Rapport de TER
Réalisation d'un ordonnanceur programmable pour Cubicle

Mattias Roux

9 mai 2014

Table des matières

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.

Je profite de cette introduction pour expliquer clairement ma démarche lors de la rédaction de ce rapport. J'ai décidé d'y mettre beaucoup de parties de code qui, bien que paraissant obscures de prime abord, se révéleront vite claires grâce aux explications. Ainsi, le code arrive avant l'explication et il sera peut-être nécessaire de s'y référer assez souvent. C'est pour cela que j'ai, dans un souci de lisibilité, décidé de rendre le code remarquable [2].

Bonne lecture.

1. Travail d'Etude et de Recherche
2. J'en profite pour remercier les auteurs de [1] qui m'ont permis de réutiliser certaines de leurs fonctions LaTeX.

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**

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

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 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 :

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

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 suffisante de ce qu'il est possible de faire avec **Cubicle** pour comprendre ce que j'ai fait avec mon ordonnanceur. 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, l'ordonnanceur se charge donc de faire tourner l'automate et de garder en mémoire les états visités.

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.

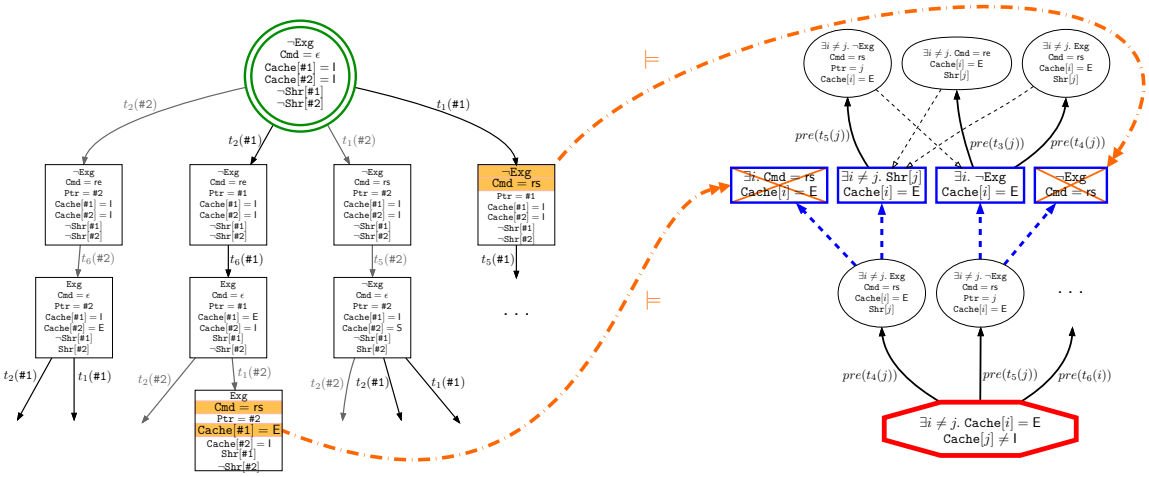

FIGURE 2.10: Exemple du fonctionnement de **BRAB** pour German-*ish*

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.

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 permettant de représenter tous les tableaux avec un seul type :

```
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 de **Cubicle**, il me fallait, pour cela, déclarer un type qui les regrouperait tous (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

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

On retiendra, ici, que le système contient un ensemble d'états [1] qui contiennent, eux, des `Hashtbl` de `value` et des `Hashtbl` de `DimArray` de `value`.

D'un point de vue apprentissage, cette partie m'a permis de comprendre assez précisément le

1. Plus d'autres choses mais qui sont importantes pour l'initialisation et l'ordonnancement et qui seront donc expliquées après.

fonctionnement des modules, des foncteurs et du polymorphisme en **OCaml**. On remarquera, en effet, que ces trois modules sont indépendants les uns des autres (une implémentation différente d'un état mais qui aurait la même signature, ne changerait rien à l'implémentation du système, par exemple.) et entièrement polymorphes. Nous manipulons ici des éléments de type `value` mais il aurait tout-à-fait été possible de manipuler des entiers ou des chaînes de caractères.

3.2 La rentrée des classes

Il convient, avant de commencer, d'expliciter les choix qui ont été faits pour les types abstraits. Reprenons notre type `abstr` de la figure 2.1. A la figure 2.2, on remarque que la variable `Abs` a ce type et c'est la seule. Dorénavant, le type `abstr` sera un type énuméré dont la seule valeur possible sera `Abs`. Ainsi, chaque variable de type abstrait prendra comme valeur elle-même, sinon, si elle a été initialisée avec une autre variable du même type, (qui ne peut pas être une valeur, car ce type n'a pas, à proprement parler, de valeur), les deux variables auront comme valeur une valeur choisie parmi elle deux (si $v : abstr = v' : abstr$ alors $v \leftarrow Hstr \quad v$ et $v' \leftarrow Hstr \quad v$).

Afin de mettre en œuvre une initialisation efficace, j'ai décidé de fonctionner avec un prétraitement me permettant de remplir des classes d'équivalence [2] et de différences.

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 le représentant actuel d'un élément est un élément et non pas une valeur.*

Sans plus tarder, voici donc les deux classes :

