Lot 3.2

Test

Prototype de génération de test d'interopérabilité pour les systèmes temporisés

Description :	Cette fourniture donne une description de l'implémentation du prototype de génération de test d'interopérabilité pour les systèmes temporisés AVERROES.
Auteur(s) :	Ismail BERRADA, Richard CASTANET, Patrick FÉLIX,
Référence :	AVERROES / Lot 3.2 / Fourniture 3.2.2 / V1.0
Date :	juin 2005
Statut :	validé
Version :	1.0

Réseau National des Technologies Logicielles
Projet subventionné par le Ministère de la Recherche et des Nouvelles Technologies

CRIL Technology, France Télécom R&D, INRIA-Futurs, LaBRI (Univ. de Bordeaux – CNRS), LIX (École Polytechnique, CNRS) LORIA, LRI (Univ. de Paris Sud – CNRS), LSV (ENS de Cachan – CNRS)

Historique

Juin 2005	V 0.1	version préliminaire
Juillet 2006	V 1.0	version finale

Table des matières

1 Représentation symbolique

La manipulation des polyèdres requiert une structure de données pour représenter les polyèdres. Cette structure de données doit permettre de tester l'inclusion de deux polyèdres et de calculer aisément les différentes opérations définies sur les polyèdres, i.e. l'intersection, le successeur et le prédécesseur, l'image par une remise à zéro,... Dans cette section, nous allons présenter la structure de données DBM (Difference Bound Matrix), ainsi que l'implantation des différentes opérations définies sur les polyèdres.

1.1 Préliminaires

Une borne est un couple $(c, \prec)$ tel que $c \in \mathbb{Z}$ et $\prec \in \{\leq, <\}$. Un ordre total est alors défini sur les bornes de la façon suivante : si $m = (c, \prec)$ et $m' = (c', \prec')$ alors $m < m'$ ssi $c < c'$ ou $c = c'$ et $\prec$ est plus stricte que $\prec'$; $m \leq m'$ si $m < m'$ ou m et m' sont identiques. Nous définissons aussi l'opération d'addition de m et m' par $m + m' = (c + c', \prec \wedge \prec')$. Le complément de $m = (c, \prec)$, noté $-m$, est la borne $(-c, \prec)$. Finalement, $min(m, m')$ est égale à m si $m \leq m'$ et à m' sinon.

Dans le reste de cette partie, $V = \{v_1, ..., v_n\}$ dénotera un ensemble de variables à valeurs dans $\mathbb{R}^{\geq 0}$ et $V_0 = V \cup \{v_0\}$ l'ensemble des variables V muni d'une variable fictive v_0 qui vaut 0 tout le temps.

1.2 Automate temporisé étendu

Une autre variante des automates temporisés est les automates temporisés *étendus*. Ces derniers, en plus des variables continues (horloges) utilisent des variables discrètes et des paramètres.

Pour un ensemble d'horloges C, un ensemble de paramètres P et un ensemble de variables V, l'ensemble des contraintes d'horloge $\Phi(C, P, V)$ est défini par la grammaire suivante :

$$\phi := \phi \mid \phi \wedge \phi \mid x \leq f(P, V) \mid f(P, V) \leq x$$

avec x une horloge de C et $f(P, V)$ une expression linéaire de P et V.

Définition 1 (ETIOA) *Un automate temporisé étendu à entrée/sortie (ETIOA) est un 10-uplet $M = (S, s_0, \Sigma, C, P, V, V_0, Pred, Ass, \rightarrow)$ tel que :*

- *S est un ensemble fini des états.*
- *s_0 est l'état initial.*
- *Σ est un alphabet fini d'actions,*
- *C est un ensemble fini d'horloges.*
- *P est un ensemble fini de paramètres.*
- *V est un ensemble fini de variables.*
- *V_0 est un ensemble de valeurs initiales pour les variables de V.*
- *$Pred = \Phi(C, P, V) \cup \tilde{P}[P, V]$, $\tilde{P}[P, V]$ est un ensemble d'inégalités linéaires sur V et P.*
- *$Ass = \{x := 0 \mid x \in C\} \cup \{v := f(P, V) \mid v \in V\}$ est un ensemble de mises à jour sur les horloges et les variables.*
- *$\rightarrow \subseteq S \times Pred \times \Sigma_\tau \times Ass \times S$ est un ensemble de transitions.*

$t = (s, pred, a, ass, s')$ est la transition de l'état s à l'état s' sur l'occurrence du symbole a. $pred \subseteq Pred$ est une contrainte sur C et V et $ass \subseteq Ass$ est un ensemble des mises à jour de C et V.

Un exemple d'ETIOA est donné dans la FIG.1.

- $S = \{s_0, s_1, s_2, s_3\}$ et s_0 est l'état initial.
- $L = \{!a, ?b, !c, ?d\}$, $C = \{x, y\}$, $P = \{\lambda\}$, $V = \{v1\}$ et $V_0 = \{2\}$.
- $Pred = \{y \geq \lambda, x \leq 1, v1 \leq 4\}$.
- $Ass = \{x := 0, y := 0, v1 := v1 + 1\}$.
- La transition de source s_2 et de destination s_3 est : $t = (s_2, \{x \leq 1\}, !c, \{v1 := v1 + 1\}, s_3)$.

