Une approche SAT pour l'analyse de l'autostabilisation d'un unisson asynchrone

Asma Khoualdia, Sami Cherif, Stéphane Devismes, Léo Robert

Laboratoire MIS UR 4290, Université de Picardie Jules Verne, Amiens, France

{prenom.nom}@u-picardie.fr

Résumé

L'unisson asynchrone est un problème de synchronisation d'horloges de référence en systèmes distribués, et notamment en autostabilisation. Il permet notamment de simuler des systèmes synchrones dans des environnements asynchrones, faciliter la détection de terminaison, partager localement des ressources, ou encore pour réaliser des calculs d'infimum. Cet article explore l'autostabilisation d'un algorithme d'unisson asynchrone en utilisant une approche basée sur la satisfiabilité propositionnelle. Nous proposons une modélisation logique de l'exécution asynchrone de l'algorithme capturant les règles de mise à jour des horloges ainsi que la détection de la convergence vers une configuration légitime ou, au contraire, de la divergence par l'existence d'interblocages ou de cycles entre configurations illégitimes.

Mots-clés

Autostabilisation, unisson, asynchronisme, satisfiabilité.

Abstract

Asynchronous unison is a benchmark clock synchronization problem in distributed systems and particularly self-stabilization. It allows to simulate a synchronous system in an asynchronous environment, help the termination detection, achieve local resource allocation, or compute infimum. This article explores the self-stabilization of an asynchronous unison algorithm using a propositional satisfiability-based approach. We propose a logical model of the asynchronous execution of this algorithm capturing the clock update rules, as well as the detection of convergence to a legitimate configuration or, conversely, divergence through the existence of daedlocks or cycles of illegitimate configurations.

Keywords

Self-stabilization, Unison, Asynchronism, Satisfiability.

1 Introduction

Un système distribué est un ensemble d'entités de calcul, ici appelées *processus*, qui sont *autonomes* et *interconnectées* entre elles. Le but de ces processus est de coopérer, via des échanges d'informations, afin de résoudre une tâche globale au système. Dans ce contexte, un algorithme est dit *autostabilisant* si, quel que soit l'état initial du système sur lequel il est déployé, il garantit le retour en temps fini, et sans intervention extérieure, à un état dit *légitime* à partir duquel la spécification du système est satisfaite [16]. Bien que la notion de panne ne soit pas explicite dans la définition des systèmes autostabilisants, la motivation première de cette approche est bien la *tolérance aux pannes*. En effet, après un nombre fini de *pannes transitoires* [1], l'état d'un système distribué peut être quelconque et ainsi ne plus vérifier sa spécification. Un algorithme autostabilisant garantit alors que le système récupère de telles fautes en temps fini.

La conception d'algorithmes autostabilisants peut sembler complexe au premier abord en raison de l'accès limité des processus à l'état global du système. Pourtant, nombre de ces algorithmes se révèlent étonnamment élégants et parfois même plus simples que leurs pendants non stabilisants. Cependant, démontrer formellement l'autostabilisation d'un algorithme s'avère souvent ardu et sujet aux erreurs. Cela est principalement dû à la nature combinatoire du problème : la convergence vers un état légitime doit être démontrée à partir de *tout* état du système.

Dans cette optique, nous nous intéressons ici à un problème simple : l'*unisson asynchrone*. On suppose un système asynchrone où les processus sont anonymes et arbitrairement connectés entre-eux. Chaque processus dispose d'une horloge locale dont les valeurs entières varient entre 0 à $K-1$ où K est appelée la *période*. À partir d'un état où l'horloge de chaque processus a une valeur quelconque de $\{0,\ldots,K-1\}$, le but est de faire converger le système vers un état dans lequel la différence entre les horloges de deux voisins quelconques est d'au plus un incrément (modulo K) à chaque instant. L'unisson est un outil d'algorithmique distribuée fondamental qui possède de nombreuses applications. Entre autres, il peut être utilisé pour simuler des systèmes synchrones dans des environnements asynchrones [15], pour faciliter la détection de terminaison [8], pour partager localement des ressources [10], ou encore pour réaliser des calculs d'infimum [9].

Couvreur *et al.* ont proposé une solution simple à ce problème [13]. L'autostabilisation de leur algorithme est démontrée pour tout $K > n^2$ où n est la taille du réseau. Cependant, l'optimalité de cette borne reste une question

1. C'est-à-dire, des perturbations temporaires de certains composants du système (liens ou processus).

ouverte. Nous proposons ici d'étudier comment la satisfiabilité propositionnelle peut aider à traiter cette question, et plus généralement comment elle peut aider à vérifier l'autostabilisation d'algorithmes distribués asynchrones. En effet, la *satisfiabilité* propositionnelle (SAT) est un problème fondamental en logique et en informatique théorique, consistant à déterminer si une formule booléenne peut être satisfaite par une assignation de valeurs. Ce problème, qui a été le premier démontré NP-complet [12], suscite un intérêt croissant car les solveurs SAT modernes parviennent à traiter efficacement de grandes instances issues d'applications concrètes, malgré la difficulté théorique établie du problème. En plus de son rôle clé en vérification logicielle et matérielle, en intelligence artificielle et en cryptographie, SAT se situe à l'intersection de plusieurs disciplines telles que la logique, la complexité et la programmation par contraintes [4].

Dans [18], nous avons initié l'analyse rigoureuse d'algorithmes autostabilisants déployés dans des réseaux de topologies variées via la satisfiabilité propositionnelle en prenant l'unisson synchrone d'Arora *et al.* [2] comme cas d'étude. Cependant, l'étude proposée se restreint aux systèmes distribués synchrones. Dans cet article, nous proposons une approche formelle basée sur SAT pour analyser rigoureusement le comportement de l'unisson asynchrone [13]. Cet algorithme suit des règles de mise à jour d'horloges distinctes, et son autostabilisation se caractérise par des propriétés différentes. Nous traduisons ces règles en contraintes logiques, permettant l'utilisation de solveurs SAT pour vérifier si la synchronisation est atteinte dans un cadre asynchrone. Cette méthodologie offre un cadre systématique pour étudier l'autostabilisation de l'algorithme dans différentes topologies et configurations initiales et avec différentes périodes.

Le reste de cet article est organisé comme suit. Dans la section 2, nous présentons l'algorithme d'unisson asynchrone et nous introduisons la satisfiabilité propositionnelle. La section 3 est consacrée à la modélisation formelle de l'algorithme de l'unisson asynchrone, où nous détaillons les variables et les contraintes logiques, avec une attention portée à la mise à jour des horloges et à la modélisation des propriétés d'autostabilisation. La section 4 présente les résultats obtenus. Enfin, nous concluons dans la section 5.

2 Préliminaires

2.1 Unisson asynchrone

Nous considérons un système distribué de $n \geq 3$ processus interconnectés. Le réseau d'interconnexion est modélisé par un graphe non-orienté connexe $G = (V, E)$, où V est un ensemble de n sommets représentant les processus et E un ensemble d'arêtes représentant les liens de communication bidirectionnels entre les processus. L'ensemble des voisins d'un processus p est noté $N(p)$. Nous considérons l'unisson asynchrone proposé par Couvreur *et al.* [13]. Cet algorithme est décrit dans le *modèle à états* dans [1]. Il s'agit d'un modèle à mémoires localement partagées : chaque nœud peut directement lire son état et ceux de ses voisins, mais peut uniquement modifier son propre état.

