LaTeX

Polytech Grenoble

INFO 4

Preuves Automatisées pour les Algorithmes Distribués Auto-Stabilisants : Exploration des Méthodes Formelles avec Why3 et SMT

Lilian Derain

Année académique 2023/2024

Contents

1 Remerciements

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

J'ai fait un stage de 3 mois 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 des techniques de preuves basées sur la logique de Hoare et des solveurs SMT. L'article why3-do présente 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 de faire avec ces outils.

I did a 3-month placement at the VERIMAG laboratory. The aim of this placement was to use formal formal methods to prove the correctness of self-stabilising distributed self-stabilising distributed algorithms. The idea of this internship is to explore proof techniques based on Hoare logic and SMT solvers. The article why3-do presents a model for proving self-stabilising distributed algorithms based on the Why3 tool. The aim of of this internship is to evaluate what can be done with these tools.

3 Contexte

3.1 Verimag

Le laboratoire VERIMAG est une unité de recherche en informatique située 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 mathématique. Le laboratoire s'intéresse à de nombreux problèmes tels que les circuits, les processeurs, des algorithmes 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 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

Mon stage avait pour objectif de prendre un article de recherche récent, le comprendre, tester l'implémentation fournie, explorer ses limitations et développer un autre exemple que celui fourni.

Au début de mon stage, j'ai commencé par lire le livre Introduction to Distributed Self-Stabilizing Algorithms pour comprendre le domaine des algorithmes auto-stabilisants, en particulier ceux liés à la circulation dans un anneau et au coloriage et j'ai également consacré du temps à installer et configurer les outils Why3 et Why3-do.

Pour pouvoir comprendre l'article Why3-do, j'ai du acquérir des connaissances sur la logique de Hoare et les SMT solveurs qui sont des composantes essentielles à Why3. De plus j'ai également étudié le fonctionnement des algorithmes distribués auto-stabilisants pour pouvoir com-

prendre leurs implémentations dans Why3-do. 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.

En parallèle, j'ai travaillé avec un dépôt Git pour versionner mon code et mes documents. Cela m'a permis de suivre mes modifications de manière précise, de revenir en arrière en cas de besoin, et de collaborer plus efficacement avec Erwan et Karine en partageant facilement mes avancées et en intégrant leurs retours dans mon travail.

4 Algorithme distribué auto-stable

4.1 Définitions

Introduction

Le concept d'auto-stabilisation a été introduit par Dijkstra en 1973 (à mettre en cible) 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. La conception 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:

- 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 faute transitoire entraînant un retour à un état illégitime, l'algorithme converge de lui-même vers un état légitime.

Pour raisonner sur ces algorithmes distribués Dijkstra a proposé un modèle d'execution dit modèle atomique à état (ASM) :

Celui-ci est constitué d'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 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 C'est le concept de démon qui permet de modéliser le non-déterminisme du système. Exemples non exhaustifs de démons :

1. **Central :** On active un seul nœud de la liste des nœuds enabled.
2. **Synchrone :** On active tous les nœuds de la liste des nœuds enabled.

3. **Distribué :** On active un nœud ou plus de la liste des nœuds enabled.

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

1. **Clôture :** 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 (steps).
3. **Correction :** Assure que l'algorithme fonctionne comme prévu et conformément à sa spécification quand il est dans un état légitime.

4.2 Exemple 1 :Anneau à jeton de Dijkstra (Token Ring Dijkstra)

Le problème du Token Ring introduit par Dijkstra est une illustration classique d'un algorithme distribué auto-stabilisant. Il s'agit de garantir la circulation continue d'un unique jeton (token) dans un réseau en anneau, permettant ainsi une coordination ordonnée entre les nœuds du réseau.

4.2.1 Description de l'algorithme

L'objectif principal est de faire en sorte qu'un seul jeton circule indéfiniment dans un réseau en anneau de n nœuds (avec n $>$ 3). Ce jeton permet la synchronisation et la gestion des ressources entre les différents nœuds du réseau.

