Rapport de stage

Lilian Derain

July 17, 2024

1 Remerciement

En introduction de ce rapport, je tiens à exprimer ma gratitude envers le laboratoire VERIMAG pour m'avoir accueilli au sein de son équipe de recherche, me permettant ainsi d'explorer le monde de la recherche scientifique.

Je souhaite également remercier mon tuteur de stage Erwan Jahier qui m'a permis de découvrir le monde de la recherche et qui avec Karine Altisen m'ont beaucoup aidé durant ce stage.

Je remercie également Nicolas Palix qui a accepté d'être mon enseignant référant pendant ce stage.

2 Résumé/Abstract

Ce document fait office de rapport sur le stage de 3 mois que j'ai pu faire au laboratoire 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 de mon stage, j'ai consacré du temps à installer et configurer les outils essentiels, tels que Why3 et why3-do. En parallèle, j'ai entrepris d'acquérir des connaissances de base en logique de Hoare, SMT solver et algorithmes distribués auto-stabilisants pour mieux comprendre et utiliser ces outils. Cette phase initiale d'apprentissage était cruciale pour établir une base solide pour mes travaux futurs. Afin de surmonter les défis techniques et méthodologiques, des échanges fréquents avec Erwan ont été établis. Ces interactions régulières ont permis de clarifier des aspects complexes et de résoudre les problèmes rencontrés au quotidien.

En outre, des réunions avec Erwan et Karine ont été organisées toutes les une à deux semaines pour faire le point sur mes avancées. Lors de ces réunions, je présentais ce que j'avais compris, le fonctionnement et les limitations des outils et concepts sur lesquels je travaillais. Je détaillais également mon avancement en exposant ce qui fonctionnait, ce qui ne fonctionnait pas, et les obstacles rencontrés. Ces séances de retour d'information m'ont permis de recevoir des suggestions précieuses sur les domaines à améliorer et sur les points à focaliser pour la semaine suivante. Erwan et Karine proposaient également de nouvelles idées pour résoudre les problèmes non résolus, ce qui m'a aidé à progresser de manière structurée et efficace tout au long de mon stage.

4 Algorithme distribué auto-stable

4.1 Définition algorithme distribué auto-stable

Introduction

Le concept d'auto-stabilisation a été introduit par Dijkstra en 1973 dans le contexte des systèmes distribués. Un système distribué est composé d'un ensemble fini de processus autonomes interconnectés par un réseau de communication, avec pour objectif d'atteindre un but global. Le design d'un algorithme distribué auto-stabilisant peut sembler complexe, car chaque processus doit se coordonner avec les autres malgré une vue partielle du système. Chaque processus se base sur son état local et les informations reçues via des supports de communication, généralement asynchrones, le reliant à une partie des autres processus.

Avantages

Tolérance aux fautes :

- L'algorithme converge d'un état illégitime à un état légitime en un nombre fini d'étapes.
- Une fois dans un état légitime, l'algorithme y reste.
- En cas de "transient fault" (faute transitoire) entraînant un retour à un état illégitime,

l'algorithme converge de lui-même vers un état légitime.

Simplicité :

- Utilisation de nombreux petits algorithmes simples au lieu d'un seul gros algorithme complexe.

Définition

Un graphe où chaque nœud contient des variables et une action de la forme 'guard -> statement'. Si la garde (condition) est vraie, alors l'action est "enabled" (activée). Un nœud a accès uniquement aux variables de ses voisins.

Étapes et exécutions

À chaque étape ("step"), on examine la liste des actions "enabled". Différentes manières de procéder existent, ce qui mène au concept de démons.

Démons

Exemples non exhaustifs de démons :

1. **Central :** On active un seul nœud parmi la liste des nœuds et on met à jour la liste des nœuds "enabled".
2. **Synchrone :** On active tous les nœuds de la liste.
3. **Distribué :** On active au moins un nœud parmi ceux "enabled".

