Utilisation de Diagrammes de Décision de Données pour la vérification fonctionnelle de systèmes matériels

Vincent Beaudenon, Emmanuelle Encrenaz
Laboratoire d'Informatique de Paris VI,
Architecture des Systèmes Intégrés et Micro-électronique
Université Pierre et Marie Curie (Paris VI), CNRS UMR 7606
12, rue Cuvier,
75252 PARIS cedex 05.
{Vincent.Beaudenon, Emmanuelle.Encrenaz}@lip6.fr

Résumé : Dans cet article nous montrons comment vérifier des propriétés CTL sur des systèmes décrits en langage ProMeLa par vérification de modèle symbolique. La représentation symbolique est basée sur les Diagrammes de décision de données (DDD), qui sont des arbres n-aires partagés. Après avoir décrit les principaux composants nécessaires à la vérification des programmes ProMeLa, nous comparons les performances de notre approche avec celle de model-checker SPIN en mettant en évidence les classes de systèmes pour lesquels SPIN est plus performant, et celles pour lesquelles l'outil que nous présentons a de meilleurs résultats.

Mots-clés : Systèmes Matériels, Preuve Formelle, Diagrammes de Décision de Données, ProMeLa, SPIN

1 Introduction

La vérification formelle consiste à définir si un système donné satisfait une propriété temporelle. Deux types de résultats peuvent être attendus : Dans un premier cas, le système ne satisfait pas la propriété voulue et il est alors utile de fournir des contre-exemples afin de procéder à des corrections ; que ce soit sur le système s'il est défectueux ou sur la propriété à vérifier si elle n'est pas pertinente. Dans un second cas, le système satisfait la propriété. Un tel résultat ne peut être obtenu qu'après avoir parcouru la totalité des scenarii envisageables, ce qui induit une utilisation importante des ressources matérielles.

Les méthodes de simulation permettent d'obtenir des résultats partiels sur un sous ensemble fini des exécutions possibles du système à vérifier. En revanche, s'il est possible, par des méthodes exhaustives de comprendre et analyser formellement les comportements de modules simples qui composent un système complexe, leur articulation induit, dans la majorité des cas, une explosion combinatoire.

Les ressources mémoire nécessaires croissent avec le nombre d'états atteints par le système et la quantité d'informations qu'ils contiennent. On cherche à limiter ces deux facteurs en conférant au système un modèle abstrait. Ce dernier doit permettre de réduire la quantité d'informations vis à vis de la propriété à vérifier, et de restreindre l'ensemble des exécutions du système tout en préservant l'exhaustivité de cas pertinents pour la propriété à vérifier.

Nous avons choisi un niveau d'abstraction pouvant se prêter à la synthèse de haut niveau [Hommais et al., 2001] et aux applications industrielles [de Kock et al., 2000] de cette dernière.

Le système à implanter et à vérifier est modélisé par un ensemble fini de processus concurrents, asynchrones, communicant par canaux bornés ou variables partagées et supportent l'attente simultanée sur plusieurs canaux différents. De tels systèmes peuvent être décrits dans le langage ProMeLa et analysés à l'aide de l'outil SPIN [Holzmann, 1997b]. SPIN propose le validation de propriétés de sûreté et de logique temporelle linéaire (LTL)[Pnueli, 1977].

Pour ce faire, le model-checker tente de synchroniser l'automate du système avec un automate permettant de reconnaitre des séquences infinies validant (ou invalidant) une propriété LTL. Ce procédé suppose de conserver les états explorés pour limiter la durée de l'analyse et accomplir la synchronisation, ce qui est extrêmement coûteux en ressources mémoire.

L'outil SPIN a permis la vérification effective de protocoles de communication et de programmes C [Holzmann, 1997b]. Il a été pourvu de nombreuses améliorations pour, entre autres, limiter l'espace de recherche [Holzmann and Peled, 1994, Peled, 1994], ou diminuer les ressources matérielles nécessaires [Holzmann, 1997a]. Il reste néanmoins limité à l'analyse de systèmes représentables sur 10^6 à 10^7 états maximum. De plus, l'appréhension de systèmes réels est conditionnée par les connaissances de l'utilisateur quant aux méthodes mises en œuvre par SPIN [Beaudenon et al., 2003]

Par ailleurs les méthodes de *vérification symbolique* permettent de vérifier des propriétés sur des systèmes de plus grande taille (de l'ordre de 10^{20} états) [Burch et al., 1990]. Ces méthodes consistent à manipuler états et transitions par *ensembles* plutôt qu'indi-

viduellement et se prêtent particulièrement bien à la vérification de formules de logique temporelle arborescente CTL. Les Outils VIS [Brayton et al., 1996] et SMV [McMillan, 1993] implantent ces techniques de vérification en se basant sur une représentation symbolique fondée sur les Diagrammes de Décision Binaires (BDD, [Bryant, 1986]).

Un BDD permet de repésenter une fonction booléenne. C'est un arbre dont chaque nœud représente une variable binaire et chaque arc une de ses deux valuations. Ces variables sont ordonnées dans l'arbre. La force des BDD réside dans sa représentation canonique et compacte. Malheureusement, une telle représentation nous ramène à la construction bit à bit des données que l'on rencontre dans le matériel et par là-même des modules dont nous voulions précisément abstraire le comportement. D'autre part elle nous oblige à borner l'ensemble des informations définissant un système.

Une comparaison entre SPIN et l'outil VIS, a été proposée pour le protocole ATM [Peng et al., 1999]. VIS est un model-checker symbolique de propriétés CTL sur des réseaux de processus *synchrones* communicant par *signaux*. Dans cette étude comparative, les modèles les plus adaptés pour chacun des outils ont étés utilisés. Il ressort que SPIN est plus rapide et plus adapté aux systèmes de petite taille. En revanche, VIS exploite la mémoire plus efficacement mais ne dispose pas des facilités de simulation de SPIN.

