Vérification de programmes C concurrents avec Cubicle : Enfoncer les barrières

Sylvain Conchon[1,2] & David Declerck[1] & Luc Maranget[3] & Alain Mebsout[1,2]

1: Université Paris Sud, CNRS, F-91405 Orsay
2: INRIA Saclay – Île-de-France, F-91893 Orsay
3: INRIA Paris – Rocquencourt, F-78153 Le Chesnay
sylvain.conchon@lri.fr, david.declerck@u-psud.fr
luc.maranget@inria.fr, alain.mebsout@lri.fr

Résumé

Toutes les bibliothèques de *threads* au standard POSIX se doivent d'implémenter une barrière de synchronisation. Une telle structure de contrôle permet à des *threads* de s'attendre en un point donné d'un programme. Il existe de nombreuses implémentations pour ces barrières, plus ou moins sophistiquées. Citons par exemple, les *sense barriers*, les *static tree barriers*, les *tournament barriers*, etc. Nous présentons dans cet article un outil pour vérifier automatiquement la sûreté de barrières de synchronisation écrites en langage C. Pour être sûre, une barrière doit garantir qu'aucun *thread* ne peut la franchir tant que les autres ne l'ont pas atteinte, et ceci quelque soit le nombre de *threads* impliqués. Notre approche consiste à compiler des programmes C avec *threads* vers des systèmes de transition paramétrés, puis à vérifier leur sûreté à l'aide d'un *model checker*. Plus concrètement, notre compilateur traduit un sous-ensemble du C vers le langage d'entrée de Cubicle. On montre le bien fondé de notre technique par la preuve de sûreté de la barrière *sense reversing* pour un nombre quelconque de *threads*. À notre connaissance, c'est la première preuve automatique d'une telle implémentation.

1. Introduction

La programmation d'applications concurrentes avec processus légers (*threads*) repose habituellement sur l'utilisation de bibliothèques spécialisées. Par exemple, si on programme en langage C sous Linux, BSD ou Mac OS, on peut utiliser les bibliothèques NPTL ou GNU Pth qui implémentent la norme POSIX IEEE 1003.1c-1995 [16]. Cette norme définit une interface pour créer et ordonnancer des *threads*, gérer des *mutexes*, des signaux, et fournit également des mécanismes de synchronisation. Parmi ces mécanismes, on trouve les barrières.

Une barrière de synchronisation est une technique qui permet à des *threads* de se retrouver en un point donné du programme. Chaque *thread* qui participe à la barrière attend tous les autres. Lorsque le dernier arrive, tous les *threads* participant reprennent leur exécution. La norme POSIX offre une implémentation des barrières mais nous allons définir et prouver la nôtre. On définit donc un type des barrières (par exemple `barrier_t`). À cette structure de données on associe une fonction `barrier_init` pour l'initialiser avec un certain nombre de participants, ainsi qu'une fonction de synchronisation `barrier_wait`.

La conception d'une barrière de synchronisation est délicate. En effet, en plus des considérations habituelles liées à l'utilisation de verrous (attente active ou passive, réutilisation, opérations bloquantes, *etc.*), une bonne implémentation de barrière doit également minimiser le temps entre l'arrivée du dernier *thread* et la sortie de la barrière du dernier *thread*. Elle doit également faire

en sorte que tous les *threads* quittent la barrière à peu près au même moment. Ainsi, on trouve de nombreux algorithmes dans la littérature comme par exemple, les barrières centralisées (*sense* et *sense-reversing barrier*), les barrières arborescentes (*static tree* et *combining tree barrier*) ou les barrières à tableaux (*dissemination* et *tournament barrier*) [15]. Bien évidemment, en plus de ces considérations, il est primordial que l'implémentation d'une barrière soit sûre, c'est-à-dire qu'elle garantisse qu'aucun *thread* ne quitte la barrière avant l'arrivée des autres. De ce fait, la programmation de barrières de synchronisation est une tâche difficile, sachant que le débuggage de code concurrent est un cauchemar.

Le problème qui nous intéresse dans cet article est de vérifier formellement et automatiquement la sûreté, c'est-à-dire la synchronisation correcte, de barrières écrites en langage C. Les barrières de synchronisation étant conçues indépendamment du nombre de *threads* pour lesquelles elles seront utilisées, il advient donc de les vérifier pour un nombre quelconque de participants, *i.e.* de façon *paramétrée*.

D'une manière générale, la vérification de programmes concurrents est une tâche ardue. Cette activité remonte aux années 70, avec des travaux de recherche sur des extensions des méthodes manuelles de Hoare et de Floyd [5, 25], et au début des années 80 avec des approches automatiques par *model checking* [6, 27]. En ce qui concerne les programmes C concurrents, leur vérification est un domaine de recherche toujours très actif [7, 11, 17, 23, 24].

Dans cet article, on propose une nouvelle approche qui consiste à compiler des programmes C avec *threads* vers des systèmes de transition paramétrés, puis à vérifier leur sûreté à l'aide du *model checker* Cubicle [8]. Pour illustrer notre propos, on utilise l'implémentation d'une barrière *sense-reversing* présentée en section 2.

Nos contributions sont les suivantes :

— Un schéma de traduction d'un sous-ensemble du langage C concurrent vers des systèmes de transition à tableaux (section 4) dont les gardes et les actions sont décrites par des formules logiques du premier ordre. Ce fragment du langage C est limité (voir section 3) mais permet d'analyser de réelles barrières de synchronisation.

— Une analyse statique par typage pour déterminer si une variable de type `int` est utilisée comme une variable booléenne ou un compteur de *threads* (section 5). Ces informations sont nécessaires pour un traitement efficace par Cubicle.

On montre (section 6) que notre outil permet de vérifier plusieurs variantes de *sense-reversing barrier*. À notre connaissance, c'est la première preuve automatique de telles implémentations. Pour obtenir ces résulats, on a étendu l'algorithme d'atteignabilité arrière de Cubicle pour gérer plus efficacement les quantificateurs universels dans les gardes.

2. La barrière de synchronisation *Sense-Reversing*

Un exemple très simple d'utilisation d'une barrière de synchronisation en langage C est présenté dans la figure 1. La fonction principale de ce programme commence par initialiser une barrière de synchronisation avec un nombre `N` de participants, puis démarre `N` *threads* qui exécutent la fonction `runner`. Chaque *thread* entre dans une boucle infinie qui commence par un appel à `barrier_wait` pour attendre tous les autres *threads*. Après la synchronisation, le *thread* affiche son identifiant `id` puis attend à nouveau tous les autres. Quand cette deuxième synchronisation a eu lieu, seul le *thread* dont l'identifiant est 0 affiche un retour chariot.

