elle a été ajoutée pour les besoins du test. En effet, contrairement à la preuve, en test on souhaite décrire certaines configurations des entrées/sorties, qui ne sont pas vérifiées à tous les instants, mais seulement après plusieurs cycles de calcul particuliers. Remarquons que toute la syntaxe LUSTRE est disponible pour définir la propriété argument d'un `reach`. On peut donc exprimer toute observation du passé.

Considérons le nœud FDB_CALC_TEMPO (c.f. figure 1). Ce nœud est la description Lustre d'une temporisation paramétrée. Ce sous-programme a été implanté manuellement afin de pouvoir gérer d'éventuelles divergences de la variable compteur (dépassements de maxInt). Nous allons donc considérer que ce nœud Lustre est la spécification de la temporisation paramétrée.

Bien que cette temporisation autorise une grande diversité de comportements, ses paramètres de contrôle doivent respecter les contraintes suivantes :

- la temporisation n'est jamais abandonnée et démarrée simultanément ;
- la durée de temporisation est constante et strictement supérieure à la durée d'un cycle ;

Ces contraintes caractérisent le domaine d'utilisation de FDB_CALC_TEMPO et elles seront respectées lors de la génération des séquences de test. Nous exprimons ces contraintes d'environnement à l'aide des assertions données dans la figure 5.

```
assert  not(demarrage and abort) ;
assert  (true -> (tempo = pre(tempo))) ;
assert  tempo > tps_cycle ;
```

FIG. 5 – *Contraintes d'environnement du nœud* FDB_CALC_TEMPO

Lorsque ces paramètres respectent les contraintes d'environnement, la temporisation doit avoir le comportement suivant.

> Le signal fin_tempo est faux à l'initialisation et lorsque demarrage est vrai.
> fin_tempo ne peut réapparaître que lorsque abort apparaît, ou lorsque la durée tempo est atteinte, sans apparition de demarrage ou abort depuis le début de la temporisation.
> Le signal fin_tempo doit alors être maintenu tant que demarrage n'apparaît pas. Ainsi, le signal fin_tempo n'est à la valeur false que lorsqu'on est en cours de temporisation.

D'après cette description, si après un demarrage le signal fin_tempo apparaît sans redémarrage ni abandon intermédiaire, alors la durée tempo est atteinte.

Un objectif de test permettant d'atteindre cette situation peut être décrit ainsi : on veut une séquence de test conduisant dans un état où fin_tempo est vrai, alors que demarrage est apparu au moins une fois, et que depuis l'apparition de demarrage, on n'a jamais observé abort. Ce qui peut s'exprimer à l'aide de la directive reach comme indiqué figure 6.

```
reach ( fin_tempo  and
        not(jamais(demarrage))  and
        jamais_depuis(abort, demarrage) );
```

FIG. 6 – fin_tempo *est atteinte après* demarrage *sans redémarrage ni abandon intermédiaire*

Les opérateurs jamais et jamais_depuis se contentent d'observer le passé de leurs arguments et sont définis en LUSTRE (c.f. figure 7).

```
node jamais (obs : bool)
returns (jamais : bool) ;
let
    jamais = if obs
             then false
             else (true -> pre(jamais)) ;
tel;

node jamais_depuis (obs,debut : bool)
returns (jamais_depuis : bool) ;
let
    jamais_depuis = if obs
                    then jamais(debut)
                    else (true -> pre(jamais_depuis)) ;
tel;
```

FIG. 7 – *Définition des opérateurs* `jamais` *et* `jamais_depuis`

Le noeud de test décrivant cet objectif et les contraintes d'environnement est le suivant :

```
node test_obj1(demarrage : bool ; abort : bool ; tempo : int)
returns (fin_tempo: bool) ;
let
    assert not(demarrage and abort) ;
    assert (true -> (tempo = pre(tempo))) ;
    assert tempo > tps_cycle ;

    fin_tempo = FDB_CALC_TEMPO(demarrage, abort, tempo) ;

    reach ( fin_tempo  and
            not(jamais(demarrage))  and
            jamais_depuis(abort, demarrage) );
tel;
```

4. Principes de l'outil GATEL

Dans cette partie, nous décrivons l'outil GATEL. L'exemple introduit dans la section 2 illustrera notre propos.