```
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

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

```
(* Hstring.t représente le type de la variable *)
type stype =
     | RGlob of Hstring.t
     | RArr of (Hstring.t * int)
```

FIGURE 3.6: stype : variable ou tableau avec type et dimension

Dont voici l'explication :

ec : $\begin{cases} \texttt{Hstring.t} & \text{Le nom de l'élément} \\ \texttt{TS.elt} & \text{Son représentant (un élément ou une valeur, cf figure 3.2)} \\ \texttt{stype} & \text{cf figure 3.6} \end{cases}$

dc : $\begin{cases} \texttt{Hstring.t} & \text{Le nom du représentant de la classe (un élément qui représente tous les éléments égaux entre eux)} \\ \texttt{ty} & \text{Le type de cette classe : abstrait ou énuméré} \\ \texttt{TS.t} & \text{L'ensemble des valeurs possibles pour cette classe} \\ \texttt{TI.t} & \text{L'ensemble des représentants différents de celui-ci} \end{cases}$

Ces structures sont initialisées avec des valeurs par défaut :

$$\text{init\_ce} : \forall e.\ \texttt{Hashtbl.add}\ ec\ (Var\ \ e)\ (e,\ \texttt{stype}(e))$$

$$\text{init\_dc} : \forall e.\ \texttt{Hashtbl.add}\ dc\ \ e\ \ (\texttt{ty}(e),\ \begin{cases} \emptyset & \texttt{ty}(e) = abstract \\ \{Var\ \ e\} & \texttt{ty}(e) = other \end{cases},\ \emptyset)$$

FIGURE 3.7: Initialisation de ec et de dc

Les classes d'équivalence et de différences étant définies et initialisées, voyons les cinq cas de figures pouvant arriver lors de l'initialisation à proprement parler [3]

Element e = Valeur v	On supprime définitivement le représentant `r` de `e` de `dc` et on modifie le représentant dans `ec` de `e` ainsi que tous les éléments ayant le même représentant par `v`. De plus, tout élément de `dc` étant différent de `r` voit la valeur `v` supprimée de ses valeurs possibles.	(3.1)
Element e = Element e'	On suppose que les représentants de `e` et de `e'` ne sont pas des valeurs, auquel cas on serait ramené au premier cas. L'un des deux éléments devient le représentant de tous les éléments de la classe du second. S'ils sont de type abstrait, la valeur possible est le représentant, sinon, c'est l'intersection des types possibles des deux classes de différences. Ensuite, on supprime la classe qui n'est pas représentante de `dc`.	(3.2)
Element e $\neq$ Valeur v	Naturellement, ici, on supprime la valeur des valeurs possibles et on ne fait rien d'autre [4]	(3.3)
Element e $\neq$ Element e'	On suppose que les représentants de `e` et `e'` ne sont pas des valeurs. On change uniquement les ensembles `TI` de chacun des représentants `r` de `e` et `r'` de `e'` en y ajoutant respectivement `r'` et `r`.	(3.4)
Element e $\neq$ Proc z	Ce cas est particulier et intervient uniquement dans le cas où une variable est différente de tous les processus (cf figure 2.4. Dans ce cas là, on détermine que la seule valeur possible pour celle-ci est un processus fantôme défini depuis le début comme étant égal au nombre de processus plus 1.	(3.5)

Bien que cela n'apparaisse pas ici car étant en cours de construction, le cas des entiers et des réels est géré aussi mais entraîne un traitement différent qu'il ne me semblait pas judicieux d'expliquer.