Notre approche consiste à appliquer des méthodes de vérification symbolique sur des systèmes décrits en ProMeLa en utilisant une structure de données ayant les bonnes propriétés de canonicité et de compacité des BDD mais permettant la manipulation de données entières et dynamiques.

Dans cette optique, bon nombre de structures inspirées des BDD ont été créées [Lai et al., 1996, Miner and Ciardo, 1999, Bryant and Chen, 1994, Ciardo et al., 1999]. Parmi ceux-ci, les Diagrammes de Décision de Données (DDD, [Couvreur et al., 2002]), se distinguent en deux points principaux : Les données représentées sont bornées mais n'ont aucun domaine de définition *a priori* ; leur recensement le long des chemins de l'arc n'est soumis à aucun ordre particulier ; et une variable peut apparaître plusieurs fois de la racine jusqu'aux feuilles, la taille du chemin n'étant pas fixée. Cette souplesse dans la représentation d'un ensemble de données étend considérablement les domaines d'application des DDD. Elle permet, par exemple de représenter des systèmes dynamiques. D'autre part, outre les opérations ensemblistes classiques (union, intersection, concaténation...), elle est munie de mécanismes de parcours appelés *homomorphismes inductifs*. Ces derniers permettent de procéder efficacement à des modifications locales sur un DDD.

Après avoir présenté les principes de la vérification symbolique de formules CTL, nous aborderons le formalisme des DDD et des homomorphismes, puis nous montrons comment nous les avons mis en œuvre pour la vérification de propriétés CTL sur des systèmes écrits en ProMeLa. Enfin nous évaluons la méthode en la comparant aux performances de l'outil SPIN pour une recherche d'accessibilité et la validation de propriétés CTL, afin de mettre en évidence les classes de systèmes sur lesquelles SPIN et notre approche symbolique sur DDD se distinguent.

2 La logique Temporelle CTL

2.1 Syntaxe et Sémantique

La logique CTL (Computation Tree Logic) a été proposée pour spécifier des propriétés temporelles de systèmes concurrents finis [Clarke et al., 1986]. À la logique propositionnelle classique, elle adjoint des opérateurs temporels auxquels sont associés des quantificateurs de chemin.

La sémantique d'une formule CTL est définie sur une *structure de Kripke*.

Définition 1 (Structure de Kripke) *Une structure de Kripke est un tuple $M = (AP, S, L, R, S_0)$ où*
- *AP est un ensemble fini de propositions atomiques.*
- *S est un ensemble fini d'états.*
- *$L : S \to 2^P$ est une fonction associant à chaque état un ensemble de propositions atomiques.*
- *$R \subseteq S \times S$ est une relation de transition. Celle-ci doit être totale ($\forall s \in S, \exists s'/(s, s') \in R$).*
- *S_0 est l'ensemble des états initiaux.*

On appelle *chemin* une séquence infinie d'états $\pi = s_0, s_1, s_2 \ldots$ telle que $\forall i \geq 0$, $(s_i, s_{i+1}) \in R$. On notera π^i le *suffixe* de π commençant à s_i.

Définition 2 (Propriété CTL [Clarke et al., 1986]) *Les formules CTL sont construites à partir de ces propositions atomiques. Leur syntaxe et sémantique sont les suivantes :*
- *Toute proposition atomique $p \in AP$ est une formule CTL.*
  $$M, s \models p \iff p \in L(s)$$
- *Soient f et g deux formules CTL alors $\neg f$, $f \wedge g$, $\mathbf{EX}f$, $\mathbf{EG}f$ et $\mathbf{E}f\mathbf{U}g$ sont des formules CTL.*
  $$M, s_0 \models \neg f \iff M, s_0 \not\models f$$
  $$M, s_0 \models f \wedge g \iff M, s_0 \models f \text{ et } M, s_0 \models g$$
  $$M, s_0 \models \mathbf{EX}f \iff \text{il existe un chemin } \pi \text{ tel que } M, \pi^1 \models f$$
  $$M, s_0 \models \mathbf{EG}f \iff \text{il existe un chemin } \pi \text{ tel que } \forall i \geq 0 M, \pi^i \models f$$
  $$M, s_0 \models \mathbf{E}f\mathbf{U}g \iff \text{il existe un chemin } \pi \text{ tel que } \exists k \geq 0 \text{ tel que } M, \pi^k \models g \text{ et } \forall 0 \leq i < k, M, \pi^i \models f$$

2.2 Vérification des Propriétés CTL

Vérifier une proposition atomique consiste à sélectionner les états de S qui la satisfont et regarder si l'état spécifié en fait partie. Ceci est aussi valable pour toute conjonction ou négation logique. La vérification de formules comprenant des opérateurs **EX**, **EG** ou **EU** s'appuie sur

la relation de transition. Par définition, les états qui satisfont $\mathbf{EX}p$ sont tout ceux qui précèdent par *une* transition les états qui vérifient le prédicat p. D'autre part, il est possible de réécrire les opérateurs **EG** et **EU** en s'appuyant sur les propriétés récurrentes suivantes :

$$M, s \models \mathbf{EG}f \iff M, s \models f \wedge \mathbf{EX}(\mathbf{EG}f)$$

$$M, s \models \mathbf{E}f\mathbf{U}g \iff \begin{cases} M, s \models g \\ \vee \\ M, s \models (f \wedge \mathbf{EX}(\mathbf{E}f\mathbf{U}g)) \end{cases}$$

Ainsi, il est possible de réécrire les opérateurs **EG** et **EU** sous forme d'un calcul de point fixe [Emerson and Clarke, 1980] fondé uniquement sur l'opérateur **EX**, c'est à dire l'évaluation de l'ensemble des prédécesseurs d'un ensemble d'états.