Si la barrière de synchronisation fonctionne correctement, c'est-à-dire si aucun *thread* ne peut quitter la barrière avant l'arrivée du dernier, le comportement attendu du programme est d'afficher (à l'infini) des lignes contenant une et une seule fois chaque chiffre entre 0 et `N`-1, dans un ordre indéterminé, comme par exemple dans la figure 2. Si au contraire, la barrière est incorrecte, le programme pourra afficher des lignes quelconques, comme dans la figure 3. Une autre manière de

```c
#include <stdio.h>
#include <pthread.h>

#define N 10

barrier_t b;

void * runner(void *id) {
  while (1) {
    /// SAFETY MARK 1
    barrier_wait(&b);
    printf(" %i ", *(int *)id);
    fflush(stdout);
    /// SAFETY MARK 2
    barrier_wait(&b);
    if (*(int *) id == 0) printf("\n");
  }
  return 0;
}
```

```c
int main()
{
  int k;
  pthread_t th[N];
  int tid[N];

  barrier_init (&b, N);
  for (k = 0; k < N; k++) {
    tid[k] = k;
    pthread_create(&(th[k]), NULL, runner, &tid[k]);
  }

  for (k = 0; k < N; k++) {
    pthread_join(th[k], NULL);
  }

  return 0;
}
```

FIGURE 1 – Exemple d'utilisation d'une barrière de synchronisation

décrire la propriété de sûreté d'une barrière est qu'il ne peut y avoir deux *threads* tels que l'un soit au point de programme indiqué par le commentaire `/// SAFETY MARK 1`, et l'autre au point indiqué par le second commentaire `/// SAFETY MARK 2`.

```
9 0 6 7 3 4 8 2 1 5
3 5 8 2 7 6 0 4 9 1
0 7 8 6 1 4 2 3 5 9
8 2 1 6 5 0 3 7 4 9
6 3 2 8 5 7 4 0 9 1
3 8 4 7 5 1 9 0 2 6
4 1 2 3 7 9 0 6 8 5
4 2 1 0 3 7 5 6 8 9
2 4 5 7 0 6 9 8 3 1
9 7 1 3 0 4 5 6 2 8
...
```

FIGURE 2 – Trace correcte

```
3 1 6 4 7 5 9 8 0
2 2 9 4 1 8 7 3 6 5 0
0
7 6 2 1 8 9 4 5 3 3 0
2 8 5 3 1 4 6 9 0
2 9 1 6 7 0 2 4
8 3 5 5 9 8 0 2 4
6 1 3 7 7 9 8 3 0
5 2 6 4 1 1 0
8 4 3 5 6 2 9 7 7 9 0
...
```

FIGURE 3 – Trace incorrecte

Intéressons nous maintenant à l'implémentation de la barrière, à savoir le type `barrier_t`, et les fonctions `barrier_init` et `barrier_wait`. Parmi les nombreuses structures existantes, nous proposons d'étudier dans cet article une barrière *sense reversing* [14]. L'implémentation présentée en figure 4 est une version légèrement modifiée de la barrière proposée par Herlihy et Shavit dans « The art of Multiprocessor Programming » [15, Sec 17.3]. La modification porte sur la suppression d'une variable « *thread specific* » qui simplifie l'utilisation des barrières. Elle est suffisamment conséquente pour que la présomption de correction qui s'applique à un algorithme provenant d'un texte de référence ne se transmette pas automatiquement à sa version modifiée.

Le type `barrier_t` est un enregistrement composé de trois champs `n`, `count` et `sense`. Ces trois champs sont initialisés par la fonction `barrier_init`. Le champ `n` contient le nombre de *threads*

```
#define DECR(x) __sync_add_and_fetch(x, -1)
#define FENCE __sync_synchronize()

typedef struct barrier_t {
  unsigned int n;
  volatile unsigned int count;
  volatile int sense;
} barrier_t;

void barrier_init(barrier_t *b, unsigned int n) {
  b->n = n;
  b->count = n;
  b->sense = 0;
}
```

```
void barrier_wait(barrier_t *b) {
  int sense, rem;
  FENCE; sense = b->sense;
  FENCE; rem = DECR(&b->count);
  if (rem == 0) {
      FENCE; b->count = b->n;
      FENCE; b->sense = !sense;
  } else {
      FENCE; while (b->sense == sense);
  }
  FENCE;
}
```

FIGURE 4 – Implémentation d'une barrière *sense reversing*

impliqués dans la barrière. L'entier `count` indique le nombre de *threads* à attendre. Enfin, le booléen `sense` (de type `int` car le type `bool` n'existe pas en C) indique le *sens* de la barrière. Ce booléen est utilisé non seulement pour la synchronisation des *threads*, mais également pour une réutilisation efficace de la barrière pour des synchronisations successives. Le compteur `count` est décrémenté à chaque appel de la fonction `barrier_wait` à l'aide d'une instruction atomique (notée `DECR`). Il est réinitialisé, et le sens de la barrière est inversé, si le *thread* qui a appelé cette fonction est le dernier entré dans la barrière. Dans le cas contraire, le *thread* est mis en pause dans une boucle qui attend que le sens de la barrière s'inverse.

Le code proposé suppose que la machine sur laquelle il s'exécutera suit le modèle "*Sequential Consistency*" (SC) [19]. Dans ce modèle l'exécution parallèle résulte de l'enchevêtrement des instructions des *threads* exécutées dans l'ordre du programme et les écritures dans la mémoire prennent effet immédiatement. Or les machines modernes ne suivent pas ce modèle, mais des modèles plus relâchés, qui autorisent entre autres l'exécution des instructions dans le désordre, les tampons en écriture etc. — Voir [2] pour une introduction récente au sujet. En pratique, on peut forcer la machine cible à se comporter selon le modèle SC en insérant des instructions spécifiques, dites barrières mémoire [1], entre chaque paire d'accès à des cases distinctes de la mémoire partagée. Le code de `barrier_wait` de la figure 4 montre un placement sûr de ces barrières mémoire (notées `FENCE`). L'optimisation du placement des barrières mémoire sort du cadre de cet article — Voir par exemple [4, 21].

3. Langages source et cible

Notre approche consiste à traduire les programmes C vers des systèmes de transition paramétrés à tableaux [12]. Ces systèmes sont parfaitement adaptés à la modélisation de programmes concurrents avec un nombre quelconque de *threads*, même si leur expressivité est limitée pour permettre une vérification efficace de leurs propriétés de sûreté par *model checking*.

Systèmes de transition à tableaux

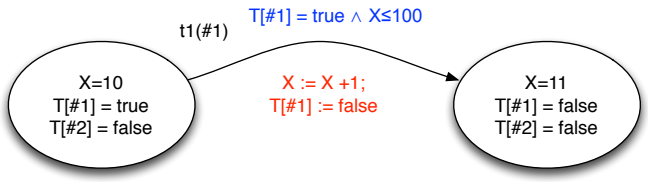
Notre langage cible est celui du *model checker* Cubicle [10]. L'état d'un système de transition est décrit par un ensemble de variables globales et de tableaux infinis indicés par des identificateurs de *thread*. Les types à notre disposition sont les entiers (`int`), les booléens (`bool`), les énumérations

1. à ne pas confondre avec les barrières de synchronisation

et le type des *threads* (`proc`). Pour décrire les transitions, on utilise dans cet article les notations de [26]. Chaque transition est représentée par une formule logique qui relie les valeurs des variables d'état avant et après la transition. Ainsi, on écrit $\mathtt{X}'$ la valeur de la variable `X` après exécution de la transition. Par exemple, si l'état d'un système est représenté par une variable entière `X` et un tableau `T`, la formule

$$t_1 : \quad \exists i.\ \mathtt{T}[i] = \mathsf{true} \wedge \mathtt{X} \leq 100 \wedge \mathtt{X}' = \mathtt{X} + 1 \wedge \mathtt{T}'[i] = \mathsf{false}$$

décrit une transition paramétrée (par i) applicable s'il existe un *thread* i tel que $\mathtt{T}[i]$ est vrai et que la valeur de `X` est inférieure à 100. Dans ce cas, l'action de cette transition est d'incrémenter la valeur de `X` et de changer la valeur de la case i de `T` à false. Graphiquement, l'application de cette transition à un système avec deux *threads* est représentée par le schéma suivant :