Désormais, pour l'initialisation, le reste du travail ne se fera qu'avec ces deux groupes.

3.3 Coloriage

Nous avons donc des classes d'équivalence dans laquelle chaque élément devra avoir la même valeur et des classes de différences où sont répertoriées les valeurs possibles pour chaque classe ainsi que les classes desquelles elles doivent être différentes.

On épargnera au lecteur la transformation de `dc` en graphes de différences car cela est surtout le résultat de la lecture de `dc` afin de remplir une table avec des groupes de différences et nous allons plutôt l'illustrer par un exemple.

3. Celle du fichier **Cubicle**

```
type state = I | S | E

var S1 : state
var S2 : state
var S3 : state
var S4 : state
var S5 : state
var S6 : state

init (z) {
     S1 = S2 &&
     S2 <> S3 &&
     S3 <> S5 &&
     S1 <> S6 &&
     S4 <> I }
```

FIGURE 3.8: Exemple d'initialisation

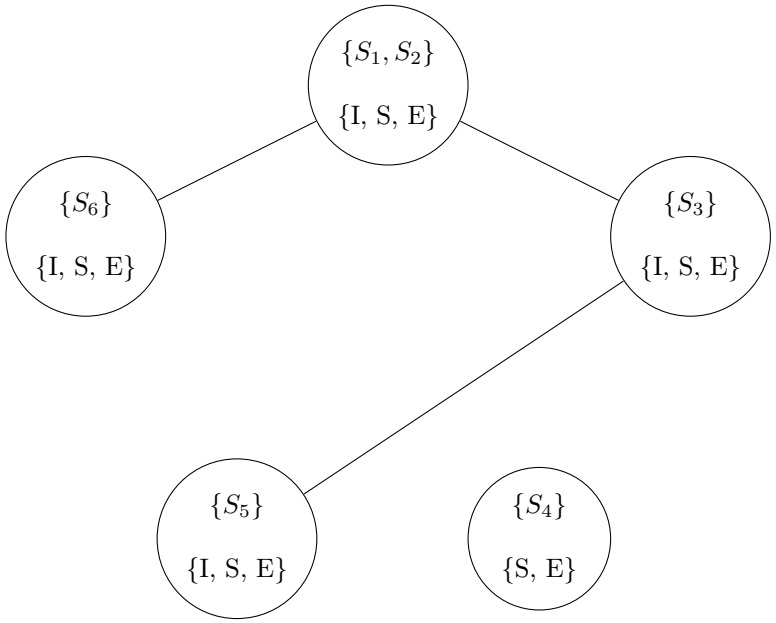

FIGURE 3.9: Le graphe correspondant

Dont une solution possible est :

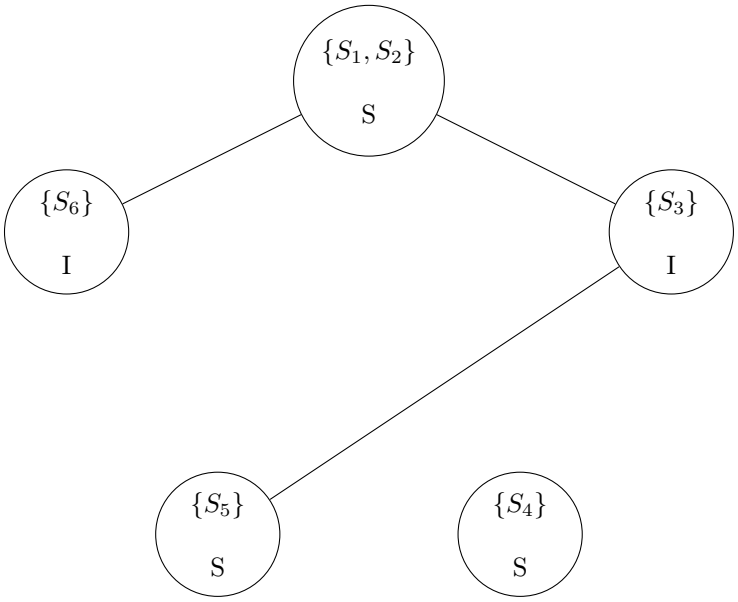

FIGURE 3.10: Le graphe solution

Solution que nous avons trouvé en appliquant l'algorithme de coloriage de graphe suivant :

```
function COLORIAGE(G(V, E))
    Q ← SORT_NODES(V)
    for all n ∈ Q do
        v ← AVAILABLE_VALUE(n)
        n.value ← v
        for all n' ∈ Q \ {n} do
            if (n, n') ∉ E then
                n'.value ← v
                Q ← Q \ {n'}
            else if n' ∈ CFC(n) then
                /* CFC(n) est la composante fortement connexe contenant n */
                n'.available_value ← n'.available_value \ {v}
            end if
        end for
    end for
end function
```