Pour un ensemble d'états donné, cet ensemble des prédécesseurs se calcule grâce à l'ensemble des états accessibles du programme qui est obtenu au préalable à partir des éléments R et S_0 de la structure de Kripke. Aussi la validation de propriétés CTL peut se résumer à l'élaboration des opérateurs suivants :

1. Pour la recherche des états accessibles :

$$post\colon \quad S \to S$$
$$post(s) = s' \iff R(s, s')$$

2. Pour l'évaluation de l'opérateur **EX** :

$$pre\colon \quad S \to S$$
$$pre(s') = s \iff R(s, s')$$

Afin de s'affranchir d'une représentation explicite des états, on utilise un type de données abstrait (muni des opérateurs classiques d'union $\cup$, d'intersection $\cap$, d'inclusion $\subset$, et d'égalité $=$) capable de recenser efficacement un ensemble d'états.

On étend les opérateurs $post$ et pre à des transformations ensemblistes $Post$ et Pre.

Soient $X, Y \subset S \times S$, on écrit $Y = Post(X)$ si $post$ réalise une surjection de X vers Y. De même on notera $X = Pre(Y)$ si pre réalise une surjection de Y vers X.

En partant d'un ensemble d'états initiaux, on applique l'opérateur $Post$ jusqu'à l'obtention d'un point fixe : L'ensemble des états accessibles. Cet ensemble permet d'évaluer l'opérateur Pre nécessaire à la vérification de propriétés CTL.

Dans la section 3 nous présentons le type de données abstrait utilisé dans cette étude pour représenter des ensembles d'états ainsi que les mécanismes mis en œuvre pour construire les opérateurs $Post$ et Pre.

3 Les Diagrammes de Décision de Données

3.1 Présentation

Les Diagrammes de Décision de Données (DDD, [Couvreur et al., 2002]) sont des structures arborescentes qui représentent un ensemble d'affectations successives. Les nœuds sont étiquetés par les variables et les arcs par les valeurs affectées à ces variables. Aucun ordonnancement de variable n'est supposé et une même variable peut y apparaître plusieurs fois. Une séquence acceptée est représentée par un chemin menant de la racine à la feuille terminale 1. Les chemins qui mènent de la racine à feuille terminale 0 correspondent aux séquences qui ne sont pas acceptées, ils ne sont pas représentés sur le DDD. La feuille $\top$ permet de caractériser les séquences indéfinies.

Définition 3 (DDD [Couvreur et al., 2002]) *Soit E un ensemble de variables, l'ensemble $\mathbb{D}$ des DDD est construit inductivement par $d \in \mathbb{D}$ si :*

- *$d \in \{0, 1, \top\}$ ou*
- *$d = (e, \alpha)$ avec :*
  - *$e \in E$*
  - *$\alpha : Dom(e) \to \mathbb{D}$, tel que $\{x \in Dom(e) | \alpha(x) \neq 0\}$ est fini.*

Où e est la variable étiquetant le nœud courant et α l'arc partant du noeud courant, étiqueté par la valeur x vers un nœud successeur non nul. On note $e \xrightarrow{a} d$, le DDD (e, α) avec $\alpha(a) = d$ et $\forall x \neq a, \alpha(x) = 0$. Lorsque toutes les séquences d'un DDD sont acceptées, on dit alors qu'il est bien défini. La figure 1 montre deux DDD qui correspondent aux affectations suivantes :

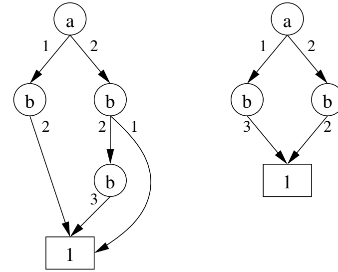

FIG. 1 – Deux DDD

- $\{a = 1, b = 2\}$, $\{a = 2, b = 1\}$ et $\{a = 2, b = 2, b = 3\}$ pour le DDD de gauche.
- $\{a = 1, b = 3\}$ et $\{a = 2, b = 2\}$ pour le DDD de droite.

Deux DDD bien définis d et d' sont égaux si

- $d = d' = 1$
- $d = (e, \alpha) \neq 0$ et $d' = (e, \alpha') \neq 0$ et $\forall x \in Dom(e), \alpha(x) = \alpha'(x)$.

Les opérations ensemblistes sur DDD sont définies inductivement. Elles peuvent être obtenues par une évaluation récursive des arborescences des DDD concernés (elle suppose que les deux DDD sont ordonnés de la même façon). La figure 2 montre le principe d'évaluation d'une union de deux DDD. Elle met en évidence l'apparition d'une séquence indéfinie issue de l'union des nœuds 1 et $b \xrightarrow{3} 1$ alors que les DDD à unir sont pourtant bien définis. Les opérateurs d'intersection et de différence fonctionnent suivant le même principe. Il est également possible de concaténer deux DDD : $(a \xrightarrow{x} 1).(b \xrightarrow{y} 1) = a \xrightarrow{x} b \xrightarrow{y} 1$.

3.2 Homomorphismes

On souhaite représenter les états d'un système matériel sur des DDD afin de profiter de la canonicité et de

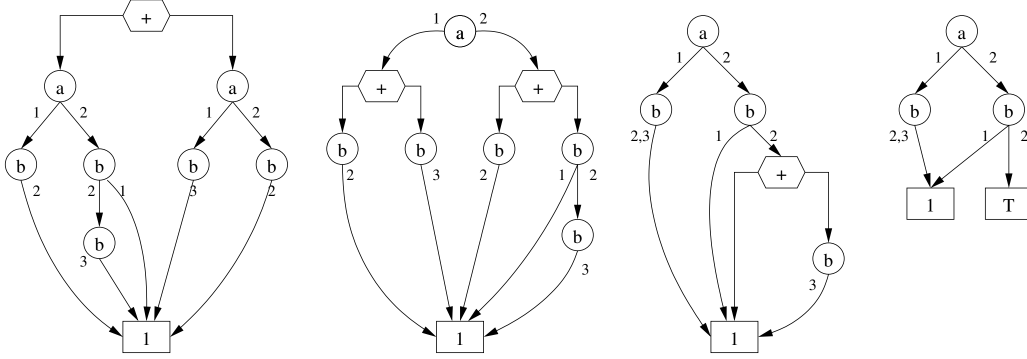

FIG. 2 – Union de deux DDD

la souplesse de la structure. Partant d'un état initial, il est nécessaire de construire un ensemble de traitements satisfaisant la relation de transition R. Les DDD sont pourvu d'un formalisme intéressant pour construire ces opérations spécialisées basé sur des homomorphismes [Couvreur et al., 2002]. Soit Φ un homomorphisme et deux DDD bien définis d et d', alors $\Phi(d + d') = \Phi(d) + \Phi(d')$. On s'attachera à un sous-ensemble particulier de ceux-ci : Les homomorphismes inductifs. Ils ont la particularité d'être définis localement, et comme, contrairement aux BDD, seule les séquences acceptées sont représentées dans un DDD bien défini, il n'est pas nécessaire de parcourir le DDD en entier pour effectuer les traitements voulus.

