Rapport de TER
Réalisation d'un ordonnanceur programmable pour Cubicle

Mattias Roux

5 mai 2014

Table des matières

Table des matières **1**

1 Introduction **3**

2 Cubicle et l'algorithme BRAB **5**
2.1 Cubicle . . . 5
2.1.1 Déclaration des types et variables typées . . . 6
2.1.2 Déclaration de l'état initial . . . 7
2.1.3 Déclaration d'états unsafe . . . 8
2.1.4 Déclaration des transitions . . . 8
2.2 BRAB . . . 9

3 Les données et leur initialisation **11**
3.1 Les structures de données . . . 11
3.2 Initialisation ? Des classes d'équivalences et de différences . . . 15
3.3 Déterministe vs non déterministe . . . 15
3.4 Colorions . . . 15

4 Les transitions et leur exécution **17**
4.1 La garde . . . 17
4.2 Mise à jour . . . 17

Bibliographie **19**

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.*

Dans le cadre de mon TER [1], j'ai travaillé au sein de l'équipe Toccata (leur site) et tout particulièrement avec Sylvain Conchon et Alain Mebsout, à l'élaboration d'un ordonnanceur programmable pour Cubicle. Voilà le rapport de ce travail commencé en janvier et que je vais continuer lors d'un stage. Ce rapport est donc à la fois un compte-rendu de ce que j'ai déjà fait et aussi un aperçu de ce que je voudrais faire.

1. Travail d'Etude et de Recherche

Chapitre 2

Cubicle et l'algorithme BRAB [1]

2.1 Cubicle

Avant d'entrer dans le vif du sujet (ici, pour les connaisseurs de **Cubicle**), il convient de présenter le langage utilisé par **Cubicle** à travers sa syntaxe et sa sémantique afin de favoriser la compréhension immédiate du fonctionnement de l'ordonnanceur qui a été réalisé pour ce langage. Voici donc une présentation succincte de la structure d'un fichier **Cubicle**. J'ai décidé d'utiliser, plutôt qu'un exemple purement théorique, le fichier **Cubicle** correspondant au protocole German-*ish*, ce fichier étant celui sur lequel j'ai le plus travaillé car assez riche syntaxiquement sans être trop lourd. 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 ↓

1. Attention, je n'entrerai pas dans les détails du langage car cela a déja été fait dans un des articles que j'eu à lire [1] et je n'expliquerai pas non plus les choix sémantiques ou syntaxiques du langage - à moins que cela ne soit nécessaire. Il ne faut pas non plus s'attendre à une explication détaillée de **BRAB**, décrite dans un autre de ces articles [2]. Je ne décrirai donc que les parties importantes à savoir pour comprendre en quoi a consisté le travail qui m'a été demandé.

2.1.1 Déclaration des types et variables typées

Cubicle est un langage typé. Les types `int`, `real` et `bool` sont reconnus par son compilateur ainsi qu'un type `proc` qui permet d'identifier les processus. De plus, il est possible pour l'utilisateur de définir deux autres sortes de types, les types énumérés ou les types abstraits [2] :

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

FIGURE 2.1: Définition de types pour **Cubicle**

Après avoir défini ses propres types, 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 2.2: Déclarations de variables et de tableaux pour **Cubicle**

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

2. Dont nous verrons plus tard qu'ils exigent deux traitement bien distincts.↓

```
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 2.3: Déclarations dans German-*ish* pour **Cubicle**

2.1.2 Déclaration de l'état initial

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

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

FIGURE 2.4: Etat initial du protocole German-*ish* pour **Cubicle**[3]

qui peut être lu comme ceci :

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

FIGURE 2.5: Etat initial du protocole German-*ish* sous forme logique

3. On remarquera que la variable `Curptr` n'est pas initialisée, je reviendrai dessus plus tard↓

2.1.3 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 2.6: Etat unsafe du protocole German-*ish* pour **Cubicle**

peut être lu comme ceci :

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

FIGURE 2.7: Etat unsafe du protocole German-*ish* sous forme logique

2.1.4 Déclaration des transitions

Reprenons notre exemple (ici, une transition de German-*ish*) :

```
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 2.8: Une transition du protocole German-*ish* pour **Cubicle**

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 :

$$gnt\_exclusive : \exists n.\ \begin{array}{l} \neg \texttt{Shrset}[n]\ \wedge\ \texttt{Curcmd} = \texttt{Reqe}\ \wedge \\ \neg \texttt{Exgntd}\ \wedge\ \texttt{Curptr} = n\ \wedge \\ \forall l.l \neq n\ \neg \texttt{Shr}[l] \end{array}$$

FIGURE 2.9: Une transition du protocole German-*ish* sous forme logique

La partie correspondant aux modifications du système n'est donc exécutée que si cette garde est vérifiée.