FIGURE 3.11: Initialisation des variables selon leurs différences

La fonction SORT_NODES(V) trie les nœuds selon deux critères. Les nœuds ayant le moins de valeurs disponibles sont les premiers, puis arrivent ceux ayant le plus haut degré (donc étant différent du plus grand nombre). Cet algorithme a néanmoins un défaut, il fait une initialisation déterministe (la fonction AVAILABLE_VALUE(n) choisit la première valeur disponible). Mais, de toutes façons, bien que cet algorithme ait été intéressant à trouver, il n'est pas très intéressant par rapport aux fichiers que j'avais en ma possession car peu de variables sont initialisées par rapport à leur différence avec d'autres. Voici donc la partie intéressante, l'initialisation multiple.

3.4 Multiplication

Dans la plupart des cas, il n'y a pas de différences. On peut donc oublier les cas 3.3 et 3.4 et se concentrer sur les cas 3.1, 3.2 et 3.5 [5].

Dans les cas 3.1 et 3.5, tout se passe pour le mieux car la classe ne peut avoir qu'une seule valeur et cela introduira donc une seule initialisation possible pour les éléments de cette classe.

Le cas qui nous intéresse est donc le 3.2. Cela entraîne simplement le fait que deux éléments doivent être égaux mais rien n'est dit sur la valeur qu'ils doivent avoir. Originellement, pour des

5. On considère dorénavant que les variables ne sont pas liées par des relations de différence.

raisons de simplicité évidente, j'avais décidé d'initialiser ces variables à une valeur choisie par défaut [6]. Le système avait donc un label `init` mais celui-ci n'était pas un ensemble d'états, c'était un état unique.

Cela s'est avéré, d'un point de vue expressif, beaucoup trop faible pour des fichiers complexes (adieu German-*ish*, dorénavant il m'a fallu travailler sur Flash). J'ai donc décidé d'initialiser toute variable non initialisée à deux de ses valeurs possibles. Cette solution fut très rapidement abandonnée du fait de l'explosion du nombre d'états qu'elle engendrait. Imaginez un peu, pour quinze variables non initialisées [7] on obtient 2^{15} états différents. Autant dire que la mémoire ne supportait pas beaucoup ce cas de figure.

Comme d'habitude, il a fallu arrêter de foncer tête baissée dans la programmation pour réfléchir plutôt à ce qui était vraiment intéressant. Alain et moi avons donc lancés plusieurs fichiers pour nous rendre compte qu'il n'y avait, en réalité, que deux cas qui posaient problèmes.

Le cas des variables de type processus non initialisés

Le cas de certains tableaux représentant le contenu des caches non initialisés

Il a donc été décidé qu'un fichier de configuration de l'ordonnanceur serait créé pour chaque fichier **Cubicle** le nécessitant. Nous y voilà enfin, l'ordonnanceur pouvait commencer à devenir programmable.

J'ai donc identifiés les cas nous posant actuellement problème et ai associé à chacun une commande spécifique dans le langage [8] que j'ai créé pour l'occasion.

```
opt not fproc
opt fproc
proc_init Nom_de_variable [0,-1]
tab_init Nom_de_tableau [(V1, 1), (V2, 2)]
```

FIGURE 3.12: Les quatre commandes possibles du langage de l'ordonnaceur

6. Par exemple, la première apparaissant dans la définition du type de la variable.
7. On aura enfin compris que "non initialisée" signifie que la variable n'est pas dans les cas 3.1 ou 3.5
8. Très réduit, pour le moment

opt not fproc	Toute variable de type processus non initialisée sera initialisée à une unique valeur par défaut.
opt fproc	Toute variable de type processus non initialisée sera initialisée à deux valeurs. Celle par défaut et le processus fantôme.
proc_init Nom_de_variable [0,-1]	Oblige la variable à prendre chacune des valeurs apparaissant dans la liste. La valeur -1 correspond au processus fantôme.
tab_init Nom_de_tableau [(V1, 1), (V2, 2)]	Pour chaque élément associé à un nombre n dans la liste, on veut que cet élément apparaisse au moins n fois dans le tableau.

