De l'avantage de nuancer les décisions binaires

Claude Marché, Denis Cousineau

De l'avantage de nuancer les décisions binaires*

Claude Marché[1] et Denis Cousineau[2]

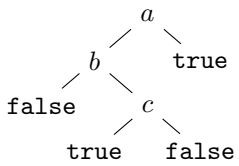
[1]Université Paris-Saclay, CNRS, ENS Paris-Saclay, Inria, LMF, 91190, Gif-sur-Yvette, France
[2]Mitsubishi Electric R&D Centre Europe, Rennes, France

Nous présentons une extension des diagrammes de décision binaire classiques, consistant à rajouter aux feuilles possibles `true` et `false` des contraintes externes issues d'un domaine de contraintes paramétrable. Nous instancions ce concept avec des contraintes linéaires sur des variables entières. Nous utilisons cette extension comme domaine abstrait pour inférer des invariants de boucle dans des programmes en WhyML, le langage de l'environnement de vérification Why3. L'application principalement visée est de vérifier des codes en langage Ladder, langage qui sert à programmer des contrôleurs logiques. De tels programmes utilisent typiquement un grand nombre de variables booléennes et simultanément quelques variables entières. Notre approche est évaluée par comparaison avec l'utilisation d'un interpréteur abstrait utilisé précédemment, et nous mettons en évidence les gains significatifs obtenus en terme de temps de calcul ainsi qu'en précision des invariants obtenus.

Mots-clés : Diagrammes de Décision Binaires, Interprétation Abstraite, Inférence d'invariants de boucle.

1 Introduction

Les *diagrammes de décision binaire*, que nous abrégeons en BDD selon la terminologie anglophone, forment une structure de données permettant de représenter efficacement des formules de la logique propositionnelle. Au tout premier abord, un BDD consiste à représenter une formule par un arbre binaire, chaque nœud étant l'objet d'une décision sur la valeur booléenne d'une variable. Ainsi une formule comme $a \vee (b \wedge \neg c)$ peut être vue comme l'arbre



où le sous-arbre à gauche d'une variable exprime le cas où la variable est interprétée à `false`, alors que le sous-arbre de droite concerne le cas elle est interprétée à `true`. Une manière équivalente de voir cet arbre, mais de façon textuelle, est l'expression

`if` a `then true else (if` b `then (if` c `then false else true)) else false`

*Ce travail est partiellement financé par un contrat bilatéral ProofInUse-MERCE entre l'équipe Inria Toccata et Mitsubishi Electric R&D Centre Europe, à Rennes.

formée de conditionnelles imbriquées. L'efficacité des BDD réside de manière cruciale au niveau de leur consommation en mémoire : en effet les arbres tels que ci-dessus sont en fait mémorisés comme des graphes acycliques où les sous-arbres identiques sont partagés en mémoire. Si l'on fixe *a priori* un ordre sur les variables, et si l'on fait l'effort de forcer un partage maximal, alors on obtient la propriété très forte que deux formules sont logiquement équivalentes si et seulement si elles sont stockées à la même case mémoire. Une telle propriété permet non seulement de décider très rapidement si une formule est insatisfaisable mais aussi permet de coder les programmes opérant sur des BDD de manière très efficace en temps de calcul, grâce aux techniques de mémoïsation classiques. Plus concrètement, dans une bibliothèque implémentant de tels BDD, comme par exemple la bibliothèque OCaml-BDD [10], l'efficacité en mémoire de la représentation et l'efficacité temporelle des opérations sur les BDD sont obtenues par des codes de nature impérative, à base de tables de hachage et de *hash-consing*. Malgré ce caractère non purement fonctionnel, une telle bibliothèque peut tout à fait être dotée d'une interface d'une nature fonctionnelle pure, cachant le caractère impératif derrière une barrière d'abstraction robuste [7].

Les BDD ont de nombreuses applications, comme la conception de circuits logiques, ou d'une manière générale la résolution de contraintes au sens large du terme. Dans cet article, notre intérêt pour les BDD est justifié par leur utilisation comme domaine abstrait pour l'inférence d'invariants de programme. Dans un contexte d'une application industrielle étudiée par Mitsubishi Electric R&D Centre Europe [5], nous avons cherché à rendre plus efficace la recherche d'invariants de boucle sur des programmes comportant un grand nombre de variables booléennes, ceci grâce aux BDD.

1.1 Contexte : inférence d'invariants par interprétation abstraite

À titre d'exemple, considérons le programme de la figure 1, un exemple jouet en langage C, spécifié par un contrat en ACSL [3]. On peut chercher à vérifier que ce code respecte son contrat en utilisant l'analyseur par interprétation abstraite fourni par Frama-C [2] :

```
> frama-c -eva -lib-entry -main toy toy.c
[...]
[eva:alarm] toy.c:4: Warning:
  function toy: postcondition got status unknown.
[...]
[eva:final-states] Values at end of function toy:
  x ∈ [2018..2059]
  b ∈ {0; 1}
```

On constate que l'état abstrait calculé en fin de fonction, qui sur-approxime l'ensemble des

```
int x,b;

/*@ requires 0 <= x <= 91;
  @ ensures x <= 2024 ;
  @*/
void toy (void) {
  b = 0;
  while (x <= 2017) {
    b = (x <= 1000);
    if (b) { x += 42; } else { x += 7; }
  }
}
```

Figure 1. Un exemple jouet en langage C, avec un contrat en ACSL

états accessibles, indique que x peut prendre n'importe quelle valeur entre 2018 et 2059, une information qui en particulier ne permet pas de conclure à la validité de la post-condition. Dans le détail, il faut comprendre que l'interpréteur abstrait a inféré un état abstrait point-fixe de la boucle, ici l'état

```
x ∈ [0..2059]
b ∈ {0; 1}
```

l'état final étant ensuite obtenu par intersection avec la négation de la condition de sortie de la boucle, soit $x > 2017$. La faiblesse ici réside donc initialement dans la trop grande sur-approximation obtenue sur l'état point-fixe de la boucle, qui peut être vu comme l'invariant de boucle $0 \leq x \leq 2059 \wedge 0 \leq b \leq 1$.

L'exemple ci-dessus illustre une situation très classique en interprétation abstraite : la précision obtenue dépend de l'expressivité des domaines abstraits utilisés. Dans le cas, présent il y a un besoin d'utiliser un domaine *relationnel*, qui ne se contente pas d'estimer les domaines de chaque variable indépendemment les unes des autres, mais exprime également des relations entre ces variables. Utiliser des domaines relationnels a cependant un coût algorithmique très important (aussi bien en terme d'utilisation mémoire que de temps de calcul) et ne peut être fait sans parcimonie. L'objectif de cet article est de proposer une forme de domaine abstrait relationnel qui sera bien adapté à notre application. À titre d'accroche pour ce qui va suivre, avec la méthode que nous proposons l'invariant de boucle obtenu sur l'exemple jouet est

$$\texttt{if } b \texttt{ then } 42 \leq x \leq 1042 \texttt{ else } 0 \leq x \leq 2024$$

ce qui, par intersection avec la négation de la condition de boucle, nous donne $2018 \leq x \leq 2024$, ce qui valide la post-condition.

1.2 Application visée : les programmes Ladder

Dans le cadre d'un projet en collaboration entre Inria et Mitsubishi Electric R&D Centre Europe, nous travaillons depuis quelques années sur la vérification de programmes Ladder. Il s'agit de codes servant à programmer des PLC (Programmable Logic Controllers), utilisés de manière très répandue dans les chaînes de montage automatisées. Nous avons publié des travaux montrant comment nous pouvons vérifier statiquement qu'un code Ladder satisfait une spécification de nature temporelle, exprimée par des diagrammes de transition [5]. Dans ces travaux, un code Ladder (et sa spécification temporelle) est traduit automatiquement en un programme en WhyML, le langage utilisé par l'environnement Why3 [11]. Si la preuve réussit, on a réussi à prouver formellement que le code Ladder respecte sa spécification. En cas d'échec, un outillage permet de remonter un contre-exemple produit par Why3 [4] vers un contre-exemple au niveau du Ladder.