4.2.2 État initial et variables

- Anneau : Le réseau est structuré sous la forme d'un anneau de n nœuds.
- Racine : Le premier nœud de l'anneau est désigné comme la racine.
- État des nœuds : Chaque nœud v possède une variable d'état state(v),

qui est une valeur entière comprise entre 0 et K (où K est une valeur supérieure au nombre de nœuds).

4.2.3 Prédicat Token

Chaque nœud possède également un prédicat nommé Token, défini comme suit :

- Pour la racine : Token(root) est vrai si state(root) != state(predecessor(root)).
- Pour les autres nœuds : Token(v) est vrai si state(v) = state(predecessor(v)).

4.2.4 Activation des nœuds

Un nœud est dit activable (enabled) lorsque son prédicat Token est vrai. Lorsque cela se produit :

- La racine prend la valeur de l'état de son prédécesseur.
- Pour les autres nœuds, leur état est incrémenté de 1 modulo K.

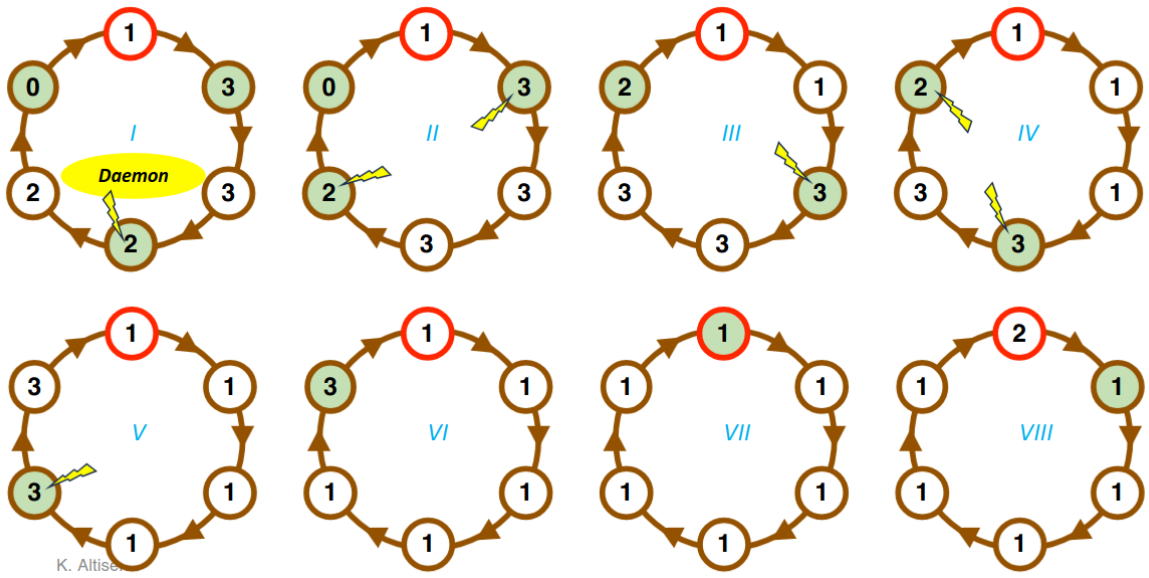

Figure 1: Exemple d'execution pour un démon distribué

4.2.5 Exemple d'exécution simple sous un démon distribué

Ici nous avons un exemple d'execution sous un démon distribué, (au moins un noeud activé à chaque step).

Les noeuds colorés représentent les noeuds activables et les noeuds avec un éclair correspondent aux noeuds activés. Pour la première étape, on remarque que trois noeuds sont activables mais seulement un est activé. La valeur du noeud activé est alors mis à jour. On remarque qu'à partir de l'étape 5, on a atteint un état légitime avec seulement un seul token (un seul noeud activable à chaque tour).

4.3 Exemple 2 : Coloring

4.3.1 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 pas la même couleur.

1. Chaque nœud v a un état $color(v)$ qui est représenté par un entier positif.
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 $color(v) = color(u)$

où voisins(v) est la liste des voisins du noeud v et color(u) est la couleur du noeud u.