Algorithme 1 Unisson asynchrone : algorithme local pour chaque processus p

Entrées :
$N(p)$: l'ensemble des voisins de p
K : entier strictement positif

Variable :
$p.c \in \{0, \ldots, K-1\}$

Prédicat :
$behind(a, b) = ((b.c - a.c) \mod K) \leq n$

Règles :

$$I(p) :: \forall q \in N(p),\ behind(p, q) \ \rightarrow\ p.c \leftarrow (p.c + 1) \bmod K$$

$$R(p) :: p.c \neq 0 \ \wedge\ (\exists q \in N(p),\ \neg behind(p, q) \wedge \neg behind(q, p)) \ \rightarrow\ p.c \leftarrow 0$$

Étant donné un paramètre entier K commun à tous les processus et appelé *période*, chaque processus p dispose d'une seule variable $p.c \in \{0, \ldots, K-1\}$ qui représente son *horloge*. La *configuration* du système est ainsi modélisée par le vecteur associant à chaque processus la valeur de son horloge. L'ensemble des configurations possibles du système sera noté $\Gamma_{n,K}$. L'algorithme local de chaque processus p est décrit par deux règles de la forme $étiquette :: garde \rightarrow action$. L'étiquette est uniquement utilisée pour identifier la règle dans le raisonnement. La garde d'une règle de p est un prédicat sur l'état de p et ceux de ses voisins. L'action correspond à une mise-à-jour de la valeur d'horloge du processus p. Dans une configuration donnée, la règle d'un processus est dite activable si sa garde est vraie. Par extension, un processus est dit activable lorsqu'une de ses règles est activable. Une configuration où aucune règle n'est activable est dite *terminale*.

Le système est asynchrone : tant que le système n'est pas dans une configuration terminale, des *itérations asynchrones* sont exécutées comme suit. Un adversaire, appelé *démon distribué inéquitable*[2], sélectionne un sous-ensemble non vide de processus activables dans la configuration courante γ_i. Les processus sélectionnés sont dits *activés* et exécutent simultanément l'action de l'une de leurs règles activables, générant ainsi une nouvelle configuration γ_{i+1}, et ainsi de suite. Une exécution de l'algorithme est donc une suite maximale (finie ou infinie) de configurations $e = \gamma_0, \gamma_1, \ldots$ telle que pour tout $i > 0$, γ_i est obtenu à partir de γ_{i-1} en une itération asynchrone de l'algorithme.

Dans l'algorithme 1, le prédicat $behind(p, q)$ est vrai lorsque le processus p est en retard d'au plus n incréments par rapport à q. On dira dans ce cas que p a un *retard admissible* par rapport à q. Ainsi, un processus p est activable pour l'incrémentation ($I(p)$) dans une configuration donnée s'il a un retard admissible avec chacun de ses voisins. Dans ce cas, si le processus est choisi par le démon, il incrémente sa valeur modulo K pour rattraper son retard. D'autre part, un processus p est activable pour la réinitialisation ($R(p)$)

2. Le démon matérialise l'asynchronisme du système.

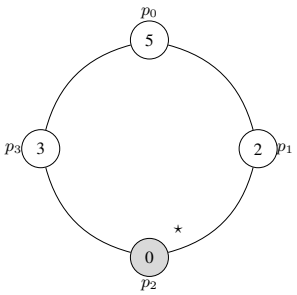


FIGURE 1 – Configuration initiale convergente γ_0 dans un anneau orienté de 4 processus avec une période $K = 9$. Un processus est grisé lorsqu'il est activable et affiche une étoile lorsqu'il est activé.

	p_0	p_1	p_2	p_3
γ_0	5	2	0*	3
γ_1	5	2	1*	3
γ_2	5	2*	2	3
γ_3	5	3	2*	3
γ_4	5	3*	3	3
γ_5	5	4	3*	3
γ_6	5	4*	4	3
γ_7	5	5	4	3*
γ_8	5	5	4	4

TABLE 1 – Valeurs des processus pour chaque configuration. Une case est grisée lorsque son processus est activable et affiche une étoile lorsqu'il est activé.

dans une configuration donnée si sa valeur d'horloge est non nulle et s'il a un voisin q avec qui son horloge n'est pas comparable : p n'a pas de retard admissible avec q et q n'a pas de retard admissible avec p. Dans ce cas, p est activable pour réinitialiser sa valeur d'horloge à 0 afin de sortir de cette situation ambiguë avec q. On note que les deux propriétés d'autorisation de l'algorithme 1 sont localement mutuellement exclusive : lorsqu'une des deux règles est activable pour un processus, l'autre ne l'est pas pour ce même processus. Ainsi, au plus une règle peut être applicable pour un processus à un instant donné.

Dans la suite, nous nous intéressons aux propriétés d'autostabilisation de l'algorithme. En particulier, on parle de convergence, définie formellement ci-dessous, quand on garantit un retour à une configuration légitime de l'algorithme où, pour chaque paire de voisins, la différence entre leurs valeurs d'horloge est d'au plus un incrément. Précisément, dans une configuration légitime, on a $p.c \in \{q.c - 1 \bmod K, q.c, q.c + 1 \bmod K\}$, pour toute paire de voisins p et q. À noter ici que, les horloges étant modulaires, l'écart entre $K - 1$ et 0 n'est que d'un seul incrément. De plus, l'ensemble des configurations légitimes est clos : toute configuration accessible depuis une configuration légitime est également légitime.

L'algorithme est démontré convergent (ou autostabilisant) pour toute période $K > n^2$ [14]. Cependant, lorsque la valeur de K est inférieure ou égale à n^2, l'exécution peut ne jamais stabiliser. On parle alors de *divergence*, définie formellement ci-dessous, quand un *cycle de configurations*

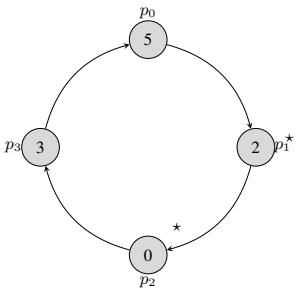


FIGURE 2 – Configuration initiale divergente γ_0 par l'existence d'un cycle (de configurations illégitimes) dans un anneau de 4 processus avec $K = 6$. Un processus est grisé lorsqu'il est activable et affiche une étoile lorsqu'il est activé.

	p_0	p_1	p_2	p_3
γ_0	5	2*	0*	3
γ_1	5*	3*	1*	3*
γ_2	0*	4*	2	4*
γ_3	1	5	2*	5
γ_4	1*	5	3	5
γ_5	2	5*	3	5*
γ_6	2*	0	3*	0*
γ_7	3*	0*	4*	1*
γ_8	4*	1*	5*	2*
γ_9	5	2	0	3

TABLE 2 – Valeurs des processus pour chaque configuration avec quatre processus. Une case est grisée lorsque le processus est activable et une étoile indique le processus effectivement activé.

illégitimes est exhibé dans un préfixe d'exécutions ou lorsqu'une exécution atteint un *interblocage*, c'est-à-dire, une configuration terminale (illégitime). Un cycle peut être répété à l'infini et ainsi donner une exécution ne contenant aucune configuration légitime. Une configuration terminale viole la spécification de l'unisson, qui exige que chaque processus incrémente son horloge infiniment souvent. On illustre ces différents cas de figures dans les exemples 1, 2 et 3.

Définition 1 (Convergence)**.** *L'algorithme de l'unisson asynchrone exécuté sur un graphe non orienté connexe de n processus est dit* convergent *si, à partir de toute configuration initiale, toutes les exécutions possibles atteignent en un nombre fini d'itérations une configuration* légitime *où, pour chaque paire de voisins, la différence entre leurs valeurs d'horloges est au plus un incrément.*

Exemple 1. *Considérons un réseau de processus G=($\{p_0, p_1, p_2, p_3\}, E$) dont la topologie est en anneau. Posons $K = 9$ et supposons la configuration initiale γ_0 donnée dans la Figure 1. À partir de γ_0, on obtient le préfixe d'exécution $\gamma_0, \gamma_1, \ldots, \gamma_8$ donné dans la Table 1. Après la dernière itération, l'algorithme atteint une configuration λ_8 légitime puisqu'en particulier, toutes les valeurs d'horloges ont un écart d'au plus 1.*

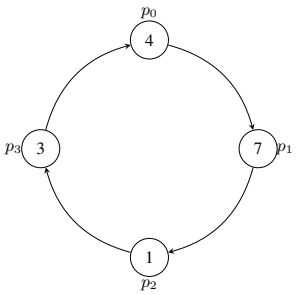


FIGURE 3 – Configuration terminale divergente γ_0 par l'existence d'un interblocage dans un anneau de 4 processus avec une période $K = 9$.