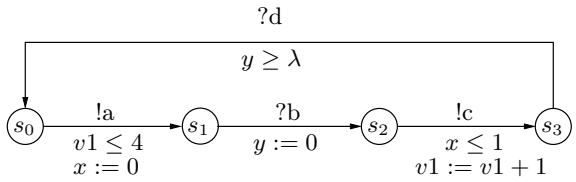

FIG. 1 – Automate temporisé étendu.

1.3 Représentation symbolique des polyèdres : DBMs

Il existe plusieurs structures de données symboliques qui permettent de représenter les polyèdres. Ici nous nous limiterons à la description des DBMs [2] (Difference Bound Matrix) ou "matrice des bornes", la structure de données introduite par David Dill. Pour plus d'informations sur les autres structures, se référer à [2, 1].

Cette représentation est particulièrement intéressante car elle donne lieu à des algorithmes simples et performants pour le calcul de la forme canonique. Par exemple, l'algorithme de Floyd-Warshall de recherche des plus courts chemins dans un graphe valué [3], présenté par la suite, est de complexité $O(n^3)$. Cet algorithme permet à la fois de calculer la forme canonique d'un polyèdre et de tester s'il est vide (en testant la présence des boucles négatives, i.e. de nombres négatifs sur la diagonale).

Définition 2 *Une matrice de bornes (DBM) de dimension n est une matrice carrée* $(n+1)\times(n+1)$*, dont les éléments sont des bornes. Si* $M = (m_{ij})$ *est une DBM, alors* m_{ij} *est l'élément de la ligne* i *et de la colonne* j*.*

L'idée derrière la représentation d'un polyèdre par une DBM $M = (m_{ij})$ est la suivante : un élément m_{ij} représentera la borne supérieure de la différence des variables $v_i - v_j$. Par exemple, la contrainte $v_i - v_j \leq 5$ sera codée par $m_{ij} := (5, \leq)$, tandis que la contrainte $v_i - v_j \geq 2$ sera codée par $m_{ji} := (-2, \leq)$. La ligne et la colonne d'indice 0 sont utilisées pour coder les contraintes sur une seule variable, par exemple $x_i \leq 3$ sera codée par $m_{i0} := (3, \leq)$ et $x \geq 4$ sera codée par $m_{0i} := (-4, \leq)$.

Formellement, considérons un polyèdre Z et $G(Z)$ son graphe de contraintes. Z est représenté par la DBM $M = (m_{ij})$ de dimension n, telle que :

$$\begin{cases} m_{ij} := (l_{ij}, \leq) & ssi\ v_j \xrightarrow{l_{ij}} v_i \text{ est un arc de } G, i \neq j \\ m_{ii} := (0, \leq) \end{cases}$$

Par exemple, si $n = 2$, le polyèdre $v_2 \leq 2 \wedge v_1 \geq 2 \wedge v_1 - v_2 \leq 3$ peut être représenté par la DBM M :

$$M = \begin{array}{c|ccc} & v_0 & v_1 & v_2 \\ v_0 & (0,\leq) & (-2,\leq) & (\infty,\leq) \\ v_1 & (\infty,\leq) & (0,\leq) & (3,\leq) \\ v_2 & (2,\leq) & (\infty,\leq) & (0,\leq) \end{array}$$

Inversement, toute DBM M de dimension n représente un polyèdre Z sur V tel que :

$$Z = \bigwedge_{v_i, v_j \in V_0, i \neq j} (v_i - v_j \leq m_{ij})$$

Par exemple, la DBM

$$M = \begin{array}{c|ccc} & v_0 & v_1 & v_2 \\ v_0 & (0,\leq) & (\infty,\leq) & (-3,\leq) \\ v_1 & (2,\leq) & (0,\leq) & (-1,\leq) \\ v_2 & (\infty,\leq) & (\infty,\leq) & (0,\leq) \end{array}$$

représente le polyèdre $Z = v_1 \leq 2 \wedge v_2 \geq 3 \wedge v_1 - v_2 \leq -1$ (ou encore $v_1 \leq 2 \wedge v_2 \geq 3$).

Coder un polyèdre par une DBM demande alors $O(n^2)$ d'espace mémoire. Plusieurs algorithmes ont été proposés pour réduire l'espace mémoire nécessaire pour représenter un polyèdre [5, 4]. Ces réductions sont basées, en général, sur des transformations du graphe de contraintes.

1.4 Implantation des opérations sur les polyèdres

Nous allons à présent décrire l'implantation des opérations sur les polyèdres en utilisant leur représentation sous forme de DBMs.

Forme_Canonique(matrice $M = (m_{ij})$)
{
  Pour $k = 0$ **à** n **Faire**
    Pour $i = 0$ **à** n **Faire**
      Pour $j = 0$ **à** n **Faire**
        $m_{ij} := min\{m_{ij}, m_{ik} + m_{kj}\}$;
      Si$(m_{ii} < (0, \leq))$ **Alors** retourner $M_\emptyset$;
  retourner M ;
}

FIG. 2 – Mise en forme canonique : algorithme de Floyd-Warshall.