Dans cette approche, le code WhyML généré se caractérise par, d'une part, un grand nombre de variables booléennes, et d'autre part, une structure de contrôle particulière : le code est formé d'une séquence de boucles successives, typiquement une dizaine à la suite. Pour que la preuve réussisse, chacune de ces boucles doit être munie d'un invariant de boucle suffisamment précis. Il n'est pas envisageable, pour l'applicabilité de la méthodologie, de supposer que l'utilisateur va donner les invariants lui-même. Ils doivent donc être générés automatiquement. C'est donc là que l'interprétation abstraite entre en jeu de manière cruciale. Pour atteindre nos objectifs, nous avons donc voulu ajouter à Why3 un moteur de génération d'invariants par interprétation abstraite. Heureusement un tel prototype existait un début de ce travail, réalisé par Lucas Baudin [1]. Nous avons donc repris le développement de ce prototype, que nous avons nommé InferLoop. Le gros point manquant au départ était justement le support des variables booléennes. Nous avons alors complété le code de InferLoop par un support des booléens dans les domaines abstraits. Ce support était analogue à ce que l'on ferait avec du code C, comme sur la figure 1, où les booléens sont vus

comme des entiers valant 0 ou 1. Ce prototype, après un peu de travail de mise au point, a permis de réaliser une première version d'un vérificateur de programmes Ladder [5]. À titre d'illustration, sur un code WhyML équivalent au code C de la figure 1, l'invariant généré est

$$(\neg b \wedge 0 \leq x \leq 91) \vee (b \wedge 42 \leq x \leq 1042) \vee (\neg b \wedge 1008 \leq x \leq 2024)$$

ce qui s'avère suffisant pour prouver la post-condition. On peut d'ailleurs remarquer la forme de l'invariant généré, une disjonction de conjonctions, qui révèle le fait que InferLoop utilise des domaines *disjonctifs* (cf Section 5).

Les résultats expérimentaux rapportés dans l'article Belo et al. [5] sont positifs dans le sens où un exemple de code Ladder non trivial réussit à être prouvé automatiquement. Le résultat est malgré tout mitigé par un constat d'inefficacité de la génération d'invariants, selon deux aspects. Un premier aspect est que InferLoop ne passe pas suffisamment à l'échelle pour traiter la dizaine de boucles en séquence qui devaient être supportées, ce qui a nécessité une adaptation en amont pour générer un code WhyML distinct pour chaque boucle, l'invariant de chaque boucle devant être réinjecté comme pré-condition de la boucle suivante. Un second aspect est que, malgré le découpage précédent, le temps de calcul des invariants était insatisfaisant : le temps passé à générer les invariants était de l'ordre de dix fois le temps passé à prouver le programme.

Pour résoudre ce problème de passage à l'échelle, nous avons cherché des solutions qui prendraient mieux en compte notre situation spécifique, à savoir des programmes faisant usage d'un grand nombre de variables booléennes. Les solutions potentielles que nous avons trouvé dans la littérature et que nous avons expérimenté (et discutées dans la section 5 ci-après), n'ont pas offert les résultats escomptés, nous nous sommes donc tournés vers une solution originale qui semblaient *a priori* mieux convenir aux besoins. Cette solution, que nous nommons *Diagrammes de décision binaires paramétriques*, consiste à généraliser les BDD classiques, en remplaçant les deux seules feuilles possibles `true` et `false` par des formules d'un domaine paramètre. Ce paramètre peut être instancié par exemple par un domaine dédié aux entiers.

1.3 Contributions et structure de cet article

La structure de cet article est la suivante : dans la section 2, nous rentrons dans le détail des opérations fournies par la bibliothèque OCaml-BDD, et comment nous nous en sommes servi pour un premier prototype d'interpréteur abstrait pour des programmes purement booléens. Nous présentons nos contributions à OCaml-BDD : des opérations supplémentaires nécessaires pour l'interprétation abstraite. Dans la section 3 nous présentons la structure de données de BDD paramétriques. Nous présentons d'abord son interface, et une partie de son implémentation, sous forme d'un foncteur OCaml `BDDparam`. Nous présentons alors l'instance de ce foncteur sur un domaine de contraintes linéaires sur les entiers. Dans la section 4 nous présentons des expérimentations conduites pour évaluer l'efficacité apportée, y compris les améliorations de rapidité et de précision sur les programmes WhyML traduits depuis Ladder. Enfin, dans la section 5, nous dressons quelques conclusions de notre travail, comparons celui-ci avec des travaux existants, et présentons des perspectives.

Le code complet de notre bibliothèque BDDinfer est disponible comme sous-répertoire du dépôt de Why3 [8].

2 Diagrammes de décision binaire

Dans un premier temps, nous nous sommes focalisé sur l'objectif de réaliser un prototype d'interpréteur abstrait générant des invariants de boucles pour des programmes avec uniquement des variables booléennes. L'idée est alors d'utiliser la bibliothèque OCaml-BDD pour implémenter un domaine abstrait de formules booléennes. Un exemple illustratif est donné

```
val random_bool () : bool
val a : ref bool
val b : ref bool
val c : ref bool

let f () =
  a := False; b := False; c := False;
  while not !c do
    if !b then c := True;
    if !a then b := True;
    let x = random_bool () in if x then a := True;
  done;
  assert { !a }
```

Figure 2. Un exemple de programme purement booléen avec une boucle, en WhyML. La fonction `random_bool` est déclarée sans corps, elle renvoie de façon non-déterministe un booléen.

---

sur la figure 2. Même très simple, un tel exemple est déjà représentatif des codes issus de programmes Ladder, où les changements d'état de devices logiques évoluent en plusieurs étapes d'itération. La fonction `random_bool` utilisé dans cet exemple est représentative des entrées lues depuis des capteurs, dont les valeurs sont non-déterministes.

2.1 Opérations nécessaires pour un interpréteur abstrait

Pour réussir à prouver l'assertion après la boucle dans l'exemple de la figure 2, il est nécessaire de découvrir un invariant suffisamment précis pour la boucle. Un tel invariant est naturellement une formule booléenne sur a, b et c, et l'on se propose de la générer sous la forme d'un BDD. Suivant les principes de base de l'interprétation abstraite, on va construire en chaque point du code un état abstrait (ici donc une formule, c'est-à-dire un BDD) qui sur-approxime les états accessibles à ce point. On va procéder par une exécution symbolique vers l'avant, en calculant l'état abstrait après chaque instruction à partir de l'état avant son exécution. Ainsi, sur l'exemple de la figure 2, au début du corps de `f` on part de l'état `true`, indiquant que toutes les valeurs de a, b et c sont possibles, puis on traverse les trois premières affectations et obtenons successivement les états $\neg a$, $\neg a \wedge \neg b$ et $\neg a \wedge \neg b \wedge \neg c$. Il faut noter donc que nous avons besoin d'opérations sur les BDD qui nous permettent de calculer le BDD représentant l'état du programme après une affectation. L'interprétation abstraite va ensuite rentrer dans le corps de la boucle, et interpréter la première conditionnelle. Pour interpréter une conditionnelle, il faut :

- intersecter l'état d'entrée avec la condition, interpréter la branche `then`
- intersecter l'état d'entrée avec la négation de la condition, interpréter la branche `else`
- calculer l'union des états obtenus, représentant donc l'ensemble des états accessibles après la conditionnelle