Définition 2 (Divergence)**.** *L'algorithme de l'unisson asynchrone exécuté sur un graphe non orienté connexe de n processus est dit* divergent *s'il existe un interblocage (c'est-à-dire, une configuration terminale) ou une paire de configurations identiques (appelée cycle) dans une exécution asynchrone où, pour (au moins) une paire de voisins, la différence entre leurs valeurs d'horloge dépasse un incrément.*

Exemple 2. *Considérons un réseau de processus $G = (\{p_0, p_1, p_2, p_3\}, E)$ dont la topologie est en anneau. Posons $K = 6$ et supposons la configuration initiale γ_0 donnée dans la Figure 2. À partir de γ_0, on obtient le préfixe d'exécution $\gamma_0, \gamma_1, \ldots, \gamma_9$ détaillé dans la Table 2. Un cycle de configurations illégitimes apparaît à partir de la configuration γ_9.*

Exemple 3. *Considérons un réseau de processus $G = (\{p_0, p_1, p_2, p_3\}, E)$ dont la topologie est en anneau et posons $K = 9$. La configuration terminale γ_0 donnée dans la Figure 3 est divergente par interblockage puisqu'aucun processus ne peut être activé.*

2.2 Satisfiabilité propositionnelle

Soit X un ensemble de variables propositionnelles. Un littéral est une variable $x \in X$ ou sa négation $\overline{x}$. Une clause est une disjonction de littéraux. Une formule en Forme Normale Conjonctive (FNC) est une conjonction de clauses. Une affectation $I : X \to \{vrai, faux\}$ associe à chaque variable une valeur booléenne et peut être représentée comme un ensemble de littéraux. Un littéral l est satisfait par une affectation I si $l \in I$, sinon il est falsifié par I. Une clause est satisfaite par une affectation I si au moins un de ses littéraux est satisfait par I, sinon elle est falsifiée par I. Une formule FNC est satisfiable s'il existe une affectation qui satisfait toutes ses clauses, sinon elle est insatisfiable. Le problème de satisfiabilité (SAT) consiste à déterminer si une formule FNC donnée est satisfiable.
Bien que SAT soit NP-complet [12], les solveurs modernes basés sur l'algorithme CDCL (Conflict Driven Clause Learning) [19] sont efficaces et parviennent à résoudre des instances impliquant un grand nombre de variables et de clauses. En effet, au delà de l'apprentissage de clauses et du retour arrière non chronologique, ces solveurs intègrent des mécanismes puissants comme, par exemple, les structures paresseuses, des heuristiques de branchement dédiées ou encore les redémarrages [7]. Par ailleurs, le problème SAT est largement utilisé pour modéliser et résoudre de nombreux problèmes combinatoires complexes issus de divers domaines, notamment en vérification formelle [5, 11] mais aussi en cryptographie, en bio-informatique et en planification [7].
Dans la suite, on s'intéresse à la modélisation du comportement de l'algorithme d'unisson asynchrone, appliqué à un graphe connecté. L'objectif est de modéliser l'exécution de l'algorithme et de caractériser ses conditions de convergence ou de divergence, à travers des formules logiques pouvant être résolues par un solveur SAT.

3 Modélisation formelle d'un algorithme d'unisson asynchrone

On rappelle que le réseau est modélisé par un graphe non orienté connexe de $n \geq 3$ processus, noté $G = (V, E)$. L'ensemble des processus est défini, sans perte de généralité, par leurs indices $V = \{0, \ldots, n-1\}$. L'ensemble des processus voisins à $p \in V$ dans G est noté $N(p)$. Les valeurs d'horloge de chaque processus sont contenues dans l'intervalle $M = \{0, \ldots, K-1\}$, où K est la période. Nous nous intéressons aux t_f premières configurations d'une exécution quelconque de l'algorithme de l'unisson asynchrone. Ces configurations sont indexées de 0 à $t_f - 1$, et l'ensemble des indices est noté $T = \{0, \ldots, t_f - 1\}$. Dans la suite, nous nous intéressons aux contraintes qui encadrent l'exécution de l'algorithme. Ces contraintes doivent non seulement représenter fidèlement l'évolution du système dans un contexte asynchrone, mais aussi permettre de détecter sa convergence ou sa divergence.

3.1 Modélisation de l'exécution

Dans un premier temps, nous nous intéressons à la formalisation des contraintes modélisant une exécution valide de l'unisson asynchrone. Plus précisément, nous décrivons les variables booléennes permettant de raisonner sur l'évolution du système et nous décrivons les contraintes permettant de représenter les règles de mise à jour, assurant ainsi l'évolution correcte des horloges au fil des itérations.

3.1.1 Sur l'unicité des horloges

Pour représenter l'évolution du réseau au fil des itérations de l'algorithme, nous introduisons les **variables d'horloge** $clk_{p,t,v}$, où $p \in V$, $t \in T$ et $v \in M$, qui sont affectées à $vrai$ si le processus p possède la valeur v dans la configuration t. Ces variables permettent de représenter l'état du système à chaque étape de l'exécution. Ainsi, pour garantir une exécution valide de l'algorithme, il est nécessaire de maintenir l'unicité de l'horloge de chaque processus à chaque itération. Précisément, il faut veiller à ce que chaque processus possède une et une seule valeur d'horloge dans chaque configuration du système. Cette contrainte peut être modélisée comme suit :

$$\sum_{v \in M} clk_{p,t,v} = 1, \qquad \forall p \in V, \forall t \in T \tag{1}$$

On peut naturellement écrire cette contrainte sous forme clausale via un encodage binomial classique comme décrit ci-dessous, nécessitant $O(n * t_f * K^2)$ clauses mais des formulations plus compactes peuvent néanmoins être obtenues grâce à l'introduction de variables auxiliaires [20].

$$\bigwedge_{\substack{(v,v')\in M^2 \\ v<v'}} (\overline{clk_{i,t,v}} \vee \overline{clk_{p,t,v'}}), \quad \forall p \in V, \forall t \in T \tag{1a}$$

$$\bigvee_{v \in M} clk_{p,t,v}, \qquad \forall p \in V, \forall t \in T \tag{1b}$$

3.1.2 Sur l'autorisation des processus

On s'intéresse ici à la sémantique d'autorisation de mise à jour des horloges des processus dans l'unisson asynchrone. Pour cela, nous considérons tout d'abord la modélisation des retards légitimes entre paires de processus $(p, q) \in V^2$ avec $p \neq q$, définie par le prédicat $behind(p, q) = ((q.c - p.c) \bmod K) \leq n$. Pour cela, nous introduisons les **variables de retard** $bhd_{p,q,t}$ pour $p, q \in V^2$ tels que $p \neq q$ et $t \in T$, indiquant que la valeur d'horloge du processus p est en retard d'au plus n incréments par rapport à celle du processus q dans la configuration t. De plus, afin de faciliter l'encodage FNC, nous introduisons des **variables de double horloge** $dclk_{p,q,t,v,u}$ représentant la conjonction des deux variables d'horloge correspondantes, c'est-à-dire le fait que p possède la valeur v et q la valeur u à l'instant t. Les contraintes suivantes expriment alors que $bhd_{p,q,t}$ est vraie si et seulement si l'une des paires (v, u) satisfait la condition de retard admissible, générant un nombre de clauses en $O(n^2 * t_f * K^2)$:

$$bhd_{p,q,t} \leftrightarrow \bigvee_{\substack{(v,u)\in M^2 \\ (u-v) \bmod K \leq n}} dclk_{p,q,t,v,u} \qquad \forall p \in V,\ \forall q \in N(p),\ \forall t \in T \tag{3}$$

$$dclk_{p,q,t,v,u} \leftrightarrow clk_{p,t,v} \wedge clk_{q,t,u} \qquad \forall p \in V,\ \forall q \in V \setminus \{p\},\ \forall v \in M^2,\ \forall t \in T \tag{4}$$