Auto-stabilisation
L'auto-stabilisation repose sur trois propriétés :

1. **Clôture (Closure) :** À partir d'un état légitime, on reste dans un état légitime.
2. **Convergence :** À partir d'un état illégitime, on atteint un état légitime en un nombre fini d'étapes.
3. **Correction (Correctness) :** Assure que l'algorithme fonctionne comme prévu et maintient les deux propriétés précédentes.

4.2 Exemple 1 : Coloring

4.2.1 Algorithme

L'algorithme de coloration vise à attribuer des couleurs aux nœuds d'un graphe de manière à ce que deux nœuds adjacents n'aient jamais la même couleur.

1. Chaque nœud v a une variable $color(v)$.
2. Chaque nœud v observe les couleurs de ses voisins.
3. Si v détecte qu'il a la même couleur qu'un de ses voisins, il change sa couleur à la première couleur différente des couleurs de ses voisins.

Formellement, l'algorithme peut être écrit comme suit :

- Garde : $\exists u \in voisins(v)$ tel que $color(v) = color(u)$
- Action : $color(v) \leftarrow \min(\{1, 2, 3\} \setminus \{color(u) \mid u \in voisins(v)\})$

4.2.2 Exécution sous démon central

Sous un démon central, à chaque étape, un seul nœud est activé. Voici un exemple simple :

1. Considérons un graphe avec trois nœuds A, B, et C formant un anneau.
2. Initialement, $color(A) = 1$, $color(B) = 1$, et $color(C) = 2$.

Le noeud A et le noeud B sont enabled,

- **Cas 1 :** le démon active le noeud A. A observe que B a la même couleur (1), donc A change sa couleur en 3 (première couleur disponible différente de 1 et 2).
  - Nouvel état : $color(A) = 3$, $color(B) = 1$, $color(C) = 2$.
- **Cas 2 :** Le démon active B. B observe que A a la même couleur (1), donc B change sa couleur en 3 (première couleur disponible différente de 1 et 2).
  - Nouvel état : $color(A) = 1$, $color(B) = 3$, $color(C) = 2$.

L'algorithme a convergé vers un état légitime où tous les nœuds adjacents ont des couleurs différentes.

4.2.3 Exécution sous démon synchrone

Sous un démon synchrone, tous les nœuds activés changent leur couleur en même temps. Voici un exemple montrant pourquoi cela peut échouer :

1. Considérons le même graphe initial avec $color(A) = 1$, $color(B) = 1$, et $color(C) = 2$.
2. Le démon synchrone active tous les nœuds enabled.
   - A et B observent qu'ils ont la même couleur. A et B changent tous les deux leur couleur en 3 (première couleur disponible différente de 1 et 2).
   - Nouvel état : $color(A) = 3$, $color(B) = 3$, $color(C) = 2$.

Ainsi, après une étape synchrone, A et B ont toujours la même couleur, ce qui montre que l'algorithme ne converge pas nécessairement vers un état légitime sous un démon synchrone.

4.3 Exemple 2 : Token Ring Dijkstra

4.3.1 Explication de l'algorithme

4.3.2 Exemple d'exécution simple sous démon central

4.3.3 Étapes d'exécution

5 Explication des outils

Pour pouvoir bien comprendre la preuve présente dans why3-do, il est nécessaire de comprendre les outils et méthodes utilisées.

5.1 Logique de Hoare

La logique de Hoare est un formalisme utilisé 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.