On voit clairement apparaître ici l'utilisation des opérations de conjonction, de négation et de disjonction, qui existent naturellement sur les BDD. L'interprétation de l'introduction d'une variable locale (`let..in..`) et d'un appel de fonction va demander des opérations moins naturellement présentes sur les BDD. L'ajout de la variable locale n'est pas difficile si on la voit comme la prise en compte d'une variable booléenne supplémentaire. L'interprétation d'un appel d'une fonction arbitraire est plus problématique. Il y a déjà au départ deux possibilités : exécuter le corps de la fonction, ou bien interpréter en une seule étape cet appel, en prenant en compte uniquement le contrat de la fonction. Notre interpréteur sait faire les deux, mais pour notre propos ici nous ne considérons que ce second cas. Ce second

cas couvre d'ailleurs le cas des affectations, les deux pouvant être vus comme des instances d'une opération générique que l'on appellera Havoc. Voici sa forme générale :

$$\mathsf{Havoc}(x_1, \ldots, x_k; C)$$

est une opération qui modifie en place les variables $x_1, \ldots, x_k$ de manière *a priori* non déterministe, mais en respectant la post-condition C. La condition C peut mentionner à la fois les valeurs des variables avant et après le Havoc. Les valeurs avant seront ici notées $\texttt{old}(x_i)$. Ainsi, une affectation $x \leftarrow e$ correspond à

$$\mathsf{Havoc}(x; x = \texttt{old}(e))$$

où $\texttt{old}(e)$ désigne l'expression e où toutes les occurrences de x sont remplacées par $\texttt{old}(x)$. De même, un appel de fonction sans corps, de la forme

```
val f (..)
  requires P
  writes x1,...,xk
  ensures Q
```

est vu comme

$$\texttt{assert}\ P;\ \mathsf{Havoc}(x_1, \ldots, x_k; Q)$$

Traiter un Havoc de façon générale dans le moteur d'interprétation est un peu plus délicat : il y a besoin de renommage, puis d'élimination des variables temporaires introduites. Nous détaillons cela ci-dessous. Enfin, pour traiter la boucle, il faut savoir calculer des *post-point-fixes*. En général, on s'appuie sur un opération spéciale du domaine : le *widening*. Avec les booléens, il se trouve que la disjonction (c.-à-d. l'union) est une opération de *widening* convenable car il ne peut y avoir de séquence indéfiniment croissante.

2.2 La bibliothèque OCaml-BDD et nos extensions

La figure 3 présente un extrait de l'interface de la bibliothèque OCaml-BDD, montrant les opérations essentielles à connaître : les constructeurs de BDD et la fonction de test de satisfaisabilité. Les variables sont codées par des entiers. En réalité, cette interface est la signature d'un foncteur qui doit être utilisé en lui donnant une valeur `max_var` qui est le numéro maximal de variable. Cette interface est purement fonctionnelle, mais son implémentation utilise du *hash-consing* pour garantir un partage maximal, ce qui permet à la fonction `is_sat` d'être implémentée par un simple test d'égalité à `zero`.

```
type variable = int
  (** A variable is an integer, ranging from 1 to [max_var] *)
type t (** The abstract type of BDDs *)

(** smart constructors *)
val zero : t (* i.e. false *)
val one : t (* i.e. true *)
val mk_var : variable -> t
val mk_not : t -> t
val mk_and : t -> t -> t
val mk_or : t -> t -> t

val is_sat : t -> bool
  (** Checks if a bdd is satisfiable *)
```

Figure 3. Interface de la bibliothèque OCaml-BDD (extrait).

Comme vu dans la section 2.1, les opérations de négation, conjonction et disjonction nous permettent de réaliser la gestion des conditionnelles et des boucles d'un programme à interpréter, mais il manque de quoi gérer l'opération Havoc. Nous réalisons le transfert d'un état abstrait S à travers une opération

$$\mathsf{Havoc}(x_1, \ldots, x_k; C)$$

en la décomposant en plusieurs étapes :

1. considérer des variables $x'_1, \ldots, x'_k$ n'apparaissant pas déjà dans S, visant à dénoter les nouvelles valeurs des variables x_i ;
2. calculer la conjonction S' de S et C', où C' est obtenue en remplaçant dans C les occurrences de x_i par x'_i et les occurrences de $\texttt{old}(x_i)$ par x_i ;
3. éliminer de S' toutes les anciennes occurrences de $x_1, \ldots, x_k$, élimination vue comme une *élimination de quantificateur existentiel* ;
4. renommer les variables $x'_1, \ldots, x'_k$ en leur version sans prime.

Expliquons cette idée d'élimination existentielle sur un exemple : imaginons un état abstrait $S = a \wedge \neg b$ et l'opération d'échange

$$\mathsf{Havoc}(a, b; a = \texttt{old}(b) \wedge b = \texttt{old}(a))$$

on introduit donc l'état ($\leftrightarrow$ dénote l'équivalence)

$$S' = (a \wedge \neg b) \wedge (a' \leftrightarrow b \wedge b' \leftrightarrow a)$$

on suppose alors que la bibliothèque de BDD fournit une opération qui permet de construire le BDD pour la formule

$$\exists a, b.S'$$

Si l'implémentation est au point, on doit obtenir une formule équivalente à $\neg a' \wedge b'$ qui après renommage donnera l'état $\neg a \wedge b$. Ce mécanisme fonctionne donc si cette opération d'élimination existentielle est fournie, ce qui n'était pas le cas dans OCaml-BDD au départ. Il s'agit donc d'une contribution que nous avons faite à cette bibliothèque. La fonction fournie a le type `(variable -> bool) -> t -> t`, la fonction en argument indiquant si une variable doit être éliminée. La figure 4 présente le code du *smart constructor* en question, pour les lecteurs et lectrices qui seraient intéressées par la façon d'implémenter une telle fonction de manière efficace, en profitant du partage.

Comme vu ci-dessus, il nous faut une opération de renommage, qui n'est pas évidente sur les BDD, dont l'ordre des variables est imposé. Nous reviendrons sur ce sujet à la section suivante, pour le moment nous pouvons considérer qu'un renommage disons de x en une variable fraîche y peut-être effectué par une conjonction avec $x \leftrightarrow y$ puis l'élimination existentielle de x.

Tous les éléments sont maintenant disponibles pour écrire un interpréteur abstrait de base, selon les principes décrits à la section 2.1. À titre d'illustration, un tel interpréteur permet de générer, sur l'exemple de la figure 2, l'invariant de boucle :

$$\begin{array}{l}\texttt{if } a \texttt{ then if } b \texttt{ then true else if } c \texttt{ then false else true}\\ \qquad\texttt{else if } b \texttt{ then if } c \texttt{ then false else true else true}\end{array}$$

Le moins que l'on puisse dire est que ce n'est pas très lisible. Ainsi, un tout dernier élément que nous avons proposé comme contribution à OCaml-BDD est la reconstruction d'une formule à partir d'un BDD, reconstruction qui essaye de reconstituer des connecteurs booléens quand c'est possible. Avec cette amélioration cosmétique, l'invariant généré est

$$\texttt{if } a \texttt{ then } (b \vee \neg c) \texttt{ else } (\neg b \wedge \neg c)$$

On peut d'ailleurs calculer la conjonction avec la négation de la condition de boucle, pour obtenir l'état de sortie de boucle $a \wedge b \wedge c$ qui en particulier permet de garantir que l'assertion finale du code de l'exemple de la figure 2 est vraie.

```
let rec quantifier_elim cache op filter b =
  try H1.find cache b with Not_found ->
    let res = match b.node with
      | Zero | One -> b
      | Node(v,l,h) ->
         let low = quantifier_elim cache op filter l in
         let high = quantifier_elim cache op filter h in
         if filter v then op low high else mk v low high
    in H1.add cache b res; res

let mk_exist filter b =
  let cache = H1.create cache_default_size in
  quantifier_elim cache mk_or filter b
```

Figure 4. Le code du constructeur d'élimination existentielle. Un cache est créé à chaque appel, pour ne calculer qu'une seule fois sur les nœuds du DAG. La descente récursive, quand elle rencontre une variable à éliminer, effectue une disjonction des sous-arbres. Le constructeur interne `mk` permet d'effectuer un *hash-consing* et également simplifie quand les deux sous-arbres sont identiques.