Voilà, donc, un bref aperçu de ce langage et j'espère que ceci a permis une compréhension précise de ce qu'il est possible de faire avec **Cubicle**. En réalité, le plus simple est de voir **Cubicle** comme un automate non déterministe avec un état initial et un ou plusieurs états terminaux que sont les états unsafe.

2.2 BRAB

Attaquons maintenant le corps du sujet, vérifier qu'un algorithme est sûr. Pour cela, on génère l'ensemble des états pouvant mener aux états unsafe et on vérifie que cet ensemble ne contient pas l'état initial. Dans ce cas là, naturellement, toute exécution partant de l'état initial ne conduira jamais à un état critique et l'algorithme est sûr.

Malheureusement, pour un protocole ayant uniquement dix variables pouvant avoir trois valeurs différentes, on a déjà un ensemble d'état potentiel contenant 3^{10} états. Si l'algorithme devait parcourir tous ces états, le temps passé serait beaucoup trop long. De plus, on cherche ici à prouver ces algorithmes quel que soit le nombre de processus. C'est là qu'intervient **BRAB** et, afin de comprendre comment il fonctionne, je préfère l'illustrer par un exemple.

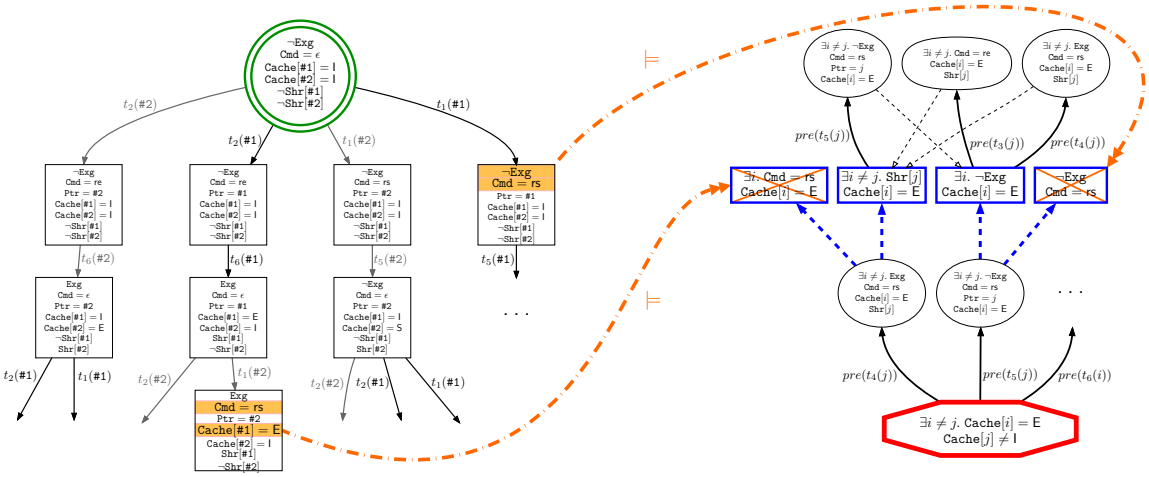

FIGURE 2.10: Exemple du fonctionnement de **BRAB** pour German-*ish*

L'état *unsafe* est représenté par l'octogone rouge. **BRAB** exécute un chaînage arrière afin de savoir quels états possibles auraient pu mener à cet état. Ces états sont malheureusement trop restrictifs et conduisent au problème évoqué précédemment. On essaye donc d'en faire des approximations qui sont représentées ici par les rectangles bleus. Or, qui dit approximation, dit chance de se tromper. Ainsi, à chaque approximation faite, on vérifie qu'elle ne représente pas un état vu depuis une exécution du programme depuis l'état initial. Si tel est le cas, l'approximation conduira forcément à une erreur qui n'est pas obligatoirement réaliste. On en cherche donc une nouvelle et ainsi de suite jusqu'à arriver à un ensemble d'état stable par le système de transitions [4].