Définition 4 (Homomorphismes Inductifs) *Soient c un DDD et $\phi(e, x)_{e \in E, x \in Dom(e)}$ une famille d'homomorphismes.*

$$\Phi(d) = \begin{cases} 0 & if\ d = 0 \\ \top & if\ d = \top \\ c & if\ d = 1 \\ \sum_{x \in Dom(e)} \phi(e, x)(\alpha(x)) & if\ d = (e, \alpha) \end{cases}$$

est un homomorphisme [Couvreur et al., 2002].

La dernière expression ($\sum_{x \in Dom(e)} \phi(e, x)(\alpha(x))$) ne s'applique qu'aux nœuds du DDD. On ne dispose alors que d'informations locales : La variable e concernée et ses valeurs possibles α. On applique, sur chacun des arcs sortants $\alpha(x)$, une fonction $\phi(e, x)$. On réunit alors les DDD obtenus. A l'instar des opérations ensemblistes, les homomorphismes inductifs peuvent faire l'objet d'une évaluation paresseuse qui peut limiter considérablement les redondances de calcul.

Prenons le cas de l'affectation constante $var = cst$, On construit l'homomorphisme $\langle\ \mathsf{setCst}\ (var, cst)\ \rangle$ comme suit :

– Les nœuds étiquetés par une autre variable que var ne sont pas concernés par l'affectation. On reporte le même homomorphisme ($\langle\ \mathsf{this}\ \rangle$) sur chacun des arcs sortants sans autre modification. Ainsi, si $e \neq var$,
$$\begin{aligned} & \langle\ \mathsf{setCst}\ (var, cst)\ \rangle\ (e, x) \\ = \quad & e \xrightarrow{x} \langle\ \mathsf{setCst}\ (var, cst)\ \rangle \\ = \quad & e \xrightarrow{x} \langle\ \mathsf{this}\ \rangle \end{aligned}$$
– En revanche, si $e = var$ tous les arcs sortants doivent être étiquetés par la valeur cst et les nœuds fils ne sont pas affectés ; on les reproduit donc à l'identique. On note :
$$\langle\ \mathsf{setCst}\ (var, cst)\ \rangle\ (var, x) = e \xrightarrow{cst} \langle\ \mathsf{id}\ \rangle$$
– Enfin, appliquer ce traitement au noeud 1 reviendrait à affecter une valeur à une variable non déclarée :
$$\langle\ \mathsf{setCst}\ (var, cst)\ \rangle\ (1) = \top$$

Nous considérons qu'aucun homomorphisme ne peut s'appliquer sur le DDD 1 car cela suppose que le traitement n'a pu être appliqué correctement (variable utilisée mais non déclarée). Aussi, nous ne nous préoccupons que des traitements à appliquer sur les nœuds, et non sur les feuilles terminales. Ce qui revient à définir un traitement pour chaque couple *(variable, arc)* d'un nœud.

Considérons à présent l'affectation $v_1 = v_2$. A l'instar de $\langle\ \mathsf{setCst}\ ()\ \rangle$ on reporte le traitement sur les fils des nœuds étiquetés par des variables (v) autres que v_1 et v_2.

$$\langle\ \mathsf{setVarVar}\ (v_1, v_2)\ \rangle\ (e, x) = e \xrightarrow{x} \langle\ \mathsf{this}\ \rangle$$

Si les valeurs de v_2 étaient situées *plus haut* dans l'arborescence, l'évaluation de l'homomorphisme se résumerait à une affectation constante pour chacune des valeurs prises par v_2.

$$\langle\ \mathsf{setVarVar}\ (v_1, v_2)\ \rangle\ (v_2, x) = v_2 \xrightarrow{x} \langle\ \mathsf{setCst}\ (v_1, x)\ \rangle$$

Mais dans le cas où les valeurs de v_2 sont situées *plus bas* que v_1 dans l'arborescence, il est alors nécessaire de faire *remonter* ces informations jusqu'à v_1. On se servira de l'homomorphisme up dont l'application locale est la suivante :

$$\langle\ \mathsf{up}\ (var, val)\ \rangle\ (e, x) = e \xrightarrow{x} var \xrightarrow{val} \langle\ \mathsf{id}\ \rangle$$

En composant n fois cet homomorphisme suivant les couples $(variable, valeur)$ rencontrés, il est possible de faire "remonter" un nœud quelconque sur n niveaux. Ici, le nœud en question devra être étiqueté par v_1 avec pour arcs sortants les valeurs de v_2. C'est l'homomorphisme $\langle\ \mathsf{down}\ ()\ \rangle$ qui se charge de composer les opérateurs $\langle\ \mathsf{up}\ (e, x)\ \rangle$ et de créer ce nœud :

$$\langle\ \mathsf{setVarVar}\ v_1, v_2\ \rangle\ (v_1, x) = \langle\ \mathsf{down}\ (v_1, v_2)\ \rangle$$

avec

$$\begin{aligned} & \langle\ \mathsf{down}\ (v_1, v_2)\ \rangle\ (e, x) \\ = \quad & \begin{cases} \langle\ \mathsf{up}\ (e, x)\ \rangle \circ \langle\ \mathsf{down}\ (v_1, v_2)\ \rangle & if\ e \neq v_2 \\ v_1 \xrightarrow{x} e \xrightarrow{x} & else \end{cases} \end{aligned}$$