4.1. Le noyau de GATEL

Comme nous l'avons mentionné en introduction, la génération de séquences de tests est réalisée par résolution d'un système de contraintes. Nous détaillerons par la suite la construction de ce système. Le noyau de GATEL est une procédure de résolution de contraintes sur des variables booléennes ou à valeur dans des intervalles d'entiers (nous ne prenons pas en compte les descriptions faisant intervenir des réels).

La résolution procède par éliminations successives de toutes les contraintes. On utilise une procédure d'instanciation (non déterministe) des variables intervenant dans ces contraintes (étape de "labelling" en programmation logique contrainte). Afin d'éviter les instanciations erronées, un mécanisme de propagation de contraintes surveille en permanence la satisfiabilité des contraintes (arithmétiques, booléennes et temporelles). Ainsi, l'instanciation d'une variable "réveille" la propagation des contraintes dans lesquelles elle intervient, qui elles-mêmes peuvent instancier d'autres variables, réveiller ou introduire d'autres contraintes.

Du fait des opérateurs temporels, les séquences à construire peuvent être très longues, et leur longueur n'est jamais connue à l'avance (celle-ci est cependant bornée par le paramètre `maxFlow` fixé par l'utilisateur). Les équations de définition des variables aux différents cycles sont donc traduites en contraintes à la demande, lorsqu'une contrainte nécessite leur valeur. Cette résolution "paresseuse" permet de minimiser le nombre moyen de contraintes, et par conséquence, l'utilisation mémoire (la mémoire nécessaire dépend aussi du nombre d'étapes de "labelling"). Ceci conduit à une exploration du programme LUSTRE "en arrière", c'est à dire des sorties vers les entrées et du dernier cycle au cycle initial.

L'efficacité de GATEL repose en grande partie sur le choix de la variable à instancier pendant la résolution. On utilise alors des heuristiques visant à réveiller le maximum de contraintes, tout en "minimisant" le branchement moyen de l'arbre de résolution (variable de plus petit domaine).

Lorsque toutes les contraintes sont résolues, on dispose d'une instanciation partielle des flots d'entrée (certaines valeurs ne sont pas nécessaires à chaque cycle). Une instance close des entrées est alors obtenue par instanciation aléatoire des variables restantes, en respectant les intervalles de valeurs possibles pour les variables entières. Les séquences de sortie correspondantes sont ensuite calculées par simple évaluation.

GATEL est implanté en PROLOG dans l'environnement de programmation logique contrainte ECLiPSe ([9]). Le mécanisme de propagation de contraintes utilise des suspensions et des variables attribuées dans des domaines finis : booléens et unions d'intervalles d'entiers.

4.2. Génération guidée par un objectif de test

Décrivons maintenant comment GATEL construit et résout le système de contraintes dérivé d'un objectif de test et de la description de l'environnement. Considérons l'objectif donné figure 6. Le système de contraintes correspondant est construit à partir des assertions à chaque cycle connu, des propriétés apparaissant dans les `reach` au dernier cycle, et des définitions des variables de flot nécessaires issues du programme LUSTRE.