3 Diagrammes de décision binaire paramétriques

Dans la section 2, nous avons montré comment la bibliothèque OCaml-BDD, augmentée de quelques ajouts, permet de réaliser un interpréteur abstrait précis et efficace sur des programmes ayant uniquement des variables booléennes. Nous nous intéressons maintenant au cas des programmes ayant des variables non booléennes, en particulier des variables entières. On peut imaginer de construire un domaine abstrait « produit cartésien », qui traiterait en parallèle les booléens et les entiers, mais alors on n'aurait aucune possibilité d'exprimer des relations entre eux, comme dans l'exemple de la figure 1. En fait, un tel domaine produit est facile à faire dans un premier temps, et sans surprise, sur l'exemple de la figure 1, l'invariant généré est $0 \leq x \leq 2058$, exactement celui trouvé par Frama-C.

Dans la mesure où les exemples d'application que l'on vise comportent malgré tout en grande majorité des variables booléennes, nous avons cherché à continuer à se baser sur les BDD. L'idée que nous proposons maintenant est de « nuancer » les arbres de décision binaires, en étendant les valeurs possibles aux feuilles de ces arbres : au lieu de se limiter aux feuilles `true` et `false`, nous proposons de mettre aux feuilles des BDD des contraintes issues d'un autre domaine, par exemple un domaine de contraintes linéaires sur les entiers.

Il s'avère impossible de continuer à utiliser la bibliothèque OCaml-BDD directement, puisque nous souhaitons modifier l'implémentation au cœur de la structure. Pour rester néanmoins le plus générique possible, nous avons développé une version fonctorisée de la bibliothèque, où l'argument du foncteur donne les éléments du domaine sous-jacent nécessaire. Nous avons appelé cela des *BDD paramétrés*. La bibliothèque `BDDparam` construite est décrite dans la section 3.1 qui suit, et est ensuite instanciée dans la section 3.2. Le code est disponible comme sous-répertoire du dépôt de Why3 [8].

3.1 Merci foncteur

La difficulté qui se présente pour réaliser notre foncteur est que nous voulons conserver la propriété essentielle des BDD, à savoir un partage maximal des sous-arbres. Il nous faut supposer que le domaine paramètre soit capable de proposer une fonction d'égalité et une fonction de hachage sur son langage de contraintes qui lui permette d'identifier efficacement les contraintes identiques. C'est une hypothèse que nous faisons. Mais une

difficulté supplémentaire va se rajouter, qui s'explique quand nous voulons instancier notre foncteur avec la bibliothèque Apron [14] : une contrainte linéaire sur des variables y est associée à un *environnement*, qui en interne associe des noms de variables à des numéros, correspondant à des indices dans les matrices qui servent à calculer efficacement sur les polyèdres. L'égalité de deux contraintes de ce langage n'a de sens que pour deux contraintes sur le même environnement.

Nous avons alors cherché à minimiser la taille de la signature du domaine paramètre, tout en permettant de supporter ces caractéristiques complexes. Le résultat est présenté sur la figure 5. Le type `p_index`, qui est explicitement un entier, est utilisé comme identifiant unique d'une contrainte, dans un contexte donné. Le type `p_context` de ces contextes est abstrait dans cette signature. Enfin, le type des contraintes est abstrait également, désigné sous le nom de `p_state`. Nous utilisons en effet un vocabulaire proche de l'application à l'interprétation abstraite, en particulier `meet` désigne la conjonction et `join` la disjonction. Nous déléguons la tâche de *hash-cons-er* les états à chaque instance : en effet, l'idée est que la table de hash-consing qu'il est nécessaire de maintenir soit une partie intégrante du

```
module type ParamDomain = sig

  type p_index = int
  type p_context
  type p_state

  val meet : p_context -> p_index -> p_index -> p_index option
  val join : p_context -> p_index -> p_index -> p_index option
  val widening : p_context -> p_index -> p_index -> p_index option
  (** when result is [None], it means [bottom] for [meet] and [top]
      for [join] and [widening] *)

  val exist_elim : p_context -> p_index -> p_context -> p_index option
  (** [exist_elim ctxi i ctx] fetches the formula [i] in context
     [ctxi], builds a corresponding formula in context [ctx] and
     returns an index for the result. When result is [None], it means
     [top]. The target context [ctx] is supposed to be a sub-context
     of [ctxi], so that this operation corresponds to existential
     elimination of variables of [ctxi] that are not in [ctx]. *)

  val entails : p_context -> p_index -> p_index -> bool
  (** returns [true] when the first formula entails the second *)

  val change : (p_state -> p_state) ->
    p_context -> p_index -> p_context  -> p_index
  (** [change f ctx_src i ctx_dst] returns an index in context
     [ctx_dst] for [f s] where [s] is the state of index [i] in
     [ctx_src] *)

  val meet_with_constraint : (p_state -> p_state) ->
    p_context -> p_index option -> p_index option
  (** [meet_with_constraint f ctx i] returns an index in context
     [ctx] for [f s] where [s] is the state of index [i] in
     the same context. The case [None] here means [bottom]  *)

end
```

Figure 5. La signature du module attendu comme argument du foncteur `BDDparam`.

contexte. Ainsi la fonction `meet` reçoit deux indices de contraintes dans le *même* contexte, et renvoie l'indice de la conjonction, ou bien `None` si la contrainte résultante est `false`. Des fonctions similaires permettent de calculer les `join` (disjonction) et `widening`, mais également l'élimination des quantificateurs existentiels, comme identifiée comme nécessaire dans la section précédente. Comme l'élimination des variables quantifiées existentiellement change les variables de la contrainte, le contexte du résultat est un autre contexte. La fonction `entails` permet décider si une constrainte est conséquence logique d'une autre. La fonction `change` est une fonction générique de mapping de formule à formule, avec contextes différents en entrée et sortie. Elle doit permettre (entre autres) un renommage de variables. Enfin, la fonction `meet_with_constraint` est similaire à `change` mais s'applique cette fois sur le même contexte en entrée et en sortie, et supporte le cas où les formules sont `false` en particulier en sortie.

```
1  type t = { tag: int; node : view }
2  and view = Zero | One | Leaf of int | Node of variable * bdd * bdd
3
4  let mk_param (i : int) : t =
5    try Hint.find param_table i with Not_found ->
6      let t = gentag () in
7      let n = { tag = t; node = Leaf i } in
8      Hint.add param_table i n; n
9
10 let mk v low high =
11   if low == high then low else hashcons_node v low high
12
13 let meet ctx b1 b2 =
14   let cache = H2.create cache_default_size in
15   let rec app ((u1,u2) as u12) = if u1 == u2 then u1 else
16       match u1.node, u2.node with
17       | Zero,_ | _, Zero -> zero
18       | One,_ -> u2 | _, One -> u1
19       | _ -> try H2.find cache u12 with Not_found ->
20           let res = match u1.node,u2.node with
21             | Node(v1,l1,h1), Node(v2,l2,h2) ->
22               if v1 == v2 then
23                 mk v1 (app (l1, l2)) (app (h1, h2))
24               else if v1 < v2 then
25                 mk v1 (app (l1, u2)) (app (h1, u2))
26               else (* v1 > v2 *)
27                 mk v2 (app (u1, l2)) (app (u1, h2))
28             | Leaf i, Leaf j ->
29               begin match D.meet ctx i j with
30                 | Some k -> mk_param k
31                 | None -> zero
32               end
33             | Node(v1,l1,h1), Leaf _ ->
34                 mk v1 (app (l1, u2)) (app (h1, u2))
35             | Leaf _,Node(v2,l2,h2) ->
36               mk v2 (app (u1, l2)) (app (u1, h2))
37             | (Zero|One),_ | _,(Zero|One) -> assert false
38           in
39           H2.add cache u12 res; res
40   in
41   app (b1, b2)
```

Figure 6. Extrait du code du foncteur `BDDparam`. L'opération `meet` utilise un cache sur des paires de BDD pour mémoïser les appels identiques. Aux lignes 28–32, on fait appel à l'opération `meet` du domaine paramètre `D` pour faire une conjonction de deux feuilles. Aux lignes 33–36 on repousse récursivement les conjonctions d'une feuille avec un nœud interne de BDD.