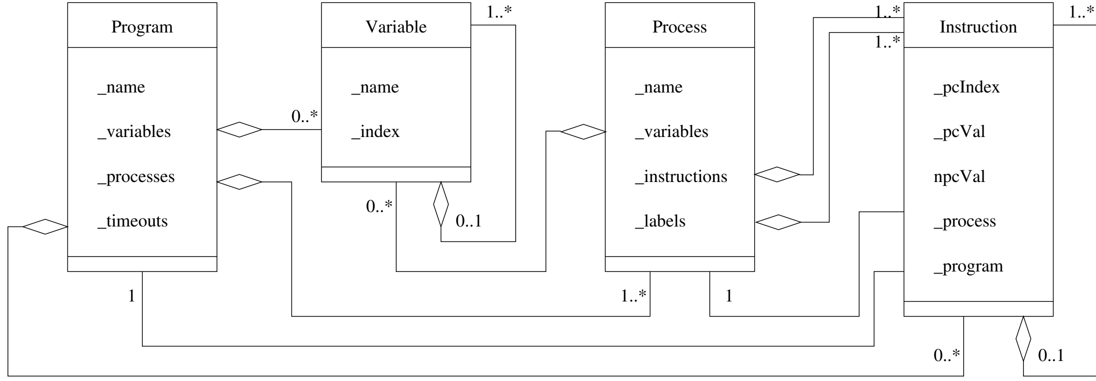

FIG. 3 – Représentation objet d'un programme ProMeLa

Exemple 5 (Utilisation de $\langle$ up () $\rangle$ et $\langle$ down () $\rangle$)
On souhaite réaliser l'affectation $b = d$ *sur le DDD* $a \xrightarrow{1} b \xrightarrow{2} c \xrightarrow{3} d \xrightarrow{4} e \xrightarrow{5} 1$.

$$\begin{aligned}
& \langle \text{setVarVar}\,(b,d) \rangle\, (a \xrightarrow{1} b \xrightarrow{2} c \xrightarrow{3} d \xrightarrow{4} e \xrightarrow{5} 1) \\
= \; & a \xrightarrow{1} \langle \text{setVarVar}\,(b,d) \rangle\, (b \xrightarrow{2} c \xrightarrow{3} d \xrightarrow{4} e \xrightarrow{5} 1) \\
= \; & a \xrightarrow{1} \langle \text{down}\,(b,d) \rangle\, (c \xrightarrow{3} d \xrightarrow{4} e \xrightarrow{5} 1) \\
= \; & a \xrightarrow{1} \langle \text{up}\,(c,3) \rangle \circ \langle \text{down}\,(b,d) \rangle\, (d \xrightarrow{4} e \xrightarrow{5} 1) \\
= \; & a \xrightarrow{1} \langle \text{up}\,(c,3) \rangle\, (b \xrightarrow{4} d \xrightarrow{4} e \xrightarrow{5} 1) \\
= \; & a \xrightarrow{1} b \xrightarrow{4} c \xrightarrow{3} d \xrightarrow{4} e \xrightarrow{5} 1
\end{aligned}$$

Des mécanismes similaires peuvent être employés pour l'affectation et la sélection portant sur des expressions plus complexes.

4 Composants d'un Programme ProMeLa

Nous montrons comment les DDD sont utilisés pour la représentation d'un état d'un programme ProMeLa et définissons les homomorphismes correspondant aux opérateurs Pre et $Post$ conformes à la sémantique du langage ProMeLa.

4.1 Modèle Objet d'un programme ProMeLa

La figure 3 montre comment est construite la représentation objet d'un programme ProMeLa. La classe `program` est propriétaire des variables globales, des canaux de communication et des processus du *programme* ProMeLa. Chaque classe `process` est propriétaire des variables locales d'un *processus* et de son compteur ordinal.
On distingue deux catégories d'instructions ProMeLa : Les instructions élémentaires (gardées, étiquetées) et les délimiteurs de blocs (sélections, boucles, etc.). Ces derniers contiennent un ensemble d'instructions structuré suivant leur particularité mais ne nécessite pas d'homomorphismes autres que ceux construits pour les instructions élémentaires.

4.2 Représentation d'un État

Un programme ProMeLa décrit un ensemble de processus asynchrones communicants par variables partagées ou canaux (FIFOs ou rendez-vous). Une variable de type entière est représentable sur un DDD à l'aide d'un nœud de type

$$variable \xrightarrow{valeur} ...$$

Un processus est construit par la concaténation des DDD représentant les variables locales et le compteur ordinal

$$program_counter \xrightarrow{pc} local_var_1 \xrightarrow{val} local_var_2 \rightarrow ...$$

D'autre part, tout type structuré statique peut être mis à plat sans remettre en cause le comportement du système à analyser. Par exemple un tableau t de taille n peut être représenté par n variables :

$$t[0] \rightarrow t[1] \rightarrow ... \rightarrow t[n-1] \rightarrow ...$$

Les communications par canaux sont de deux types : Les rendez-vous sont effectués par des affectations gardées ; et les FIFOs sont représentées suivant le principe d'encodage qui est proposé par [Couvreur et al., 2002] : On utilise une même variable pour chaque place occupée de chaque FIFO du système. Les nœuds de ces variables sont représentés successivement sur le DDD. Ils sont encadrés par deux nœuds étiquetés par la même variable : Le premier indiquant la taille du canal et le dernier indiquant qu'il n'y a plus d'autre place occupée dans la FIFO. Celui-ci a un unique arc sortant étiqueté par une valeur qui ne peut être contenue dans la FIFO ($\#$).
Ainsi, une Fifo est représentée de la façon suivante :

$$f \xrightarrow{size} f \xrightarrow{1^{st}elt} ...f \xrightarrow{i^{th}elt} f \xrightarrow{\#} ...$$

On représente les états d'un programme ProMeLa en concaténant les DDD représentant l'ensemble des variables, canaux et processus sur un unique DDD.

