Rapport de TER
Réalisation d'un ordonnanceur programmable pour Cubicle

Mattias Roux

2 mai 2014

Table des matières

Table des matières **1**

1 Introduction **3**
1.1 Cubicle et l'algorithme BRAB 3

Bibliographie **7**

Chapitre 1

Introduction

La mise au point des programmes concurrents est très difficile. Ceci est principalement dû au non-déterminisme de leur exécution : si on exécute plusieurs fois le même programme, on obtient rarement le même résultat. Ceci est aggravé par le fait que le modèle threads/mémoire partagée est très difficile à programmer.

Pour réduire le nombre de bugs dans ces programmes, on a recours à des outils de vérification, appelés model checker, qui tentent de vérifier la sûreté d'un programme concurrent en explorant (statiquement) tous ses comportements possibles à l'exécution.

Cubicle est un model checker conçu pour vérifier des propriétés de sûreté d'algorithmes faisant intervenir un nombre quelconque de processus. Le fonctionnement de Cubicle repose en partie sur l'analyse de traces d'exécution d'un programme pour un nombre fini de processus.

1.1 Cubicle et l'algorithme BRAB

Avant d'entrer dans le vif du sujet, il convient de présenter plus précisément le langage utilisé par Cubicle. Attention, nous n'entrerons pas, ici, dans les détails du langage car cela a déja été fait dans [1]. Néanmoins, il convient d'en décrire la syntaxe afin de favoriser la compréhension immédiate du fonctionnement de l'ordonnanceur qui a été réalisé pour ce langage. Voici donc, brièvement, une présentation succincte d'un fichier Cubicle. Celui-ci se décompose en plusieurs parties :

- La déclaration des types et de variables typées ↓
- La déclaration de l'état initial ↓
- La déclaration d'états unsafe ↓
- La déclaration de l'ensemble des transitions ↓

Déclaration des types et variables typées

Les types `int`, `real` et `bool` sont reconnus par le compilateur de Cubicle ainsi qu'un type `proc` qui permet d'identifier les processus. Enfin, il est possible pour l'utilisateur de définir deux autres sortes de types, les types énumérés ou les types abstraits (dont nous verrons plus tard qu'ils exigent deux traitement bien distincts) :

```
type state = Invalid | Shared | Exclusive (* Enuméré *)
type abstr      (* Abstrait *)
```

FIGURE 1.1: Définition de types pour Cubicle

Après cette définition, il est possible de déclarer des variables globales ainsi que des tableaux indexés par des variables de type `proc`

```
var Timer : int
var Abs : abstr
array Arr[proc] : state
```

FIGURE 1.2: Déclarations de variables et de tableaux pour Cubicle

Ainsi, dans le cadre de German-*ish*, on doit écrire :

```
type msg = Empty | Reqs | Reqe
type state = Invalid | Shared | Exclusive

var Exgntd : bool
var Curcmd : msg
var Curptr : proc

array Cache[proc] : state
array Shrset[proc] : bool
```

FIGURE 1.3: Déclarations dans German-*ish* pour Cubicle

Déclaration de l'état initial

Un état initial admet une représentation logique et son équivalent dans Cubicle, par exemple, celui de German-*ish* en Cubicle est le suivant :

```
init (z) { Cache[z] = Invalid && Shrset[z] = False &&
           Exgntd = False && Curcmd = Empty }
```

FIGURE 1.4: Etat initial du protocole German-*ish* pour Cubicle

qui peut être lu comme ceci (on remarquera que la variable `Curptr` n'est pas initialisée, nous reviendrons dessus plus tard) :

$$\forall z.\ \mathtt{Cache}[z] = \mathsf{Invalid} \ \wedge\ \neg\mathtt{Shrset}[z] \ \wedge\ \neg\mathtt{Exgntd} \ \wedge\ \mathtt{CurCmd} = \mathsf{Empty}$$

FIGURE 1.5: Etat initial du protocole German-*ish* sous forme logique

Déclaration d'états unsafe

De la même manière que pour l'état initial, on définit un ou plusieurs états unsafe. Par exemple, l'état unsafe de German-*ish* :

```
unsafe (z1 z2) { Cache[z1] = Exclusive && Cache[z2] = Shared }
```