L'état initial d'un tel système paramétré est défini par une formule logique qui décrit les valeurs des variables globales et des tableaux pour tous les indices, *i.e.* pour tous les *threads*. Par exemple, on décrit l'état initial d'un système où `X` vaut 0 et `T` contient false pour tous les *threads* de la manière suivante :

$$\forall i.\ \neg \mathtt{T}[i] \wedge \mathtt{X} = 0$$

Les propriétés de sûreté de ces systèmes de transition sont exprimées sous forme négatives, comme des formules caractérisant les états dangereux. Par exemple, la formule suivante

$$\exists i\, j.\ i \neq j \wedge \mathtt{X} \geq 1 \wedge \mathtt{T}[i] = \mathsf{false} \wedge \mathtt{T}[j] = \mathsf{false}$$

exprime que les mauvais états du système sont ceux où il existe deux *threads* i et j distincts, tels que $\mathtt{T}[i]$ et $\mathtt{T}[j]$ sont faux et `X` est plus grand que 1.

Fragment du langage C supporté

L'expressivité de notre langage cible étant limitée, notre outil accepte un fragment volontairement restreint du langage C, mais suffisant pour écrire l'implémentation de la barrière de synchronisation donnée en figure 4 et son utilisation en figure 1.

Ainsi, les programmes acceptés ne peuvent utiliser que les types `int` et `void` (les qualificateurs `volatile` et `unsigned` sont acceptés mais ignorés). Les déclarations de structures non récursives sont également possibles. L'utilisation de pointeurs est restreinte au passage d'arguments par référence, et au retour de type `void *`. Les opérateurs et relations sur ces types sont les suivants :

- les opérations `+`, `-`, `*` et `!` ;
- les relations (`==`, `<`, `<=`, `>` et `>=`) ;
- l'accès aux champs d'une structure (`.` et `->`) ;
- les opérateurs d'adresse et de déréférencement (`&` et `*`).

Le jeu d'instructions se résume aux affectations (`=`, `++`, `--`), aux conditionnelles (avec et sans `else`), aux boucles `while` et `for`, ainsi qu'aux instructions `return` et `assert`. Les fonctions d'entrée-sortie `printf` et `fflush` sont acceptées mais ignorées dans notre schéma de compilation.

4. Schéma de compilation

Les programmes C que l'on considère sont composés d'un *thread* principal et (éventuellement) de *threads* enfants démarrés par celui-ci.

Modèle mémoire

Notre modèle mémoire pour ces programmes est très simple. L'état d'un programme est uniquement défini par un ensemble de variables globales (partagées entre tous les *threads*) et le pointeur d'instruction et la pile de chaque *thread*. Pour simplifier le schéma de traduction, les variables locales d'un *thread* sont directement stockées dans sa pile. Pour des raisons d'efficacité, on isole les valeurs de type `int` utilisées de fait comme des booléens (cf. section 5), en séparant la pile d'un *thread* en une pile d'entiers et une pile de booléens.

En ce qui concerne les déclarations globales, la traduction des variables de type `int` est immédiate : pour chaque variable `x`, on utilise une variable `GLOBAL_x`. Les structures sont quant à elles traduites *à plat*, à l'aide d'une variable par champ (ou champ de champ dans le cas de structures imbriquées). Par exemple, la variable globale `x` définie de la manière suivante

```
struct t1 { int a; int b;} ;
struct t2 { int c; struct t1 s;} ;

struct t2 x;
```

est transformée en trois variables globales `STRUCT_x_c`, `STRUCT_x_s_a` et `STRUCT_x_s_b` de type `int`. Une limitation de ce modèle *à plat* est qu'il nous conduit, pour des raisons de simplicité, à interdire l'affectation de structures (il faut passer par l'affectation manuelle de chaque champ). Il nous pousse également à restreindre le passage de structures par adresse, plutôt que par valeur, et à interdire de renvoyer des structures comme résultat d'appel de fonction.

Le pointeur d'instruction du *thread* principal est matérialisé par une variable globale `PC_M`, et celui du *thread* enfant i par l'élément d'indice i d'un tableau `PC` (*i.e.*, `PC[`i`]`). Le type de ces pointeurs est une énumération d'étiquettes de la forme

$$\texttt{type t} \;=\; \mathsf{Idle} \mid \mathsf{End} \mid \mathsf{L}_1 \mid \mathsf{L}_2 \mid \ldots$$

où les constructeurs (étiquettes) L_k représentent les points de programme des *threads*, auxquels on ajoute les deux constructeurs Idle et End, pour indiquer respectivement qu'un *thread* n'est pas démarré ou qu'il est terminé.

Notre langage cible ne permettant pas de manipuler directement des piles, on représente une pile de taille n par n variables. Par exemple, si le *thread* principal a une pile d'entiers de hauteur 2, et une pile de booléens de hauteur 3, alors les cases des piles sont représentées par les variables `STACK_M_INT_0`, `STACK_M_INT_1` et `STACK_M_BOOL_0`, `STACK_M_BOOL_1`, `STACK_M_BOOL_2`.

Les piles d'un *thread* enfant i sont représentées par des variables locales, qui sont encodées par les cases de tableaux indicés par i. Par exemple, si i a une pile d'entiers de hauteur 2 et une pile de booléens de hauteur 1, alors ses piles sont interprétées par les variables `STACK_INT_0[i]`, `STACK_INT_1[i]` et `STACK_BOOL_0[i]`.

Compilation des expressions

Dans chaque *thread*, la compilation des expressions se fait de manière récursive par la compilation des sous-expressions à l'aide des piles `STACK_INT` et `STACK_BOOL`, dont la taille maximale peut être calculée statiquement.

La compilation d'une expression produit une séquence d'*affectations atomiques* de la forme `X` $\leftarrow$ `E`, où `E` est soit une constante, une variable globale, une case de pile, ou une relation (ou opération) élémentaire entre ces dernières. Ces affectations constituent les opérations *atomiques* de notre langage source.

À titre d'exemple, pour traduire l'expression `c == (a > 1)` dans le *thread* principal, il faut générer du code qui stocke dans les piles l'évaluation des sous-expressions `c`, `1`, `a`, `a > 1` et `c == (a > 1)`. En réutilisant l'espace quand cela est possible, il suffit d'une pile d'entiers et d'une pile de booléens de taille 2 pour compiler cette expression. En prenant les variables `STACK_M_INT_0` et `STACK_M_INT_1` pour coder la pile d'entiers, et les variables `STACK_M_BOOL_0` et `STACK_M_BOOL_1` pour la pile de booléens, la compilation de l'expression précédente correspond alors aux affectations suivantes :

`c`	STACK_M_BOOL_0 $\leftarrow$ GLOBAL_c
`1`	STACK_M_INT_0 $\leftarrow$ 1
`a`	STACK_M_INT_1 $\leftarrow$ GLOBAL_a
`a>1`	STACK_M_BOOL_1 $\leftarrow$ STACK_M_INT_1 $>$ STACK_M_INT_0
`c == (a>1)`	STACK_M_BOOL_0 $\leftarrow$ STACK_M_BOOL_0 $=$ STACK_M_BOOL_1

Notons l'importance de calculer au plus juste la taille des piles pour la compilation des expressions. En effet, moins les piles sont hautes et moins le nombre de variables pour les représenter est élevé, ce qui aide considérablement la phase de *model checking* par Cubicle.