- Action : $color(v) \leftarrow \min(\{n \in N*\} \setminus \{color(u) \mid u \in voisins(v)\})$

4.3.2 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 activables. Le démon choisit alors quel noeud sera activé :

- **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).
  - Nouvelle configuration : $color(A) = 3$, $color(B) = 1$, $color(C) = 2$.
- **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).
  - Nouvelle configuration : $color(A) = 1$, $color(B) = 3$, $color(C) = 2$.

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

4.3.3 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 ne pas converger :

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 tout les deux la couleur 3. On a de nouveaux les noeuds A et B enabled car ils ont tous les deux la couleur 3. On retrouve la configuration initiale à un rennomage près, ce qui montre que l'algorithme ne converge pas nécessairement vers un état légitime sous un démon synchrone.

5 Explication des outils

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

5.1 Logique de Hoare

5.1.1 Définition

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

- P est la précondition, une assertion sur l'état des variables 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 des variables du programme après l'exécution de C.

Un triplet 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} ⇒ {P} if (b) then C1 else C2 {Q}.
4. **Règle de la boucle** : Pour une boucle while (b) do C,
   - {I ∧ b} C {I} ⇒ {I} while (b) do C {I ∧ ¬b}, où I est un invariant de boucle.

5.1.2 Exemple

Considérons la fonction free utilisée dans notre implémentation du coloring dans l'outil why3 en 7.1 Cette fonction renvoie le plus petit entier qui n'est pas présent dans un ensemble :

```
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)
```

Dans le langage de Why3, whyMl, les préconditions sont annotées le mot-clef `requires` et les postconditions par le mot-clef `ensures`.

Lors de l'appel d'une précondition ou postcondition, l'outil Why3 va générer pour chaque appel de cette fontion, une condition de vérification (VC) qui sera soumise à un solveur SMT.

Préconditions (requires) `requires { cardinal colors < k }` stipule que la taille (cardinal) de l'ensemble `colors` doit être strictement inférieure à `k`. Cela signifie qu'il existe au moins un entier dans l'intervalle `[0, k)` qui n'est pas présent dans l'ensemble `colors`. En d'autres termes, l'ensemble `colors` ne peut pas contenir tous les entiers de 0 à k.

Postconditions (ensures) `ensures { not mem result colors }` Cette postcondition garantit que le résultat (l'entier retourné par la fonction `free`) ne fait pas partie de l'ensemble `colors`. Autrement dit, `result` est un entier qui n'est pas présent dans l'ensemble `colors`.

`ensures { forall x. (0 <= x < k /\ not mem x colors) -> result <= x }` Cette postcondition assure que le résultat est le plus petit entier possible qui n'est pas dans `colors` et qui se trouve dans l'intervalle [0, k). Pour tout entier `x` dans l'intervalle [0, k) qui n'est pas dans `colors`, le résultat doit être inférieur ou égal à `x`. Cela garantit que la fonction retourne le plus petit entier manquant.

`ensures { 0 <= result < k }` Cette postcondition garantit que le résultat se situe dans l'intervalle `[0, k)`. Le résultat doit être un entier non négatif et strictement inférieur à `k`.

Détails de l'implémentation `min_elt (diff (interval 0 k) colors)` Cette expression utilise des opérations sur les ensembles pour trouver le résultat.

- `interval 0 k` représente l'ensemble des entiers de 0 à k-1.
- `diff (interval 0 k) colors` représente l'ensemble des entiers de 0 à k-1 qui ne sont pas dans `colors`.
- `min_elt` trouve le plus petit élément de cet ensemble, qui est le plus petit entier manquant dans `colors`.

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 utilisant des théories (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 utiliser différents solveurs SMT tel que Z3, altErgo et CVC4

5.3 Why3

Why3 est un outil pour la vérification formelle basé sur la logique de Hoare, 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, basé sur Ml, pour écrire des programmes vérifiables tout en permettant l'extraction de code vers OCaml. 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 Why3 do The Way Of Harmonious Distributed System Proofs

Maintenant nous avons connaissance des différents outils nous pouvons aborder why3-do. Dans cet article, les auteurs 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.