On rappelle qu'un processus $p \in V$ est activable quand la garde d'une de ses règles est vérifiée et il peut donc exécuter son action s'il est choisi par le démon. L'algorithme de l'unisson asynchrone présente deux règles d'autorisation différentes. Tout d'abord, un processus p est autorisé à incrémenter son horloge si elle a au plus n incréments de retard sur les valeurs d'horloge de chacun de ses voisins. Pour représenter cette sémantique, nous introduisons les **variables d'autorisation à l'incrémentation** $inc_{p,t}$ où $p \in V$ et $t \in T$ qui seront affectées à $vrai$ si le processus p a un retard admissible par rapport à tous les processus dans son voisinage $N(p)$ à la configuration t. Cette condition peut alors être exprimée par la contrainte suivante générant $O(n^2 * t_f)$ clauses :

$$inc_{p,t} \leftrightarrow \bigwedge_{q \in N(p)} bhd_{p,q,t} \qquad \forall p \in V,\ \forall t \in T \tag{5}$$

L'algorithme prévoit également une seconde règle permettant à un processus de réinitialiser sa valeur d'horloge lorsque sa situation par rapport à ses voisins devient ambiguë. Plus précisément, un processus p est autorisé à se réinitialiser si sa valeur d'horloge est non nulle et s'il existe au moins un voisin dont la valeur d'horloge n'est pas dans une relation de retard admissible avec celle de p. Pour représenter cette sémantique, nous introduisons les **variables d'autorisation à la réinitialisation** $res_{p,t}$ où $p \in V$ et $t \in T$, qui seront affectées à $vrai$ si le processus p est autorisé à réinitialiser son horloge à la configuration t. Cette condition repose sur l'existence d'un voisin q avec qui l'horloge de p n'est pas comparable. Pour cela, nous introduisons **les variables de double retard** $dbhd_{p,q,t}$ indiquant qu'au moins l'un des deux processus p et q est en retard admissible par rapport à l'autre. Ces conditions peuvent alors être exprimées par les contraintes suivantes générant $O(n^2 * t_f)$ clauses :

$$res_{p,t} \leftrightarrow \overline{clk_{p,t,0}} \wedge \Big(\bigvee_{q \in N(p)} \overline{dbhd_{p,q,t}} \Big) \qquad \forall p \in V,\ \forall t \in T \tag{6}$$

$$dbhd_{p,q,t} \leftrightarrow bhd_{p,q,t} \vee bhd_{q,p,t} \qquad \forall p \in V,\ \forall t \in T \tag{7}$$

3.1.3 Sur l'activation des processus

Dans cette section, nous nous intéressons à la prise en compte de l'asynchronisme, qui induit différentes exécutions possibles à partir de chaque configuration. Précisément, l'asynchronisme du système est généré par un démon distribué inéquitable, qui active à chaque étape un sous-ensemble arbitraire non vide de processus activables. Lorsqu'il est activé, un processus modifie sa valeur d'horloge conformément aux règles définies dans l'algorithme 1 selon son statut d'autorisation (incrémentation ou ré-initialisation). À l'instant t, notons A_t l'ensemble des processus activables. Sous un démon distribué inéquitable, n'importe quel sous-ensemble non vide de A_t peut être activé simultanément. Chaque choix de sous-ensemble correspond donc à une évolution possible du système à partir de la configuration t. Le nombre total d'exécutions possibles à cette étape est alors égal au nombre de sous-ensembles non vides de A_t, soit $2^{|A_t|} - 1$. La prise en compte conjointe de l'asynchronisme et de la diversité des configurations initiales induit donc une combinatoire particulièrement importante.

Pour modéliser la sémantique de l'asynchronisme, nous introduisons les **variables d'activation** $act_{p,t}$ où $p \in V$ et $t \in T$. Ces variables sont affectées à $vrai$ si le processus p est activé à la configuration t, et leur choix est laissé libre au solveur pour simuler les décisions d'activation d'un démon distribué inéquitable. Afin de garantir la cohérence du modèle, deux contraintes doivent être imposées : un processus activé doit être activable (pour l'incrémentation ou de la ré-initialisation de son horloge), et l'ensemble des processus activés à chaque étape doit être non vide s'il n'y a pas d'interblocage. En effet, il convient de souligner qu'un

blocage peut survenir au niveau des mécanismes d'autorisation. Dans certaines situations, le système peut atteindre un état dans lequel aucun processus n'est en mesure de progresser. Dans ce cas, on parle de phénomène d'interblocage (*deadlock*), le système entre alors dans un état de divergence, caractérisé par l'absence totale d'évolution du système (toute les valeurs d'horloge restent figées).
Dans le modèle décrit dans la section 2, lorsqu'un interblocage apparaît dans la configuration t d'une exécution, l'exécution termine : les configurations t' avec $t' > t$ ne sont pas définies. Or, dans notre modélisation, la plupart des variables sont définies pour toutes les valeurs $t \in [0..t_f - 1]$. Pour palier à ce problème, nous complémentons toutes exécutions atteignant un interblocage en t avec un suffixe constitué de $t_f - t - 1$ configurations identiques à la configuration t. Cette astuce nous permettra d'ailleurs de gérer l'existence d'un interblocage de manière analogue à l'existence d'un cycle de configurations illégitimes.
Pour modéliser une situation d'interblocage, nous introduisons les **variables d'interblocage** $dlock_t$ où $t \in T$. Ces variables sont affectées à $vrai$ si aucun processus n'est activable à la configuration t. Les trois contraintes sont exprimées ci-dessous, introduisant ainsi un nombre de clauses bornée en $O(n * t_f)$:

$$act_{p,t} \rightarrow (inc_{p,t} \vee res_{p,t}), \quad \forall p \in V,\ \forall t \in T \tag{8}$$

$$\overline{dlock_t} \rightarrow \bigvee_{p \in V} act_{p,t}, \quad \forall t \in T \tag{9}$$

$$dlock_t \leftrightarrow \bigwedge_{p \in V} (\overline{inc_{p,t}} \wedge \overline{res_{p,t}}), \quad \forall t \in T \tag{10}$$

3.1.4 Sur la mise à jour des horloges

On s'intéresse maintenant à la mise à jour des horloges des processus à chaque itération de l'algorithme. On rappelle que, dans un système asynchrone contrôlé par un démon distribué inéquitable, tous les processus activables ne sont pas nécessairement activés simultanément, et seuls certains d'entre eux sont autorisés à modifier leur état en fonction de la vérification des propriétés d'autorisation. Il est donc nécessaire de distinguer les situations possibles afin de modéliser fidèlement l'évolution des configurations du système. Plus précisément, à chaque étape t, la mise à jour des horloges dépend du statut de chaque processus, en fonction de son activation et de son autorisation. Un processus peut donc se trouver dans l'une des trois situation suivantes : (i) Il n'est pas activable et conserve donc son état courant ; (ii) Il est activable mais non activé et, dans ce cas, maintient également son état inchangé ; (iii) Il est activable et activé, auquel cas il met à jour sa valeur conformément aux deux règles de l'algorithme. Ces trois cas couvrent l'ensemble des comportements possibles et suivent donc des règles spécifiques qui sont détaillées ci-dessous.