Compte tenu de la simplicité de notre modèle mémoire, les appels de fonctions sont simplement *inlinés*. Le passage de paramètres par valeur est alors simulé par l'introduction de variables intermédiaires contenant les copies des arguments. Ces variables sont allouées sur les piles `STACK_M_BOOL` et `STACK_M_INT`. Par exemple, l'appel de la fonction `fct` dans le programme

```
int a, b;

int fct(int y, int *z) { return y + *z; }

int main() { return fct(a,&b); }
```

est *inliné* de la manière suivante

```
int a, b;

int fct(int y, int *z) { return y + *z; }

int main() {
  int x;
  x = a;
  return x + b;
}
```

La traduction de l'appel de fonction correspond alors aux affectations de piles suivantes, où `STACK_M_INT_1` est utilisée comme variable intermédiaire (`x`) :

`a`	STACK_M_INT_0 $\leftarrow$ GLOBAL_a
`x = a`	STACK_M_INT_1 $\leftarrow$ STACK_M_INT_0
`b`	STACK_M_INT_2 $\leftarrow$ GLOBAL_b
`x + b`	STACK_M_INT_1 $\leftarrow$ STACK_M_INT_1 $+$ STACK_M_INT_2
`return x + b`	STACK_M_INT_0 $\leftarrow$ STACK_M_INT_1

Dans la suite, on note $\mathcal{E}(i, \texttt{exp})$ la séquence d'affectations *atomiques* qui résulte de la traduction d'une expression `exp` pour un *thread* quelconque i (principal ou enfant). Notons que la dernière affectation atomique de la séquence $\mathcal{E}(i, \texttt{exp})$ met le résultat de l'évaluation de *exp* dans le niveau 0 de la pile.

Compilation des instructions

Pour compiler les instructions du langage source, on définit une fonction $\mathcal{C}$ qui prend en arguments un identificateur de *thread*, une instruction, et deux étiquettes, et renvoie un ensemble de transitions. Ainsi, $\mathcal{C}(i, \texttt{instr}, \mathsf{L}_0, \mathsf{L}_1)$ génère le code cible pour le *thread* i qui correspond à l'exécution de `instr` depuis le point de programme L_0 jusqu'au point L_1. Ce code cible correspond à un ensemble de transitions, c'est-à-dire à un ensemble de formules comme décrites en section 3.

Affectations. La compilation d'une affectation atomique $\texttt{X} \leftarrow \texttt{E}$ est donnée par la transition

$$\mathcal{C}(i, \texttt{X} \leftarrow \texttt{E}, \mathsf{L}_0, \mathsf{L}_1) \quad = \quad \{\exists i.\ \texttt{PC}[i] = \mathsf{L}_0 \wedge \texttt{X}' = \texttt{E} \wedge \texttt{PC}'[i] = \mathsf{L}_1\}$$

qui s'applique lorsque le point de programme $\texttt{PC}[i]$ du *thread* i est en L_0. Elle met à jour la variable `X` puis change le point de programme en L_1. Une affectation quelconque `x = exp` du langage C, où `exp` est supposée être de type `int`, se traduit alors par

$$\mathcal{C}(i, \texttt{x} = \texttt{exp}, \mathsf{L}_0, \mathsf{L}_1) \quad = \quad \mathcal{C}(i, \mathcal{E}(i, \texttt{exp}); \texttt{x} \leftarrow \texttt{STACK\_INT\_0}[i], \mathsf{L}_0, \mathsf{L}_1)$$

Séquence d'instructions. La compilation d'une séquence d'instructions `instr1; instr2` est donnée par la définition

$$\begin{aligned}\mathcal{C}(i, \texttt{instr1}; \texttt{instr2}, \mathsf{L}_0, \mathsf{L}_1) \quad = \quad & \texttt{let } \mathsf{L} = \texttt{fresh\_label}() \texttt{ in} \\ & \mathcal{C}(i, \texttt{instr1}, \mathsf{L}_0, \mathsf{L}) \cup \mathcal{C}(i, \texttt{instr2}, \mathsf{L}, \mathsf{L}_1)\end{aligned}$$

qui compile récursivement l'instruction `instr1` entre le point de programme L_0 et un point de programme intermédiaire L, puis récursivement l'instruction `instr2` entre les points L et L_1.

Conditionnelles. La compilation des instructions conditionnelles nécessite quant à elle de créer trois nouvelles étiquettes L, $\mathsf{L}_{\texttt{then}}$ et $\mathsf{L}_{\texttt{else}}$. La première étiquette est utilisée pour la compilation de la condition. Les deux dernières permettent d'aiguiller les transitions vers les branches `then` ou `else`.

$$\begin{aligned}&\mathcal{C}(i, \texttt{if (exp) \{ instr1 \} else \{ instr2 \}}, \mathsf{L}_0, \mathsf{L}_1) = \\ &\qquad \texttt{let } \mathsf{L} = \texttt{fresh\_label}() \texttt{ in} \\ &\qquad \texttt{let } \mathsf{L}_{\texttt{then}} = \texttt{fresh\_label}() \texttt{ in} \\ &\qquad \texttt{let } \mathsf{L}_{\texttt{else}} = \texttt{fresh\_label}() \texttt{ in} \\ &\qquad \mathcal{C}(i, \mathcal{E}(i, \texttt{exp}), \mathsf{L}_0, \mathsf{L}) \cup \\ &\qquad \{\exists i.\ \texttt{PC}[i] = \mathsf{L} \wedge \texttt{STACK\_BOOL\_0}[i] = \texttt{true} \wedge \texttt{PC}'[i] = \mathsf{L}_{\texttt{then}}\ , \\ &\qquad\ \exists i.\ \texttt{PC}[i] = \mathsf{L} \wedge \texttt{STACK\_BOOL\_0}[i] = \texttt{false} \wedge \texttt{PC}'[i] = \mathsf{L}_{\texttt{else}}\} \cup \\ &\qquad \mathcal{C}(i, \texttt{instr1}, \mathsf{L}_{\texttt{then}}, \mathsf{L}_1) \ \cup \ \mathcal{C}(i, \texttt{instr2}, \mathsf{L}_{\texttt{else}}, \mathsf{L}_1)\end{aligned}$$

Boucles. La compilation des boucles `while` est similaire aux conditionnelles. Elle ne nécessite que deux étiquettes intermédiaires.

$$
\begin{aligned}
&\mathcal{C}(i, \texttt{while}\,(\texttt{exp})\,\{\,\texttt{instr}\,\}, \mathsf{L}_0, \mathsf{L}_1) = \\
&\qquad \texttt{let}\ \mathsf{L} = \texttt{fresh\_label}()\ \texttt{in} \\
&\qquad \texttt{let}\ \mathsf{L}_{\texttt{while}} = \texttt{fresh\_label}()\ \texttt{in} \\
&\qquad \mathcal{C}(i, \mathcal{E}(i, \texttt{exp}), \mathsf{L}_0, \mathsf{L}) \cup \\
&\qquad \{\exists i.\ \texttt{PC}[i] = \mathsf{L} \wedge \texttt{STACK\_BOOL\_0}[i] = \texttt{true} \wedge \texttt{PC}'[i] = \mathsf{L}_{\texttt{while}}\ , \\
&\qquad\ \exists i.\ \texttt{PC}[i] = \mathsf{L} \wedge \texttt{STACK\_BOOL\_0}[i] = \texttt{false} \wedge \texttt{PC}'[i] = \mathsf{L}_1\} \cup \\
&\qquad \mathcal{C}(i, \texttt{instr1}, \mathsf{L}_{\texttt{while}}, \mathsf{L}_0)
\end{aligned}
$$

Compteurs de *threads*

Le programme d'utilisation de la barrière de synchronisation donné en figure 1 fonctionne pour un nombre de *threads* fixé à 10 par la directive :

```
# define N 10
```

Cependant, pour vérifier la propriété de bonne synchronisation de la barrière pour un nombre quelconque de *threads*, nous ne devons pas tenir compte de cette constante, mais seulement considérer `N` comme un paramètre du système.

