UQAC
Université du Québec
à Chicoutimi

MÉMOIRE

PRÉSENTÉ À

L'UNIVERSITÉ DU QUÉBEC À CHICOUTIMI

COMME EXIGENCE PARTIELLE

DE LA MAÎTRISE EN INFORMATIQUE

PAR

KUN XIE

L'AMÉLIORATION DE LA COLLECTE DE DONNÉES ET DE L'ÉVALUATION DE FORMULES POUR LA VÉRIFICATION DE L'EXÉCUTION

MAI 2016

Résumé

Les technologies de l'information sont devenues une partie importante de notre vie. Bien que ces magnifiques techniques nous facilitent la vie et facilitent la vie, les accidents et les catastrophes causés par des dysfonctionnements de logiciels provoquent beaucoup de pertes de vies et de richesse qui peuvent en fait être évitées. La vérification et la validation de logiciels sont un ensemble de techniques visant à vérifier la fonctionnalité et à évaluer la qualité logicielle. La vérification de l'exécution est l'une des techniques couramment utilisées dans le domaine industriel. Son origine provient d'autres techniques de vérification, mais elle a aussi ses propres fonctionnalités et caractéristiques.

Le but de cette recherche est d'explorer les méthodes et les solutions pour améliorer deux aspects de la vérification de l'exécution : la collecte de données et l'évaluation de formules. Dans la première partie, nous présentons un canal de communication unidirectionnelle basé sur des codes optiques qui sont applicables pour la transmission de données dans un environnement spécifique. Ensuite, dans la deuxième partie, nous introduisons notre solution de l'évaluation hors ligne de logiques temporelles basées sur la manipulation bitmap et la compression bitmap. Les deux parties ont été écrites sous forme d'articles à publier, dont l'un a été publié, tandis que l'autre est en cours d'examen.

Mots-clefs: vérification de l'exécution, logique temporelle linéaire, compression de bitmap, protocoles de communication optique, code QR.

Remerciements

Je voudrais remercier toutes les personnes qui m'ont aidé à terminer mes études.

Je voudrais d'abord exprimer ma sincère gratitude à mon directeur, le professeur **Sylvain Hallé** pour sa supervision, son soutien, sa patience et sa générosité. J'ai la chance d'avoir un superviseur qui a vraiment à cœur ses étudiants et qui est toujours patient à répondre à toutes sortes de questions et de problèmes. **M. Sylvain Hallé** m'a aussi accordé l'opportunité et la confiance de prendre part aux travaux de rédaction d'articles pour les journaux et conférences de grande renommée. Grâce à lui, je garderai un formidable souvenir de l'UQAC.

Je voudrais exprimer ma gratitude au professeur **Sylvain Boivin** pour son orientation, son soutien et son aide. Il m'a motivé à poursuivre mes études à l'université, m'a encouragé dans l'apprentissage de la langue française et m'a introduit au proffesseur **Sylvain Hallé**.

Je voudrais également remercier **Mme Marjolaine Hénault** pour sa gentillesse, sa générosité, son encouragement et son aide précieuse dans mon apprentissage du français, les amis demeurant à Chicoutimi : **Ran Wei, Ping Li, Jian Qin et al.** pour leur amitié, leur encouragement, leur soutien et leur aide, ainsi que les amis et les étudiants travaillant chez LIF : **Edmond la Chance, Francis Guérin, Daehli Nadeau-Otis et al.** pour leur amitié et leur soutien.

Enfin, je tiens à exprimer mes plus vifs remerciements à mon épouse **Moon Ji Hyun**, dont la compagnie, le dévouement, les paroles encourageantes et la meilleure technique de cuisine sur la terre ont été la motivation essentielle de l'achèvement de ma maîtrise.

TABLE DES MATIÈRES

TABLE DES FIGURES

LISTE DES TABLEAUX

CHAPITRE 1

INTRODUCTION

Au cours des récentes décennies, un grand nombre de systèmes de logiciels ont été introduits dans presque tous les domaines de notre vie (Clarke et al., 1999). Alors que les gens apprécient les facilités apportées par ces systèmes, il y a toujours le risque d'une défaillance dans les systèmes. Un échec comme un jeu vidéo ayant plusieurs bogues est ennuyeux mais tolérable, mais d'autres échecs sont fatals et inacceptables, par exemple des bogues dans les instruments médicaux, les systèmes de contrôle du véhicule automatisé ou les systèmes aéronautiques. Un exemple récent est le satellite astronomique japonais Hitomi, qui a causé une perte de 286 millions de dollars à Japan Aerospace Exploration Agency (JAXA). Il a été lancé le 17 février 2016 et a été officiellement déclaré perdu le 28 avril de la même année à cause d'une erreur de logiciel (Witze, 2016).