4.3 Instructions

Toutes les instructions sont gardées par la valeur d'un compteur ordinal, et éventuellement une condition supplémentaire nécessaire à son franchissement. Elles procèdent ensuite à l'évolution du compteur ordinal et aux modifications décrites en ProMeLa.
Dans le cas d'un programme statique ProMeLa (sans instanciation dynamique de processus), l'ensemble des transitions peut se résumer à l'articulation des opérateurs suivants :

1. Sélection des états satisfaisant une expression booléenne.
2. Affectation d'une expression entière.
3. Écriture d'une expression dans une FIFO.
4. Lecture dans une FIFO d'une variable donnée.
5. Prise d'information concernant une FIFO (vide, pleine, non vide ou non pleine).

Les homomorphismes correspondants à l'opérateur $Post$ de ces instructions ne posent aucune difficulté d'implémentation.
Pour évaluer l'opérateur Pre, la principale difficulté provient de l'irréversibilité de certaines instructions (Lecture dans une FIFO, Affectation d'une expression ne comportant pas la variable modifiée). Ces dernières sont basées sur le calcul de prédécesseur d'une affectation de constante. L'ensemble des états prédécesseurs à l'exécution d'une instruction d'affectation $var = cst$ n'est pas déterminé si nous ne connaissons pas *a priori* la borne de var, ou mieux, l'ensemble des valeurs effectivement prises par var dans le contexte du franchissement de l'instruction $var = cst$.
Le calcul des prédécesseurs d'une affectation constante fait intervenir deux ensembles d'états :

– TO, l'ensemble des états dont nous voulons connaître les prédécesseurs (par l'instruction $var = cst$)
– et C, l'ensemble des états candidats qui est une approximation des états représentant le contexte d'exécution de $var = cst$. Il est contenu dans l'ensemble des états accessibles.

Nous partons des hypothèses suivantes :

– La valeur initiale de la variable modifiée peut être quelconque.
– La seule variable modifiée est var.

La solution peut se décomposer en trois traitements :

1. Sélectionner dans le DDD TO les états pour lesquels $var = cst$.
2. Étendre dans ce nouveau DDD la valeur de var à toutes les valeurs possibles *proposées par les états candidats* (car on ne connait pas la borne de var).
3. Procéder à une intersection de ce dernier DDD avec l'ensemble des candidats.

L'homomorphisme $sweet$ réalise ces trois traitements en un seul parcours. Il est paramétré par var, cst et les états dont on veut calculer les prédécesseurs TO. Il s'applique à l'ensemble des états candidats C :

$$\begin{aligned} & \langle\ \mathsf{preSetCst}\ (var, cst, C\ \rangle\ (TO) \\ = & \ \ \langle\ \mathsf{sweet}\ (var, cst, TO)\ \rangle\ (C) \end{aligned}$$

On détaille ici l'application locale de l'homomorphisme $\langle$ sweet (v, c, To) $\rangle$ à un nœud quelconque (e, x). Plusieurs cas se présentent :

1. Il n'existe pas de prédécesseur (ni de successeur) à un Tuple *absent*.

$$\langle\ \mathsf{sweet}\ (v, c, To)\ \rangle\ = \emptyset$$

si $To = \emptyset$.

2. Tant qu'il n'a pas rencontré var, il procède à une intersection locale.

$$\begin{aligned} & \langle\ \mathsf{sweet}\ (v, c, To)\ \rangle\ (e, x) \\ = & \ \ e \xrightarrow{x} \langle\ \mathsf{sweet}\ (v, c, To|_x)\ \rangle \end{aligned}$$

si $e \neq v$ où $To|_x$ désigne le nœud pointé par l'arc issu de la racine de To, étiqueté par x.

3. Une fois var rencontrée, on étend son ensemble de valeurs prises à celles qui figurent dans l'ensemble des états candidats (plus précisément toutes les valeurs étiquetant les arcs issus du nœud courant). A partir de là, l'intersection (pour les variables restantes) se fait par un produit classique de DDD.

$$\begin{aligned} & \langle\ \mathsf{sweet}\ (v, c, To)\ \rangle\ (e, x) \\ = & \ \ e \xrightarrow{x} (\langle\ \mathsf{id}\ \rangle\ \cap To|_c)i \end{aligned}$$

Une évaluation de l'opérateur $sweet$ est donnée sur la figure 4. Dans cet exemple, on cherche à calculer l'ensemble des états prédécesseurs de TO par l'application de $b = 2$, sur l'ensemble des candidats désignés par C. On applique à C l'homomorphisme $\langle$ sweet $(b, 2, TO)$ $\rangle$. Ce qui revient à effectuer un parcours attelé entre TO et C jusqu'à rencontrer la variable b. Le DDD TO n'accepte pas $a = 2$, donc l'application de sweet sur $a \xrightarrow{2}$ ... s'évalue en une feuille nulle, et l'homomorphisme se propage dans C sur l'arc $a \xrightarrow{1} C_b$. Par définition, toutes les valeurs des arcs sortants du nœud C_b sont acceptables. L'ensemble des états dont elles sont le support $((b \xrightarrow{2} C_{c1}) + (b \xrightarrow{3} C_{c2}))$ est intersecté avec les états de TO représentant le contexte $b = 2$ ($b \xrightarrow{2} d3$) à partir des nœuds fils (quatrième schéma).
Le résultat est donné à droite. On a bien sélectionné dans C les états prédécesseurs de TO par le franchissement de $b = 2$ respectant le contexte d'exécution de cette instruction.