1.4.1 Forme canonique

Il faut noter que deux DBMs différentes peuvent représenter le même polyèdre. La mise sous forme canonique permet d'affirmer que deux DBMs représentent le même polyèdre si elles sont identiques. Cette forme réduite est obtenue en appliquant un algorithme des plus courts chemins. L'algorithme est dû à Floyd-Warshall (FIG.2) permet ce calcul. Si un élément de la diagonale est négatif, alors le polyèdre représenté par cette matrice est vide et l'algorithme retourne la matrice $M_\emptyset$ dont chaque élément est égal à $(0, <)$. Par convention, $M_\emptyset$ représentera la forme canonique du polyèdre vide.

Par la suite, la forme canonique d'une DBM M sera notée $cf(M)$. Si $M = cf(M)$, alors M est dite *canonique*. Soient $M = (m_{ij})$ et $M' = (m'_{ij})$ les DBMs canoniques de dimension n représentant respectivement les polyèdres Z et Z'.

1.4.2 Test d'inclusion : $Z \subseteq Z'$

$$Z \subseteq Z' \; ssi \; \forall 0 \leq i, j \leq n, \; m_{ij} \leq m'_{ij}$$

Ainsi, tester l'inclusion de deux polyèdres se fait en temps linéaire (taille de la matrice).

1.4.3 Intersection : $Z \cap Z'$

$Z \cap Z'$ est représenté par la DBM $M'' = (m''_{ij})$ telle que :

$$m'' = min(m_{ij}, m'_{ij})$$

Notons qu'il peut arriver que M'' ne soit pas canonique.

1.4.4 Successeur : $Z^{\uparrow}$

$Z^{\uparrow}$ est représenté par la DBM canonique $M' = (m'_{ij})$ telle que :

$$m'_{ij} = \begin{cases} (\infty, \leq) & si\ j = 0 \\ m_{ij} & sinon \end{cases}$$

1.4.5 Prédécesseur : $Z^{\downarrow}$

$Z^{\uparrow}$ est représenté par la DBM (pas forcement canonique) $M' = (m'_{ij})$ telle que :

$$m'_{ij} = \begin{cases} (0, \leq) & si\ i = 0 \\ m_{ij} & sinon \end{cases}$$

1.4.6 Successeur après ré-initialisation : $Z[X := 0]$

Soit $X \subseteq V$ un ensemble de variables à remettre à zéro. Soit la fonction totale $\lambda : V_0 \mapsto V_0$ définie par : pour toute $v_i \in V_0$,

$$\lambda(v_i) = \begin{cases} v_i & si\ v_i \notin X \\ v_0 & sinon \end{cases}$$

Notons d'abord que la remise à zéro d'une horloge $v_i \in X$ est équivalent à remplacer v_i par $v_0 = \lambda(v_i)$. Maintenant, lorsqu'on remplace v_i par v_j, alors v_i et v_j deviennent égales et toutes les contraintes sur v_j deviennent des contraintes sur v_i. Ainsi $Z[X := 0]$ est représenté par la DBM (pas forcement canonique) $M' = (m'_{ij})$ telle que pour toutes $v_i, v_j \in V_0$,

$$si\ \lambda(v_i) = v_j\ alors\ ligne_i(M') = ligne_j(M)\ et\ colonne_i(M') = colonne_j(M)$$

où $ligne_i(M)$ (resp. $colonne_i(M)$) représente la ligne (resp. colonne) de M d'indice i. Ainsi, M' est obtenue par des remplacements de lignes et de colonnes.

1.4.7 Prédécesseur après ré-initialisation : $[X := 0]Z$

Rappelons qu'une variable v_i de $[X := 0]Z$ est remplacée dans Z par $\lambda(v_i)$. Maintenant, supposons qu'on a deux contraintes $v_k - v_l \leq l_{kl}$ et $v_r - v_s \leq v_{rs}$ et qu'on remplace (i) v_k et v_r par v_i et (ii) v_l et v_s par v_j. On obtient alors les contraintes $v_i - v_j \leq v_{kl}$ et $v_i - v_j \leq v_{rs}$ et ainsi $v_i - v_j \leq min(v_{kl}, v_{rs})$. Ainsi, $[X := 0]Z$ est représenté par la DBM $M' = (m'_{ij})$ telle que pour toute $v_i \in V_0$:

$$m'_{ij} = min\{m_{kl} \,|\, \lambda(v_k) = v_i \wedge \lambda(v_l) = v_j\}$$

1.4.8 c-clôture : $close(Z, c)$

$close(Z, c)$ est représenté par la DBM $M' = (m'_{ij})$ telle que pour $0 \leq i \neq j \leq n$:

$$m'_{ij} = \begin{cases} (0, \leq) & si\ m_{ij} > (c, \leq) \\ (-c, \leq) & si\ m_{ij} + (c, \leq) < (0, \leq) \\ m_{ij} & sinon \end{cases}$$

Ainsi, une borne supérieure telle que $v_i \leq c'$, avec $c' > c$, est remplacée par $v_i \leq \infty$ et une borne inférieure telle que $v_i \geq c'$, avec $c' > c$, est remplacée par $v_i \geq c$. Les autres bornes restent inchangées.