FIGURE 1.6: Etat unsafe du protocole German-*ish* pour Cubicle

peut être lu comme ceci :

$$\Theta :\ \exists z1, z2.\ z1 \neq z2 \ \wedge\ \mathtt{Cache}[z1] = \mathsf{Exclusive} \ \wedge\ \mathtt{Cache}[z2] = \mathsf{Shared}$$

FIGURE 1.7: Etat unsafe du protocole German-*ish* sous forme logique

Déclaration des transitions

Reprenons notre exemple :
et intéressons nous à la partie *requires* qui correspond à ce qui sera appelé *garde*. D'un point de vue logique, la garde peut être lue comme ceci :

$$\mathtt{Ptr} = i \ \wedge\ \mathtt{Cmd} = \mathsf{re} \ \wedge\ \neg\mathtt{Exg} \ \wedge\ \forall j.\ \neg\mathtt{Shr}[j]$$

$$\mathtt{Cmd}' = \epsilon \ \wedge\ \mathtt{Exg}' \ \wedge\ \mathtt{Shr}'[i] \ \wedge\ \mathtt{Cache}'[i] = \mathsf{E}$$

```
transition gnt_exclusive (n)
requires { Shrset[n] = False && Curcmd = Reqe &&
           Exgntd = False && Curptr = n &&
           forall_other l. Shrset[l] = False }
{
  Curcmd := Empty;
  Exgntd := True;
  Shrset[n] := True;
  Cache[n] := Exclusive;
}
```

FIGURE 1.8: Une transition du protocole German-*ish* pour Cubicle

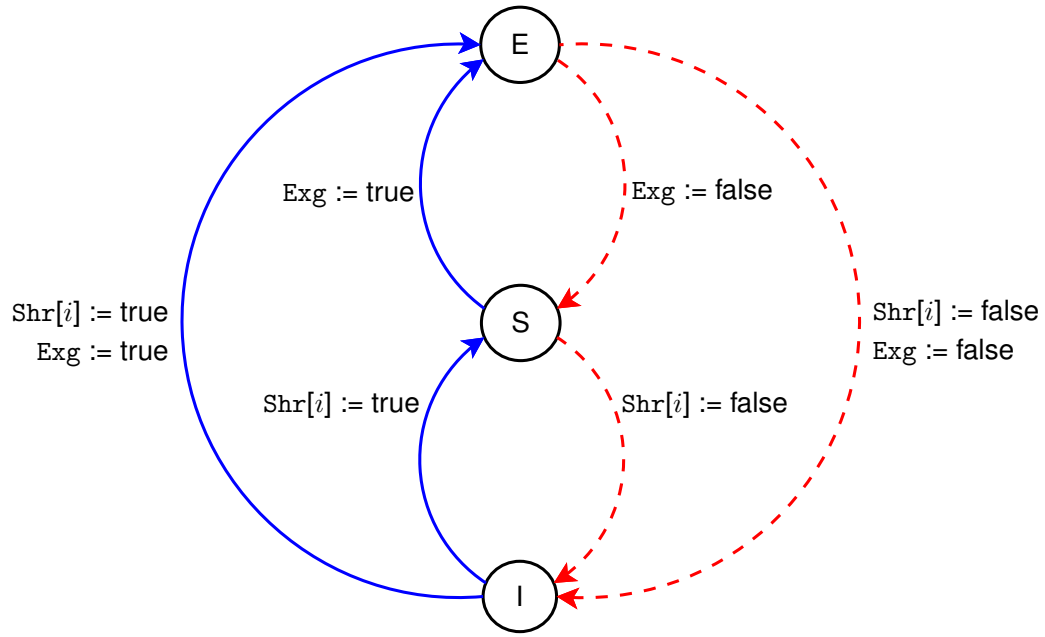

FIGURE 1.9: Diagramme d'états du protocole German-*ish*

Ici Figure 1.9

Bibliographie

[1] Sylvain Conchon, Alain Mebsout, Fatiha Zaïdi
Vérification de systèmes paramétrés avec Cubicle.
https://www.lri.fr/~conchon/TER/2013/1/cubicle.pdf