5 Résultats

Cette section compare les performances de l'outil que nous avons développé sur DDD avec celles de SPIN pour le calcul de l'ensemble des états accessibles et la vérification de certaines propriétés CTL.
Pour l'étude de systèmes faiblements parallèles, nous avons étudié la version du protocole "Sliding Window" proposée pour SPIN par [Holzmann, 1997b]. Dans ce cas, SPIN se montre beaucoup plus efficace que notre outil, sur tous les plans : Sur un processeur Intel PIII cadencé à 1GHz, une seconde et 8Mo ont suffi pour parcourir l'ensemble des états accessibles comprenant 400000 états, quand l'approche ProMeLa-DDD nécessite une demiheure et 27Mo de mémoire.
Pour l'étude de systèmes massivement parallèles nous avons choisi l'exemple du dîner de philosophes. Nous avons également étudié le problème de l'élection sur un anneau (leader election). Il combine les caractéristiques des deux précédents systèmes : On retrouve une topologie circulaire comparable à celle du dîner des philosophes et les informations ne circulent que dans un sens,

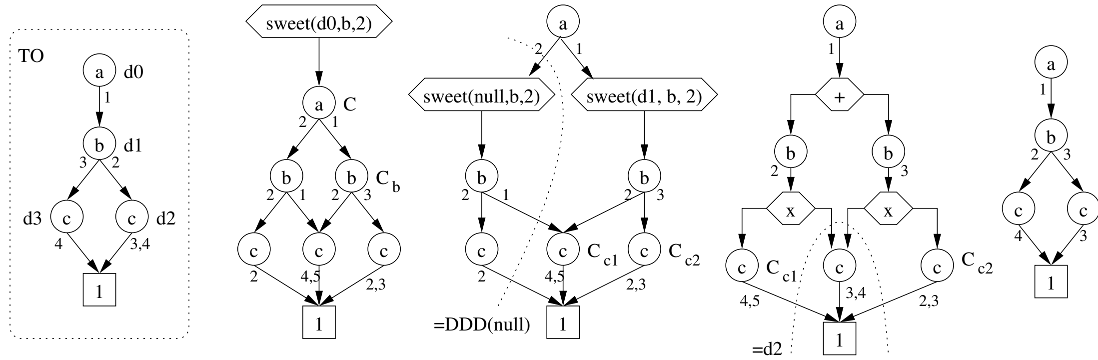

FIG. 4 – Évaluation de l'opérateur $sweet$

Philosophers									Leaders								
	SPIN		DDD			DDD-R				SPIN		DDD			DDD-R		
p	tr	m	tr	tf	m	tr	tf	m	n	tr	m	tr	ts	m	tr	ts	m
8	1	5.7	5	16	10	7	16	7.4	2	<1	2.6	1	<1	4.6	<1	<1	4.1
10	6	37	61	360	27	17	37	12.7	4	<1	5.3	4	2	5.5	4	2	5.8
13	260	165	280	388	161	52	122	23.9	6	1892	744	65	28	179	58	15	14.5
15	*	*	393	*	642	92	142	33.7	7	*	*	288	79	45.7	141	32	27
20	*	*	*	*	*	313	702	80.8	10	*	*	*	*	*	997	226	109
40	*	*	*	*	*	5484	10826	759	15	*	*	*	*	*	7815	1961	525

TAB. 1 – Comparatif SPIN/ProMeLa-DDD

comme dans le cas du protocole Sliding Window, ce qui séquentialise de fait les traitements.

Les résultats comparatifs sont donnés sur le tableau 1. Pour les deux systèmes (Philosophes et Élection) nous comparons SPIN et notre approche (DDD) pour une recherche d'états accessibles et de validation de propriétés CTL lorsque les variables sont ordonnées sur le DDD suivant l'ordre de déclaration dans le programme ProMeLa. Les performances sont données avec p philosophes ou n nœuds dans l'anneau d'élection. La mémoire m nécessaire est exprimée en Mo. Les temps de calcul, tr pour la recherche d'accessibilité, tf pour l'évaluation d'une propriété de vivacité, et ts pour l'évaluation d'une propriété de sûreté, sont exprimés en secondes. Les colonnes "DDD-R" correspondent aux mêmes calculs que "DDD" lorsque l'ordre des variables sur le DDD s'appuie sur la topologie du réseau. L'approche ProMeLa-DDD est capable d'appréhender des systèmes beaucoup plus importants pour peu que l'ordonnancement des variables ait bien été choisi.

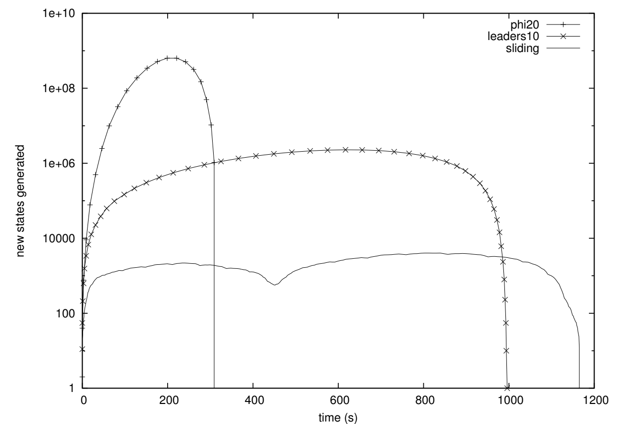

FIG. 5 – Nombre de nouveaux états générés à chaque itération de l'opérateur $Post$

Néanmoins, nous avons vu que pour le protocole Sliding Window, SPIN obtient de bien meilleures performances que nous. La figure 5 montre la nature très différente des trois systèmes étudiés (Sliding Window, Leader et Philosophers) : Elle donne (sur une échelle logarithmique) le nombre de nouveaux états par itération de l'opérateur $Post$, l'abscisse représente le déploiement du calcul en fonction du temps, ce qui permet de distinguer le nombre d'itérations du temps de calcul. Dans le cas du dîner des philosophes et, dans une moindre mesure, de l'élection sur un anneau, la quantité d'états générés (nouveaux états après chaque application de l'opérateur $Post$) est particulièrement importante (10^9 et 10^6) tandis qu'elle stagne en dessous de 5000 pour le protocole "Sliding Window". L'emploi de méthodes symboliques n'est effectivement intéressant que lorsque l'ensemble des états générés, suite à l'application de l'opérateur $Post$, croît fortement, comme c'est généralement le cas des systèmes massivement parallèles ou fortement indéterministes. D'autre part la mise à plat des types structurés présente deux inconvénients importants : Tout d'abord cela augmente le nombre de variables représentées, ce qui rallonge la mise en œuvre des mécanismes de parcours. Ensuite, l'affectation, la lecture ou l'écriture d'un type structuré se traduit en ensemble d'instruction variable par variable, contrairement à SPIN qui procède en une seule opération.