Notre foncteur prend un module de la signature précédente en argument. La signature de sortie de ce foncteur est très similaire à l'extrait de la figure 3, et en particulier elle reste purement fonctionnelle. Un ajout est que pour implémenter une version plus efficace du renommage de variable que la méthode proposée brièvement à la fin de la section 2.1, nous ajoutons explicitement une telle fonction de renommage.

Un extrait de l'implémentation de notre foncteur `BDDparam` est donné sur la figure 6, montrant le type de donné interne d'arbre, les constructeurs `mk_param` et `mk` pour respectivement les contraintes paramétres aux feuilles des BDD et leurs nœuds internes, et enfin la fonction

```
module ApronDom = struct

  type p_index = int
  type p_state = Polka.strict Polka.t Abstract1.t
  type p_context = {
    apron_env : Environment.t;
    mutable counter : int;
    state_to_int : int H.t;
    int_to_state : p_state Hint.t;
  }

  let get ctx i =
    try Hint.find ctx.int_to_state i with Not_found ->
      Format.eprintf "get:␣index␣%d␣not␣found␣in␣ctx@." i; assert false

  let record ctx s =
    try H.find ctx.state_to_int s with Not_found ->
      let i = ctx.counter in
      ctx.counter <- ctx.counter + 1;
      Hint.add ctx.int_to_state i s;
      H.add ctx.state_to_int s i; i

  let meet ctx i j =
    let si = get ctx i in let sj = get ctx j in
    let s = Abstract1.meet man si sj in
    if Abstract1.is_bottom man s then None else Some (record ctx s)

  let exist_elim ctxi i ctx =
    let si = get ctxi i in
    let s = Abstract1.change_environment man si ctx.apron_env false in
    if Abstract1.is_top man s then None else Some (record ctx s)

  let entails ctx i j =
    let si = get ctx i in let sj = get ctx j in Abstract1.is_leq man si sj

  let change f ctxi i ctx =
    let si = get ctxi i in let s = f si in record ctx s

  let meet_with_constraint f ctxi i =
    let s = match i with
      | Some i -> let si = get ctxi i in f si
      | None -> f (Abstract1.top man ctxi.apron_env)
    in if Abstract1.is_bottom man s then None else Some (record ctxi s)

end
```

Figure 7. L'instance de la signature de la figure 5 avec le support des contraintes linéaires fournies par la bibliothèque Apron. Le contexte contient un environnement Apron, et deux tables de hachage réciproques l'une de l'autre. La fonction `get` permet de récupérer une formule à partir de son indice, et la fonction `record` enregistre une formule dans ce contexte, en lui attribuant un nouvel indice si elle n'y est pas déjà.

`meet`, qui fait appel à l'opération meet du domaine paramètre au niveau des feuilles. Le module `Hint` fournit des tables de hachage indexées par des entiers.

3.2 Une instance : un domaine abstrait mixte entiers-booléens

Notre instance avec des contraintes linéaires sur les entiers utilise la bibliothèque Apron [14]. Le module à passer en argument du foncteur est donné sur la figure 7. Le module `H` fournit des tables de hachage indexées par des formules Apron, pour lesquelles des fonctions de hachage et de comparaison sont disponibles.

4 Évaluations expérimentales

Nos évaluations se basent sur une comparaison avec le prototype InferLoop précédent de Why3. Comme annoncé en accroche d'introduction, BDDinfer fait aussi bien que InferLoop sur le code de la figure 1, en générant un invariant permettant de prouver la post-condition. Nous montrons ci-dessus un premier exemple qui montre que BDDinfer peut gagner significativement en précision. Puis, dans un second temps, nous comparons avec InferLoop en terme de temps de calcul, cette fois sur du code WhyML généré automatiquement à partir de Ladder.

4.1 Un exemple de gain de précision

Le code de la figure 8 est un exemple jouet, construit par inspiration et réduction d'exemples réels de modélisation de codes Ladder. Avec InferLoop, l'invariant généré est

$$\neg s_1 \vee (0 \leq c \leq 42 \wedge s_1 \wedge \neg serr)$$

L'invariant est insuffisamment précis pour que l'assertion soit prouvable. Avec BDDinfer, l'invariant généré est

$$\texttt{if } serr \texttt{ then } (\neg s_1 \wedge \neg s_0 \wedge 43 \leq c) \texttt{ else } (\texttt{if } s_1 \texttt{ then } \neg s_0 \wedge 0 \leq c \leq 42 \texttt{ else } s_0 \wedge c = 0)$$

on voit que l'invariant en forme de conditionnelle capture précisément les différents phases d'itération. L'assertion est prouvable. Noter que l'on peut produire une variation de cet

```
let f ()
  requires { s0 = True}
  requires { s1 = serr = False}
  requires { c = 0 }
  =
   while (not !serr) do
     if !s1 then c := !c +1;
     let tmp = random_bool () in
     if !s0 && tmp then
        (s0 := False; s1 := True; c := 0)
     else
       if (!s1 && (!c > 42)) then
         (s1 := False; serr := True)
   done;
   assert { !serr ∧ !c ≥ 43}
```

Figure 8. Un exemple illustrant l'importance de la précision des invariants.

exemple en insérant un `break` après `serr := True` (et du coup mettre la condition de boucle à `True`). L'invariant généré est alors

$$(\texttt{if } s_1 \texttt{ then } \neg s_0 \wedge 0 \leq c \leq 42 \texttt{ else } s_0 \wedge c = 0) \wedge \neg serr$$

L'assertion reste prouvable, et dans les faits on peut même se passer complètement de la variable `serr`.

4.2 Évaluation sur du code Ladder

Belo et al. [5] présentent une façon de vérifier qu'un programme Ladder satisfait une spécification donnée sous la forme d'un diagramme de temps. Un tel diagramme de temps représente un scénario temporel de changement de valeur des variables d'entrées du programme, assorti de la spécification des valeurs des variables de sortie associées. Cette vérification consiste en la traduction automatique d'un code Ladder et de sa spécification temporelle en un programme WhyML. La faisabilité de l'approche est démontrée en se basant sur le cas d'usage d'un programme de contrôle d'un chariot (figure 9). Son code Ladder est donné figure 10 est sa traduction automatique est détaillée dans la figure 11. Nous renvoyons à Belo et al. [5] pour les explications concernant la sémantique du langage Ladder, les détails d'implémentation en WhyML des instructions `RST`, `SET`, `PLS`, ainsi que celles dédiées à la gestion des minuteurs.

Le comportement attendu du chariot, et donc de son code Ladder, est spécifié par un diagramme de temps, donné dans la figure 12. Un tel diagramme de temps spécifie des changements séquentiels des valeurs d'entrée du programme (et spécifie les valeurs de sortie attendues). Nous appelons ces changements de valeurs d'entrée des *évènements*. Entre deux évènements, les valeurs de sortie sont spécifiées stables. Nous appelons *états* les ensembles d'exécutions du programme entre deux évènements consécutifs. Le nombre d'exécutions du programme durant un état est arbitrairement grand.