1.4.9 Extraction de valuations de bornes

Le calcul des valuations de bornes (minimales et maximales) est direct. Pour tout $k \in [0, n]$:

1. La valuation $\nu_k^M(Z)$ est définie par :
   - Si $k = 0$ alors $\nu_k^M(Z)(v_i) = m_{i0}$.
   - Sinon $\nu_k^M(Z)(v_i) = -m_{0k} + m_{ik}$, et $\nu_k^M(Z)(v_k) = -m_{0k}$

   pour tout $i \in [1, n]$, $i \neq k$.
2. La valuation $\nu_k^m(Z)$ est définie par :
   - Si $k = 0$ alors $\nu_k^M(Z)(v_i) = -m_{0i}$.
   - Sinon $\nu_k^m(Z)(v_i) = m_{k0} - m_{ki}$, et $\nu_k^m(Z)(v_k) = m_{k0}$

   pour tout $i \in [1, n]$, $i \neq k$.

2 Trace symbolique

Dans cette section, nous étudierons l'exécutablité d'un chemin dans un ETIOA à travers la définition de la trace symbolique.

2.1 Préliminaires

Considérons un ETIOA $M = (S, s_0, L, C, P, V, V_0, Pred, Ass, \rightarrow)$ et un chemin $\rho = t_1...t_n$ de M de l'état initial tel que $t_i = (s_{i-1}, pred_i, a_i, ass_i, s_i)$ pour tout $i \in [1, n]$.

Définition 3 *ρ est dit exécutable (ou faisable) s'il existe des valeurs pour les paramètres P et les variables V et des instants de tir des transitions t_i tels que les prédicats $pred_i$ soient vérifiés, pour tout $i \in [1, n]$.* □

Considérons le chemin ρ défini par :

$$\rho \ : \ s_0 \xrightarrow{x \leq 2/?a/y:=0} s_1 \xrightarrow{x \geq p \wedge y \leq 2/!b/v=p+1} s_2 \xrightarrow{v \geq 5/?c} s_3$$

avec p un paramètre et v une variable. On peut remarquer que :

- Le prédicat $v \geq 5$ de la transition de source s_2 et de destination s_3 est vérifié si et seulement si $p \geq 4$. En effet, la dernière valeur de v avant cette transition correspond à l'affectation $v = p + 1$.
- Le prédicat $x \geq p \wedge y \leq 2$ de la transition de source s_2 et de destination s_3 est vérifié si et seulement si $p \leq 4$. En effet, le temps d'attente maximale dans l'état s_1 correspond à la borne de $y \leq 2$ du fait que y est remise à zéro dans la transition entrante à s_1, ce qui implique que la valeur de x est inférieure à 4.

En conclusion, la seule valeur de p pour que le chemin ρ soit exécutable est 4. Une trace temporelle dans ce cas est $(?a, 2).(!b, 4).(?c, 5)$. □

Comme nous venons de le voir dans cet exemple, pour décider de l'exécutabilité d'un chemin, on est amené à résoudre un système de contraintes associé à ρ. Dans la section qui suit, nous allons décrire comme construire le système de contraintes associé à ρ à travers le calcul de la trace symbolique.

2.2 Trace symbolique

Tout d'abord, rappelons que pour un ETIOA $M = (S, s_0, \Sigma, C, P, V, V_0, Pred, Ass, \rightarrow)$, l'ensemble des prédicats $Pred$ et l'ensemble des mises à jour Ass sont définis par [1] :

- $Pred = \Phi(C, P, V) \cup \tilde{P}[P, V]$ tel que :

[1] Pour plus de détails, voir la section 1.2

1. $\tilde{P}[P,V]$ est un ensemble d'inégalités linéaires sur V et P.
2. $\Phi(C,P,V)$ est défini par :

$$\phi := \phi_1 \mid \phi_2 \mid \phi_1 \wedge \phi_2,\ \phi_1 := x \leq f(P,V),\ \phi_2 := f(P,V) \leq x$$

avec x une horloge de C et $f(P,V)$ une expression linéaire de P et V.

– $Ass = \{x := 0 \mid x \in C\} \cup \{v := f(P,V) \mid v \in V\}$

Par la suite, nous supposons que pour l'ETIOA $M = (S, s_0, \Sigma, C, P, V, V_0, Pred, Ass, \rightarrow)$ et le chemin $\rho = t_1...t_n$ de M partant de l'état initial, les ensembles C, V et V_0 sont définis par : $C = \{c_1, ..., c_k\}$, $V = \{v_1, ..., v_m\}$ et $V_0 = \{v_{0_1}, ..., v_{0_m}\}$.

Pour pouvoir décider de l'exécutabilité de ρ, la première étape consiste à remplacer les occurrences des variables par leurs valeurs qui peuvent changer d'une transition à l'autre. Dans la deuxième étape, les différentes horloges sont remplacées, dans chaque transition, par leurs instants de tir. Le chemin ne contenant que des paramètres et des instants de tir de chaque transition ainsi obtenu est appelé *la trace symbolique de* ρ. La procédure **SymbolicTrace** de la FIG.3 calcule la trace symbolique associée à un chemin ρ.

Procédure SymbolicTrace :
Entrée : Un chemin initial $\rho = t_1...t_n$, $t_i = (s_{i-1}, a, pred_i, ass_i, s_i)$, d'un ETIOA $M = (S, s_0, \Sigma, C, P, V, V_0, Pred, Ass, \rightarrow)$.
Sortie : Une trace symbolique $\rho' = t'_1...t'_n$, $t'_i = (s_{i-1}, a, pred'_i, \emptyset, s_i)$.
Variables Temporaires : Deux vecteurs de contexte $V1$ de taille m et $V2$ de taille k.
Début

/*Initialisation des vecteurs */
1. **Pour** $i := 1$ **à** m **Faire**
2. $V1[i] := v_{0_i}$

3. **Pour** $i := 1$ **à** k **Faire**
4. $V2[i] := 0$

/*Mise à jour des vecteurs */
5. **Pour** $i := 1$ **à** n **Faire**
6. $pred'_i := UpdatePredicates(pred_i, V1, V2, i)$;
7. $UpdateContext(ass_i, V1, V2, i)$;

Fin

FIG. 3 – Procédure SymbolicTrace.