Tout d'abord, il est important de comprendre que leur modélisation est séparée en deux parties distinctes. La première partie est un modèle instanciable qui est la partie commune aux algorithmes distribués auto-stabilisants. La seconde est propre aux spécifités de l'algorithme et de la topologie dans laquelle celui-ci est appliqué.

6.1 Explication du modèle : modelReadallEnabled

Nous commençons par décrire le modèle nommé modelReadallEnabled. Celui-ci est composé de deux modules Config et Steps.

6.1.1 Module Config

Dans le module Config (nommé World dans leur papier), on définit les différents types qui seront utilisés dans l'algorithme. Nous aurons alors le type node, qui représente les noeuds du graphe et le type state qui représente les états que peut prendre un noeud. On définit alors le type Config comme étant une fonction associant les noeuds à leurs états (map). Les types node et state ne sont pas définis dans le modèle car ils dépendent de l'algorithme. Lorsque l'on décrira l'algorithme il faudra spécifier leurs types, puis instancier le module Config avec ces types à l'aide de mot clé Clone.

```
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
```

6.1.2 Module Steps

Le second module est une formalisation en whyml du modèle atomique à état. 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-clef `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 qui est que l'état initial est fixé et doit être instancié, ce qui est une limitation très restrictive pour un algorithme distribué auto-stable; on voudrait pouvoir faire des preuves pour toutes configuartions initiales.

Dans le code whyml ci-dessous `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 initiale.

On remarque l'utilisation du mot-clef `ghost`. Cette notation indique que le prédicat ici est seulement présent dans les spécifications et non dans le code exécutable. Cela permet d'ajouter des informations pour faciliter la preuve de programme.

```
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 activable.

`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 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 mise 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 lemme `indpred_step` assure que le prédicat `inductive` est conservé après un `step` ce qui est facilement prouvable grâce aux postconditions de `handleEnbld`. Le lemme `step_preserves_states` 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'étapes 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'algorithme de Token Ring de Dijkstra.

La preuve est séparée en deux parties, la cloture : 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.

6.2.1 Cloture

Pour définir l'algorithme, ils vont implémenter les différents types, prédicats et fonctions du modèle à instancier, c'est à dire à chaque fois qu'il y avait le mot-clef `val`.

Dans un premier temps ils choisissent de définir les noeuds et les états comme des entiers. Ils peuvent alors définir le type `config` en clonant le module `Config` du modèle.

```
type node = int
type state = int


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

Puis ils définissent un noeud valide comme un entier compris entre `0` et `n_nodes`. 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 ils définissent différentes fonctions et prédicats pour pouvoir instancier leur modèle.

Une fois que tout est définit, il suffit d'utiliser de prouver que tout état atteignable depuis l'état initial prouve `oneToken`.

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



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

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

```
  let function initState (n:node) : state
  = if n=n_nodes-1 then 1 else 0

  predicate has_token (c:config) (n:node) =
  (n = 0 /\  c n =  c (n_nodes-1))
  \/
  (n > 0 /\ n < n_nodes /\ c n <> c (n-1))

  let ghost predicate enabled (c : config) (n:node)
  = has_token c n

  let function handleEnbld (n:node) (c:config) : state
  = if n = 0 then incre (c (n_nodes-1))
             else c (n-1)

  predicate inv (c:config) =
    (forall n :int. 0 <= n < n_nodes -> 0 <= c n < k_states)
    /\
    atMostOneToken c n_nodes

  let ghost predicate indpred (c:config) = inv (c)


  (* Cloning the Steps module will generate VCs to ensure that indpred is an
      inductive invariant *)
  clone modelReadallEnabled.Steps with
    type node,
    type state,
    val validNd,
    val case_node,
    val case_state,
    val initState,
    val indpred,
    val enabled,
    val handleEnbld


  predicate oneToken (w:world) = atMostOneToken w n_nodes /\ atLeastOneToken w
      n_nodes

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

end
```

6.2.2 Terminaison

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

```
  predicate initConv (c:config) =
    exists j : int. 0<=j<n_nodes /\ (forall k :int. 0<k<=j -> c k = c 0) /\ noOcc_from
      c (c 0) (j+1)

  predicate converged (c:config) =
    forall n :node. validNd n -> c n = c 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 récursive 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
        formulation *)
= measureNodes w n_nodes
```