Soit S une séquence de valeurs finie, notons S_N la dernière valeur de S et plus généralement, S_{N-i} la valeur de S au cycle précédant de i cycles le dernier cycle ($0 \leq i < N$). Ainsi, $\mathtt{pre}(S_N)$ sera noté S_{N-1}. Les contraintes correspondant à notre exemple sont les suivantes :

$$
\begin{array}{l}
\ \ \mathtt{not(demarrage}_N\ \mathtt{and\ abort}_N\mathtt{)\ =\ true} \\
\wedge\ \mathtt{(true\ \text{->}\ (tempo}_N\ \mathtt{=\ tempo}_{N-1}\mathtt{))\ =\ true} \\
\wedge\ \mathtt{(tempo}_N\ \mathtt{>\ 50)\ =\ true} \\
\\
\wedge\ \mathtt{fin\_tempo}_N\ \mathtt{=\ true} \\
\wedge\ \mathtt{not(jamais(demarrage}_N\mathtt{))\ =\ true} \\
\wedge\ \mathtt{jamais\_depuis(abort}_N\mathtt{,\ demarrage}_N\mathtt{)\ =\ true}
\end{array}
$$

Lors de l'introduction des contraintes, le mécanisme de propagation tente de les simplifier au maximum. Lorsqu'une contrainte ne peut plus évoluer, elle "s'endort" en attente sur les variables dont l'instanciation pourrait relancer son évolution.

Les flots de données devant être synchronisés, une contrainte de synchronisation (que nous laisserons implicite) impose une longueur identique aux séquences de valeurs de tous les flots référencés. Cette contrainte gère le statut de chaque cycle connu (à valeur dans l'ensemble `{initial, non_initial}`). Lorsque le statut devient `non_initial`, elle crée un cycle précédent, propage les assertions à ce cycle et s'endort en attendant que le statut de ce nouveau cycle soit connu. Chaque contrainte d'initialisation (opérateur `->`) consulte ces informations de synchronisation pour propager son premier ou son second argument selon le statut de son cycle. Si ce statut n'est pas connu, la contrainte d'initialisation s'endort en attente sur l'instantiation du statut.

L'introduction des contraintes de notre exemple provoque les traitements suivants:

- `not(demarrage`$_N$ `and abort`$_N$`) = true` devient `demarrage`$_N$ `and abort`$_N$ `= false` et s'endort en attente sur l'instanciation de `demarrage`$_N$ ou de `abort`$_N$;
- le statut du cycle N n'étant pas connu, la contrainte `(true -> (tempo`$_N$ `= tempo`$_{N-1}$`)) = true` s'endort en attente sur ce statut ;
- la contrainte `(tempo`$_N$ `> 50) = true` est résolue en réduisant l'intervalle de `tempo`$_N$ à `[51..maxInt]` (les valeurs `minInt` et `maxInt` sont paramétrés par l'utilisateur) ;
- la contrainte `fin_tempo`$_N$ `= true` est remplacée par la définition de `fin_tempo`$_N$ et la substitution `fin_tempo`$_N$ $\leftarrow$ `true` est mémorisée ; la contrainte définissant `fin_tempo`$_N$ s'endort en attente sur `abort`$_N$;
- la contrainte `not(jamais(demarrage`$_N$`)) = true` est remplacée par la définition de `jamais`$_N$ et la substitution `jamais`$_N$ $\leftarrow$ `false` est mémorisée ; la contrainte définissant `jamais`$_N$ s'endort en attente sur `demarrage`$_N$;
- la contrainte `jamais_depuis(abort`$_N$`, demarrage`$_N$`)) = true` est remplacée par la définition de `jamais_depuis`$_N$ et la substitution `jamais_depuis`$_N$ $\leftarrow$ `true` est mémorisée ; la contrainte définissant `jamais_depuis`$_N$ s'endort en attente sur `abort`$_N$.

