Rapport de stage

Lilian Derain

July 3, 2024

1 Remerciement

2 Résumé/Abstract

Stage de 3 mois au labo Verimag. L'objectif de ce stage est d'utiliser des méthodes formelles pour prouver de façon la plus automatisée possible la correction d'algorithmes distribués auto-stabilisants. L'idée de ce stage est d'explorer d'autres techniques de preuves, basées sur la logique de séparation et des solveurs SMT. En particulier l'article why3-do se présentant comme un modèle pour la preuve d'algorithmes distribués auto-stabilisants basé sur l'outil Why3 L'objectif de ce stage est d'évaluer ce qu'il est possible et impossible de faire avec ces outils compte tenu de leur maturité actuelle.

3 Contexte

3.1 Verimag

Le laboratoire VERIMAG est un centre de recherche en informatique situé à Grenoble. Fondé en 1991, le laboratoire est affilié à l'UGA. Le laboratoire vise à produire des outils théoriques et techniques sur les systèmes informatisés en mettant en place une rigueur mathématiques. Le laboratoire agit sur de nombreux problèmes tels que les circuits, les processeurs, des al- gorithmes distribués et des systèmes intégrants de l'IA. L'une des spécialités du laboratoire est la vérifcation formelle. Il s'agit d'une approche mathématique permettant de garantir que des systèmes répondent bien aux spécifications et propriétés. Erwan Jahier et Karine Altisen utilise font de la vérification formelle dans le cadre des algorithmes distribués auto-stabilisants. Ce sont des algorithmes qui peuvent atteindre un état correct depuis n'importe quel état sans intervention externe.

3.2 Organisation de travail

Au début -> installations des outils (why3 et why3-do) et apprentissage des connaissances de bases (logique de hoare, smt solver, algo distribué auto-stabilisant)

Echanges fréquents avec Erwan sur les questions/problèmes rencontrés

Et réunion toutes les une semaine ou deux aver Erwan et Karine pour mettre au point les avancements depuis la dernière réunion :

1. Expliquer ce que j'ai compris -> le fonctionnement et les limitations de ce sur quoi je travaille
2. Présenter mon avancement actuel : ce que j'ai réussi, ce qui marche (pas), là où je suis bloqué
3. Avoir un retour sur ce quoi je dois me focaliser pour la prochaine semaine, des nouvelles idées pour faire face aux problèmes que je n'ai pas su résoudre

4 Algorithme distribué auto-stable

4.1 Définition algorithme distribué auto-stable

Le concept s'auto stabilisation a été introduit par Dijkstra en 1973 dans le contexte des systèmes distribués, c'est à dire des systèmes fait d'un ensemble fini de process autonome interconnecté grâce à un réseau de communication, qui a pour but d'atteindre un objectif global. Dans ce cas le design d'un algorithme distribué auto-stable peut sembler un peu complexe car chaque process doivent se coordiner entre eux bien que les process ont une vue partielle du système. Son état local et ses informations transmises via des supports de communication, généralement asynchrones, reliant le processus à une partie des autres processus.

Tolérances aux fautes: L'algo converge d'un état illégitime à un état légitime dans un nombre fini d'étape Une fois que l'algo est dans un état légitime, il le reste S'il y a une "transient fault" et que l'on retourne dans un état illégitime alors l'algorithme converge de lui-même jusqu'à atteindre un état légitime.

Plein de petits algos simples à la place d'un seul gros qui gère tout.

Un graphe où chaque noeuds contient des variables et une action de la forme guard -> statement . Si la guarde vaut true alors l'action est "enabled" Un noeud à accès uniquement aux variables de ses voisins

A chaque étape "step", on regarde la liste des actions "enabled". Cependant on peut choisir différentes manières de procéder donc concept de de démon

Exemples non exhaustifs de démons :

1. Central : On active un seul noeud parmi la liste des noeuds et on mets à jour la liste des noeuds enabled
2. Synchrone : On active tous les noeuds de la liste
3. Distribué : On active au moins un noeuds parmi ceux enabled

3 propriétés Closure : depuis un état légitime on reste dans un état légitime Convergence : depuis un état illégitime on atteint un état légitime avec un nombre fini d'étape Correctness