Pour faire cela, Why3do détaille 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` (la mesure de la configuration décroit lorsque l'on fait un step) et `converged_oneToken` (lorsque la mesure vaut 0, alors il y a un seul token et il se trouve à la racine).

```
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_states = 0 }
  ensures  { n>1 -> k = n-1 -> result + n_nodes * mod (convState w - convState w')
      k_states = n-1-n_nodes }
  ensures  { n>0 -> k = 0 -> convState w' <> convState w -> result + n_nodes * (
      diffZero w (convState w)) <= 0 }
  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 why3-do

Nous allons maintenant montrer comment j'ai implémenté et prouvé l'algorithme de coloration en nous basant sur le modèle proposé par why3-do. Nous allons suivre les méthodes présentées dans le chapitre précédent pour cette implémentation.

7.1 Cloture

Pour la cloture, notre objectif est d'instancier le modèle.

Pour cela, nous créons un nouveau module et nous importons les différentes bibliothèques 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 cloture. 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.

Nous nous plaçons dans un anneau, 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_three : 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 définit alors le type `config` comme étant le map entre les noeuds et les états.

```
type node = int

type state = {k: int}

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

Ici nous ajoutons le prédicat 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
```

Le prédicat is_neighbor vérifie si deux nœuds i et j sont voisins dans un anneau de nœuds, on préfère détailler chaque cas pour ne pas utiliser l'arithmétique modulaire qui peut poser problème pour les prouveurs SMT.

```
predicate is_neighbor (i : int) (j : int) =
  validNd i -> validNd j ->
   if i = 0          then j=1 \/ j=n_nodes-1 else
   if i = n_nodes-1 then j=0 \/ j=n_nodes-2 else
   j = i-1 \/ j = i+1

predicate is_neighbor_in_a_ring (i:int) (j:int) = is_neighbor i j
```

Le lemme `is_neigbhor_symetric` affirme que le prédicat `is_neighbor` est symétrique. En d'autres termes, si i est un voisin de j, alors j est aussi un voisin de i. `not_its_own_neighbor` stipule qu'aucun nœud n'est son propre voisin. Si i est un nœud valide, il ne peut pas être voisin de lui-même. Ces deux lemmes sont nécessaires pour prouver les clauses de la prochaine fonction `neighbor_set`.

```
lemma is_neighbor_symmetric: forall i j. is_neighbor i j = is_neighbor j i

lemma not_its_own_neighbor : forall i.  validNd i -> not (is_neighbor i i)
```

La fonction `neighbor_set` prend un nœud i en entrée et retourne l'ensemble de ses voisins.

Les voisins du nœud i sont calculés de la manière suivante : next est le nœud immédiatement après i, calculé comme `(i+1) mod n_nodes`. prev est le nœud immédiatement avant i, calculé comme `(i-1) mod n_nodes`. L'ensemble des voisins est ensuite créé en ajoutant `next` et prev˜ à un ensemble vide.

Clauses `ensures` (Conditions que la fonction garantit après son exécution) :

Les deux premières clauses : Les nœuds prev et next sont bien des voisins de i.

Clause suivante : Aucun nœud n'est son propre voisin (c'est-à-dire que i n'est pas dans le résultat).

Clause de cardinalité : L'ensemble des voisins a exactement 2 éléments, ce qui est spécifique à la structure en anneau où chaque nœud a exactement deux voisins.

Clause de validité : Tous les nœuds dans l'ensemble résultant sont des nœuds valides.

Clauses de voisinage : Chaque nœud dans l'ensemble résultant est un voisin de i, et chaque voisin valide de i est dans l'ensemble résultant.