Les contraintes obtenues à la fin de cette première étape de propagation sont les suivantes :

```
  demarrage_N and abort_N = false
∧ (true -> (tempo_N = tempo_N-1)) = true
∧ tempo_N ∈ [51..maxInt] (*)

∧ fin_tempo_N ← true  (*)
∧ fin_tempo_N =
    if abort_N
    then true
    else if en_route_N
         then (compteur_N >= tempo_N)
         else true
∧ jamais_N ← false  (*)
∧ jamais_N =
    if demarrage_N
    then false
    else (true -> jamais_N-1)
∧ jamais_depuis_N ← true  (*)
∧ jamais_depuis_N =
    if abort_N
    then jamais(demarrage_N)
    else (true -> jamais_depuis_N-1)
```

Certaines contraintes sont annotées par le symbole * pour indiquer qu'elles sont résolues (instanciation d'une variable ou bien appartenance d'une variable à un intervalle).

A ce stade du calcul, les contraintes ne peuvent plus évoluer. La procédure de résolution doit alors choisir une variable à instancier de manière à faire évoluer/disparaître le maximum de contraintes (“labelling”). Nous utilisons une heuristique classique qui consiste à choisir la variable de plus petit domaine sur laquelle un nombre maximum de contraintes sont en attente. Dans notre exemple, il s'agit de la variable booléenne $\texttt{abort}_N$ sur laquelle trois contraintes sont en attente. La procédure de résolution l'instancie alors de manière aléatoire par `true` ou `false`, en mémorisant le choix effectué de façon à pouvoir explorer l'autre possibilité en cas d'échec (“backtrack”).

Plaçons nous dans le cas où la valuation choisie pour $\texttt{abort}_N$ est `false`. Le “réveil” de ces trois contraintes provoque les évolutions suivantes :

- la contrainte $\texttt{demarrage}_N$ `and false = false` est résolue et disparaît ;
- la contrainte définissant $\texttt{fin\_tempo}_N$ devient `if` $\texttt{en\_route}_N$ `then (`$\texttt{compteur}_N$ `>=` $\texttt{tempo}_N$`) else true` puis s'endort en attente sur $\texttt{en\_route}_N$; la valeur de $\texttt{en\_route}_N$ est alors nécessaire, on introduit sa définition qui s'endort en attente sur le statut du cycle N ;
- la contrainte définissant $\texttt{jamais\_depuis}_N$ devient `(true ->`$\texttt{jamais\_depuis}_{N-1}$`)` puis s'endort en attente sur le statut du cycle N.

On obtient le système de contraintes suivant :

```
  abort_N ← false  (*)
∧ (true -> (tempo_N = tempo_N-1)) = true
∧ tempo_N ∈ [51..maxInt] (*)

∧ fin_tempo_N ← true  (*)
∧ fin_tempo_N =
    if en_route_N
    then (compteur_N >= tempo_N)
    else true
∧ en_route_N =
    demarrage_N and not(abort_N)
    ->
    if en_route_N-1
    then not(abort_N)
    else (demarrage_N and not(abort_N))
∧ jamais_N ← false  (*)
∧ jamais_N =
    if demarrage_N
    then false
    else (true -> jamais_N-1)
∧ jamais_depuis_N ← true  (*)
∧ jamais_depuis_N =
    true
    ->
    jamais_depuis_N-1
```

A nouveau, les contraintes ne peuvent plus évoluer. La procédure de résolution doit élire une nouvelle variable à instancier. Le principe de fonctionnement étant suffisamment illustré, nous ne détaillerons pas d'avantage le calcul.

Remarquons que lors de l'instanciation aléatoire de $\texttt{abort}_N$, la valeur `true` aurait conduit à une insatisfiabilité. En effet, notre objectif de test impose que l'on n'ait jamais observé `abort` depuis `demarrage`. Donc, lorsque `abort` est vrai, on ne peut pas avoir eu `demarrage` par le passé (c.f. définition de `jamais_depuis` figure 7), ce qui est incompatible avec la partie de l'objectif imposant qu'il y ait eu au moins un `demarrage` (`reach not(jamais(demarrage))`).