L'entrée de l'algorithme est un chemin ρ de M. La sortie est un chemin ρ' dont le prédicat, de chaque transition, est en fonction des paramètres P, des valeurs V_0 et de l'ensemble $\{h_1, ..., h_n\}$ correspondant aux instants de franchissement des transitions selon une horloge globale h. La procédure utilise deux vecteurs $V1$ et $V2$. $V1$ contient les valeurs courantes des variables V dans chaque étape de la procédure. Ces valeurs peuvent être en fonction des paramètres P. $V2$ est un vecteur d'entiers : $V2[q]$ stocke l'indice de la dernière transition où l'horloge $c_q \in C$ a été réinitialisée. *SymbolicTrace* se compose d'une phase d'initialisation des vecteurs (lignes 1-4) et une phase de mise à jour des vecteurs (lignes 5-7).

Dans la phase d'initialisation, la valeur courante de chaque variable v_i correspond à sa valeur initiale v_{0i} dans V_0 : $V1[i] := v_{0i}$. De plus, toutes les horloges sont initialisées dans l'état initial : $V2[i] := 0$.

Procédure UpdatePredicates :
Entrée : Un prédicat $pred = p_1(c_1, ..., c_k, P, v_1, ..., v_m) \wedge p_2(P, v_1, ..., v_m)$, un indice i, $V1$ et $V2$ deux vecteurs.
Sortie : Un prédicat $predUpdated$.
Début

1. $predUpdate := p_1(h_i - h_{V2[1]}, ..., h_i - h_{V2[k]}, P, V1[1], ..., V[m]) \wedge p_2(P, V1[1], ..., V[m])$

Fin

FIG. 4 – Procédure UpdatePredicates.

Dans la phase de mise à jour, le prédicat $pred_i$ de la transition courante t_i (ligne 6) est utilisé pour calculer le nouveau prédicat $pred'_i$ de t'_i. $Pred_i$ est obtenu en remplaçant les variables et les horloges de $pred_i$ par leurs valeurs courantes dans $V1$ et $V2$. Ce remplacement est réalisé par l'appel à la procédure **UpdatePredicates**. L'affectation ass_i de t_i est alors utilisée pour calculer les nouvelles valeurs des variables de $V1$ ainsi que les nouvelles remises à jour des horloges dans $V2$, après le franchissement de t_i. Cette mise à jour est réalisée à travers l'appel à la procédure **UpdateContext** (ligne 7).

La procédure **UpdatePredicates** est illustrée dans la FIG.4. Rappelons qu'un prédicat s'écrit comme une conjonction d'une contrainte d'horloges $p_1(c_1, ..., c_k, P, v_1, ..., v_m)$ et une contrainte de variables $p_2(P, v_1, ..., v_m)$. Dans la contrainte d'horloges, toute horloge c_p réinitialisée pour la dernière fois dans la transition $t_{V2[p]}$ est remplacée par $h_i - h_{V2[p]}$, où i est l'indice de l'étape courante (ligne 1, partie $p_1(h_i - h_{V2[1]}, ..., h_i - h_{V2[k]}, P, ...)$). En effet, $h_i - h_{V2[p]}$ correspond au temps écoulé depuis la dernière ré-initialisation de c_p. Dans la contrainte de variable, toute variable v_j est remplacée par sa valeur courante dans l'étape i. Cette valeur vaut $V1[j]$ (ligne 1, partie $p_1(..., P, V1[1], ..., V[m])$. Cette démarche est aussi appliquée à $p_2(P, v_1, ..., v_m)$.

Procédure UpdateContext :
Entrée : Une affectation ass, un indice i et deux vecteurs $V1$ et $V2$.
Sortie : Deux vecteurs $V1$ et $V2$.
Début

1. **Pour** $j := 1$ **à** m **Faire**
2. **Si** $v_j := f(v_1, .., v_m, P) \in ass$ **Alors** $V1[j] := f(V[1], .., V[m], P)$

3. **Pour** $j := 1$ **à** m **Faire**
4. **Si** $c_j := 0 \in ass$ **Alors** $V2[j] := i$

Fin

FIG. 5 – Procédure UpdateContext.

La procédure **UpdateContext** est illustrée dans la FIG.5. Elle met à jour les valeurs courantes des variables dans $V1$ en tenant compte des nouvelle affectations (lignes 1-2) et les ré-initialisations des horloges en positionnant les champs de $V2$ à l'indice de l'étape courante (lignes 3-4).