(i) Processus non activables. Un processus est *non activable* à l'étape t lorsqu'il ne vérifie aucune propriété d'autorisation : il n'est autorisé ni à incrémenter sa valeur d'horloge ni à la ré-initialiser. Dans ce cas, son état doit rester inchangé entre les configurations t et $t + 1$ (en particulier, en cas d'interblocage). Autrement dit, si le processus $p \in V$ a la valeur v dans la configuration t et qu'il n'est activable pour aucune des deux propriétés d'autorisation ($\overline{inc_{p,t}} \wedge \overline{res_{p,t}}$), alors il maintient la valeur v dans la configuration $t + 1$. Cette contrainte est exprimée ci-dessous, introduisant un nombre de clauses en $O(n * t_f * K)$:

$$\begin{aligned} clk_{p,t,v} \wedge \overline{inc_{p,t}} \wedge \overline{res_{p,t}} \rightarrow clk_{p,t+1,v} \\ \forall p \in V,\ \forall t \in T \setminus \{t_f - 1\}\ \forall v \in V \end{aligned} \tag{11}$$

(ii) Processus activables et non activés. Un processus est *activable mais non activé* à l'étape t lorsqu'il est autorisé à changer de valeur mais qu'il n'est pas sélectionné par le démon pour exécuter une transition à cette étape. Dans ce cas, bien que le processus pourrait modifier sa valeur, il conserve son état actuel, c'est-à-dire que son état reste inchangé entre les étapes t et $t+1$. Autrement dit, si le processus $p \in V$ possède la valeur $v \in M$ à l'étape $t \in T \setminus t_f - 1$ et n'est pas activé ($act_{p,t} = faux$), alors il conservera cette valeur à l'étape suivante. Cette contrainte est représentée ci-dessous et induit un nombre de clauses borné en $O(n * t_f * K)$:

$$\begin{aligned} clk_{p,t,v} \wedge \overline{act_{p,t}} \rightarrow clk_{p,t+1,v}, \\ \forall p \in V,\ \forall t \in T \setminus \{t_f - 1\}\ , \forall v \in M \end{aligned} \tag{12}$$

On note ici que l'introduction de cette contrainte rend la contrainte (11) redondante. En effet, par contraposée de la contrainte (8), un processus non activable ne peut pas être activé. Néanmoins, la contrainte (11) est conservée afin de renforcer le pouvoir de propagation du solveur et de permettre une mise à jour plus efficace des horloges des processus non activables. De plus, il est important de noter qu'en cas d'interblocage, même si l'exécution courante doit s'arrêter, les processus ne sont pas activés et conservent donc leurs valeurs jusqu'à l'instant $t_f - 1$ grâce à la contrainte (12) (ou (11)), ce qui permettra de détecter les phénomènes de convergence et de divergence décrits dans la section 3.2.

(iii) Processus activables et activés. Un processus est *activable et activé* à l'étape t lorsqu'il est autorisé à modifier sa valeur locale et qu'il est effectivement sélectionné par le démon pour exécuter une transition à cette étape. Dans ce cas, le processus met nécessairement à jour son état conformément aux règles de l'algorithme de l'unisson asynchrone décrites dans l'algorithme 1. Ainsi, chaque processus activé doit mettre à jour sa valeur selon son statut d'autorisation : s'il est autorisé à l'incrémentation, il doit incrémenter sa valeur modulo K à la prochaine configuration ; et, si il est autorisé à réinitialiser sa valeur, elle doit devenir nulle à la prochaine configuration. Ces contraintes, décrites formellement ci-dessous, génèrent un nombre de clauses en $O(n * t_f * K)$:

$$\begin{aligned} clk_{p,t,v} \wedge inc_{p,t} \wedge act_{p,t} \rightarrow clk_{p,t+1,(v+1) \bmod K} \\ \forall p \in V, \forall t \in T \setminus \{t_f - 1\},\ \forall v \in M \end{aligned} \tag{13}$$

$$\begin{aligned} clk_{p,t,v} \wedge res_{p,t} \wedge act_{p,t} \rightarrow clk_{p,t+1,0} \\ \forall p \in V, \forall t \in T \setminus \{t_f - 1\},\ \forall v \in M \end{aligned} \tag{14}$$

Ainsi, l'ensemble des contraintes définies ci-dessus permet de modéliser de manière exhaustive toutes les exécutions valides de l'algorithme asynchrone. L'utilisation conjointe des variables d'activation et d'autorisation garantit une représentation compacte et efficace des transitions. Dans la section suivante, nous nous intéresserons à la formalisation des contraintes permettant d'analyser les propriétés d'autostabilisation de l'algorithme.

3.2 Analyse de l'autostabilisation

Cette section est consacrée à l'analyse du comportement d'autostabilisation de l'algorithme de l'unisson asynchrone. Pour ce faire, nous distinguons deux aspects complémentaires : d'une part, la convergence, qui garantit l'atteinte d'une configuration légitime à partir de toute configuration initiale en un nombre fini d'itérations ; d'autre part, la divergence, qui permet d'identifier et de caractériser les exécutions qui ne peuvent pas mener à un état légitime. Cette analyse doit prendre en compte l'ensemble des exécutions possibles induites par l'asynchronisme de l'algorithme.

3.2.1 Convergence

Nous rappelons que la convergence est établie si, à partir de toute configuration initiale, l'exécution asynchrone de l'algorithme aboutit à une configuration légitime dans laquelle la valeur d'horloge de chaque processus p est en retard d'au plus un incrément par rapport à chacune des horloges de ses voisins. Plus précisément, si p possède la valeur d'horloge $v \in M$, alors la valeur d'horloge u d'un voisin $q \in N(p)$ appartient nécessairement à l'ensemble $\{v-1 \bmod K,\ v,\ v+1 \bmod K\}$. On rappelle ici que, les horloges étant modulaires, l'écart entre $K-1$ et 0 n'est que d'un seul incrément. De plus, l'ensemble des configurations légitimes est clos : toute configuration accessible depuis une configuration légitime est également légitime.

Plutôt que de considérer toutes les exécutions asynchrones directement, nous raisonnons par contradiction. Nous chercherons à vérifier s'il existe un chemin d'exécution asynchrone issu d'une configuration initiale arbitraire où l'algorithme ne converge pas, c'est-à-dire où la configuration à l'étape $t_f - 1$ comporte au moins un processus p possédant la valeur v et un voisin $q \in N(p)$ avec une valeur u dont l'écart par rapport à v est supérieur à un incrément, c'est-à-dire que $u \notin \{v-1 \bmod K,\ v,\ v+1 \bmod K\}$. Cette situation viole la propriété de convergence sur l'écart légitime et il suffit donc de considérer un seul chemin qui échoue pour aboutir à une contradiction ; sinon, tous les chemins d'exécution asynchrone possibles respectent la propriété de convergence et l'algorithme converge au plus tard après t_f configurations.

Pour formaliser cette propriété, on introduit les **variables de retard légitime** $obhd_{p,q,v}$ pour deux processus distincts $(p,q) \in V^2$ et $v \in M$, qui seront affecté à $vrai$ si le processus p possède une valeur d'horloge avec un écart d'au plus 1 par rapport à q à la dernière configuration $t_f - 1$. Ainsi, la non-convergence est clairement exprimée par l'existence d'un couple de processus voisins (p,q) et d'une valeur d'horloge $v \in M$ du processus p à la configuration $t_f - 1$ tel que q n'est pas en retard légitime par rapport à p ($obhd_{p,q,v} = faux$). Pour faciliter la réécriture sous format FNC, nous introduisons également les **variables d'horloges légitimes** $oclk_{p,v}$ pour $p \in V$ et $v \in M$, qui seront affecté à $vrai$ si le processus p possède une valeur d'horloge avec un écart d'au plus 1 par rapport à la valeur v à la dernière configuration $t_f - 1$. Ces contraintes, permettant l'analyse de la converge, sont présentées ci-dessous, générant un nombre de clauses borné en $O(n^2 * K)$:

$$\bigvee_{\substack{p \in V \\ q \in N(p) \\ v \in M}} \overline{obhd_{p,q,v}} \tag{15}$$

$$obhd_{p,q,v} \leftrightarrow (\overline{clk_{p,t_f-1,v}} \ \vee \ oclk_{q,v}) \quad \forall p \in V,\ \forall q \in V \setminus \{p\},\ \forall v \in M \tag{16}$$

$$oclk_{p,v} \leftrightarrow (clk_{p,t_f-1,v} \ \vee \ clk_{p,t_f-1,(v-1) \bmod K} \ \vee \ clk_{p,t_f-1,(v+1) \bmod K}) \quad \forall p \in V,\ \forall v \in M \tag{17}$$

Ainsi, l'algorithme de l'unisson asynchrone converge sur un graphe connexe non orienté de taille n en t_f configurations si et seulement si la formule induite par les contraintes présentées est insatisfiable. Cette formule permettant d'analyser la convergence contient globalement $O(n^2 * t_f * K^2)$ clauses.