Lorsque toutes les contraintes sont résolues, les variables d'entrée restantes sont instanciées aléatoirement et les sorties sont évaluées. On peut alors obtenir les séquences de valeurs suivantes :

Cycle	demarrage	abort	tempo	fin_tempo
1	true	false	53	false
2	false	false	53	true
3	false	false	53	true
4	false	false	53	true

Cette séquence de valeurs n'est pas minimale. En effet, ses deux premiers cycles suffisent pour atteindre l'objectif considéré. Cependant, elle est intéressante car elle illustre le fait que lorsque la durée de temporisation est atteinte (cycle 2), le signal `fin_tempo` doit être maintenu tant que le signal `demarrage` n'apparaît pas (cycles 3 et 4).

```
reach ( auparavant( fin_tempo  and
                    not(jamais(demarrage))  and
                    jamais_depuis(abort, demarrage) ) and
        jamais_depuis(not(fin_tempo), fin_tempo);


node auparavant(obs : bool)
returns (auparavant: bool) ;
let
  auparavant =
    false
    ->
    if pre(obs)
    then true
    else pre(auparavant);
tel;
```

FIG. 8 – *Maintient de* `fin_tempo`

Nous pouvons définir un nouvel objectif de test assurant la couverture de notre objectif initial et une telle situation de maintien de la temporisation. Ce nouvel objectif est donné dans la figure 8. L'opérateur `auparavant` impose la réalisation de l'objectif initial dans un passé strict. La contrainte `jamais_depuis(not(fin_tempo), fin_tempo)` assure le maintien de `fin_tempo`.

Pour ce nouvel objectif, GATeL engendre la séquence de valeurs donnée en figure 9. L'objectif initial est atteint au cycle 5 à partir duquel `fin_tempo` est maintenu. Remarquons que cet objectif n'imposait aucune condition sur `demarrage` et `abort` depuis l'apparition de `fin_tempo`. Le fait que `demarrage` soit maintenu à `false` est une condition nécessaire (et suffisante) pour le maintien de `fin_tempo` à `true`.

4.3. Décomposition en sous-domaines

Nous venons d'illustrer comment obtenir *une* séquence de test par résolution d'un système de contraintes dérivé d'un objectif et de contraintes d'environnements. Ne retenir qu'une seule solution pour ce système de contraintes revient à considérer que toute solution est équivalente. En d'autres termes, nous faisons l'hypothèse que toutes les séquence de test atteignant l'objectif ont la même

Cycle	demarrage	abort	tempo	fin_tempo
1	true	false	211	false
2	false	false	211	false
3	false	false	211	false
4	false	false	211	false
5	false	false	211	true
6	false	true	211	true
7	false	true	211	true
8	false	false	211	true

FIG. 9 – *Séquence de test pour l'objectif : maintien de* `fin_tempo`

capacité de détection d'erreurs. De telles *hypothèses d'uniformité* ([3]) sont classiques mais sont souvent trop fortes. La démarche communément admise en test consiste alors à décomposer le domaine de validité des tests en une union de sous-domaines plus petits, sur lesquels il semble raisonnable d'appliquer des hypothèses d'uniformité.

4.3.1. Dépliage des opérateurs LUSTRE

Dans notre cas, le domaine de validité des tests est caractérisé par un système de contraintes. Nous proposons une technique de découpage en sous-domaines inspirée de la technique de dépliage proposée dans l'outil LOFT [12]. Elle permet de découper récursivement des sous-domaines selon des sous-cas prédéfinis pour certains opérateurs LUSTRE.

Par exemple, si le système de contraintes contient une contrainte $\mathtt{S}_i$ `= if` $\mathtt{Cond}_i$ `then ExpThen else ExpElse` où $\mathtt{Cond}_i$ est une variable, nous pouvons dériver deux sous-domaines par dépliage du `if then else`. Le premier sous-domaine contient tous les tests tels que $\mathtt{Cond}_i$ est vraie, tandis que le second contient les tests tels que $\mathtt{Cond}_i$ est fausse. Ces deux sous-domaines sont définis par les deux systèmes de contraintes obtenus par propagation des valuations possibles de $\mathtt{Cond}_i$. Ils peuvent à leur tour être découpés, par dépliage d'un des opérateurs apparaissant dans leurs contraintes de définition.