```
let function neighbor_set (i:node): set
  requires { validNd i }

  ensures { is_neighbor (mod (i-1) n_nodes) i }
  ensures { is_neighbor (mod (i+1) n_nodes) i }
  ensures { forall c. mem c result -> c <> i}
  ensures {  cardinal result = 2 }
  ensures { forall j. mem j result -> validNd j }
  ensures { forall j. mem j result -> is_neighbor  i j }
  ensures { forall j. validNd j -> is_neighbor i j -> mem j result }
  =
  let next = mod (i+1) n_nodes in
  let prev = mod (i-1) n_nodes in
  add next (add prev (empty()))
```

La fonction `free` prend en entrée un ensemble `colors` de valeurs entières et un entier `k`. Elle retourne la plus petite valeur entière qui n'est pas présente dans l'ensemble colors et qui se trouve dans l'interval `[0, k-1]`.

```
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)
```

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 (n: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 c. La postcondition `ensures { forall coul.  mem coul result <-> exists x.  mem x nl /\ coul = (c x).k }` signifie que pour chaque couleur `coul`, `coul` est un élément du résultat (`result`) si et seulement si il existe un élément `x` dans l'ensemble `nl` tel que `coul` est la couleur associée à `x` dans la configuration `c`. Cette postcondition est nécessaire pour prouver la suite du programme.

```
let rec function get_colors (nl:set) (c:config) : set =
   requires { cardinal nl < 3 }
   ensures { cardinal result <= cardinal nl }
   ensures { forall coul. mem coul result <-> exists x. mem x nl /\ coul = (c x).k }
   variant { cardinal nl }
   if is_empty nl then empty () else
    let n = choose nl in
    let coul = color c n in
    add coul (get_colors (remove n nl) c)
```

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.

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

On instancie notre modèle en spécifiant qu'un noeud est activalbe 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 predicate enabled (c:config) (n:node)
   ensures { validNd n -> result = conflict c n }
   =  validNd n &&  mem (c n).k (get_colors (neighbor_set n) c)


  let function handleEnbld (n:node) (c:config) : state
    requires { validNd n }
    ensures { not mem result.k (get_colors (neighbor_set n) c) }
    ensures { forall i. mem i (neighbor_set n) -> not conflict c i -> not conflict (
        set c n result) i }
    =
    { k = free (get_colors (neighbor_set n) c) 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 (c: config) =
  (forall n: node. 0 <= n < n_nodes -> 0 <= (c n).k <= k_color)

let ghost predicate indpred (c:config) = inv c


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

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) s'il n'y a pas de conflit pour la configuration alors le configuration est bien coloriée.

```
predicate colored (c:config) =
  forall n1: node.
    validNd n1 ->
    couleurDifferentes c n1

predicate no_conflict (c: config) = forall n: node. not(conflict c n)

goal goal_colored : forall c:config. reachable c -> no_conflict c -> colored c
```

```
end
```

7.2 Terminaison

Nous allons maintenant montrer la terminaison de l'algorithme. Pour cela, nous allons utiliser une fonction de potentiel. Cette fonction permettra d'évaluer la valeur d'une configuration.

Nous commençons alors par définir le potentiel d'un noeud comme étant 1 s'il y a un conflit, c'est à dire un noeud de la même couleur parmi ses voisins.

```
let ghost function potential_node (c:config) (n:int) : int
  ensures { 0 <= result <= 1 }
  ensures { result = 1 <-> conflict c n }
  =
  if conflict c n then 1 else 0
```

Puis nous définissons la fonction de potentiel sur la configuration entière en faisant la somme de tout ses noeuds.

La fonction récursive `potential_i` calcule le potentiel de conflit pour les `i` premiers nœuds d'une configuration donnée `c`. Elle prend en entrée cette configuration ainsi qu'un entier `i`, représentant un index de nœud.

La clause `requires` garantit que l'index `i` correspond bien à un nœud valide. Le résultat renvoyé par la fonction est toujours non négatif. Si tous les nœuds de `0` à `i` ne présentent aucun conflit (c'est-à-dire que la fonction `potential_node c j` renvoie 0 pour chaque `j` dans cet interval), alors le résultat sera 0 ; sinon, il sera strictement supérieur à 0.

La fonction commence par calculer le potentiel de conflit pour le nœud `i` actuel (`pot_i`), puis ajoute ce potentiel à celui des nœuds précédents de manière récursive.

La clause `variant { i }` assure que la récursion terminera correctement, car l'index `i` diminue à chaque appel récursif.

La fonction potential est juste un appel de `potential_i` pour tous les noeuds, en gardant les mêmes clauses.