Malheureusement, il n'est pas possible de manipuler directement ce paramètre dans notre langage cible. Seul le type `proc`, c'est-à-dire l'ensemble des indices des *threads*, permet de le représenter, `N` étant la cardinalité de cet ensemble. Puisque le domaine des tableaux est le type `proc`, on peut simplement représenter la valeur `N` comme un tableau de booléens où toutes les cases contiennent `true`. Ce tableau joue alors le rôle d'un *encodage unaire* de `N`. De la même manière, chaque variable entière `X` manipulant `N` est codée à l'aide d'un tableau de booléens. Le nombre de cases du tableau `X` ayant pour valeur `true` représente le nombre de 1 composant l'encodage unaire de `X`. On autorisera donc seulement certaines opérations sur les variables qui manipulent `N` comme un entier en C :

— l'affectation à `N` où à 0 (`Global_x = N`, `Global_x = 0`)
— le test d'égalité avec `N` et 0 (`Global_x == N`, `Global_x == 0`)
— l'incrémentation et la décrémentation atomique (*e.g.* avec la macro `DECR(&Global_x)`)

Ces variables servent donc essentiellement de compteurs de *threads*.

La compilation d'une affectation à 0 se fait en mettant toutes les cases à `false` du tableau correspondant et de manière similaire, une affectation à `N` se fait en mettant toutes les cases à `true`, de la manière suivante :

$$
\begin{aligned}
\mathcal{C}(i, \texttt{Global\_x} \leftarrow \texttt{N}, \mathsf{L}_0, \mathsf{L}_1) &= \{\exists i.\ \texttt{PC}[i] = \mathsf{L}_0 \wedge \forall j.\ \texttt{Global\_x}'[j] = \mathsf{true} \wedge \texttt{PC}'[i] = \mathsf{L}_1\} \\
\mathcal{C}(i, \texttt{Global\_x} \leftarrow 0, \mathsf{L}_0, \mathsf{L}_1) &= \{\exists i.\ \texttt{PC}[i] = \mathsf{L}_0 \wedge \forall j.\ \texttt{Global\_x}'[j] = \mathsf{false} \wedge \texttt{PC}'[i] = \mathsf{L}_1\}
\end{aligned}
$$

La compilation d'un test d'égalité à 0 ou à `N` sur la pile booléenne est décrite par :

$$
\begin{aligned}
&\mathcal{C}(i, \texttt{STACK\_BOOL\_}k[i] \leftarrow \texttt{Global\_x} == \texttt{N}, \mathsf{L}_0, \mathsf{L}_1) \quad = \\
&\qquad \{\exists i.\ \texttt{PC}[i] = \mathsf{L}_0 \wedge \forall j.\ \texttt{Global\_x}[j] = \mathsf{true} \wedge \texttt{STACK\_BOOL\_}k'[i] = \mathsf{true} \wedge \texttt{PC}'[i] = \mathsf{L}_1, \\
&\qquad\ \exists i.\ \texttt{PC}[i] = \mathsf{L}_0 \wedge \exists j.\ \texttt{Global\_x}[j] = \mathsf{false} \wedge \texttt{STACK\_BOOL\_}k'[i] = \mathsf{false} \wedge \texttt{PC}'[i] = \mathsf{L}_1\} \\
&\mathcal{C}(i, \texttt{STACK\_BOOL\_}k[i] \leftarrow \texttt{Global\_x} == 0, \mathsf{L}_0, \mathsf{L}_1) \quad = \\
&\qquad \{\exists i.\ \texttt{PC}[i] = \mathsf{L}_0 \wedge \forall j.\ \texttt{Global\_x}[j] = \mathsf{false} \wedge \texttt{STACK\_BOOL\_}k'[i] = \mathsf{true} \wedge \texttt{PC}'[i] = \mathsf{L}_1, \\
&\qquad\ \exists i.\ \texttt{PC}[i] = \mathsf{L}_0 \wedge \exists j.\ \texttt{Global\_x}[j] = \mathsf{true} \wedge \texttt{STACK\_BOOL\_}k'[i] = \mathsf{false} \wedge \texttt{PC}'[i] = \mathsf{L}_1\}
\end{aligned}
$$

Enfin, la traduction de la décrémentation atomique (implémentée à l'aide des instructions machine idoines, par le truchement d'un *atomic builtin* de GCC) consiste simplement à changer la valeur d'une case contenant `true` à `false` (*i.e* enlever un 1 de l'encodage unaire) :

$$\mathcal{C}(i, \texttt{DECR(\&Global\_x)}, \mathsf{L}_0, \mathsf{L}_1) = \\ \exists i.\ \mathtt{PC}[i] = \mathsf{L}_0 \wedge \exists j.\ \mathtt{Global\_x}[j] = \mathsf{true} \wedge \mathtt{Global\_x'}[j] = \mathsf{false} \wedge \mathtt{PC'}[i] = \mathsf{L}_1$$

Propriété de sûreté

La traduction de la propriété de sûreté consiste simplement à récolter dans un ensemble S tous les points de programmes correspondant à des annotations de la forme `/// SAFETY MARK` i. On génère alors la formule suivante qui caractérise les états dangereux correspondants à ces annotations :

$$\bigvee_{\substack{(m_1, m_2) \in S \times S \\ m_1 \neq m_2}} \exists i, j.\, \mathtt{PC}[i] = m_1 \wedge \mathtt{PC}[j] = m_2$$

5. Typage

Il est important de pouvoir dire quand une variable entière (de type `int`) est en réalité utilisée comme un booléen pour que la phase de *model checking* effectuée par Cubicle soit la plus efficace possible. En effet le traitement de formules avec variables booléennes ne demande qu'un raisonnement propositionnel et ne fait donc appel qu'à la partie SAT du solveur SMT de Cubicle. En revanche l'utilisation d'entiers et plus particulièrement d'opérations arithmétiques demande un raisonnement spécifique de la théorie de l'arithmétique dont les appels sont coûteux.

Dans le fragment du C utilisé ici, on peut (parfois) décider statiquement si une variable est booléenne. Dans cette section on propose une analyse de typage qui, bien que limitée, donne suffisamment d'informations pour permettre la vérification de nos barrières.