Pour modéliser l'exécution du programme Ladder en regard du diagramme de temps spécifié, Belo et al. proposent l'encodage suivant. Si E_i est un évènement du diagramme de temps, et S_i est son état associé. Soit X_i la variable d'entrée dont la valeur change à l'évènement E_i (sur l'exemple du chariot et son diagramme de la figure 12, X_1 est X0). Soit G_i la conjonction d'assertions concernant les valeurs d'entrée du programme pour l'évènement E_i et l'état S_i (sur l'exemple du chariot, G_1 est X0∧¬X1∧¬X2∧X3∧X4). On

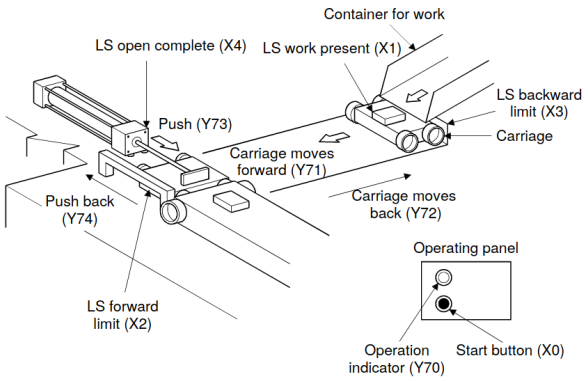


Figure 9. Carriage line control : le chariot transporte un objet le long de sa ligne, puis revient à son point initial, après qu'un bras mécanique ait enlevé l'objet du chariot.

note I_i la conjonction d'assertions concernant les valeurs de sorties pour l'évènement E_i et l'état S_i (sur l'exemple du chariot, I_1 est la formule Y70∧¬Y71∧¬Y72∧¬Y73∧¬Y74).

La modélisation en WhyML d'un évènement E_i et de l'état associé S_i se fait sous la forme d'une boucle de style *do-while*. Les boucles *do-while* ne sont pas présentes dans la syntaxe de WhyML, nous utilisons donc deux fois le corps du programme Ladder traduit en WhyML, une première fois pour modéliser l'exécution (unique) correspondant à E_i, et une deuxième fois dans le corps d'une boucle *while* attenante qui simule les exécutions (dont le nombre est arbitraire, possiblement zéro) de l'état associé S_i. La condition de garde de la boucle correspond à l'hypothèse G_i sur les valeurs des variables d'entrée pour E_i et S_i. La spécification concernant les valeurs des variables de sortie est donnée sous la forme

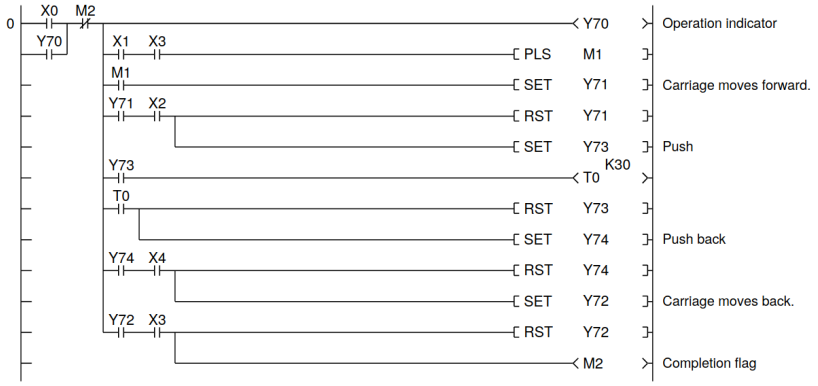
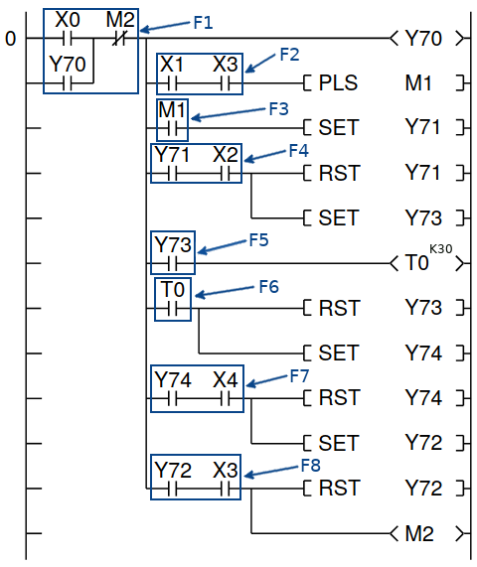


Figure 10. Carriage line control : programme Ladder



(a) Programme Ladder

```
let scan () =
  let f1 = (!x0 || !y70)
        && (not !m2) in
  y70 := f1;
  let f2 = !x1 && !x3 in
  pls (f1 && f2) m1 cc0;
  let f3 = !m1 in
  set (f1 && f3) y71;
  let f4 = !y71 && !x2 in
  rst (f1 && f4) y71;
  set (f1 && f4) y73;
  let f5 = !y73 in
  timer_coil (f1 && f5) t0 30;
  let f6 = timer_contact t0 in
  rst (f1 && f6) y73;
  set (f1 && f6) y74;
  let f7 = !y74 && !x4 in
  rst (f1 && f7) y74;
  set (f1 && f7) y72;
  let f8 = !y72 && !x3 in
  rst (f1 && f8) y72;
  m2 := f1 && f8
```

(b) Encodage WhyML

Figure 11. Encodage d'une exécution du programme Ladder

de l'invariant I_i à prouver : l'initialisation de l'invariant correspond à la spécification de l'évènement tandis que sa préservation correspond à celle de l'état. Enfin, nous modélisons l'aspect non-déterministe du changement d'état par une affectation de la variable d'état X_{i+1} dont le changement déclenche l'évènement suivant E_{i+1}, à l'aide de la fonction `random_bool`.

Nous résumons cette formalisation par le pseudo-code

```
(* evenement E_i *)
scan();
X_{i+1} := random_bool();

(* etat S_i *)
while G_i do
  invariant { I_i }
  scan();
  X_{i+1} := random_bool();
done
```

dans lequel `scan()` représente l'exécution du code WhyML donnée en figure 11b.

Notre objectif initial était de directement encoder la succession de tous les évènements et états dans une même fonction WhyML afin de lancer la vérification grâce au calcul de plus faible pré-condition de Why3 et l'appel à des prouveurs SMT. Malheureusement les performances de l'interpréteur abstrait InferLoop de Why3 ne le permettaient pas, son temps de calcul semblant exponentiel en le nombre de boucles successives traitées (voir ci-dessous). Nous avons alors dû implémenter un mécanisme permettant de traiter chaque paire d'évènement et état dans une fonction séparée, en ré-injectant l'invariant généré pour l'état S_i comme pré-condition pour la formalisation de l'évènement E_{i+1} et de l'état S_{i+1}. Cela nous a permis de contourner ce problème de performances, en atteignant la vérification complète du cas d'usage présenté en un temps acceptable de l'ordre de 25 secondes (dont plus de 17 pour la génération d'invariants).

Nous avons donc comparé les performances du nouvel interpréteur abstrait BDDinfer avec celles de l'ancien InferLoop. Dans ce but, nous chronométrons la génération d'invariants sur différents fichiers WhyML. Les résultats sont présentés sur la figure 13. L'utilisation de BDD paramétrés dans le nouveau moteur d'interprétation abstraite BDDinfer de Why3 permet d'obtenir de bien meilleures performances. Cette amélioration des performances nous permet ainsi de ne plus avoir à créer des fonctions WhyML distinctes pour les différents évènements et états, et surtout de ne plus compliquer l'outillage nécessaire pour propager les invariants

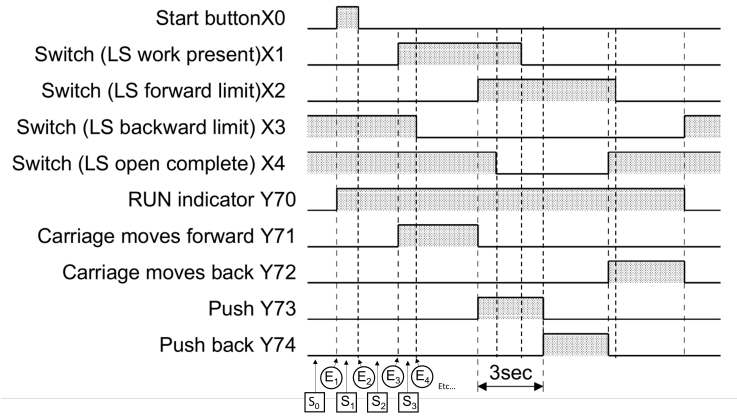


Figure 12. Diagramme de temps

év./état	1	2	3	4	5	6	7	8	9	10
InferLoop	1.02	0.95	2.05	0.97	5.45	1.49	1.21	1.44	1.29	0.86
BDDinfer	0.12	0.10	0.12	0.10	0.33	0.10	0.09	0.13	0.12	0.08
év./états	1	1→2	1→3	1→4	1→5	1→6	1→7	1→8	1→9	1→10
InferLoop	1.02	33.5	292	1089	-	-	-	-	-	-
BDDinfer	0.12	0.19	0.30	0.42	0.53	0.91	1.00	1.10	1.22	1.35
# vars	47	95	143	191	239	287	335	379	427	475