1.1 CONTEXTE

De toute évidence, la fiabilité d'un système est critique, et un système fiable doit avoir la capacité de fonctionner strictement selon sa spécification dans une période définie (Avižienis et al., 2004). La vérification et la validation de logiciels est le processus pour mesurer cette

capacité et évaluer la qualité logicielle (IEEE, 2012).

la vérification de l'exécution (Leucker et Schallhart, 2009) est une approche de la vérification et de la validation de logiciels qui est applicable pour vérifier si le comportement d'un système informatique satisfait ou viole certains propriétés. Normalement, la vérification de l'exécution n'a pas d'influence sur l'exécution du système en cours d'exécution, même si une violation des propriétés est détectée. À cet effet, un moniteur est utilisé pour analyser les traces finies collectées puis produire un verdict qui est généralement une valeur de vérité. Par conséquent, un système de vérification à l'exécution doit avoir au moins deux éléments essentiels : la collecte de données et l'évaluation de formules.

De nos jours, afin de répondre à la demande de l'analyse de la quantité de données de traces rapidement croissantes, diverses solutions ont été proposées. Certains chercheurs comme Barre et al. (2012) ont porté des méthodologies existantes aux cadres du calcul distribué. Cependant, peu importe la quantité de processeurs et de mémoires qu'un système de cloud possède, il y a toujours une limite pour leurs utilisations. Ainsi, d'autres chercheurs ont essayé d'optimiser les algorithmes, tels que Havelund et Roşu (2001).

Bitmap est une méthode efficace pour réduire le coût de l'espace grâce à sa structure concise, et les caractéristiques telles que le parallélisme du niveau de bits ou l'affinité du cache CPU sont en mesure d'accroître la performance des opérations (Culpepper et Moffat, 2010). Il est largement appliqué dans les applications qui ont une exigence sérieuse de l'espace et de l'efficacité, par exemple les bases de données et les moteurs de recherche (Kaser et Lemire, 2014). Si un bitmap est limité, c'est-à-dire que la fraction de bits utilisés est faible, le bitmap peut occuper moins d'espace de stockage à l'aide de l'algorithme de compression de bitmap (Antoshenkov, 1995).

Avant que les moniteurs analysent les traces, la collecte de données joue un rôle important

(Casley et Kumar, 1988). Pour les différents systèmes, il existe des solutions correspondantes de la collecte de données. Zwijze-Koning et De Jong (2005) ont revu les techniques de collecte de données pour l'analyse de réseau. Calabrese et al. (2011) ont présenté un système de surveillance en temps réel avec la collecte de données à résolution élevée et à définition élevée de l'utilisation du téléphone cellulaire d'une ville italienne. Shabtai et Elovici (2010) ont développé un système de détection d'intrusion basé sur l'hôte pour les appareils mobiles d'Android en rassemblant les données des événements de systèmes et des interactions d'utilisateurs. Comme cela est indiqué dans les exemples, divers moyens sont utilisés pour extraire et transférer les données vers la location où la vérification a lieu. La lumière visible est également un moyen de communication efficace, comme le suggèrent Komine et Nakagawa (2004), particulièrement dans des environnements restreints où le câble ou la communication radio sont peu pratiques, comme Vasilescu et al. (2005).

Différents codes-barres ont été appliqués dans divers domaines à partir des systèmes traditionnels e-commerce à la croissance rapide d'appareils mobiles (Gao et al., 2007), parce que les codes-barres numériques fournissent une méthode simple mais précise avec un fable coût de distribution et de reconnaissance. Comparé avec le fameux code-barre 1-D UPC qui ne peut encoder que des chiffres, les codes-barres 2-D qui sont apparus à la fin des années 1980, peuvent non seulement encoder les données alphanumériques et les données même binaires, mais également fournir une capacité beaucoup plus grande. Quick Response Code (code QR) (Denso Wave Inc., 2015) est devenu l'un des 2-D codes-barres les plus populaires en raison de sa précision, sa capacité considérable, son impression relativement petite et sa grande efficacité. Il a été mis sur presque tous les types de surface visible, comme le papier, le téléphone et l'écran d'ordinateur, les vitrines des magasins (Okazaki et al., 2012).

1.2 OBJECTIFS DU MÉMOIRE

Les objectifs de cette recherche sont centrés sur le développement de méthodes ou de techniques qui est capable de fournir de l'assistance aux deux aspects mentionnés de la vérification de l'exécution : la collecte de données et l'évaluation de formules.

Le premier objectif principal et la contribution étaient de présenter une nouvelle méthode de la collecte de données et de discuter de sa faisabilité et de sa performance. Les codes QR sont considérés comme rapides et de grande capacité, et le fait le plus important est que son utilisation ne nécessite qu'une surface (par exemple l'écran) comme émetteur et une caméra comme récepteur, qui sont toutes deux devenues générales dans presque tous les ordinateurs portables et les téléphones mobiles ces dernières années. Si un code QR contenant une certaine quantité de données est considéré comme un paquet de données du réseau, une séquence de codes QR est similaire à un flux de données du réseau. Notre principale préoccupation ici était la bande passante du canal de la communication unidirectionnelle composée de codes QR, les facteurs critiques qui affectent la performance, ainsi que la méthode de l'application de ce canal de communication à notre pratique de la vérification de l'exécution.