Pour simplifier cette analyse, on force la distinction entre les types `bool` et `int`. Ainsi, les opérateurs arithmétiques ou de comparaison (sauf `==` et `!=`) ne pourront être utilisés que sur des valeurs de type `int`, de même, les constantes différentes de 0 et 1 seront de type `int`. Les opérateurs logiques (`&&`, `!`, etc.) seront eux exclusivement réservés aux valeurs de type `bool`. Enfin, seules les constantes 0 et 1 peuvent représenter à la fois un entier ou un booléen. Pour cette raison, on introduit des variables de type α afin de représenter le type `int`$\cup$`bool`. Regardons sur trois exemples la manière dont fonctionne notre analyse.

```
int x, y, z;

x = 0;
y = x;
z = y;
if (z) { x = x + 1; }
```

```
int x, y;

y = 0;
if (y == 0)
  { x = 0; }
```

```
int x, y, z;

x = 0 && 1;
if (x != y && y != z && x != z)
  { ... }
```

Dans le programme de gauche, juste avant l'instruction `if`, on donne le même type α aux trois variables `x`, `y` et `z`. L'analyse de la conditionnelle force alors `z` à être un booléen, tandis que `x` est utilisé comme un entier. Le programme est donc rejeté. Dans le programme du milieu, la conditionnelle force `y` a être de type `bool`, tandis que `x` reste de type α. Comme `x` peut n'avoir jamais été initialisée, et donc contenir une valeur arbitraire, on choisit de lui donner le type `int`. Enfin, dans le programme de droite, après avoir typé l'expression booléenne de la conditionnelle, on dit que `x`, `y` et `z` sont de type `bool`, mais seul `x` est initialisé. Si on donnait réellement le type `bool` à ces trois variables, la condition serait nécessairement fausse. Comme pour le programme précédent, `y` et `z` n'étant pas initialisées, elles

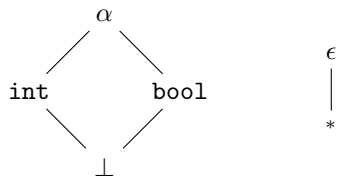
peuvent contenir des valeurs arbitraires et rendre la condition vraie. On donne donc le type `int` à `y` et `z` et ce programme est rejeté.

Pour simplifier notre présentation, on limite la grammaire des expressions et des instructions de notre langage source à :

$$e := x \mid \texttt{c} \mid e{=}{=}e \mid e{+}e \mid e{<}e \mid e \,\texttt{\&\&}\, e$$

$$i := x{=}e \mid \texttt{if}(e, i, i) \mid \texttt{while}(e, i)$$

Pour faire notre analyse, on va utiliser une algèbre de type définie par $\tau := \texttt{int} \mid \texttt{bool} \mid \alpha$, ainsi que des marques d'*initialisation* $\mu := \epsilon \mid {}^*$. Les treillis $(\tau, \sqcup, \sqcap, \sqsubseteq)$ et $(\mu, \bigvee, \bigwedge, <)$ associés à ces ensembles sont représentés ci-dessous.



Notre algorithme de typage est défini par un ensemble de règles d'inférence dirigées par la syntaxe des expressions et des instructions. Les règles pour les expressions sont données à l'aide de séquents de la forme $\Gamma \vdash e : \tau, \mu$ signifiant « dans l'environnement Γ, l'expression e est bien typée de type τ avec une indication d'initialisation μ ». C'est seulement lorsque μ vaut * que l'on est sûr que e produit nécessairement une valeur de type τ.

Pour typer les expressions, on utilise une structure union-find sur les types dotée de deux fonctions `union` (qui réunit deux classes) et `find` (qui renvoie le représentant de la classe d'un élément). Cette structure est *globale* et utilisée par effet de bord. Elle est initialisée avec autant de classes singleton $\{\alpha_x\}$ qu'il y a de variables de terme x. De plus, elle est compatible avec le treillis des types de manière à ce que l'opération $\texttt{union}(\tau_1, \tau_2)$ ne soit applicable que si $\tau_1 \sqcap \tau_2 \neq \bot$. Par ailleurs, cette structure doit s'assurer que le représentant d'une classe est la plus petite valeur dans le treillis. On utilise également un dictionnaire Γ qui associe chaque variable x d'un terme e à une marque μ, initialisée à ϵ. Les règles pour typer les expressions sont données ci-dessous :

$$\frac{\texttt{c} \in \{\texttt{0}, \texttt{1}\} \qquad \alpha \text{ est une variable fraîche}}{\Gamma \vdash \texttt{c} : \alpha, {}^*} \qquad \frac{\texttt{c} \notin \{\texttt{0}, \texttt{1}\}}{\Gamma \vdash \texttt{c} : \texttt{int}, {}^*} \qquad \frac{}{\Gamma \vdash x : \texttt{find}(\alpha_x), \Gamma(x)}$$

$$\frac{e_1 : \tau_1, \mu_1 \qquad e_2 : \tau_2, \mu_2 \qquad \texttt{union}(\tau_1, \tau_2)}{\Gamma \vdash e_1{=}{=}e_2 : \texttt{bool}, {}^*}$$

$$\frac{e_1 : \tau_1, \mu_1 \qquad e_2 : \tau_2, \mu_2 \qquad \texttt{union}(\tau_1, \texttt{int}) \qquad \texttt{union}(\tau_2, \texttt{int})}{\Gamma \vdash e_1{+}e_2 : \texttt{int}, {}^*}$$

$$\frac{e_1 : \tau_1, \mu_1 \qquad e_2 : \tau_2, \mu_2 \qquad \texttt{union}(\tau_1, \texttt{int}) \qquad \texttt{union}(\tau_2, \texttt{int})}{\Gamma \vdash e_1{<}e_2 : \texttt{bool}, {}^*}$$

$$\frac{e_1 : \tau_1, \mu_1 \qquad e_2 : \tau_2, \mu_2 \qquad \texttt{union}(\tau_1, \texttt{bool}) \qquad \texttt{union}(\tau_2, \texttt{bool})}{\Gamma \vdash e_1 \,\texttt{\&\&}\, e_2 : \texttt{bool}, {}^*}$$

Les règles pour typer les instructions sont données plus bas à l'aide de séquents de la forme $\Gamma \vdash_i i : \Gamma'$, signifiant « dans l'environnement Γ, l'instruction i est bien typée et produit les nouvelles marques d'initialisation Γ' ». Pour gérer les marques d'initialisation, on définit la fonction `merge` sur

les dictionnaires telle que $\texttt{merge}(\Gamma_1, \Gamma_2)$ renvoie le dictionnaire qui, pour chaque couple clé-valeur $\alpha \mapsto \mu_1$ de Γ_1 et $\alpha \mapsto \mu_2$ de Γ_2, associe α à la valeur $\mu_1 \bigvee \mu_2$.

$$\frac{\Gamma \vdash e : \tau, \mu \qquad \texttt{union}(\tau, \texttt{find}(\alpha_x))}{\Gamma \vdash_i x = e : \Gamma \cup \alpha_x \mapsto \mu} \qquad \frac{\Gamma \vdash e : \tau, \mu \qquad \texttt{union}(\tau, \texttt{bool}) \qquad \Gamma \vdash_i i : \Gamma'}{\Gamma \vdash_i \texttt{while}(e, i) : \texttt{merge}(\Gamma, \Gamma')}$$

$$\frac{\Gamma \vdash e : \tau, \mu \qquad \texttt{union}(\tau, \texttt{bool}) \qquad \Gamma \vdash_i i_1 : \Gamma_1 \qquad \Gamma \vdash_i i_2 : \Gamma_2}{\Gamma \vdash_i \texttt{if}(e, i_1, i_2) : \texttt{merge}(\Gamma_1, \Gamma_2)}$$

À la fin de cette analyse, les types des variables peuvent être extraits des informations présentes dans la structure d'union-find et dans le dictionnaire Γ. Si pour une variable x, $\Gamma(x) = \epsilon$, alors la variable x n'a pas été initialisée. Son type, n'étant pas contraint, sera donc `int`. Si au contraire $\Gamma(x) = {}^*$ et $\texttt{find}(\alpha_x)$ est différent de `int`, la variable x a été initialisée et n'a pu être affectée qu'aux valeurs 0 ou 1, alors son type sera `bool`. Ces nouvelles informations de typage sont propagées dans la structure union-find afin de vérifier leur cohérence.