Il me semble maintenant que le fonctionnement de base est assez clair. Venons-en donc à la partie qui m'intéresse, la génération d'un nuage d'états assez expressifs pour permettre à **BRAB** de se tromper le moins possible [5]. En effet, si, lors d'une approximation, **BRAB** cherche à savoir si celle-ci est judicieuse et qu'il obtient une réponse affirmative alors qu'il s'avère que ce n'est pas le cas, il continuera à chercher des pré-images de cette approximation ce qui le conduira inévitablement à se rendre compte de l'erreur commise et à devoir faire un redémarrage depuis un état précédant cette approximation. Ces redémarrages sont coûteux en temps. Voilà pourquoi il a semblé intéressant à l'équipe travaillant sur ce projet de faire un ordonnanceur programmable afin de pouvoir générer des états qui soient judicieux, assez nombreux, assez expressifs. Nous voilà donc entrés dans le vif du sujet.

4. Toute pré-image donne un état déjà existant dans cet ensemble
5. Pas du tout serait même encore mieux.

Chapitre 3

Les données et leur initialisation

Cette partie est, de loin, celle qui m'a demandé le plus de temps, de corrections, de modifications. Loin d'être triviale, l'initialisation du système est une partie ô combien importante car d'elle découle le bon déroulement de l'ordonnancement.

Précisons néanmoins ce qui n'a pas encore été fait pour cet ordonnanceur. J'ai, de commun accord avec Alain Mebsout et Sylvain Conchon, décidé de ne pas m'occuper des entiers et des flottants qui représentent une partie assez restreinte des fichiers **Cubicle** et qui n'apporte pas un grand intérêt pour le travail demandé. De plus, il n'y a pas réellement de booléen dans la sémantique de **Cubicle** car ceux-ci sont remplacés par un type énuméré :

```
type bool = False | True
```

Ainsi, je n'ai eu à m'intéresser qu'aux types énumérés, aux types abstraits et aux types des processus.

3.1 Les structures de données

Il serait judicieux de commencer la description de cet ordonnanceur par une présentation de ses structures de données. Plusieurs choix se sont proposés à moi. Convertir un fichier **Cubicle** en fichier **OCaml** et exécuter ce fichier (compilation) ou, au contraire, exécuter un fichier **Cubicle** grâce à un programme **OCAml** (interprétation). J'ai choisi une espèce de mélange des deux et, pour cela, il a fallu que je crée des structures de données pour représenter les tableaux, les variables globales, un état du système, un ensemble de ces états. Pour le moment, je vais uniquement

Premièrement, avant de commencer, un petit travail de réflexion s'imposait pour pouvoir gérer les tableaux. En effet, bien que cela n'ait pas été mentionné auparavant, il faut savoir que les tableaux de **Cubicle** sont potentiellement multidimensionnels. Valait-il mieux les gérer un par un ou les regrouper sous un module pouvant représenter un tableau simple comme un tableau à cinq dimensions. J'ai, premièrement, décidé de ne faire que des tableaux à une dimension et de gérer

les multidimensionnels en les écrasant. Malheureusement, cela entraînait de sérieuses complications lorsqu'il s'agissait d'accéder à des éléments en particulier et, d'un point de vue de la lisibilité, était catastrophique. J'ai donc décidé de créer un module :

```
module type DA = sig

  type 'a t (=
    | Arr of 'a array
    | Mat of 'a t array)

  type 'a dima (= {dim:int; darr:'a t})

  val init : int -> int -> 'a -> 'a dima
  val minit : int -> int -> ('a * int) list -> 'a -> 'a dima
  val get : 'a dima -> int list -> 'a
  val set : 'a dima -> int list -> 'a -> unit
  val print : 'a dima -> (Format.formatter -> 'a -> unit) -> unit
  val copy : 'a dima -> 'a dima
  val dim : 'a dima -> int
  val equal : ('a -> 'a -> bool) -> 'a dima -> 'a dima -> bool

end
```

FIGURE 3.1: Le module représentant un tableau

Les tableaux faits, je pouvais les remplir. Malheureusement, bien que mon module `DA` soit polymorphe, je ne pouvais pas y stocker tous les types de variables, il me fallait, pour cela, déclarer un type qui les regrouperait tous [1] (énumérés, abstraits, types de bases...).

```
type value =
  | Var of Hstring.t
  | Hstr of Hstring.t
  | Proc of int
```