4.2 Example 1 : Coloring

- Algorithme
- Exemple simple d'execution sous démon centrale
- Exemple simple d'execution sous démon synchrone (pour montrer que ça marche pas)
- Preuve

4.3 Example 2 : Token Ring Dijkstra

- Explication de l'algorithme pour pouvoir expliquer Why3do
- Exemple d'éxecution simple sous démon central

5 Explication des outils

5.1 Logique de Hoare

La logique de Hoare est un formalisme utilisé en informatique pour raisonner sur la correction des programmes impératifs. Elle utilise des triples de Hoare, notés {P} C {Q}, où :

- P est la précondition, une assertion sur l'état du programme avant l'exécution de l'instruction ou du bloc d'instructions C.

- C est l'instruction ou le bloc d'instructions du programme.
- Q est la postcondition, une assertion sur l'état du programme après l'exécution de C.

Un triple de Hoare {P} C {Q} signifie que si la précondition P est vraie avant l'exécution de C, alors la postcondition Q sera vraie après l'exécution de C, à condition que C termine son exécution.

Principes clés de la logique de Hoare

1. **Règle de l'assignation** : Pour une instruction d'assignation x := e,
   - {P[e/x]} x := e {P}, où P[e/x] est l'assertion P avec toutes les occurrences de x remplacées par e.
2. **Règle de composition** : Pour deux instructions C1 et C2,
   - Si {P} C1 {Q} et {Q} C2 {R}, alors {P} C1; C2 {R}.
3. **Règle de la conditionnelle** : Pour une instruction if (b) then C1 else C2,
   - {P b} C1 {Q} et {P ¬b} C2 {Q} impliquent {P} if (b) then C1 else C2 {Q}.
4. **Règle de la boucle** : Pour une boucle while (b) do C,
   - {I b} C {I} implique {I} while (b) do C {I ¬b}, où I est un invariant de boucle.

Considérons l'algorithme simple suivant qui incrémente une variable x :

```
{ x = 0 }
x := x + 1
{ x = 1 }
```

La précondition est x = 0, l'instruction est x := x + 1, et la postcondition est x = 1. Le triple de Hoare correspondant est { x = 0 } x := x + 1 { x = 1 }.

En conclusion, la logique de Hoare permet de structurer le raisonnement sur les programmes en termes de préconditions et de postconditions, facilitant la vérification formelle de leur correction.

5.2 SMT solver

5.3 Explication Why3

- Fonctionnement
- IDE
- Logique de preuve / informations utiles

6 Why3doTheWayOfHarmoniousDistributedSystemProofs

6.1 Explication du modèle : modelReadallEnable

6.2 Explication de selfstab-ring

- Closure
- Terminaison
- Avantages et Limitations (Comparaison avec Sasa)
  - Lien direct avec Ocaml
  - Seulement le démon central
  - Etat initial fixé
  - Topologie fixée
  - Accès à tous les états et pas uniquement les voisins

7 Implémentation et preuve de l'algorithme de coloring basé sur le modèle de l'artefact

7.1 Essai d'utilisation de org mode

```
print("hello world")
```

```
(* On va prouver la closure et la terminaison de l'algo de coloration dans une topologi
module Coloring_Ring
  use list.List
  use int.Int
  use int.EuclideanDivision
  use map.Map
  use list.Mem
```

```
  use list.Length
  use ref.Ref
  use set.SetAppInt

  (* PARTIE 1: CLOSURE *)

  (* number of processes *)
  val constant n_nodes : int

  axiom at_least_two : 2 < n_nodes

  (* Nombre de couleurs *)
  val constant k_color : int

  (* On suppose qu'il y a plus de couleurs que de noeuds (une couleur par noeud) *)
  axiom k_color_bound : n_nodes <= k_color

  type node = int
}
```

7.2 Explication détaillée

7.3 Problèmes rencontrés et solutions

8 Autres différents essais et faisabilité

- Pour tout état initial
- Pour toute topologie
- Noeuds peuvent accéder uniquement à leurs voisins
- Autres démons
  - Démon