```
let rec ghost function potential_i (c:config) (i:int) : int
    requires { validNd i }
    ensures  { result >= 0 }
    ensures  { result = 0 <-> forall j :int. 0<=j<=i -> potential_node c j = 0 }
    variant  { i }
  =
       let pot_i = if conflict c i then 1 else 0 in
       if i=0 then pot_i
              else pot_i + potential_i c (i-1)


let ghost function potential (c:config) : int
    ensures  { result >= 0 }
    ensures  {  exists n:node. conflict c n -> (potential_node c n = 1) -> (result>
        0)}
    ensures  { result = 0 <-> forall i :int. 0 <= i < n_nodes -> potential_node c i =
         0 }

  = potential_i c (n_nodes-1)
```

Nous pouvons maintenant écrire quelques lemmes qui seront nécessaire pour faire les preuves suivantes.

- `step_on_n_preserves_other_nodes` affirme que lorsqu'une étape de transformation (`step`) est effectuée sur un nœud `n` dans la configuration `c`,

la valeur de tous les autres nœuds (`i`) reste inchangée dans la nouvelle configuration `c'` si `i` n'est pas égal à `n`.

- `no_conflict_after_step` affirme que si une étape de transformation (`step`) est effectuée sur un nœud `k` dans la configuration `c`,

alors le nœud `k` ne sera plus en conflit dans la nouvelle configuration `c'`.

- `step_influences_neighborhood_only` affirme que lorsqu'une étape de transformation (`step`) est effectuée sur un nœud `n` dans la configuration `c`,

seuls les nœuds voisins ou le nœud lui-même peuvent voir leur valeur modifiée dans la nouvelle configuration `c'`, tandis que tous les autres nœuds restent

```
lemma step_on_n_preserves_other_nodes :
    forall c c' :config, n i :node. step c n c' -> i<>n -> c i = c' i


lemma no_conflict_after_step: forall c, c', k. step c k c' -> not(conflict c' k)

lemma step_influences_neighborhood_only: forall c c' n i.
  step c n c' ->
     not is_neighbor i n /\ i <> n -> c i = c' i
```

Nous voulons maintenant montrer que cette fonction de potentiel décroît à chaque fois qu'un nœud est activé. Pour cela, nous définissons la fonction `deltaPotAfterStep_i`.

Explication :

- Cette fonction récursive calcule la différence de potentiel de conflit entre deux configurations, `c` et `c'`,

pour les `i` premiers nœuds après l'application d'une étape de transformation (`step`) sur un nœud spécifique `n`.

- Paramètres :
  - `c` et `c'` : Les configurations avant et après l'étape de transformation.
  - `i` : L'index du nœud jusqu'auquel on calcule la variation de potentiel.
  - `n` : Le nœud sur lequel la transformation (`step`) a été appliquée.
  - `pos` : Un booléen indiquant si l'index `i` a déjà atteint ou dépassé le nœud `n`.
  - `delta` : La somme partielle de la variation de potentiel accumulée jusqu'à `i`.
- Corps de la fonction :
  - La fonction commence par calculer la différence de potentiel pour le nœud `i` (c'est-à-dire la différence entre `potential_node c i` et `potential_node c' i`).
  - Si `i` est égal à 0, la fonction retourne cette différence.
  - Sinon, la fonction additionne cette différence à celle des nœuds précédents (calculée récursivement en appelant `deltaPotAfterStep_i` avec `i-1`).
- Clauses ensures :
  - Si `i` est égal à `n`, la différence doit être positive.
  - Si `pos` est faux (c'est-à-dire si `i` n'a pas encore atteint `n`), la différence doit être positive.

– Le résultat est non négatif et correspond à la différence de potentiel pour les `i` premiers nœuds entre `c` et `c'`.

La fonction `deltaPotAfterStep` est un appel à `deltaPotAfterStep_i` pour calculer la différence de potentiel pour tous les nœuds d'une configuration.