FIGURE 3.2: Le type des éléments

1. Oui, c'est bien une citation détournée.

Cela étant fait, je pouvais enfin définir ce qu'était un état de mon point de vue avec un nouveau module.

```
module type St = sig

  (* A dimensional array *)
  type 'a da
  (* The state : global variables and arrays *)
  type 'a t = {globs :(Hstring.t, 'a) Hashtbl.t;
               arrs : (Hstring.t, 'a da) Hashtbl.t}

  val init : unit -> 'a t

  val equal : ('a -> 'a -> bool) -> 'a t -> 'a t -> bool
  val hash : 'a t -> int

  (* Get a global variable value *)
  val get_v : 'a t -> Hstring.t -> 'a
  (* Get an array by its name *)
  val get_a : 'a t -> Hstring.t -> 'a da
  (* Get an element in an array by its name and a a param list *)
  val get_e : 'a t -> Hstring.t -> int list -> 'a

  (* Set a global variable value *)
  val set_v : 'a t -> Hstring.t -> 'a -> unit
  (* Set an array by its name and a new array *)
  val set_a : 'a t -> Hstring.t -> 'a da -> unit
  (* Set an element in an array by its name, a param list and a value *)
  val set_e : 'a t -> Hstring.t -> int list -> 'a -> unit

  val copy : 'a t -> 'a t

end
```

FIGURE 3.3: Le module représentant un état du système

Finalement, un système finissait par être représenté comme ceci :

```
module type Sys = sig

  (* A state *)
  type 'a s
  (* A dimensional array *)
  type 'a da
  type 'a set
  (* A record with a readable state and a writable state *)
  type 'a t = {syst : 'a set; init : 'a set; read_st : 'a s; write_st : 'a s}

  val init : unit -> 'a t

  val get_v : 'a t -> Hstring.t -> 'a
  val get_a : 'a t -> Hstring.t -> 'a da
  val get_e : 'a t -> Hstring.t -> int list -> 'a

  val set_v : 'a t -> Hstring.t -> 'a -> unit
  val set_a : 'a t -> Hstring.t -> 'a da -> unit
  val set_e : 'a t -> Hstring.t -> int list -> 'a -> unit

  val exists : ('a s -> bool) -> 'a t -> bool

  val exists_init : ('a -> 'a -> bool) -> 'a s -> 'a t -> bool

  val update_init : 'a t -> (Hstring.t * 'a s) -> 'a t

  val get_init : 'a t -> 'a set

  val new_init : Hstring.t -> 'a t -> 'a s -> 'a t

  val update_s : Hstring.t -> 'a t -> 'a t

end
```

FIGURE 3.4: Le module représentant le système

Bien que tout cela puisse paraître obscur pour le moment, pas d'inquiétude, je vais expliquer ces choix d'implémentation dans les parties suivantes. On retiendra simplement que le système contient un ensemble d'états (plus d'autres choses mais qui sont importantes pour l'initialisation et l'ordonnancement) qui contiennent, eux, des `Hashtbl` de `value` et des `Hashtbl` de `DimArray` de `value`.

3.2 Initialisation ? Des classes d'équivalences et de différences

Quatre cas de figures peuvent arriver lors de l'initialisation. Les deux plus simples sont lorsqu'un élément est égal à une valeur ou différent d'une valeur. Deux cas le sont moins : lorsque deux éléments sont égaux ou différents.

Afin de mettre en œuvre une initialisation efficace, j'ai décidé de fonctionner avec un prétraitement grâce auquel je remplis des classes d'équivalence et de différence[2].

Petite astuce, on aura remarqué dans la figure 3.2, la présence de `Var of Hstring.t`. Cette valeur, qui n'existe que pour l'initialisation, permet de savoir que la valeur actuelle d'un élément est lui-même ou son représentant qui est aussi un élément.

```
module TS = Set.Make (value)
(* Ensemble regroupant des noms d'éléments, tableaux ou variables *)
module TI = Set.Make (Hstring.t)

val ec : (Hstring.t, TS.elt * stype) Hashtbl.t
val dc : (Hstring.t, ty * TS.t * TI.t) Hashtbl.t
```

FIGURE 3.5: Les classes d'équivalence et de différences

Ainsi, dans le cas où un élément est égal à une valeur, on va chercher son représentant dans `ec` et on modifie celui-ci avec la valeur.

3.3 Déterministe vs non déterministe

3.4 Colorions

2. Je me suis basé, ici, sur une simplification des structures Union-Find

Chapitre 4

Les transitions et leur exécution

4.1 La garde

4.2 Mise à jour

Bibliographie

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

[2] Sylvain Conchon, Amit Goel, Sava Krstić, Alain Mebsout, Fatiha Zaïdi
Invariants for Finite Instances and Beyond
https://www.lri.fr/~conchon/TER/2013/1/invariants.pdf

[3] Sylvain Conchon, David Declerck, Luc Maranget, Alain Mebsout
Vérification de programmes C concurrents avec **Cubicle** : Enfoncer les barrières
https://www.lri.fr/~conchon/TER/2013/1/c2cub.pdf