Théorème 1. *L'algorithme de l'unisson asynchrone converge sur un graphe connexe non orienté de taille n en au plus t_f configurations ssi la formule induite par les contraintes (1-17) est insatisfiable.*

3.2.2 Divergence

La divergence de l'algorithme de l'unisson asynchrone correspond à l'existence d'une exécution qui n'atteint jamais une configuration légitime. Autrement dit, cela implique nécessairement l'existence d'un interblocage ou d'un cycle de configurations illégitimes (puisque l'ensemble des configurations légitimes est clos). En raison de l'asynchronisme, chaque configuration peut engendrer plusieurs exécutions possibles. Ainsi, pour établir la divergence, il suffit de montrer l'existence d'un interblocage ou d'un cycle de configurations identiques et illégitimes le long d'un même chemin d'exécution asynchrone.

Plus formellement, dans le cas d'un cycle, on recherche deux configurations $0 \leq t_1 < t_2 < t_f$ formant une paire de configurations identiques et illégitimes. Sans perte de généralité, on se restreint au cas où $t_1 = 0$, c'est-à-dire à la comparaison entre une configuration atteinte à l'étape $t \neq 0$ et la configuration initiale. Ce choix est justifié par le fait que toute configuration peut être considérée comme initiale, et donc l'analyse peut toujours être ramenée à cette configuration. Afin de garantir que le cycle considéré ne contient que des configurations illégitimes, il suffit d'imposer que la configuration initiale soit elle-même illégitime. Comme annoncé précédemment, les cas d'interblocage sont également

détecté à l'aide de cette méthode car, dans notre formalisation, ils deviennent des cas particuliers de cycles. En effet, toute exécution atteignant une configuration interbloquée peut aussi être ramené à une exécution où cette configuration illégitime est initiale. Dans ce cas, grâce à la conservation des valeurs d'horloge réalisée dans notre formalisation, un cycle sera détecté en une seule itération puisque les deux premières configurations seront identiques.

Pour raisonner sur l'existence de cycles, on introduit les **variables de cycles** cyc_t, pour $t \in T$, indiquant l'existence d'un cycle entre la configuration initiale et la configuration t. L'existence d'un cycle de configurations illégitimes est alors modélisée par les contraintes (18) et (19), générant deux clauses. La première clause exprime l'existence d'au moins une configuration non initiale formant un cycle tandis que la deuxième impose que la configuration initiale soit illégitime (et donc celle à t aussi), en garantissant l'existence d'un retard non légitime entre un processus et l'un de ses voisins.

$$\bigvee_{t \in T \setminus \{0\}} cyc_t \tag{18}$$

$$\bigvee_{\substack{p \in V \\ q \in N(p) \\ v \in M}} \overline{obhd_{p,q,v}} \tag{19}$$

Il est important de noter que la contrainte (19) a une sémantique légèrement différente de celle de la convergence (cf. (15)) puisqu'on l'applique à la première configuration au lieu de la dernière. Il est donc nécessaire de mettre à jour la définition des variables de retard et d'écart légitimes afin prendre en compte cette nouvelle sémantique. De plus, afin de définir la sémantique des variables de cycle cyc_t, il est nécessaire d'imposer que, lorsque cyc_t est vraie, la configuration à l'étape t coïncide avec la configuration initiale. À cette fin, nous introduisons les **variables de similarité d'horloge** $sim_{p,t,v}$, indiquant que le processus p possède la valeur v simultanément aux configurations 0 et t. L'ensemble de ces contraintes sont définies ci-dessous, générant un nombre de clauses borné en $O(n^2 * t_f * K)$.

$$obhd_{p,q,v} \leftrightarrow (\overline{clk_{p,0,v}} \ \vee \ oclk_{q,v}) \quad \forall p \in V,\ \forall q \in V \setminus \{p\},\ \forall v \in M \tag{20}$$

$$oclk_{p,v} \leftrightarrow (clk_{p,0,v} \ \vee \ clk_{p,0,(v-1) \bmod K} \ \vee \ clk_{p,0,(v+1) \bmod K}) \quad \forall p \in V,\ \forall v \in M \tag{21}$$

$$cyc_t \leftrightarrow \bigvee_{v \in M} sim_{p,t,v} \quad \forall t \in T \setminus \{0\}, \forall p \in V \tag{22}$$

$$sim_{p,t,v} \leftrightarrow clk_{p,0,v} \wedge clk_{p,t,v} \quad \forall t \in T \setminus \{0\},\ \forall p \in V,\ \forall v \in M \tag{23}$$

Ainsi, la formule induite par les contraintes (1-14 ;18-23) est satisfiable si et seulement si l'algorithme de l'unisson asynchrone diverge en au plus t_f configurations. En effet, un modèle de cette formule permet d'exhiber une configuration initiale ainsi qu'une branche de l'arbre des exécutions asynchrone contenant un cycle de configurations illégitimes (y compris en cas d'interblocage). En termes de complexité, l'encodage global reste du même ordre que celui de la convergence, soit un nombre de clauses borné en $O(n^2 * t_f * K^2)$.

Théorème 2. *L'algorithme de l'unisson asynchrone diverge sur un graphe connexe non orienté de taille n en au plus t_f configurations ssi la formule induite par les contraintes (1-14 ;18-23) est satisfiable.*

4 Évaluation expérimentale

Nous avons considéré trois topologies classiques : anneaux, chaînes et étoiles. Pour chaque type de topologie, nous avons généré différentes instances en variant les tailles de graphes n de 3 jusqu'à 5 noeuds, et la période K de $n+1$ à n^2+1. Ainsi, on obtient 41 instances de convergence et divergence (CNV et DIV) correspondant respectivement aux formules induites par les contraintes (1-17) et (1-14 ;18-23) pour chaque topologie, et totalisant 246 instances. Nous avons utilisé l'encodage Cardinality Network [3] pour réécrire la contrainte de cardinalité (1)[3]. Enfin, on fixe $t_f = 100$ pour toutes les instances. Nos modèles ont été encodés en Python en utilisant la bibliothèque PySAT[4] [17]. Nous avons résolu ces instances en utilisant le solveur Cadical [6]. Les tests ont été réalisés sur la plateforme MatriCS[5], sur une machine équipée d'un processeur Intel Core i7 cadencé à 3,80 GHz, sous Ubuntu 22.04. Un temps limite de 3600 s a été alloué à chaque instance.

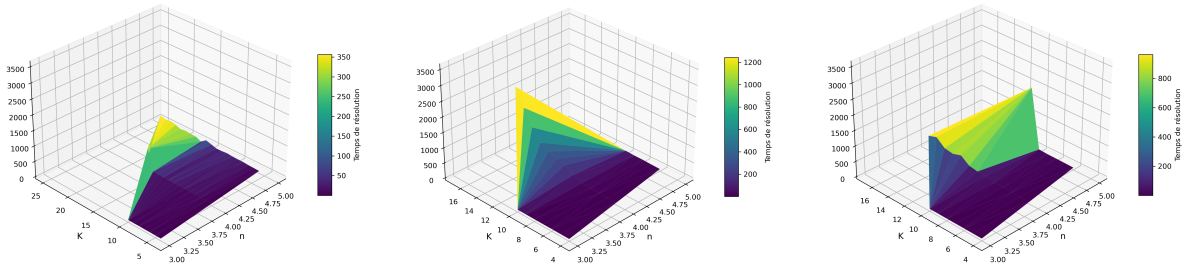
Tout d'abord, nous analysons le temps de résolution et le nombre d'instances résolues sur différentes topologies par taille de graphe. Les résultats obtenus sont présentés dans le tableau 3, en fonction de la taille du graphe et de la propriété analysée (convergence/divergence). Par ailleurs, les figures 4 et 5 illustrent l'évolution du temps de résolution des instances générées respectivement pour la vérification de la convergence et de la divergence, en fonction de la taille du graphe n et de la période K sur les différentes topologies. Il est remarquable que le taux d'évolution du temps de résolution dans les anneaux est beaucoup plus faible par rapport aux chaînes et aux étoiles.