```
let rec ghost function deltaPotAfterStep_i (c:config) (c':config) (i:int) (n:node) (
    pos : bool) (delta:int) : int
    requires { validNd i }
    requires { validNd n }
    requires { step c n c' }
    requires { pos <-> exists j. i<j<n_nodes /\ j = n }

    ensures { i = n -> result > 0 }
    ensures  { not pos -> result > 0 }
    ensures  { result >= 0 }
    ensures  { result = potential_i c i - potential_i c' i  }
    variant  { i }
  =
    let delta_i_n = potential_node c i - potential_node c' i in
    if i = 0 then delta_i_n
             else delta_i_n + deltaPotAfterStep_i c c' (i-1) n (pos || i=n ) (delta+
                 delta_i_n)

let ghost function deltaPotAfterStep (c:config) (c':config) (k:node) : int
   requires { validNd k }
   requires { step c k c' }
    ensures { result > 0 }
    ensures { result = potential c - potential c'}
  = deltaPotAfterStep_i c c' (n_nodes-1) k false 0
```

- `the_potential_decreases_after_a_step` affirme que si une étape de transformation (`step`) est effectuée sur un nœud valide `n`

dans une configuration `c`, alors le potentiel total de conflit de la configuration `c'` après la transformation sera strictement inférieur à celui de la configuration initiale `c`.

- `config_potential_is_null_iff_all_states_one_are_null` établit une équivalence entre le potentiel total d'une configuration et les potentiels de chaque nœud.

Il affirme que le potentiel total d'une configuration `c` est nul si et seulement si tous les nœuds ont un potentiel nul (c'est-à-dire qu'aucun nœud n'est en conflit).

```
lemma the_potential_decreases_after_a_step : forall c c' : config, n :node.
  validNd n -> step c n c' -> potential c > potential c'

lemma config_potential_is_null_iff_all_states_one_are_null :
  forall c : config. (potential c = 0) <-> (forall n: node. 0 <= n < n_nodes ->
      potential_node  c n = 0)
```

Il ne reste plus qu'à créer les prédicats `no_conflict` et `colored` pour pouvoir écrire nos 'goal'.

- `no_conflict` indique qu'il n'y a aucun conflit dans la configuration `c`.
- `colored` indique que la configuration `c` est correctement colorée, ce qui signifie que tous les nœuds valides n'ont aucun conflit.

```
predicate no_conflict (c: config) = forall n: node. not(conflict c n)

predicate colored (c: config) =
  forall n: node.
    validNd n -> not conflict c n
```

`goal_colored` affirme que pour toute configuration `c` atteignable à partir d'une configuration initiale `c_init`, si `c` n'a aucun conflit (`no_conflict c`), alors `c` est correctement colorée (`colored c`).

`goal_converged` affirme que pour toute configuration `c`, le potentiel de `c` est nul si et seulement si `c` est correctement colorée.

```
goal goal_colored : forall c, c_init :config. reachable c -> no_conflict c -> colored
     c

goal goal_converged : forall c : config. (potential c = 0) <-> colored c
```

Nous avons alors la convergence puisque nous avons prouvé que la fonction de potentiel est positive, elle décroît à chaque `step`, et quand elle est nulle, nous avons atteint la configuration voulue.

7.3 Quelques remarques

7.3.1 Différents solveurs SMT

Nous observons que d'avoir différents solveurs SMT peut être très avantageux, la plupart des solveurs n'arrivent pas à prouver La fonction `free` définit en 7.1 là où CVC5 met moins de 3 secondes.

7.3.2 Topologie

Pour essayer de limiter l'impact de la topologie dans l'algorithme nous n'avons que neighbor_set qui dépends de la topologie en anneau, alors que dans leur implémentation du token ring, celle-ci était présente à de nombreux endroits

7.4 Problèmes rencontrés

Lors de la mise en œuvre du programme, plusieurs défis importants ont été identifiés. Tout d'abord, la présence implicite d'axiomes faux 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. Par ailleurs, le modèle s'est révélé trop restrictif par rapport aux algorithmes distribués autostabilisants en raison de plusieurs limitations structurelles : la présence d'un unique démon central, une configuration initiale prédéterminée et une topologie fixe.

8 Perspectives

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.