6. Expérimentations

On présente dans cette section les expérimentations réalisées sur différentes barrières de synchronisation. Pour toutes ces barrières on vérifie la propriété de bonne synchronisation.

On a testé notre outil sur cinq versions de barrières *sense-reversing* : `sb_alt.c` est une version avec deux fonctions `wait` alternantes, `sb.c` est la version proposée dans [15], `sb_nice.c` est la version présentée en section 2 comportant notre modification, et `sb_single.c` est une version non réutilisable (cette même barrière est utilisée – à tort – deux fois dans `sb_single_us.c`, d'où l'erreur), enfin la version `sb_loop.c` présentée plus loin est une utilisation répétée dans une boucle infinie de la barrière `sb.c`. Pour chacune, on donne les temps d'exécution de Cubicle avec et sans l'analyse de typage faite sur les booléens (présentée en section 5). On précise aussi le nombre de nœuds visités, le nombre d'invariants inférés (Inv.) ainsi que le nombre de redémarrages de la recherche (Restarts). Tous les résultats ont été obtenus sur une machine 64 bits avec un processeur Intel® Xeon® @ 3.2 GHz et 24 Go de mémoire en lançant Cubicle de manière séquentielle et avec inférence d'invariants pour deux *threads* (option `-brab 2`). Nous renvoyons le lecteur intéressé par ce mécanisme de découverte d'invariants à [9]. On note par ☠ les exemples non-sûrs dont la propriété n'est pas vérifiée et pour lesquels Cubicle expose une trace d'erreur.

	Avec typage				Sans typage			
	nœuds	Inv.	Restarts	Temps	nœuds	Inv.	Restarts	Temps
`sb_alt`	215	152	7	**7,64s**	598	180	53	11m27s
`sb.c` [15]	331	226	10	**20,7s**	414	156	34	5m21s
`sb_nice.c`	240	106	9	**11,6s**	303	139	49	28m8s
`sb_single.c`	165	115	5	**3,11s**	174	99	54	17m44s
`sb_single_us.c`☠	810	/	0	**5,06s**	811	/	0	6,13s

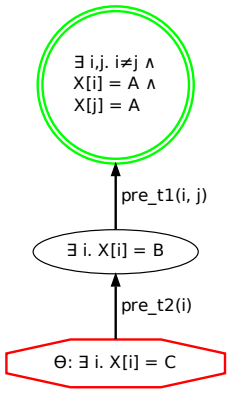
Ces résultats montrent clairement que l'analyse de typage améliore grandement l'efficacité de Cubicle. Par contre, une analyse précise des traces des expérimentations montre que le mécanisme de recherche d'invariants de Cubicle est perturbé par la présence de traces d'erreur fallacieuses. En effet, on peut se rendre compte grâce à la section 4 que la compilation des boucles et compteurs de *threads* introduit des quantificateurs universels dans les gardes. Par exemple, si `cpt` est un compteur de *threads*, la condition `cpt == 0` se traduira par $\forall i.\ \texttt{Cpt}[i] = \texttt{false}$ dans les systèmes de transition

à tableaux. Malheureusement l'utilisation de gardes dites *universelles* nous fait perdre la propriété de la décidabilité de la sûreté pour ces systèmes. En effet, les seuls moyens connus à ce jour pour traiter efficacement ces quantificateurs universels consistent tous en une forme d'*approximation*. Que ce soit dans le framework de *Regular model checking* à l'aide d'*abstractions monotones* [1] ou dans le framework *Model checking modulo theories* grâce au *Crash failure model* [3, 22], la sur-approximation introduite permet au système abstrait d'avoir des comportements qui n'existent pas dans le système réel et l'analyse pourra donc potentiellement exhiber des contre-exemples et traces d'erreur fallacieux.

Pour comprendre ce phénomène, donnons nous par exemple un tableau $\mathtt{X}$ indicé par des identificateurs de *threads* et à valeurs dans une énumération à trois constructeurs $\mathsf{A} \mid \mathsf{B} \mid \mathsf{C}$. On définit l'état initial du système par la formule $I : \forall i.\ \mathtt{X}[i] = \mathsf{A}$ et l'ensemble de ses transitions défini par :

$$\tau = \left\{ \begin{array}{lrll} t_1 : & \exists i, j. & i \neq j \wedge \mathtt{X}[i] = \mathsf{A} \wedge \mathtt{X}[j] = \mathsf{A} & \wedge\ \mathtt{X}'[i] = \mathsf{B} \\ t_2 : & \exists i. & \mathtt{X}[i] = \mathsf{B} \wedge \forall j.\ j \neq i \implies \mathtt{X}[j] \neq \mathsf{A} & \wedge\ \mathtt{X}'[i] = \mathsf{C} \end{array} \right\}$$

On veut vérifier qu'aucun état atteignable du système ne satisfait la mauvaise formule $\Theta : \exists i.\ \mathtt{X}[i] = \mathsf{C}$. Cependant, en appliquant l'algorithme classique d'atteignabilité arrière de Cubicle, on obtient successivement les nœuds représentés par l'arbre ci-dessous :



Le nœud $\exists i.\ \mathtt{X}[i] = \mathsf{B}$ est la pré-image de Θ par t_2 et indique qu'un état où il existe un *thread* i tel que $\mathtt{X}[i] = \mathsf{B}$ peut atteindre Θ en un pas. Pour calculer cet état, Cubicle a fait une approximation car il doit considérer qu'il soit possible qu'il n'y ait qu'*un seul thread* dans le programme. Ce nœud étant lui-même atteignable en un pas par l'état initial I, on obtient la trace d'erreur $I \longrightarrow t_1(i, j) \longrightarrow t_2(i) \longrightarrow \Theta$.

Il s'agit d'une trace fallacieuse car, pour exister, elle suppose que le programme contient *au moins* deux *threads*, ce qui aurait changé la première pré-image de Θ calculée par Cubicle. En effet, après avoir appliqué $t_1(i, j)$ à l'état $\mathtt{X}[i] = \mathsf{A} \wedge \mathtt{X}[j] = \mathsf{A}$, on obtient l'état $\mathtt{X}[i] = \mathsf{B} \wedge \mathtt{X}[j] = \mathsf{A}$ et la transition t_2 n'est plus applicable à cause de sa garde universelle.