Règles 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

Les SMT (Satisfiability Modulo Theories) solveurs sont des outils puissants utilisés pour vérifier la satisfiabilité d'expressions logiques sous certaines contraintes théoriques, c'est-à-dire, il vérifie qu'il existe une affectation des variables qui rend la formule vraie, en tenant compte de certaines théories de fond (comme l'arithmétique, les tableaux, les bit-vectors, etc.). Les SMT solveurs étendent les capacités des solveurs SAT (Satisfiability), qui vérifient la satisfiabilité de formules en logique propositionnelle. Alors que les solveurs SAT se concentrent sur des expressions booléennes, les SMT solveurs traitent des expressions plus complexes en combinant les solveurs SAT avec des solveurs spécifiques pour diverses théories. Les formules SMT sont souvent converties en une forme équivalente que les solveurs SAT peuvent traiter. Nous allons par la suite utilisé différents solveurs SMT tel que Z3, altErgo et CVC4

5.3 Why3

Why3 est un outil avancé pour la vérification formelle de programmes, permettant de garantir leur correction en utilisant des techniques sophistiquées. Il permet de spécifier des propriétés formelles à l'aide de contrats, incluant préconditions, postconditions, invariants de boucle et variants. Ces spécifications sont utilisées pour vérifier que le code respecte les propriétés définies en générant des obligations de preuve que Why3 tente de prouver automatiquement ou manuellement en utilisant la logique de Hoare.

L'IDE de Why3 simplifie le processus de vérification en offrant des outils pour visualiser et interagir avec les spécifications et les obligations de preuve. Why3 utilise le langage WhyML, inspiré d'OCaml, pour écrire des programmes vérifiables tout en permettant l'extraction de code vers OCaml pour une intégration pratique. Il peut également collaborer avec divers prouveurs SMT comme AltErgo, Z3 et CVC4, ainsi que des assistants de preuve interactifs comme Coq, offrant une grande flexibilité pour la vérification formelle.

6 Why3doTheWayOfHarmoniousDistributedSystemProofs

Nous allons maintenant nous intéresser à l'article why3-do, Dans leur papier ils proposent une implémentation d'un modèle instanciable pour des algorithmes distribués auto-stable ainsi que l'algo du Token Ring de Disjkstra et sa preuve.

6.1 Explication du modèle : modelReadallEnable

Ils commencent par créer le module Config (nommé World dans leur papier) qui représente le graphe de noeud et d'état du système. Les types node et state sont à instancier.

```
module Config
  use int.Int
  use map.Map
  use list.List
  use list.Append
  use list.Mem
  use list.Map as Lmap

  type node
  type state
  type config =  map node state

end
```

Le second module est un modèle de mémoire localement partagé. De la même manière que Config c'est un modèle à instancier, on retrouve les types node et state et différentes fonctions ou prédicats à instancier grâce au mot clé "val" Le prédicat validNd indique si un noeud n est valide, ce prédicat permet de mettre des conditions sur les noeuds case_node et case_state assurent que le résultat est toujours vrai pour un noeud/état donné.

```
module Steps
  use int.Int
  use map.Map
  use list.List
  use list.Mem
```

```
  use list.Append
  use list.Map as Lmap


  type node
  type state
  type config = map node state

  val predicate validNd (n:node)

  val predicate case_node (node)
    ensures { result }

  val predicate case_state (state)
    ensures { result }
```

On remarque un premier problème dans le modèle de why3-do est que l'état initial est fixé et doit être instancié, ce qui est en contradiction avec le principe l'algorithme stabilisant auto-stable. indpred est un prédicat inductif qui va faire office d'invariant dans la preuve des programmes instanciés, de plus ce prédicat doit être vrai pour la configuration initial.

```
val function initState (node) : state

constant initConfig : config = initState

val ghost predicate indpred (c : config)
  ensures { c=initConfig -> result }
```

Le prédicat enabled est la condition pour laquelle le noeud est enabled. let ghost function step_enbld (c: config) (n: node) (s: state): config: met à jour la configuration c en remplaçant l'état du nœud n par s. handleEnbld est une fonction qui gère un nœud activé dans une configuration, c'est la fonction qui va choisir la nouvelle valeur de l'état pour un noeud en s'assurant la préservation du prédicat inductif après la mise à jour.

```
val ghost predicate enabled (config) (n:node)
  requires { validNd n }

let ghost function step_enbld (c:config) (n:node) (s:state) : config =
  set c n s

val function handleEnbld (n:node) (c : config) : state
  requires { validNd n }
  requires { enabled c n}
  requires { indpred c }
  requires { case_node n }
  ensures  { indpred (step_enbld c n result) }
```