Les résultats indiquent que, sur l'ensemble des instances générées, le pourcentages d'instances résolues pour la détection de la convergence (resp. divergence) est de $97,56\%$ (resp. $97,56\%$) pour les anneaux, $60,98\%$ (resp. $85,37\%$) pour les chaînes et $63,41\%$ (resp. 100%) pour les étoiles. Plus précisément, pour les tailles $n = 3$ et $n = 4$, toutes les instances (c'est-à-dire, 7 pour $n = 3$ et 13 pour $n = 5$) ont

3. Pour encoder la contrainte $\sum_{i=1}^{h} l_i = 1$, cet encodage génère $O(h)$ clauses et variables auxiliaires.
4. https://pysathq.github.io/
5. https://www.matrics.u-picardie.fr/catalogue-de-services/compte-matrics/

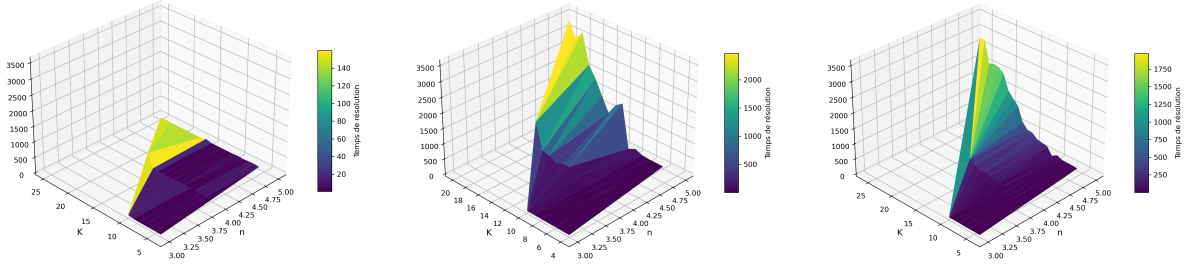
n	$\#I$	Anneaux		Chaînes		Étoiles	
		CNV	**DIV**	**CNV**	**DIV**	**CNV**	**DIV**
3	7	11.08 [7]	36.87 [7]	14.25 [7]	25.42 [7]	12.49 [7]	26.06 [7]
4	13	871.83 [13]	438.01 [13]	6947.44 [13]	2613.05 [13]	2160.56 [13]	1158.38 [13]
5	21	1584.10 [20]	170.98 [20]	102.51 [5]	18589.88 [15]	2020.86 [6]	20362.94 [21]
Total	**41**	**2467.01 [40]**	**645.86 [40]**	**7064.20 [25]**	**21228.35 [35]**	**4193.90 [26]**	**21547.38 [41]**

TABLE 3 – Temps de résolution en secondes pour les topologies en anneau, chaîne et étoile. Pour chaque topologie, les sommes des temps de résolution pour la détection de convergence (CNV) et divergence (DIV) sur toutes les périodes (K) pour une taille de graphe donnée (n) sont reportées. Pour une taille n donnée, la colonne $\#I$ indique le nombre d'instances générées avec CNV ou DIV et le nombre d'instances résolues est indiqué entre '[]'.



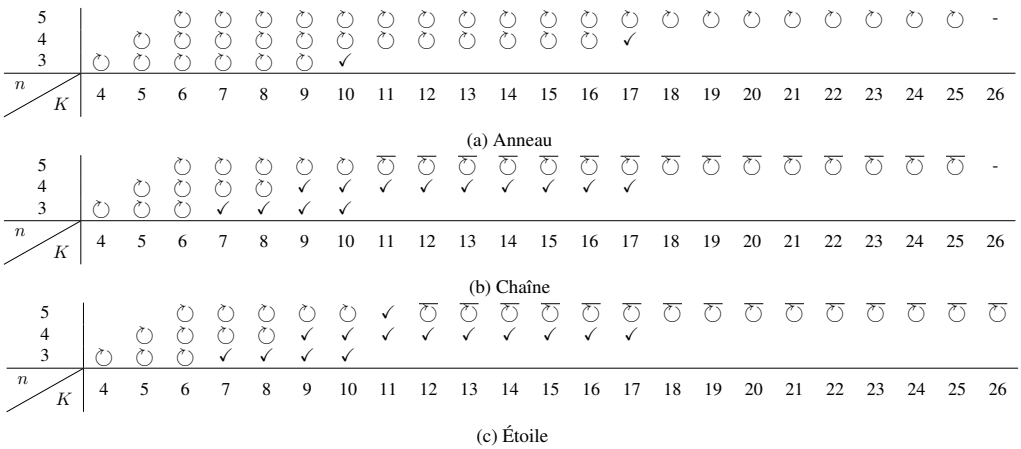
(a) Anneau (b) Chaîne (c) Étoile

FIGURE 4 – Temps de résolution en secondes pour la convergence en fonction du nombre de processus n et de la période K.



(a) Anneau (b) Chaîne (c) Étoile

FIGURE 5 – Temps de résolution en secondes pour la divergence en fonction du nombre de processus n et de la période K.



(a) Anneau

(b) Chaîne

(c) Étoile

FIGURE 6 – Résultats de convergence et de divergence de l'algorithme de l'unisson asynchrone sur différents types de graphes pour des tailles $n = 3, 4, 5$ et périodes $K = n + 1 \dots n^2 + 1$.

été résolues à la fois pour la convergence et la divergence pour toutes les topologies. Avec les graphes de plus grande taille $n = 5$, 20 instances de convergence et de divergence relatives à la topologie en anneau ont été résolues montrant un comportement divergent. Avec les topologies en chaîne et en étoile, nous remarquons que la résolution est beaucoup plus difficile surtout dans le cas de la divergence avec seulement 5 instances dans les chaînes (respectivement 6 dans les étoiles) pour le cas de la convergence menant à une baisse drastique du nombre d'instances résolues. En effet, seules 5 ou 6 instances sont résolues sur les 21 instances générées. Par ailleurs, nous remarquons que toutes les instances ont été résolues dans le cas de la divergence pour les étoiles. Par contre, seulement 15 instances instances ont été résolues dans le cas de la divergence pour les chaînes.

En terme de temps de résolution, pour un même nombre d'instance résolues sur les tailles de graphe de 3 et 4, nous remarquons que le modèle est plus performant sur les graphes en anneaux que sur les chaînes et les étoiles avec une accélération de $87,32\%$ et de $82,00\%$ par rapport aux graphes en chaînes et de $59,37\%$ et $59,91\%$ par rapport aux graphes en étoiles respectivement dans le cas de la convergence et de la divergence. Pour les graphes de taille 5, le modèle a résolu dans le cas des anneaux 20 instances de plus entre convergence et divergence par rapport aux chaînes et 13 instances de plus par rapport aux étoiles.

Les résultats présentés dans la Figure 6 illustrent les comportements exhibés de l'algorithme d'unisson asynchrone en termes de convergence et de divergence sur les différents types de topologies (anneaux, chaînes et étoiles) en fonction de la taille n et de la période K. Les signes (✓), (↻), ($\overline{↻}$) et ($-$) indiquent respectivement que l'algorithme converge, diverge, ne diverge pas dans la limite du temps alloué ou que les instances de convergence et de divergence n'ont pas été résolues. Pour toutes les topologies, tous les comportements ont pu être exhibés pour les graphes de taille 3 et 4, révélant une convergence alignée avec les résultats prouvés dans [14] lorsque $K = n^2 + 1$. En revanche, pour les graphes de taille 5, certaines instances n'ont pas été résolues dans la limite d'une heure. Ceci est traduit par la complexité accrue du modèle dans des graphes de taille plus grande.