Supposons que $\rho' = t'_1...t'_n$, $t'_i = (s_{i-1}, a, pred'_i, \emptyset, s_i)$ pour $i \in [1, n]$, est la trace symbolique associée au chemin ρ. Alors ρ est exécutable si et seulement si le prédicat $pred' = \bigwedge_{i\in[1,n]} pred'_i$ est vrai pour certaines valeurs des paramètres P et des instants de tirs $(h_i)_{i\in[1,n]}$. □

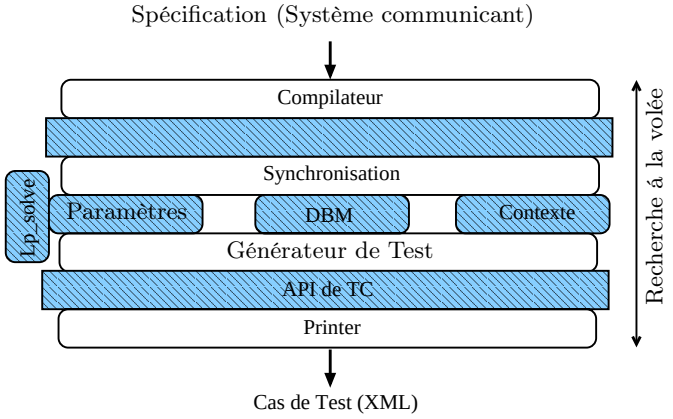

FIG. 6 – Architecture logicielle

3 L'outil TGSE

TGSE pour génération de test, simulation et émulation (Test génération, simulation and emulation) est un ensemble de logiciels regroupant différentes activités du test que nous avons développé au LaBRI. Il comporte un générateur de séquences de test, pour les systèmes temporisés et étendus, basé sur 1) le modèle CS, 2) le critère de couverture exprimé sous forme de coloriage et 3) la méthodologie *gga*. Il comporte aussi un simulateur, à travers la plate-forme Califé, permettant l'exécution graphique d'une séquence générée par TGSE. Finalement, l'émulateur temps réel de TGSE permet l'exécution réelle des différents systèmes. L'émulateur joue aussi le rôle d'un générateur de code.

Par la suite, nous allons décrire les principes et les algorithmes mis en oeuvre dans le générateur de test de TGSE.

3.1 Architecture du générateur de test de TGSE

L'architecture du générateur de test de TGSE est illustrée dans la FIG.6. Elle comporte plusieurs fonctionnalités réalisées par différents modules.

3.1.1 Module Compilation

L'entrée de TGSE est une description du système sous une syntaxe simple permettant de définir les différentes entités du système. Dans un souci de réutilisation des composantes, chaque entité est définie dans un fichier séparé. Un fichier système décrit les chemins d'accès aux entités ainsi que les vecteurs de synchronisation. Dans la version actuelle, TGSE ne supporte qu'une topologie statique. La syntaxe de description des entités est la suivante. Lorsqu'une entité définit un état à couvrir, la variable final_states est positionnée à l'indice de cet état. Ceci est équivalent à colorier cet état par la couleur *rouge*. Une transition est décrite par six champs :

1. (numéro_état,étiquette),
2. événement (*nop* pour une action interne),
3. prédicat_horloges (# pour un prédicat vrai),
4. prédicat_variables (# pour un prédicat vrai),
5. mise_à_jour_horloges (de la forme $x := 0$ et # pour l'absence de reset)
6. mise_à_jour_variables (de la forme $v := v + p$ et # pour l'absence d'affectation).

Finalement, le module de compilation traduit le système sous test en structures de données C utilisées par le module de synchronisation. Ci-dessous, des fichiers d'entrées du protocole CSMA/CD composé d'un bus, deux émetteurs et un objectif de test :

```
****************************************** P_AUTO Sender { nb_states = 4
initial_state = 10 clocks = x


(10,Wait), (10,Wait), ?CD, #, #, x:=0, # (10,Wait), (11,Transmit), !begin, #,
#, x:=0, # (10,Wait), (12,Retry), ?busy, #, #, x:=0, # (10,Wait), (12,Retry),
?CD, #, #, x:=0, # (11,Transmit), (12,Retry), ?CD, x[0,Sig[, #, x:=0,#
(11,Transmit), (11,Transmit), ?busy, #, #, #, # (11,Transmit), (13,Finish),
!end, #, x[lambda,lambda], #, # (12,Retry), (11,Transmit), !begin, x[0,2*4], #,
x:=0,# (12,Retry), (12,Retry), ?CD, x[0,2*Sig], #, x:=0,# (12,Retry),
(12,Retry), ?busy, x[0,2*Sig], #, x:=0,# (13,Finish), (10,Wait), nop, #, #,
x:=0, # } *****************Sender.aut******************


****************************************** TESTER TP { nb_states = 2
initial_state = 10 final_states = 11


(10,toto), (11,titi), ?busy, #, #, #, # }
*********************Tp.aut*******************


****************************************** P_AUTO Bus { nb_states = 3
initial_state = 10 clocks = y


(10,Idle), (11,Active), ?begin, #, #,y:=0, # (11,Active), (10,Idle), ?end, #,
#, y:=0, # (11,Active), (12,Collision), ?begin, y[0,Sig[, #, y:=0,#
(11,Active), (11,Idle), !busy, y[Sig,+inf], #, #,# (12,Collision), (10,Idle),
!CD, #, #, #, # } *****************Bus.aut**********************


****************************************** SYSTEM CSMA_CD_2 {
nb_automatons:3 nb_vectors: 8 clocks : S parameters = Sig Lambda
automaton_files:[../Example/Sender.aut ../Example/Bus.aut
                 ../Example/Sender.aut ]
tester_file: Tp.aut


VECTORS

<?busy ,!busy ,?busy ,?busy> <* ,?end ,!end ,*> <* ,?begin ,!begin ,!begin>
<!end ,?end ,* ,*> <!begin ,?begin ,* ,!begin> <* ,?begin ,!begin ,*> <!begin
,?begin ,* ,*> <?CD ,!CD ,?CD ,*> }
******************CSMA-CD.sys*****************
```