Nous voilà arrivés à la fin de l'initialisation. Et on comprend maintenant pourquoi le label `init` du système est un ensemble d'états. Celui-ci représente en effet tous les états initiaux du système en fonction de ce qui a été décidé dans le fichier *.sched* associé au fichier *.cub*.

Chapitre 4

Les transitions et leur exécution

Nous voilà maintenant avec un nombre connus d'états initiaux et la volonté de générer des états possibles du système à partir de ces états. En réalité, cette partie n'occupe qu'une place minime dans le travail que j'ai dû faire car elle est assez triviale et, comme je l'avais dit précédemment, il suffit simplement d'exécuter différentes transitions d'un automate. Je ne vais donc pas passer beaucoup de temps à expliquer ce que j'ai fait. Brièvement, on parcourt l'ensemble des transitions du fichier **Cubicle** et on sauvegarde dans une table les gardes et les mises-à-jour qu'elles entraînent. A chaque exécution, on parcourt toute la table à la recherche des transitions possibles (par une procédure de filtrage dont **OCaml** munit toutes ses structures) et on garde dans une liste l'ensemble des mises-à-jour possibles. On choisit ensuite aléatoirement une de ces mises-à-jour puis on l'exécute et on recommence à partir du filtrage.

Ce qui nous donne :

```
function SELECT_AND_EXECUTE_TRANSITION(t_list : (garde * update) list)
    U ← ∅
    for all (g, u) ∈ t_list do
        if g () then
            U ← U ∪ {u}
        end if
    end for
    UPD ← RANDOM_ELEMENT(U)
    UPD ( )
end function
function SCHEDULE(nb_exec, t_list)
    for i = 0 to nb_exec do
        SELECT_AND_EXECUTE_TRANSITION(t_list)
    end for
end function
```

```
function SCHEDULE_ALL(nb_exec, t_list, system)
    for all i ∈ system.init do
        system.read_st ← i
        system.write_st ← i
        SCHEDULE(nb_exec, t_list)
    end for
end function
```

FIGURE 4.1: Ordonnancement

On comprend donc maintenant les labels `write_st` et `read_st` du système. Le premier correspond à l'état sur lequel seront effectuées les mises-à-jour, le deuxième, celui sur lequel seront exécutées les vérifications des gardes. Chaque appel à UPD sauvegarde l'actuel état `read_st`, et remplace `write_st` et `read_st` par le nouvel état obtenu. Avant de continuer, précisons que ces deux états sont obligatoires car lors d'une mise-à-jour, tout se fait en simultané, ainsi, si $V1 = A$ et la mise-à-jour à effectuer demande que $V1 \leftarrow B$ et si $V1 = A$ alors $V2 \leftarrow Vrai$, $V2$ devra être égal à $Vrai$. Il ne faut donc pas que la modification de l'état modifie les variables sur lesquelles on teste.

Voyons donc comment la garde et la mise-à-jour d'une transition ont été créées par rapport au fichier **Cubicle**.

4.1 La garde

Cette partie admet un intérêt qu'il ne faut pas négliger. Comme nous le montrions dans la figure 2.9, il faut trouver un processus [1] qui vérifie cette garde. Cela revient donc à dire que dans le cas de la figure 2.5 et avec trois processus, il faudra vérifier que cette garde est vraie pour le processus 1, le processus 2 ou le processus 3. J'ai donc réutilisé la fonction all_permutations de Sylvain Conchon et Alain Mebsout qui à deux listes $l_1 : \alpha$ et $l_2 : \beta$ telles que l_1 contient moins d'éléments que l_2 associe une liste $l_3 : (\alpha * \beta)\, list$ contenant une liste de chaque élément de l_1 associé à un unique élément de l_2.

```
all_permutations [1;2] ['a'; 'b'];;
- : (int * char) list list = [[(1, 'a'); (2, 'b')]; [(1, 'b'); (2, 'a')]]
```

FIGURE 4.2: Exemple d'exécution d'all_permutations

1. Il se peut qu'il en faille deux ou plus mais le raisonnement est le même

Chaque garde est donc transformée en une fonction qui à `unit` associe une disjonction de garde appliquée à chaque liste de l_3. Dans notre cas, l_1 est la liste des paramètres de la transition et l_2 la liste des processus, l_3 associe donc à chaque liste de paramètres une liste de processus possibles.