step est un invariant inductif qui décrit la une transition d'une configuration à une autre après l'activation d'un noeud. La ligne step c n (step_enbld c n (handleEnbld n c)) signifie que l'on passe de la configuration c à la configuration dans laquelle le noeud n a été mis à jour. Cela implique qu'il n'y a qu'un seul noeud qui peut être activé au même moment, donc cela fixe le démon comme un démon central.

```
inductive step config node config =
| step_enbld : forall c: config , n :node.
    validNd n ->
      enabled c n ->
step c n (step_enbld c n (handleEnbld n c))
```

Le premier lemme assure que le prédicat inductiv est conservé après un step ce qui est facilement prouvable grâce au ensure de handleEnbld. Le second assure que lorsque un noeud est modifié alors tout les autres n'ont pas changé. step_TR est la fermeture transitive de step :

- Cas de base : Pour toute configuration c peut s'atteindre elle-même en 0 étape.
- Cas inductif : Pour toute configuration c, c', et c" il existe une séquence de steps pour aller de c à c' et il existe une transition unique pour aller de c' à c" via le noeud n.

```
lemma indpred_step :
  forall c c' :config, n :node. step c n c' -> indpred c -> indpred c'

lemma step_preserves_states :
  forall c c' :config, n1 n2 :node. step c n1 c' -> n2<>n1 -> c n2 = c' n2


inductive step_TR config config int =
| base : forall c: config. step_TR c c 0
| step : forall c c' c'' :config, n :node, steps :int.
  step_TR c c' steps -> step c' n c'' -> step_TR c c'' (steps+1)
```

Le lemme noNegative_step_TR assure que le nombre de steps pour passer d'une configuration à une autre est toujours positif. Le prédicat reachable vaut vrai s'il existe un nombre d'étape pour atteindre la configuration c depuis la configuration initiale. indpred_manySteps assure que le prédicat inductif est conservé par plusieurs étapes. indpred_reachable étend le lemme précédent à l'aide de reachable en assurant que le prédicat inductif reste vrai pour toutes les configurations atteignables.

```
lemma noNegative_step_TR : forall c c': config, steps :int.
  step_TR c c' steps -> steps >= 0

predicate reachable (c:config) = exists steps :int. step_TR initConfig c steps

lemma indpred_manySteps :
  forall c c' :config, steps :int . step_TR c c' steps -> indpred c -> indpred c'

lemma indpred_reachable :
  forall c: config. reachable c -> indpred c
```

**

6.2 Explication de selfstab-ring

Nous allons faire une explication rapide de la preuve de l'alogithme de Token Ring de Dijkstra

La preuve est séparé en deux parties, la closure : depuis un état légitime, on reste dans un état légitime puis la terminaison : à partir d'un état non légitime, en un nombre fini d'étape, on atteint un état légitime.