Figure 13. Comparaison des performances de l'ancien générateur d'invariant InferLoop de Why3 et du nouveau générateur BDDinfer basé sur les BDD paramétrés. La première partie de la table compare les performances, mesurées en secondes de temps CPU, pour une unique paire évènement/état. Pour chacun de ces tests, qui ne comportent qu'une seule boucle, le temps de calcul est de manière systématique de l'ordre de 10 à 20 fois plus rapide avec BDDinfer. La seconde partie compare les performances pour des séries d'évènement/état du point de départ jusqu'à un état E_i donné pour i de plus en plus grand. Chaque test contient donc une boucle de plus que le précédent. Avec InferLoop, l'explosion combinatoire est spectaculaire, il est même impossible d'atteindre le quatrième évènement dans un temps raisonnable. Alors qu'avec BDDinfer, le temps de calcul augmente plus ou moins linéairement avec le nombre de boucles, et reste très raisonnable, moins de deux secondes pour la série complète. La toute dernière ligne donne le nombre de variables booléennes utilisées dans les BDD sous-jacents. On peut constater que la bibliothèque `BDDparam` tient bien le choc avec un nombre de variables allant jusqu'à environ 500.

générés pour un état E_i vers l'évènement suivant. Au final, nous sommes capables de traiter le cas d'usage complet en moins de 2 secondes, contre 25 précédemment.

Cette amélioration des performances nous permet également de simplifier notre implémentation de la génération de scénarios d'erreur quand un diagramme de temps n'est pas vérifié. En effet, quand l'encodage WhyML d'une paire d'état et d'évènement n'était pas prouvé par Why3, nous reconstruisions alors un scénario d'erreur Ladder composé de deux types d'éléments distincts. En premier lieu, nous collectons les domaines générés par l'interpréteur abstrait qui sont retournés comme valeurs possibles des variables internes du programme lors de l'exécution des états. En un second temps, on collecte des valeurs concrètes pour l'exécution des évènements, provenant du contre-exemple fourni par le prouveur SMT, lors de la tentative de preuve de la concaténation des modélisations WhyML des états et évènements successifs depuis le premier jusqu'à celui incriminé (en ré-injectant à nouveau les invariants générés pour tous les états concernés). Nous pouvons maintenant générer de tels scénarios d'erreur directement sans avoir à générer les invariants des états un à un, pour les ré-injecter *a posteriori*.

De plus la précision accrue du nouvel interpréteur abstrait BDDinfer a permis de construire de meilleurs scénarios d'erreur en cas de diagramme de temps non vérifié. En effet, pour modéliser la spécification concernant les minuteurs, nous introduisons une variable fraîche de comptage des exécutions pour chaque minuteur spécifié, et tentons d'obtenir l'identification de cette variable fraîche avec celle correspondant au minuteur grâce aux domaines relationnels de l'interpréteur abstrait (voir [5] pour plus de détail). Malheureusement le domaine généré n'était pas toujours suffisamment précis et il arrivait qu'il ne soit pas plus précis que $[0, +\infty[$ (typiquement lorsque l'on modifiait le programme pour que le nombre d'exécutions entre les évènements 5 et 8 ne corresponde plus au minuteur spécifié dans le diagramme de temps). Le nouvel interpréteur abstrait paraît bien plus précis et nous n'avons pas pu le mettre en

défaut sur les exemples que nous avions.

En conclusion, les améliorations appportées par ce nouvel interpréteur abstrait utilisant les BDD paramétrés nous ont permis de simplifier grandement notre façon de modéliser un programme Ladder et son diagramme de temps associé. L'amélioration des performances permet également de nous rassurer sur le passage à l'échelle de notre démarche de vérification à des programmes Ladder et diagrammes de temps plus conséquents. Enfin, l'amélioration de la précision des invariants générés et des domaines associés, a permis d'améliorer nettement les scénarios d'erreur quand une violation du diagramme de temps est détectée. De plus, cette amélioration de précision permet également d'entrevoir la possibilité d'appliquer cette méthodologie de vérification à des formes de spécifications temporelles plus complexes : passer d'un scénario représenté par un diagramme de temps linéaire, à un ensemble de scénarios décrits par une certaine forme d'automate.

5 Conclusions

Nous avons présenté un interpréteur abstrait basé sur une version originale de BDD, paramétrés par un domaine. Cette version se présente sous forme d'un foncteur OCaml. Nous avons montré comment instancier ce foncteur sur un domaine de contraintes linéaires sur les entiers. Nous avons évalué notre interpréteur sur des exemples, issus d'un contexte d'utilisation industrielle pour l'analyse statique de programmes Ladder pour les PLC. Ces exemples se caractérisent par un grand nombre de variables booléennes, mais aussi des variables entières. Nos évaluations montrent que nos BDD paramétrés sont particulièrement bien adaptés à ce type de programmes, aussi bien en terme de précision des invariants générés que de temps de calcul.

Une question que nous n'avons pas abordée est celle de la correction de l'approche : sommes-nous sûr que les invariants générés sont bien des invariants ? Nous n'avons pas de garantie forte, puisque nous dépendons du code OCaml de la bibliothèque `BDDparam` et de celui de l'interpréteur abstrait BDDinfer lui-même. Néanmoins notre implémentation fournit un garde-fou très sûr : nous demandons à Why3 de reprouver que chaque invariant généré est initialement vrai et préservé par une itération. De cette façon nous avons une très forte garantie de correction de ces invariants. Cette méthode a d'ailleurs pu quelques fois détecter un bug dans notre implémentation !

5.1 Travaux connexes

Nous n'avons pas été facilement en mesure de comparer notre approche avec des approches existantes en interprétation abstraite. D'une part au niveau des outils eux-mêmes avec lesquels expérimenter, nous n'avons pas eu un grand succès, en particulier parce que nous recherchions des bibliothèques en OCaml. Nous avons tenté d'utiliser la bibliothèque MOPSA [15], mais l'API fournie est de trop haut niveau, elle ne permet pas de récupérer des invariants qui seraient généré au niveau de chaque boucle. Nous avons tenté d'utiliser la bibliothèque BDDApron [13] dont le nom est prometteur, mais nous avons dû renoncer par manque de compréhension de son interface et de son API, et qui plus est cette bibliothèque ne semble pas maintenue, et n'a pas de paquet OPAM compatible avec les versions d'OCaml supérieures à 4.08. BDDApron a été utilisée par Schrammel et Jeannet [17] pour analyser statiquement des codes Lustre, qui sont d'une certaine façon similaire à des programmes Ladder, par l'aspect data-flow synchrone. Cette similarité renforce l'idée que des domaines à base de BDD sont bien adaptés dans un tel contexte.

Dans la communauté de l'interprétation abstraite en général, l'utilisation de BDD ne semble pas du tout être populaire. On peut arguer que de toute façon les programmes où les variables booléennes sont prépondérantes ne sont pas une cible habituelle. Néanmoins, notre approche nous semble aller plus loin que la simple idée d'interpréter les opérations booléennes : la structure de BDD encode très naturellement des disjonctions de contraintes.