Le cas du `forall other l` est particulier. Il faut prendre tous les processus qui n'ont pas été pris dans la sous-liste de l_3 qu'on est en train de traiter et vérifier qu'ils vérifient tous la condition demandée. On crée donc une conjonction de cette condition appliquée à chacun de ces processus.

Ainsi, pour trois processus, la garde de la figure 2.5 devient :

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

( Shrset[1] = False && Curcmd = Reqe && Exgntd = False && Curptr = 1 &&
  Shrset[2] = False && Shrset[3] = False ) ||
( Shrset[2] = False && Curcmd = Reqe && Exgntd = False && Curptr = 2 &&
  Shrset[1] = False && Shrset[3] = False ) ||
( Shrset[3] = False && Curcmd = Reqe && Exgntd = False && Curptr = 3 &&
  Shrset[1] = False && Shrset[2] = False )
```

FIGURE 4.3: La garde de German-*ish* après transformation

4.2 Mise à jour

Les mises-à-jour ne sont intéressantes que lorsqu'il s'agit de modifier des tableaux (la mise à jour de variable revient uniquement à modifier la variable, cela ne semble pas très compliqué).

Une mise-à-jour de tableau revient à mettre à jour les cases sous certaines conditions. En réalité, cela revient à créer une garde pour chaque case et à l'exécuter séquentiellement pour savoir quelle condition la case vérifie pour être mise à jour. Vous excuserez la brièveté de cette explication mais l'intérêt de cette partie est moindre et par trop ressemblant à la partie relative aux gardes. Il est donc conseillé de se référer à celle-ci pour tout compréhension supplémentaire.

Chapitre 5

Conclusion

Afin que ce TER serve à quelque-chose, nous avons branché mon ordonnanceur au programme de l'équipe. Pour ce faire, **BRAB** envoie à l'ordonnanceur une liste de formules représentant des sous-états du système et celui-ci se charge de trouver la première formule pour laquelle aucun des états qu'il a visités ne la contient. S'il la trouve, il la renvoie, sinon il annonce qu'il n'a rien trouvé [1]. Et ça marche ! Enfin, pas pour certains protocoles mais ce sont les plus compliqués et je vais travailler dessus lors de mon stage.

Ce travail, bien que pas terminé mais vraiment prometteur, m'a permis de mettre en application différentes choses que j'ai vues lors de mes études (théorie des graphes, algorithmique, programmation fonctionnelle, programmation impérative) afin de mener à bien la création d'un ordonnanceur programmable. Il était donc à la fois compliqué mais vraiment passionnant pour tout ce qu'il demandait de croisement de connaissances.

Bien que cet ordonnanceur n'en soit encore qu'à ses balbutiements, le fait est que certains résultats obtenus sont vraiment engageants pour la suite. Et c'est surtout cela qui m'a plu. Me rendre compte qu'un travail de recherche ne consiste pas forcément à trouver la solution mais à penser qu'explorer tel ou tel chemin permettra de la trouver.

J'espère que vous n'aurez pas été trop perdu dans la construction de ce rapport mais il est trop difficile d'être exhaustif pour un TER si long. La partie relative aux transitions et aux mises-à-jour aurait peut-être pu bénéficier d'un meilleur traitement mais il s'avère que je ne la trouve vraiment pas intéressante pour la compréhension de l'ordonnanceur ainsi que du travail fourni.

Puisque je ne sais jamais où il faut mettre les remerciements (à vrai dire, je ne sais pas s'il faut en mettre mais ça me fait plaisir), je les place ici. Merci, donc, à Sylvain Conchon pour ce sujet vraiment intéressant et la patience dont il a fait preuve, à Alain Mebsout pour sa patience indéfectible malgré une soutenance de thèse approchant ainsi qu'à Kim Nguyễn pour les cafés. Enfin, merci à mes deux camarades de bureau, Albin Coquereau et Thibaut Tachon qui ont supportés mes chants et mes innombrables discussions inutiles.

1. Ce cas signifie que toutes les approximations trouvées par **BRAB** découlent de l'ordonnancement depuis les états initiaux.

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

[4] Wikipedia
https://en.wikipedia.org/wiki/Main_Page