Dans un premier temps ils instancient le modèle en suivant la même implémentation que dans la partie 4.3 Puis définissent has_token, atMostOneToken et atLeastOneToken qui permettent de définir l'invariant indpred. L'invariant est alors : les noeuds sont entre 0 et une borne max, et les état sont entre 0 et une borne max et il y a au moins token. On note également que l'état initial choisi est la racine vaut 1 et les autres noeuds valent 0. L'état initial est donc bien légitime. Puis il suffit d'utiliser de prouver que tout état atteignable depuis l'état initial prouve oneToken.

```
(** {1 Self-stabilizing mutual exclusion on a ring (Closure)} *)
module SelfStab_Ring_Closure

  use int.Int
  use int.EuclideanDivision
  use list.List
  use list.Append
  use list.Mem
  use list.Map as Lmap
  use map.Map

  (* Basic Setup: nodes, packets, inputs, outputs, states *)

  type node = int

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

  let predicate validNd (n:node) = 0 <= n < n_nodes

  axiom n_nodes_bounds : 2 < n_nodes


  type state = int

  val constant k_states : int
```

```
axiom k_states_lower_bound : n_nodes < k_states

let function incre (x:state) : state
= mod (x+1) k_states

let predicate case_node (_node) = true
let predicate case_state (_state) = true

(* clone World theory to get additional types/functions *)
clone modelReadallEnabled.World with
  type node,
  type state


(* System initialization: node states and messages      *)
let function initState (n:node) : state
= if n=n_nodes-1 then 1 else 0

(* defining when a node in the ring has the token *)
predicate has_token (lS:map node state) (i:node) =
(i = 0 /\  lS i =  lS (n_nodes-1))
\/
(i > 0 /\ i < n_nodes /\ lS i <> lS (i-1))

(* enabling predicate *)
let ghost predicate enabled (lS:map node state) (i:node)
= has_token lS i

(* handling function *)
let function handleEnbld (h:node) (lS:map node state) : state
= if h = 0 then incre (lS (n_nodes-1))
   else lS (h-1)

(* helper definitions for invariant predicate *)
let rec ghost predicate atLeastOneToken (lS:map node state) (n:int)
  requires { 0 <= n <= n_nodes }
  ensures { result <-> exists k :int. 0<=k<n /\ has_token lS k }
  variant { n }
= n > 0 && (has_token lS (n-1) || atLeastOneToken lS (n-1))
```

```
  val ghost predicate atMostOneToken (lS:map node state) (n:int)
    requires { 0 <= n <= n_nodes }
    ensures { result <-> forall i j :int. 0<=i<n -> 0<=j<n -> has_token lS i -> has_tok


  (* crucial lemma to achieve an unbounded proof *)
  (* of the atLeastOneTokenLm lemma              *)
  lemma first_last : forall n: int, lS :map node state.
    n >= 0 ->
      (forall j :int. 0<j<=n -> lS j = lS (j-1)) ->
lS 0 = lS n

  lemma atLeastOneTokenLm : forall w :world. atLeastOneToken w n_nodes



  (* candidate invariant predicate *)
  predicate inv (lS:map node state) =
    (forall n :int. 0 <= n < n_nodes -> 0 <= lS n < k_states)
    /\
    atMostOneToken lS n_nodes
    (* /\ *)
    (* atLeastOneToken lS n_nodes *)

  let ghost predicate indpred (w:world) = inv ( w)


  (* Cloning the Steps module will generate VCs to ensure that indpred is an inductive
  clone modelReadallEnabled.Steps with
    type node,
    type state,
    val validNd,
    val case_node,
    val case_state,
    val initState,
    val indpred,
    val enabled,
    val handleEnbld
  (* SYSTEM PROPERTIES TO BE PROVED FROM INVARIANT *)
  predicate oneToken (w:world) = atMostOneToken w n_nodes /\ atLeastOneToken w n_nodes
```

```
  goal oneToken : forall w :world. reachable w -> oneToken w

end
```

Dans la seconde partie le modèle est réinstancié avec des valeurs initiales différentes (pire cas possible ?) L'état initial n'est plus un état légitime et on cherche à atteindre l'état converged qui est défini comme tous les noeuds ont la même valeur. Cet état est un état légitime.

```
predicate initConv (w:world) =
  exists j : int. 0<=j<n_nodes /\ (forall k :int. 0<k<=j -> w k = w 0) /\ noOcc_from w

predicate converged (w :world) =
  forall i :node. validNd i -> w i = w 0
```

Pour prouver qu'en partant d'un état initial, on arrive bien à cet état légitime, la méthode proposée est d'utiliser une fonction de potentiel measureAllNodes, une fonction réccursive qui somme le potentiel de chaque noeud.