A chaque étape de découpage, GATEL indique les différents opérateurs *dépliables* (opérateurs apparaissant en tête d'une expression dont l'évaluation est nécessaire). Ainsi l'utilisateur peut choisir la nature et le nombre des découpages.

Nous donnons ci-dessous un extrait de la liste des opérateurs (et leurs sous-cas) pour lesquels GATEL peut proposer un dépliage.

- $\mathtt{A}_i$ `= not(Exp)` : 2 cas correspondants aux valuations de $\mathtt{A}_i$;
- $\mathtt{A}_i = \mathtt{Exp}_1$ `and` $\mathtt{Exp}_2$: 3 possibilités selon le paramètre utilisateur `nbCasDefBool`,
  - **and** séquentiel (par défaut) : 2 cas correspondants aux valuations de $\mathtt{A}_i$;
  - **and** paresseux : 3 cas,
    1. $\mathtt{A}_i \leftarrow \mathtt{true} \wedge \mathtt{true} = \mathtt{Exp}_1 \wedge \mathtt{true} = \mathtt{Exp}_2$,
    2. $\mathtt{A}_i \leftarrow \mathtt{false} \wedge \mathtt{false} = \mathtt{Exp}_1$,
    3. $\mathtt{A}_i \leftarrow \mathtt{false} \wedge \mathtt{false} = \mathtt{Exp}_2$;
  - **and** normal : 4 cas correspondants à sa table de vérité ;
- $\mathtt{A}_i = \mathtt{Exp}_1$ `=<` $\mathtt{Exp}_2$: 2 cas (valuations de $\mathtt{A}_i$) ou 3 cas ($=$, $<$ ou $>$), selon le paramètre utilisateur `nbCasDefComp`,
- $\mathtt{A}_i = \mathtt{Exp}_1$ `->` $\mathtt{Exp}_2$:
  1. le statut du cycle i est `initial`,
  2. le statut du cycle i est `non_initial`.

De plus, GATeL permet d'appliquer une décomposition en sous-domaines à partir des contraintes définissant une sortie sélectionnée. Le dépliage permet alors de construire les différents prédicats de chemins du calcul de cette sortie à un instant quelconque. On obtient ainsi une couverture *structurelle* des expressions participant au calcul de la sortie sélectionnée.

4.3.2. Couverture structurelle d'un objectif de test

Nous illustrons maintenant cette technique de dépliage en l'appliquant au système de contraintes dérivé de l'objectif "maintien de `fin_tempo`" (figure 8). Ce système de contraintes contient la contrainte définissant `fin_tempo` au dernier cycle :

```
fin_tempoN =
  if abortN
  then true
  else if en_routeN
       then (compteurN >= tempoN)
       else true
```

GATeL permet donc de déplier l'opérateur `if then else` apparaissant en tête. En effectuant le dépliage de cet opérateur nous obtenons deux sous-domaines correspondants aux valuations de $\texttt{abort}_N$. Cette première étape de dépliage nous permet de tester l'influence de `abort` sur le maintien de `fin_tempo`. Dans le cas $\texttt{abort}_N \leftarrow \texttt{false}$, la contrainte définissant $\texttt{fin\_tempo}_N$ devient :

```
fin_tempoN =
  if en_routeN
  then (compteurN >= tempoN)
  else true
```

A nouveau, l'opérateur `if then else` apparaît en tête et est dépliable. Son dépliage produit deux sous-domaines correspondants aux valuations de $\texttt{en\_route}_N$. Nous décidons de stopper la décomposition et demandons la génération de séquences de test pour les trois sous-domaines obtenus.