3.1.2 Module Synchronisation

Ce module implémente les fonctionnalités relatives au calcul de la sémantique du système communicant. Ce calcul est réalisé à la volée. Il implémente l'API $SynchronizationOnVectors()$ qui permet de choisir une synchronisation possible dans l'état courant du système, en se basant sur les vecteurs de synchronisation. Cette API retourne une structure de donnée "Element" qui contient les états d'arrivée, les transitions choisies, ainsi que le contexte relatif à cette synchronisation. Le choix des transitions, des vecteurs de synchronisation ainsi que des automates qui les réalisent

est paramétré par des variables pour chaque donnée. Les valeurs possibles de ces variables sont RANDOM, pour un choix aléatoire et FIFO pour un respect de l'ordre d'apparition dans la définition du système. La construction de la sémantique est paramétrée par le nombre maximal d'apparitions de la même transition dans un chemin. Ainsi, à chaque transition, on associe une variable *Lock* qui contient le nombre maximal d'apparitions de cette dernière dans un cas de test généré. Ce module implémente aussi les APIs *getInitialStates* qui retourne l'état initial de $\zeta(S)$ et *getSuccessors* qui retourne les successeurs d'un état courant de $\zeta(S)$.

Sous-module Contexte. Il implémente des fonctionnalités relatives à la mise à jour des variables, des prédicats, de la trace symbolique (les APIs $UpdateContext$, $UpdatePredicates$, $SymbolicTrace$,...).

Sous-module DBM. C'est une librairie qui implémente les opérations sur polyèdres et le calcul des diagnostics de bornes. Ses principales APIs utilisables par le module de génération sont *post*(), *pred*(), $TimedDiagnostics()$.

Sous-module Paramètres. Pour une trace symbolique, ce sous-module implémente les APIs $checkSymbolicTrace$ et $getParameterValues$. $checkSymbolicTrace$ construit le système de contraintes associé à la trace symbolique, interagit avec l'outil de programmation linéaire *lp_solve* et détermine si le système admet une solution. $getParameterValues$ instancie les valeurs des paramètres dans le cas où le système de contraintes admet une solution.

3.1.3 Module Générateur de Test

C'est le coeur de l'outil. Il implémente un algorithme *gga* de recherche en profondeur à la volée de l'automate sémantique du système. L'algorithme *gga* calcule d'une manière aléatoire et uniforme (qui dépend des paramètres d'entrée) un chemin de l'état initial qui se termine dans un état à couvrir. Le chemin ainsi généré est décoré par les différents verdicts. La section suivante présente en détail *gga*.

Sous-module API pour TC. Ce sous-module implémente spécialement l'API $TC()$ qui décore un chemin obtenu par *gga* par les différents verdicts.

3.1.4 Module Printer

La génération de cas de test se termine par l'API $writeTrace()$ du module Printer. $writeTrace()$ transforme les structures de données du cas de test généré en format XML. Un exemple de la sortie de TGSE est ci-dessus.

```
<?xml version="1.0" encoding="ISO-8859-1"?> <!DOCTYPE Trace SYSTEM "trace.dtd">
<Trace Label="CSMA_CD_2">
    <StateSync>

        <State Index="1" Component="Sender">
            <Loc Label="Wait"/>
            <Bounds/>
        </State>
        <State Index="2" Component="Bus">
            <Loc Label="Idle"/>
            <Bounds/>
        </State>
        <State Index="3" Component="Sender">
            <Loc Label="Wait"/>
            <Bounds/>
        </State>
```

```
        <State Index="4" Component="TP">
            <Loc Label="init"/>
            <Bounds/>
        </State>
        <Diag/>
        <Glob/>

    </StateSync>
    <Action Label="Action">
        <Comp Label="epsilon" Index="0"/>
        <Comp Label="begin" Type="Wait" Index="1"/>
        <Comp Label="begin" Type="Send" Index="2"/>
        <Comp Label="epsilon" Index="3"/>
    </Action>
    <StateSync>

        <State Index="1" Component="Sender">
            <Loc Label="Wait"/>
            <Bounds/>
        </State>
        <State Index="2" Component="Bus">
            <Loc Label="Active"/>
            <Bounds/>
        </State>
        <State Index="3" Component="Sender">
            <Loc Label="Transmit"/>
            <Bounds/>
        </State>
        <State Index="4" Component="TP">
            <Loc Label="init"/>
            <Bounds/>
        </State>
        <Diag/>
        <Glob/>

    </StateSync>
    <Action Label="Action">
        <Comp Label="busy" Type="Wait" Index="0"/>
        <Comp Label="busy" Type="Send" Index="1"/>
        <Comp Label="busy" Type="Wait" Index="2"/>
        <Comp Label="busy" Type="Wait" Index="3"/>
    </Action>
    <StateSync>

        <State Index="1" Component="Sender">
            <Loc Label="Transmit"/>
            <Bounds/>
        </State>
        <State Index="2" Component="Bus">
            <Loc Label="Collision"/>
            <Bounds/>
        </State>
        <State Index="3" Component="Sender">
            <Loc Label="Transmit"/>
```

```
            <Bounds/>
        </State>
        <State Index="4" Component="TP">
            <Loc Label="Final"/>
            <Bounds/>
        </State>
        <Diag/>
        <Glob/>

    </StateSync>
</Trace>
```

Signalons finalement que dans le cas où aucun état à couvrir n'est accessible dans le parcours courant, l'algorithme *gga* est relancé automatiquement pour une nouvelle tentative (le lancement est paramétrable). De plus, il est possible de générer un cas de test minimal en nombre de transitions pour un nombre de tentatives donné.