6 Conclusion

Nous avons développé un Model-Checker CTL pour systèmes statiques décrits en ProMeLa, avec lequel il est possible de vérifier des propriétés de sûreté et de vivacité sur des classes de système où SPIN est mis en défaut.
Le model-checker est basé sur les Diagrammes de Décision de Données qui reprennent aux BDD la notion d'arborescence partagée et de canonicité, mais permet de manipuler directement des variables entières, et est doté de mécanismes d'évaluations et de modifications locales.
A l'instar des BDD, l'ordonnancement des variables s'avère critique. Une comparaison avec d'autre approches symboliques comme l'utilisation de Diagrammes de Décision Binaires est en cours.
Nous étudions un moyen de spécifier l'instanciation dynamique de processus et introduisons la hiérarchie dans les DDD pour pallier le problème de la multiplicité des variables.

BIBLIOGRAPHIE

[Beaudenon et al., 2003] Beaudenon, V., Encrenaz, E., and Desbarbieux, J.-L. (2003). Design validation of zcsp with spin. In *Proceedings of the Third International Conference on Application of Concurrency to System Design (ACSD'03)*, page 102. IEEE Computer Society.

[Brayton et al., 1996] Brayton, R. K., Hachtel, G. D., Sangiovanni-Vincentelli, A., Somenzi, F., Aziz, A., Cheng, S. T., Edwards, S., Khatri, S., Kukimoto, Y., Pardo, A., Qadeer, S., Ranjan, R. K., Sarwary, S., Shiple, T. R., Swamy, G., and Villa, T. (1996). VIS : a system for verification and synthesis. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, volume 1102, pages 428–432, New Brunswick, NJ, USA. Springer Verlag.

[Bryant, 1986] Bryant, R. E. (1986). Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, 35(8) :677–691.

[Bryant and Chen, 1994] Bryant, R. E. and Chen, Y.-A. (1994). Verification of arithmetic functions with binary moment diagrams. Technical Report CS-94-160.

[Burch et al., 1990] Burch, J., Clarke, E., McMillan, K., Dill, D., and Hwang, L. (1990). Symbolic Model Checking : 10^{20} States and Beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 1–33, Washington, D.C. IEEE Computer Society Press.

[Ciardo et al., 1999] Ciardo, G., Luettgen, G., and Siminiceanu, R. (1999). Efficient symbolic state-space construction for asynchronous systems. Technical report.

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

[Couvreur et al., 2002] Couvreur, J.-M., Encrenaz, E., Paviot-Adet, E., Poitrenaud, D., and Wacrenier, P.-A. (2002). Data decision diagrams for petri net analysis. In *Proceedings of the 23rd International Conference on Applications and Theory of Petri Nets*, pages 101–120. Springer-Verlag.

[de Kock et al., 2000] de Kock, E. A., Smits, W. J. M., van der Wolf, P., Brunel, J.-Y., Kruijtzer, W. M., Lieverse, P., Vissers, K. A., and Essink, G. (2000). Yapi : application modeling for signal processing systems. In *Proceedings of the 37th conference on Design automation*, pages 402–405. ACM Press.

[Emerson and Clarke, 1980] Emerson, E. A. and Clarke, E. M. (1980). Characterizing correctness properties of parallel programs using fixpoints. In *Proceedings of the 7th Colloquium on Automata, Languages and Programming*, pages 169–181. Springer-Verlag.

[Holzmann, 1997a] Holzmann, G. (1997a). State Compression in Spin. Twente University, The Netherlands.

[Holzmann and Peled, 1994] Holzmann, G. and Peled, D. (1994). An improvement in formal verification. In *Proc. Formal Description Techniques, FORTE94*, pages 197–211, Berne, Switzerland. Chapman & Hall.

[Holzmann, 1997b] Holzmann, G. J. (1997b). The Model Checker Spin. *IEEE Trans. on Software Engineering*, 23(5) :279–295. Special issue on Formal Methods in Software Practice.

[Hommais et al., 2001] Hommais, D., Pétrot, F., and Augé, I. (2001). A practical tool box for system level communication synthesis. In *Proceedings of the ninth international symposium on Hardware/software codesign*, pages 48–53. ACM Press.

[Lai et al., 1996] Lai, Y.-T., Pedram, M., and Vrudhula, S. B. K. (1996). Formal verification using edge-valued binary decision diagrams. *IEEE Trans. Comput.*, 45(2) :247–255.

[McMillan, 1993] McMillan, K. L. (1993). *Symbolic Model Checking*. Kluwer Academic Publishers.

[Miner and Ciardo, 1999] Miner, A. S. and Ciardo, G. (1999). Efficient reachability set generation and storage using decision diagrams. In Donatelli, Susanna and Kleijn, Jetty, editors, *Lecture Notes in Computer Science : Application and Theory of Petri Nets 1999, 20th International Conference, ICATPN'99, Williamsburg, Virginia, USA*, volume 1630, pages 6–25. Springer-Verlag.

[Peled, 1994] Peled, D. (1994). Combining partial order reductions with on-the-fly model-checking. In *Proc. Sixth Int Conf. Computer Aided Verification (CAV94)*, pages 377–390.

[Peng et al., 1999] Peng, H., Tahar, S., and Khendek, F. (1999). Comparison of spin and vis for protocol verification.

[Pnueli, 1977] Pnueli, A. (1977). The Temporal Logic of Programs. In *18th IEEE Symp. Foundations of Computer Science*, pages 46–57.