On associe une valeur entière positive au graphe en fonction de ses états. On montre que lorsque l'on atteint converged alors la mesure vaut 0. Et que cette mesure décroit bien d'au moins 1 à chaque step.

```
let ghost function measureAllNodes (w:world) : int
  requires { indpred w }
  ensures  { result >= 0 }
  (* ensures  { result = 0 -> converged w } *)
  ensures  { not (converged w) -> result > 0 }   (* equivalent to the previous formulat
= measureNodes w n_nodes
```

Pour faire cela, Wh3do détaillent cas par cas les différentes exécutions possibles comme nous pouvons le voir dans les nombreux ensures de measureDeltaNodes. Cela permet à Why3 de prouver step_decresasesMeasure et converged_oneToken

```
let rec ghost function measureDeltaNodes (w:world) (w':world) (n:int) (k:node) : int
  requires { indpred w }
  requires { 1 <= n <= n_nodes }
  requires { validNd k }
  requires { step w k w' }
  ensures  { result = measureNodes w' n - measureNodes w n }
  ensures  { 0 <= n-1 < k -> result + n_nodes * mod (convState w - convState w') k_stat
  ensures  { n>1 -> k = n-1 -> result + n_nodes * mod (convState w - convState w') k_st
  ensures  { n>0 -> k = 0 -> convState w' <> convState w -> result + n_nodes * (diffZer
  ensures  { n>0 -> k = 0 -> convState w' =  convState w -> result < 0 }
  ensures  { 0 <= k < n-1 -> result < 0 }
  ensures  { n = n_nodes -> result < 0 }
  variant  { n }
= if n=1 then measureDeltaNode w w' 0 k
  else measureDeltaNode w w' (n-1) k + measureDeltaNodes w w' (n-1) k

goal step_decreasesMeasure : forall w w' :world, k :node.
  indpred w -> step w k w' -> measureAllNodes w' < measureAllNodes w


goal converged_oneToken : forall w :world. converged w -> oneToken w /\ has_token w 0
```

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

7.1 Explication détaillée

Nous allons maintenant implémenter l'algorithme de coloration sur un anneau présenté en partie . dans why3 et prouver la closure et la terminaison.

Tout d'abord, nous créons un nouveau module et nous importons les différents bibliothèque dont nous aurons besoin.

- Les listes
- Les entiers
- La division euclidienne pour pouvoir faire des modulos
- Les maps (associations de valeurs d'un type à un autre)

- Les ensembles d'entiers

```
module Coloring_Ring
 use list.List
 use int.Int
 use int.EuclideanDivision
 use map.Map
 use list.Mem
 use set.SetAppInt
```

Dans un premier temps, on souhaite montrer la closure. Pour cela, on commence à définir les objets et règles avec lesquels on veut travailler. On définit n_nodes, une constante entière qui représente le nombre de noeud de notre système. De la même manière on définit k_color qui représente le nombre de couleurs disponibles.

Ici nous avons besoin de règles de base initiale supplémentaire: Nous nous plaçons dans un anneau donc il nous faut au minimum 3 noeuds. Il faut s'assurer que le graphe soit coloriable, pour cela il suffit de s'assurer qu'il y a plus de couleurs disponibles que de noeuds, c'est à dire au moins une couleur par noeud. Pour cela on utilise le mot clé axiom, il faut être très délicat avec ce mot clé car il est facile d'écrire une contradiction.

```
val constant n_nodes : int

axiom at_least_two : 2 < n_nodes

val constant k_color : int

axiom k_color_bound : n_nodes <= k_color
```

Puis on définit les types que l'on va utiliser. On choisit de représenter les noeuds par des entiers. Pour l'état du noeud (sa couleur) on utilise une structure contenant un entier, on pourrait utiliser seulement un entier mais ça permet d'être plus général. Dans d'autres algorithmes on peut avoir des états représentés par deux entiers par exemple. Avoir une structure permet également la possibilité de mettre une condition sur l'état sous forme d'invariant (exemple en commentaire)