1. La première séquence de valeurs atteint l'objectif de test dans le cas où `abort` est vrai au dernier cycle :

Cycle	demarrage	abort	tempo	fin_tempo
1	true	false	81	false
2	false	false	81	true
3	false	true	81	true

2. La seconde séquence de valeurs atteint l'objectif de test lorsque `abort` est faux au dernier cycle et le flot `en_route` est vrai au dernier cycle. Par conséquent, `abort` n'a pas pu apparaître pendant le maintien de `fin_tempo` :

Cycle	demarrage	abort	tempo	fin_tempo
1	true	false	61	false
2	false	false	61	true
3	false	false	61	true
4	false	false	61	true

3. La dernière séquence de valeurs atteint l'objectif de test lorsque `abort` et `en_route` sont faux au dernier cycle. Par conséquent, `abort` est apparu avant le dernier cycle, pendant le maintien

de `fin_tempo` (cycles 4, 6 et 7) :

Cycle	demarrage	abort	tempo	fin_tempo
1	true	false	75	false
2	false	false	75	true
3	false	false	75	true
4	false	**true**	75	true
5	false	false	75	true
6	false	**true**	75	true
7	false	**true**	75	true
8	false	false	75	true
9	false	**false**	75	true

5. Conclusion

Actuellement GATeL a des performances permettant de traiter des textes Lustre relativement complexes. Ces performances sont illustrées sur un des objectifs de test de l'étude de cas SRIC [4]. Celui-ci nécessite d'engendrer des séquences de plus de 1000 cycles pour des contraintes complexes (expressions booléennes, dont les arguments font intervenir des comparaisons et des temporisations ayant pour arguments ces mêmes contraintes à des cycles antérieurs). Lors de cette expérience, cet objectif de test n'avait pas pu être traité par les autres partenaires (VERIMAG avec son outil Lurette [13], et le LAAS par sa méthode de test statistique structurel [14]). Au LRI, la génération des séquences de test était réalisée par Loft [12] (GATeL était en cours de définition) en utilisant une traduction systématique de Lustre en spécifications algébriques [11]. Après un important travail de simplification (non automatisé), Loft pouvait traiter cet objectif, sur une station Ultra-Sparc II, en 30 minutes de calcul et 600 mega-octets de mémoire. Pour ce même objectif et sans effectuer aucune simplification de la description Lustre, GATeL traite cet objectif en 15 minutes et 280 mega-octets de mémoire. Bien que le gain apparent ne semble pas très important, il faut remarquer que cet exemple ne pouvait pas être traité par Loft sans le simplifier.

Les performances de GATeL sont bien plus importantes si on simplifie la description Lustre en exploitant les informations relatives aux contextes d'appel des sous-nœuds [7] (propagation de constantes, assertions ...). On obtient alors une définition spécialisée, permettant d'ignorer statiquement des pans entiers du calcul effectué par GATeL à chaque cycle. En effectuant ce pré-traitement (automatisé), l'objectif de test est traité en 90 secondes de calcul et 10 mega-octets de mémoire. Ces résultats devront être confirmés sur d'autres études de cas complexes.

Comme nous l'avons vu section 4, GATeL permet d'exprimer et de couvrir des objectifs de test corresondant à n'importe qu'elle observation du passé, ce qui n'est pas le cas des autres outils tels que Lurette et Lutess [5] qui ne peuvent traiter que des propriétés invariantes. Contrairement aux approches du LAAS et de Lutess, GATeL n'offre pas de couverture probabiliste des objectifs de test. En contrepartie, nous proposons une méthode entièrement automatisée, ce qui n'est pas le cas de la méthode du LAAS, et nous couvrons une part importante des programmes Lustre, ce qui n'est pas le cas de Lutess qui ne traite que des calculs booléens. Cependant, la partie du langage Lustre considérée par GATeL devra être étendue : structuration des flots par des tuples, opérateurs `current` et `when`, prise en compte des réels. Ce dernier point semble le plus difficile à traiter, car la précision du calcul peut non seulement influer sur les sorties d'un cycle, mais aussi provoquer des décalages temporels dans l'apparition de certaines sorties (e.g. comparaisons de seuil en argument de temporisations).