Pour gérer de telles disjonctions, on trouve des approches comme la *powerset completion* [9], le *state partitioning* ou *trace partionining* [16], les analyses *path-sensitive* [12]. On trouve dans la littérature une notion de *decision tree abstract domains* [6] dont les présentations sont beaucoup plus abstraites que la nôtre, et dont il ne semble pas exister d'implémentation disponible. Il reste que les idées proposées et le vocabulaire utilisé par Chen et Cousot [6] sont clairement similaires aux nôtres, et nous avons, sans le savoir au départ, redécouvert cette idée d'utiliser des BDD. Par contre, dans l'article en question on ne voit pas apparaître le besoin pratique d'une opération d'élimination des quantifications existentielles, qui est pour nous un ajout indispensable. Urban et Miné [18] utilisent aussi une structure similaire appelé *Decision Tree Abstract Domain*, ceci pour un but très spécifique de preuve de terminaison. On y retrouve l'idée que les arbres de décision binaires sont une structure intéressante comme alternative au partitioning.

De notre point de vue, il nous semble que les BDD ont une plus grande popularité dans les communautés SMT, model-checking et résolution de contraintes, que dans l'interprétation abstraite. Au vu de nos expérimentations, cela semblerait plutôt un oubli dommageable.

5.2 Perspectives

Notre prototype BDDinfer est actuellement très spécialisé sur du code ne comportant que des variables entières et des variables booléennes. Dès que l'on lui propose un code qui comporte d'autres types de données, aucun invariant n'est inféré. À court terme nous souhaitons étendre le support en rajoutant un domaine générique de fonctions non-interprétés (classe UF dans le jargon des solveurs SMT) qui permettra de traiter beaucoup plus de programmes. Cela sera l'occasion d'instancier notre foncteur de BDD paramétrés sur un autre domaine que des contraintes linéaires.

Comme évoqué en fin de la section 4, les améliorations d'efficacité et de précision nous permet maintenant la possibilité d'appliquer cette méthodologie de vérification à des formes de spécifications temporelles plus complexes.

Une question ouverte pour nous est d'identifier l'intérêt que peut avoir notre méthode très centrée sur les booléens dans la communauté de l'interprétation abstraite en général. Comme il ressort ci-dessus, nous ne savons pas bien comparer notre approche avec les techniques existantes de gestion des disjonctions. N'étant pas des spécialistes de l'interprétation abstraite, nous serions heureux que la publication de cet article puisse nous permettre d'obtenir un retour d'experts de cette communauté.

Remerciements

Nous remercions Cláudio Belo Lourenço, qui a mis à jour le prototype InferLoop en lui ajoutant un support des variables booléennes, et en le rendant plus robuste ; Yacine El Haddad, qui a mis en place le prototype initial de l'interpréteur abstrait BDDinfer ; et Jean-Christophe Filliâtre, dont la bibliothèque OCaml-BDD a permis le succès de notre approche, et qui a accepté d'y intégrer nos patches proposant des extensions. Nous remercions également Florian Faissole, Inoue Hiroaki et David Mentré qui ont participé à définir notre méthode pour la vérification de programmes Ladder en regard d'un diagramme de temps.

Références

[1] Lucas Baudin. Deductive verification with the help of abstract interpretation. Technical report, Univ Paris-Sud, November 2017. URL : `https://hal.inria.fr/hal-01634318`.

[2] Patrick Baudin, François Bobot, David Bühler, Loïc Correnson, Florent Kirchner, Nikolai Kosmatov, André Maroneze, Valentin Perrelle, Virgile Prevosto, Julien Signoles,

and Nicky Williams. The dogged pursuit of bug-free C programs : The Frama-C software analysis platform. *Communications of the ACM*, 64(8) :56–68, 2021. doi:10.1145/3470569.

[3] Patrick Baudin, Pascal Cuoq, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL : ANSI/ISO C Specification Language, version 1.16*, 2020. URL : https://frama-c.com/html/acsl.html.

[4] Benedikt Becker, Cláudio Belo Lourenço, and Claude Marché. Explaining counterexamples with giant-step assertion checking. In José Creissac Campos and Andrei Paskevich, editors, *6th Workshop on Formal Integrated Development Environments (F-IDE 2021)*, Electronic Proceedings in Theoretical Computer Science, May 2021. URL : https://hal.inria.fr/hal-03217393, doi:10.4204/EPTCS.338.10.

[5] Cláudio Belo Lourenço, Denis Cousineau, Florian Faissole, Claude Marché, David Mentré, and Hiroaki Inoue. Automated formal analysis of temporal properties of Ladder programs. *International Journal on Software Tools for Technology Transfer*, 24(6) :977–997, 2022. URL : https://hal.inria.fr/hal-03737869, doi:10.1007/s10009-022-00680-0.

[6] Junjie Chen and Patrick Cousot. A binary decision tree abstract domain functor. In Sandrine Blazy and Thomas Jensen, editors, *Static Analysis*, pages 36–53, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[7] Sylvain Conchon and Jean-Christophe Filliâtre. Semi-persistent data structures. In *17th European Symposium on Programming (ESOP'08)*, Budapest, Hungary, April 2008. URL : https://inria.hal.science/hal-04045849, doi:10.1007/978-3-540-78739-6_25.

[8] Yacine El Haddad and Claude Marché. Source code for `bddinfer` library. Git repository, May 2023. https://gitlab.inria.fr/why3/why3/-/tree/master/src/bddinfer.

[9] Gilberto Filé and Francesco Ranzato. The powerset operator on abstract interpretations. *Theoretical Computer Science*, 222(1) :77–111, 1999. doi:10.1016/S0304-3975(98)00007-3.

[10] Jean-Christophe Filliâtre. An OCaml library for binary decision diagrams, version 0.4. Git repository, 2018. https://github.com/backtracking/ocaml-bdd.

[11] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — where programs meet provers. In Matthias Felleisen and Philippa Gardner, editors, *Proceedings of the 22nd European Symposium on Programming*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer, March 2013. URL : http://hal.inria.fr/hal-00789533.

[12] Julien Henry, David Monniaux, and Matthieu Moy. PAGAI : A path sensitive static analyser. In *3d Workshop on Tools for Automatic Program Analysis (TAPAS'2012)*, volume 289 of *Electronic Notes in Theoretical Computer Science*, pages 15–25, 2012. doi:10.1016/j.entcs.2012.11.003.

[13] Bertrand Jeannet. La bibliothèque BDDAPRON de domaines abstraits logico-numériques. Web site https://pop-art.inrialpes.fr/people/bjeannet/bjeannet-forge/bddapron/.

[14] Bertrand Jeannet and Antoine Miné. Apron : A library of numerical abstract domains for static analysis. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification*, pages 661–667. Springer, 2009.

[15] Matthieu Journault, Antoine Miné, Raphaël Monat, and Abdelraouf Ouadjaout. Démonstration de la plateforme Mopsa d'analyse statique de programmes par interprétation abstraite. In *JFLA 2021 - 32 èmes Journées Francophones des Langages Applicatifs*, 2021. URL : https://hal.science/hal-03190426.

[16] Xavier Rival and Laurent Mauborgne. The trace partitioning abstract domain. *ACM Trans. Program. Lang. Syst.*, 29(5), 2007. doi:10.1145/1275497.1275501.

[17] Peter Schrammel and Bertrand Jeannet. Logico-numerical abstract acceleration and application to the verification of data-flow programs. In Eran Yahav, editor, *Static Analysis Symposium*, volume 6887 of *Lecture Notes in Computer Science*, pages 233–248. Springer, 2011. doi:10.1007/978-3-642-23702-7_19.

[18] Caterina Urban and Antoine Miné. A decision tree abstract domain for proving conditional termination. In Markus Müller-Olm and Helmut Seidl, editors, *Static Analysis Symposium*, volume 8723 of *Lecture Notes in Computer Science*, pages 302–318. Springer, 2014. doi:10.1007/978-3-319-10936-7_19.