On définit alors le type config comme étant le map entre les noeuds et les états.

```
  type node = int

  type state = {k: int}
(*  invariant{ 0 <= k  *)

  type config = map node state
```

Pour donner un exemple d'axiome problèmatique, on peut prendre le suivant: Si on souhaite dire que pour toute configuration c, pour tout noeud n, la couleur de ce noeud dans cette configuration c n est positive. On pourrait croire que c'est une possibilité pour ajouter une condition sur notre configuration mais cet axiom est une contradiction car il existe des configurations (ie des map d'int vers {int}) tel que l'état n'est pas positif. Donc si nous écrivons cet axiom nous considérons vrai une contradiction ce qui permet à why3 de prouver sans souci que 1+1 = 3.

```
axiom max_color : forall c: config. forall n. 0 <= c n
```

Pour ajouter des propriétés, il est préférable de définir des prédicats et de vérifier qu'ils soient vrais. Ici nous ajoutons le prédicats qui signifie qu'un noeud est valide s'il est compris entre 0 et n_nodes le nombre de noeuds

```
let predicate validNd (n:node) = 0 <= n < n_nodes
```

Ici nous nous plaçons dans une topologie en anneau et donc définissons neighbor_set qui renvoie l'ensemble des couleurs des voisins d'un noeud dans un anneau. On assure que un noeud n'est pas son propre voisin et que son cardinal vaut 2 dans cette topologie.

```
let function neighbor_set (i:node): set
  ensures { forall c. mem c result -> c <> i}
  ensures {  cardinal result = 2 }
  =
  add (mod (i-1) n_nodes) (add (mod (i+1) n_nodes) (empty()))
```

La fonction suivante permet de calculer la plus petite valeur non présente dans un ensemble. Grâce à cette fonction nous observons que d'avoir différents solveurs SMT peut être très avantageux, la plupart des solveurs n'arrivent pas à prouver les ensures demandés là où CVC5 met moins de 3 secondes.

```
let function free (colors: set) (k:int) : int =
  requires { cardinal colors < k }
  ensures { not mem result colors }
  ensures { forall x. (0 <= x < k /\ not mem x colors) -> result <= x }
  ensures { 0 <= result < k }
  min_elt (diff (interval 0 k) colors)


  clone modelReadallEnabled.World with
    type node,
    type state
```

Nous devons une fois de plus choisir la valeur initial des états, il semble cohérent de les fixer tous à 0, la plus petite valeur possible.

```
  let function initState (k:node) : state =
  {k=0}

let function color (c:config) (n:node)  : int =
   (c n).k
```

La fonction get_colors récupère les couleurs associées à un ensemble d'éléments nl à partir d'une configuration w. La postcondition ensures { forall c. mem c result <-> exists x. mem x nl /\ c = (w x).k } signifie que pour chaque couleur c, c est un élément du résultat (result) si et seulement si il existe un élément x dans l'ensemble nl tel que c est la couleur associée à x dans la configuration w. Cette postcondition est nécessaire pour prouver la suite du programme

```
let rec function get_colors (nl:set) (w:config) : set =
   requires { cardinal nl < 3 }
```

```
   ensures { cardinal result <= cardinal nl }

   ensures { forall c. mem c result <-> exists x. mem x nl /\ c = (w x).k }

   variant { cardinal nl }
   if is_empty nl then empty () else
    let n = choose nl in
    let c = color w n in
    add c (get_colors (remove n nl) w)
```

Le prédicat conflict est défini tel que pour un noeud, on regarde ses voisins et on vérifie s'il en existe un qui a la même couleur. On définit également couleurDifferentes comme étant la négationd de conflit.

```
predicate conflict (w: world) (n:node) =
   exists c : int.
    mem c (get_colors (neighbor_set n) w)
   /\ (w n).k = c

predicate couleurDifferentes (w: config) (n: node) = not  conflict w n
```