Pour remédier à ce problème, on a modifié l'algorithme de Cubicle pour vérifier que les traces sont tout le temps possibles au moment où on effectue notre analyse d'atteignabilité arrière. Cette vérification met en œuvre une forme d'exploration en avant symbolique. Les résultats de nos expérimentations avec ce nouvel algorithme sont présentés dans la table ci-dessous.

	Avec raffinement				Sans raffinement			
	nœuds	Inv.	Restarts	Temps	nœuds	Inv.	Restarts	Temps
`sb_alt`	216	30	0	**1,70s**	215	152	7	7,64s
`sb.c` [15]	484	67	0	**3,26s**	331	226	10	20,7s
`sb_nice.c`	363	52	0	**2,06s**	240	106	9	11,6s
`sb_single.c`	192	27	0	**0,97s**	165	115	5	3,11s
`sb_single_us.c`	810	/	0	6,91s	810	/	0	**5,06s**
`sb_loop.c`	2146	257	0	**45,6s**	1274	1577	33	14m49s

7. Conclusion

Nous avons présenté dans cet article un outil pour vérifier la sûreté de barrières de synchronisation écrites en langage C pour un nombre quelconque de *threads*. Notre technique consiste à compiler un fragment du langage C avec *threads* vers le langage d'entrée du *model checker* Cubicle. Nous avons mené des expérimentations sur plusieurs implémentations de barrières *sense reversing*. Les résultats obtenus montrent la viabilité de l'approche.

D'autres approches existent pour la vérification de programmes C concurrents. On trouve par exemple des approches semi-manuelles comme la vérification déductive avec VCC [7] ou celles basées sur des assistants à la preuve comme Isabelle ou Coq (*e.g.* projet L4.verified [18]). En ce qui concerne les techniques automatiques, on trouve des approches par interprétation abstraite, comme DUET [11], qui s'attaquent à la vérification de propriétés simples (comme l'absence de déréférencement de pointeurs nuls) pour des programmes paramétrés. On trouve également des *model checkers* comme VeriSoft [13], CMC [23] ou CHESS [24] qui sont limités à la vérification de programmes C avec un nombre fixe (et petit) de *threads*. Parmi ces approches par *model checking*, on peut citer les travaux récents de Jiang et Jonsson [17] qui compilent un fragment du C vers le langage de Spin. Là encore, l'approche ne permet pas de prouver la sûreté pour un nombre quelconque de *threads*. Enfin on peut noter des techniques plus éloignées utilisant des systèmes de permissions pour prouver des modèles de barrières de synchronisation [20].

Ce travail constitue à nos yeux une approche prometteuse pour la vérification de programmes C concurrents. Pour poursuivre, nous envisageons d'étendre le modèle mémoire de notre compilateur pour y ajouter par exemple la notion de registre, de mémoire cache etc. Pour gagner en flexibilité, nous pensons qu'il serait intéressant de placer cette phase de vérification sur un langage intermédiaire d'un vrai compilateur C. Cela nous permettrait également de simplifier le schéma de traduction en vue de faire une preuve formelle de sa correction.

Références

[1] P. A. Abdulla, N. B. Henda, G. Delzanno, and A. Rezine. Handling parameterized systems with non-atomic global conditions. In *Verification, Model Checking, and Abstract Interpretation*, pages 22–36. Springer, 2008.

[2] S. V. Adve and H.-J. Boehm. Memory models : a case for rethinking parallel languages and hardware. *Commun. ACM*, 53(8) :90–101, Aug. 2010.

[3] F. Alberti, S. Ghilardi, E. Pagani, S. Ranise, and G. P. Rossi. Universal guards, relativization of quantifiers, and failure models in model checking modulo theories. *JSAT*, 8(1/2) :29–61, 2012.

[4] J. Alglave and L. Maranget. Stability in weak memory models. In *CAV*, pages 50–66, 2011.

[5] E. A. Ashcroft. Proving assertions about parallel programs. *Journal of Computer and System Sciences*, 10(1) :110–135, 1975.

[6] E. M. Clarke and E. A. Emerson. *Design and synthesis of synchronization skeletons using branching time temporal logic*. Springer, 1982.

[7] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC : A practical system for verifying concurrent C. In *Theorem Proving in Higher Order Logics*, pages 23–42. Springer, 2009.

[8] S. Conchon, A. Goel, S. Krstić, A. Mebsout, and F. Zaïdi. Cubicle : A Parallel SMT-based Model Checker for Parameterized Systems. In *CAV*, pages 718–724. Springer, 2012.

[9] S. Conchon, A. Goel, S. Krstic, A. Mebsout, and F. Zaidi. Invariants for Finite Instances and Beyond. In *FMCAD*, Portland, Oregon, USA, October 2013.

[10] S. Conchon, A. Mebsout, and F. Zaïdi. Vérification de systèmes paramétrés avec Cubicle. In *Vingt-quatrièmes Journées Francophones des Langages Applicatifs*, Aussois, France, Feb. 2013.

[11] A. Farzan and Z. Kincaid. Verification of parameterized concurrent programs by modular reasoning about data and control. In J. Field and M. Hicks, editors, *POPL*, pages 297–308. ACM, 2012.

[12] S. Ghilardi and S. Ranise. Backward reachability of array-based systems by SMT solving : Termination and invariant synthesis. *LMCS*, 6(4), 2010.

[13] P. Godefroid. Model checking for programming languages using VeriSoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 174–186. ACM, 1997.

[14] D. Hensgen, R. Finkel, and U. Manber. Two algorithms for barrier synchronization. *International Journal of Parallel Programming*, 17(1) :1–17, 1988.

[15] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming, Revised Reprint.* Elsevier, 2012.

[16] IEEE. *IEEE 1003.1c-1995 : Information Technology — Portable Operating System Interface (POSIX) - System Application Program Interface (API) Amendment 2 : Threads Extension (C Language).* 1995.

[17] K. Jiang and B. Jonsson. Using spin to model check concurrent algorithms, using a translation from c to promela. In *In Second Swedish Workshop on Multi-Core Computing*, 2009.

[18] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, et al. seL4 : Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220. ACM, 2009.

[19] L. Lamport. How to Make a Correct Multiprocess Program Execute Correctly on a Multiprocessor. *IEEE Trans. Comput.*, 1979.

[20] D.-K. Le, W.-N. Chin, and Y.-M. Teo. Verfication of static and dynamic barrier synchronization using bounded permissions. In *ICFEM*, 2013.

[21] J. Lee and D. Padua. Hiding relaxed memory consistency with a compiler. *IEEE Transactions on Computers*, 50 :824–833, 2001.

[22] N. A. Lynch. *Distributed algorithms.* Morgan Kaufmann, 1996.

[23] M. Musuvathi, D. Y. Park, A. Chou, D. R. Engler, and D. L. Dill. CMC : A pragmatic approach to model checking real code. *ACM SIGOPS Operating Systems Review*, 36(SI) :75–88, 2002.

[24] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, pages 267–280. USENIX Association, 2008.

[25] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs i. *Acta informatica*, 6(4) :319–340, 1976.

[26] A. Pnueli, S. Ruah, and L. Zuck. Automatic deductive verification with invisible invariants. In *TACAS*, volume 2031, pages 82–97. Springer, 2001.

[27] J.-P. Queille and J. Sifakis. Specification and verification of concurrent systems in cesar. In *International Symposium on Programming*, pages 337–351. Springer, 1982.