4 Algorithme de génération gga

Dans sa version actuelle, TGSE ne supporte pas tout le cadre formel présenté dans ce document et qu'il est en cours d'extension. Nous ne présentons ici que la partie supportée par TGSE.

Hypothèses. L'algorithme *gga* implémenté considère un système communicant sous test (S, col) tel que :

1. S est une topologie statique,
2. col est un coloriage distribué à trois couleurs : rouge, noir et bleu tel que $friend(rouge) = \{noir\}$. Le rouge définit les états à couvrir (*Accept*) et le bleu les états à éviter (*Reject*).

Ainsi, un état d'un automate est modélisé par un couple $(s, couleur)$.

Structures de données. Nous utilisons les structures de données suivantes :
- *States* : un (n+1)-uplet $(s_{tr}, s_1, ..., s_n)$.
- *Context* : une structure contenant deux champs $Variable$ et *Reset* :
  1. $Variable$: un vecteur contenant les valeurs des variables discrètes.
  2. *Reset* : un vecteur d'entiers contenant, à chaque étape i, les indices de transitions où les horloges ont été remises à zéro pour la dernière fois.
- $Transitions$: un tableau de $n + 1$ champs. $Transition[i]$ correspond à un pointeur sur la transition courante de l'automate M_i de $Comp(S)$.
- *Element* : une structure de donnée contenant trois champs : un champ de type *States*, un champ de type *Context* et un champ de type $Transitions$.
- *Path* : une pile d'éléments. Elle est gérée par les opérations "push", "top" et "pop".

Description de gga. L'algorithme de génération *gga*, appliqué à un CS S, réalise une recherche en profondeur de $\zeta(S)$. Durant la traversée de $\zeta(S)$, *gga* calcule un chemin symbolique (SymbolicPath) dans le cas d'une spécification temporisée, et une trace symbolique (SymbolicTrace) dans le cas d'une spécification étendue (type =Extended). Lorsqu'un état à couvrir est rencontré, la traversée s'achève par l'appel à $SymbolicPath$ ou à $SymbolicTrace$. Dans ce cas, l'appel à l'API $TC()$ permet de décorer le chemin ainsi choisi par les différents verdicts. La FIG.7 illustre le pseudo-code en langage C de *gga*.

```
Fonction gga() :
  Begin
      States := getInitialStates(), SymbolicState = ∅, Element := NULL, Path := ∅,
      SymbolicTrace := ∅, SymbolicPath := ∅

      Do

          Element := SynchronizationOnVectors(States) ;
          push(Element,Path)

          If (Element ≠ NULL) Then

              If type = Extended Then
                  push(SymbolicTrace, SymbolicTrace(Element))
                  If checkSymbolicTrace(SymbolicTrace) = false Then
                      pop(SymbolicTrace)
                      pop(Path)

              Else
                  SymbolicState = post(SymbolicTrace,Element)
                  If SymbolicState ≠ ∅ Then
                      push(SymbolicTrace,SymbolicState)
                  Else
                      pop(Path)

              States := getSuccessors(Element) ;

          Else
              pop(Path)
              States := getSuccessors(top(Path))

          If AcceptStates(States) = true Then
              exit(EXIT_SUCCESS)

      While Path ≠ ∅

      If Path ≠ ∅
          If type = Extended Then
              getParameterValues(SymbolicTrace)
          Else
              TimedDiagnostics(SymbolicPath)
          TC(Path) ;

  End
```

FIG. 7 – Algorithme de génération *gga*.

Références

[1] P. Bouyer. Untameable timed automata !. *In Proc. 20th Ann. Symp. STACS'2003*, vol 2607 of LNCS, 620-631, Springer. Feb 2004, Berlin, Germany, 2003.

[2] D. Dill. Timing assumptions and verification of finite-state concurrent systems. In J. Sifakis, editor, *Proceeding of first CAV*, number 407 in LNCS, Springer-Verlag, France, 1989.

[3] Robert W. Floyd. Algorithm 97 (shortest path). *Communications of the ACM*,18(3) :165-172, 1964.

[4] K. G. Larsen, F. Larsson, P. Pettersson and Wang Yi. Efficient verification of real-time systems : compact data structure and state-space reduction. *In Proc 18 th IEEE Real-Time Systems Symposium, RTSS'97*, IEEE Computer Society Press, San Francisco, California, USA, December 1997.

[5] C. Daws and S. Yovine. Reducing the number of clock variables of timed automata. *In Proc 1996 IEEE Real-Time Systems Symposium, RTSS'96*, IEEE Computer Society Press, Washington DC, USA, December 1996.