On instancie notre modèle en spécifiant qu'un noeud est enabled quand il a un conflit (au moins un voisin ayant la même couleur que lui). Et la fonction de calcul de la nouvelle couleur (handleEnbld), sera un appel à free. Ici k est la couleur de l'état (par rapport à l'invariant de définition)

```
let ghost predicate enabled (lS:config) (i:node) =
  conflict lS i


let function handleEnbld (n:node) (w:world) : state
  ensures { not mem result.k (get_colors (neighbor_set n) w) }
  =
  { k = free (get_colors (neighbor_set n) w) k_color }
```

Dans notre invariant nous voulons garantir que que les états des noeuds sont valides et donc le reste durant l'exécution du programme. Nous pouvons alors instancier/cloner modelReadallEnabled

```
predicate inv (w: config) =
  (forall n: node. 0 <= n < n_nodes -> 0 <= (w n).k <= k_color)

let ghost predicate indpred (w:world) = inv w


clone modelReadallEnabled.Steps with
    type node, type state, val validNd, val indpred, val enabled, val handleEnbld, val
```

On définit alors le prédicat colored qui vérifie que tout les noeuds sont bien coloriés, puis no_conflict, un prédicat qui renvoie true si pour tout les noeuds il n'y a pas de conflits. On peut alors écrire notre goal : goal_colored qui spécifie que pour tout état atteignable (depuis la configuration initiale) alors s'il n'y a pas de conflit pour la configuration alors le configuration est bien colorié.

```
predicate colored (w: world) =
  forall n1: node.
    validNd n1 ->
    couleurDifferentes w n1

predicate no_conflict (w: world) = forall n: node. not(conflict w n)
  goal goal_colored : forall w:world. reachable w -> no_conflict w -> colored w

end
```

7.2 (*PARTIE 2: TERMINAISON *)

J'ai supprimé ce que j'avais puisque c'était faux.

7.3 Problèmes rencontrés

Lors de la mise en œuvre du programme, plusieurs défis importants ont été identifiés. Tout d'abord, la présence implicite de la valeur true = false

au début du programme a entraîné des erreurs significatives, soulignant l'importance cruciale d'exécuter le détecteur de dysfonctionnements ("smoke detector") en amont pour éviter ces problèmes. En outre, la documentation insuffisante concernant le modèle utilisé dans la seconde partie du projet a compliqué sa compréhension, nécessitant des efforts supplémentaires pour en saisir les subtilités. Par ailleurs, le modèle s'est révélé peu fidèle aux algorithmes distribués stabilisants en raison de plusieurs limitations structurelles : la présence d'un unique démon central, un état initial prédéterminé et une topologie fixe. Ces facteurs ont contribué à une divergence significative entre le modèle théorique et les algorithmes pratiques, ce qui a nécessité une réévaluation approfondie des approches adoptées.

8 Autres différents essais et faisabilité

Dans le cadre de l'exploration des possibilités pour améliorer le modèle, plusieurs angles ont été envisagés mais certains sont restés partiellement développés. Tout d'abord, l'exploration de l'impact d'un état initial varié sur le modèle a été considérée, mais cette approche s'est révélée complexe et difficile à mettre en œuvre. En ce qui concerne la prise en compte de différentes topologies, cet aspect n'a pas été suffisamment approfondi durant le stage, laissant un potentiel non exploité dans ce domaine. De même, la possibilité pour les nœuds d'accéder uniquement à leurs voisins immédiats n'a pas été développée, ce qui aurait pu introduire des contraintes intéressantes dans la dynamique du modèle. Enfin, l'idée de permettre l'activation simultanée de plusieurs nœuds à chaque étape a été envisagée avec l'implémentation d'un démon synchrone. Bien que cette approche n'ait pas encore été testée, cependant le modèle a été soumis à des vérifications de type "smoke test", de plus elle a été validée par la vérification formelle à l'aide de l'outil Why3. Cette validation préliminaire suggère une faisabilité potentielle, mais des tests pratiques seront nécessaires pour confirmer son efficacité et sa robustesse.

9 Conclusion