Nous remarquons une tendance claire de convergence et de divergence pour les topologies et tailles étudiées. Tout, d'abord, la convergence est constatée pour toutes les instances résolues avec $K = n^2 + 1$. De plus, on arrive à constater la divergence pour $K = n^2$ pour les anneaux de taille $n \in \{3, 4, 5\}$. Ces résultats s'alignent avec ceux établis dans [14] pour la convergence et montrent que la borne de $n^2 + 1$ est optimale dans le cadre général. On remarque également que la divergence est exhibée pour $K \leq 2n$ sur les topologies en chaîne et en étoile tandis que les instances qui ont été résolues au dessus de cette borne sont toujours convergentes. Ces observations nous conduisent à conjecturer que l'algorithme de l'unisson asynchrone est autostabilisant pour la borne raffinée $K \geq 2n+1$ pour les structures de graphes acycliques.

5 Conclusion

Dans cet article, nous avons présenté une approche formelle basée sur SAT pour analyser le comportement d'un algorithme autostabilisant d'unisson asynchrone. Plus précisément, notre étude porte sur la détection des cas où cet algorithme présente un comportement de convergence ou de divergence. En modélisant les états de l'algorithme ainsi que son exécution à l'aide de contraintes logiques, nous avons montré comment les solveurs SAT peuvent être exploités pour démontrer l'autostabilisation de l'unisson asynchrone appliqué à différentes topologies classiques et en tenant en compte de différentes périodes ou, à défaut, pour exhiber sa divergence. Nos résultats nous ont permis d'établir l'optimalité de la borne de convergence établie dans [14] ($K = n^2+1$). Cependant, nos observations laissent conjecturer que cette borne pourrait être raffinée sous certaines hypothèses sur la structure topologique des réseaux considérés. En particulier, l'acyclicité pourrait permettre d'obtenir une borne de convergence plus précise, notamment pour des périodes à partir de $K = 2n + 1$.

Cette étude ouvre la voie à plusieurs pistes de recherche. Tout d'abord, l'extension de notre analyse à d'autres types de graphes classiques, tels que les grilles, les tores ou bien des graphes aléatoires, permettrait d'analyser de façon plus approfondie l'impact d'autres topologies distribuées sur l'autostabilisation de l'algorithme. Il serait également intéressant d'étudier d'autres formulations plus compactes des contraintes afin de réduire la complexité du modèle. De surcroît, pour la contrainte de convergence, il serait intéressant d'envisager une analyse plus fine en établissant des contraintes redondantes sur l'ensemble des configurations au lieu de se focaliser uniquement sur la vérification de la synchronisation au niveau de la dernière à l'instar de [18]. Enfin, il serait pertinent de prouver formellement l'exactitude de la borne établie ou encore de la raffiner davantage. Ces perspectives favorisent une compréhension plus approfondie des dynamiques de synchronisation dans les systèmes distribués asynchrones complexes, et à l'élaboration de nouvelles méthodes plus optimisées pour couvrir un spectre plus vaste d'exemples d'applications.

Remerciements

Ce travail est partiellement soutenu par les projets ANR-24-CE23-6126 (BforSAT) et ANR-22-EXES-0009 (PIA4 MAIA, France 2030), financés par l'agence nationale de recherche (ANR). Il a bénéficié d'un accès aux ressources HPC de la « Plateforme MatriCS » de l'Université de Picardie Jules Verne, cofinancée par l'Union Européenne avec le Fonds Européen de Développement Régional (FEDER) et le Conseil Régional des Hauts-De-France.

Références

[1] Karine Altisen, Stéphane Devismes, and Erwan Jahier. sasa : a simulator of self-stabilizing algorithms. *Comput. J.*, 66(4) :796–814, 2023.

[2] Anish Arora, Shlomi Dolev, and Mohamed G. Gouda. Maintaining digital clocks in step. *Parallel Processing Letters*, 1 :11–18, 1991.

[3] Roberto Asín, Robert Nieuwenhuis, Albert Oliveras, and Enric Rodríguez-Carbonell. Cardinality networks and their applications. In Oliver Kullmann, editor, *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*, volume 5584 of *Lecture Notes in Computer Science*, pages 167–180. Springer, 2009.

[4] Armin Biere. Handbook of satisfiability. In *Frontiers in Artificial Intelligence and Applications*, pages 75–98. IOS Press, 2009.

[5] Armin Biere et al. Symbolic model checking without bdds. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 193–207, 1999.

[6] Armin Biere, Tobias Faller, Katalin Fazekas, Mathias Fleury, Nils Froleyks, and Florian Pollitt. CaDiCaL 2.0. In Arie Gurfinkel and Vijay Ganesh, editors, *International Conference on Computer Aided Verification, CAV*, volume 14681 of *LNCS*, pages 133–152. Springer, 2024.

[7] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability - Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2021.

[8] L. Blin, C. Johnen, G. Le Bouder, and F. Petit. Silent anonymous snap-stabilizing termination detection. In *41st International Symposium on Reliable Distributed Systems, (SRDS'22)*, pages 156–165, 2022.

[9] C. Boulinier and F. Petit. Self-stabilizing wavelets and rho-hops coordination. In *22nd IEEE International Symposium on Parallel and Distributed Processing, (IPDPS'08)*, pages 1–8, 2008.

[10] C. Boulinier, F. Petit, and V. Villain. When graph theory helps self-stabilization. In *23rd Annual Symposium on Principles of Distributed Computing, (PODC'04)*, pages 150–159, 2004.

[11] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Proceedings of the Workshop on Logic of Programs*, volume 131 of *LNCS*, pages 52–71. Springer, 1982.

[12] Stephen A. Cook. The complexity of theorem-proving procedures. In Michael A. Harrison, Ranan B. Banerji, and Jeffrey D. Ullman, editors, *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, May 3-5, 1971, Shaker Heights, Ohio, USA*, pages 151–158. ACM, 1971.

[13] Jean-Michel Couvreur, Nissim Francez, and Mohamed G. Gouda. Asynchronous unison (extended abstract). In *Proceedings of the 12th International Conference on Distributed Computing Systems, Yokohama, Japan, June 9-12, 1992*, pages 486–493. IEEE Computer Society, 1992.

[14] Jean-Michel Couvreur, Nissim Francez, and Mohamed G. Gouda. Asynchronous unison (extended abstract). In *The 12th International Conference on Distributed Computing Systems (ICDCS)*, pages 486–493. IEEE Computer Society, 1992.

[15] A. K. Datta, S. Devismes, and L. L. Larmore. A silent self-stabilizing algorithm for the generalized minimal *k*-dominating set problem. *Theoretical Computer Science*, 753 :35–63, 2019.

[16] Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17(11) :643–644, 1974.

[17] Alexey Ignatiev, António Morgado, and João Marques-Silva. Pysat : A python toolkit for prototyping with SAT oracles. In Olaf Beyersdorff and Christoph M. Wintersteiger, editors, *Theory and Applications of Satisfiability Testing - SAT 2018 - 21st International Conference, SAT 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings*, volume 10929 of *Lecture Notes in Computer Science*, pages 428–437. Springer, 2018.

[18] Asma Khoualdia, Sami Cherif, Stéphane Devismes, and Léo Robert. Analyzing self-stabilization of synchronous unison via propositional satisfiability. In Maria Garcia de la Banda, editor, *31st International Conference on Principles and Practice of Constraint Programming, CP 2025, Glasgow, Scotland, August 10-15, 2025*, volume 340 of *LIPIcs*, pages 19 :1–19 :21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2025.

[19] João P. Marques-Silva and Karem A. Sakallah. Grasp : A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5) :506–521, 1999.

[20] Olivier Roussel and Vasco Manquinho. Pseudo-boolean and cardinality constraints. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability - Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 1087–1129. IOS Press, 2021.