L'utilisation de GATeL nécessite une bonne compréhension du mécanisme de propagation de contraintes, notamment lors du découpage du domaine de test par dépliage des opérations. Cette difficulté pourrait être réduite en fournissant des informations intermédiaires à l'utilisateur en cours de traitement (analyse et mise en forme des contraintes courantes, construction de l'arbre des choix ...).

Nous avons vu que la syntaxe actuelle procure un grand pouvoir d'expression, en particulier pour les objectifs de test. Cependant, il serait intéressant de simplifier la description de ces objectifs, par exemple en prenant en compte des représentations utilisées dans le domaine d'application.

Références

[1] Y. Amalric. Projet APIS-REST, GATeL, aperçu méthodologique et description du générateur de tests. Rapport CEA/LETI/DEIN/SLA/99-22, août 1998.

[2] A. Arnould and N. Perot. Projet APIS-REST génération automatique de test lustre : GATeL. Rapport Rapport/DTA/LETI/DEIN/SLA/98-08, January 1998.

[3] G. Bernot, M.C. Gaudel, and B. Marre. Software testing based on formal specifications: a theory and a tool. *Software Engineering Journal*, 6(6):387–405, 1991.

[4] Ouvrage collectif. Opération 2 : vérification et génération de tests pour un système de comptage de neutrons. *Action Forma, Bilan de la première année*, Paris, janvier 1998.

[5] L du Bousquet, F. Ouabdesselam, J.-L. Richier, and N. Zuanon. Lutess: a specification-driven testing environment for synchronous software. *21st International Conference on Software Engineering.* ACM, mai 1999.

[6] L. du Bousquet and N. Zuanon. An overview of lutess, a specification-based tool for testing synchronous software. *14th IEEE International Conference on Automated Software Engineering*, USA, octobre 1999.

[7] F. Delevacq. Spécialisation de programmes LUSTRE. Rapport de stage de DESS II, LRI, Orsay, septembre 1999.

[8] J. Dick and A. Faivre. Automating the generation and sequencing of test cases from model-based specifications. In *FME'93: Industial-Strenth Formal Methods, First International Symposium of Formal Methods Europe*, Springer Verlag, *LNCS* volume 670, pages 268–284, Odense, Danemark, avril 1993.

[9] ECLiPSe Constraint Logic Programming System. *Environnement, librairies et documentations.* http://www.icparc.ic.ac.uk/eclipse.

[10] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language LUSTRE. *Proceeding of the IEEE*, 79(9):1305–1320, septembre 1991.

[11] N.C. Hsiao. *Sélection de tests de propriétés de sûteté à partir d'une modélisation algébrique de programmes LUSTRE.* Thèse, Université de Paris-Sud, 91405 Orsay, France, 1994.

[12] B. Marre. Toward automatic test data set selection using algebraic specifications and logic programming. *ICLP'91, Eigth International Conference on Logic Programming*, pages 25–28, Paris, France, 1991. MIT Press.

[13] P. Raymond, D. Weber, X. Nicollin, and N. Halbwachs. Automatic testing of reactive systems. *19th IEEE Real-Time Systems Symposium.* IEEE, 1998.

[14] P. Thevenod-Fosse. Unit and integration testing of lustre programs: a case study from the nuclear industry. *9th European Workshop on Dependable Computing (EWDC-9)*, pages 121–124, Gdansk, Pologne, mai 1998. Rapport LAAS No98078.

[15] L. Van Aertryck, M. Benveniste, and D. Le Metayer. Casting: a formally based software test generation method. *proc. IEEE Int. Conference on Formal Engineering Methods*, pages 101–111, 1997.