Le deuxième objectif était de proposer une solution pour améliorer la performance du système de la vérification de l'exécution. Les bitmaps ont été démontrés par de nombreuses solutions pour leur capacité de l'amélioration de la performance. Parce que les états de logiques temporelles sont souvent exprimés avec les valeurs booléennes, c'est-à-dire vrai ou faux, les bitmaps sont prévus pour améliorer le calcul de formules LTL. Par conséquent, l'une de nos contributions était la solution de correspondre des états de logiques temporelles à des bits et de concevoir des algorithmes nécessaires pour mettre en œuvre les opérations LTL. Comme Kaser et Lemire (2014) le suggèrent, un bitmap faible est une perte d'espace. Une contribution supplémentaire était donc l'observation de l'impact des algorithmes de compression bitmap

sur le calcul de formules LTL.

Il est important de mentionner que notre travail et la réalisation du canal de communication de code QR ont été publiés dans la revue IEEE Access, vol. 4, pp. 284-298, 2016. Une autre partie de notre recherche, l'évaluation de formules LTL avec les manipulations de bitmap est en cours de révision pour sa publication dans les actes de la conférence internationale : Runtime Verification 2016 (RV'16) à Madrid, Espagne en 2016.

1.3 MÉTHODOLOGIE

Cette recherche a suivi une méthodologie en trois étapes.

La première étape était de développer le canal de la communication unidirectionnelle de codes QR qui correspond à notre deuxième objectif principal. Les paquets de données ont été encodés aux codes QR et décodés à partir des codes QR avec une librairie open source, et un protocole spécifique dédié à la sérialisation et à la transmission des données structurées sur des canaux de communication limités a été montré. Alors que l'expérience était en cours d'exécution, nous avons continué à optimiser notre solution basée sur le résultat de la première expérience pour améliorer le taux de correction et la vitesse de reconnaissance. Une webcam commune et un moniteur LED de 19 pouces ont été utilisés comme récepteur et émetteur dans l'expérience. Dans la dernière partie de cette étape, les codes QR étaient imprimés sur du papier de bureau et glissés devant la webcam afin de vérifier une allégation selon laquelle le protocole peut accepter des paquets de données entrantes sans ordre.

La deuxième étape a pour but de définir la relation de correspondance entre les états de logiques temporelles et les bitmaps, et aussi de concevoir les algorithmes des opérateurs de logiques temporelles. La séquence temporelle d'états d'une proposition atomique peut être

mappée dans un bitmap où la valeur de chaque bit est 0 ou 1, ce qui correspond à juste titre à la valeur de type booléen d'états de logiques temporelles. Nous avons catégorisé les opérateurs habituels LTL définis dans Huth et Ryan (2004) en trois groupes : les opérateurs de logiques propositionnelles, les opérateurs de logiques temporelles unaires et les opérateurs de logiques temporelles binaires. Chaque groupe avait ses caractéristiques et ses difficultés, en particulier les opérateurs de logiques temporelles binaires qui doivent énumérer deux bitmaps en même temps et prendre soin de plus de conditions que les autres groupes.

Dans la dernière étape, nous avons mis en œuvre notre solution avec un langage de programmation informatique populaire. Une interface de l'opération de bitmap a été extraite afin d'adapter les algorithmes de compression bitmap qui ont tous été mis en œuvre dans les librairies open source. Après le travail de programmation, une référence approfondie a été faite pour observer la vitesse de traitement sans compression ainsi que la performance de la vitesse et de l'espace avec les algorithmes de compression bitmaps.

1.4 ORGANISATION DU MÉMOIRE

Ce mémoire comprend cinq chapitres. Le contenu de chaque chapitre est le suivant.

Le premier chapitre introduit l'arrière-plan du mémoire, présente les tâches, décrit les méthodes appliquées dans la recherche et à la fin précise la structure du mémoire.

Le chapitre deux nous met au courant de la connaissance pertinente de la vérification de l'exécution, des logiques temporelles linéaires et introduit certains systèmes de la vérification de l'exécution.

Le chapitre trois est l'une des contributions de cette recherche. Il présente la solution du canal de la communication unidirectionnelle constitué de codes QR vacillants. Il est en fait une

version traduite et reformatée de la publication “Real-Time Streaming Communication with Optical Codes” (Xie et al., 2016).

Le chapitre quatre est une autre contribution, la solution de calcul de formules LTL avec l’aide des bitmaps. Le chapitre détaille la définition de la cartographie, les algorithmes et les expériences. Il est basé sur l’article “Offline Evaluation of LTL Formulæ with Bitmap Manipulations”.

Enfin, le chapitre cinq conclut cette recherche avec le résumé de nos contributions et un regard fixé sur notre travail à venir.

CHAPITRE 2

REVUE DE LA VÉRIFICATION DE L'EXÉCUTION ET DES TRAVAUX CONNEXES

Dans ce chapitre, on revoit tout d'abord la vérification de l'exécution, y compris son histoire, sa définition et la comparaison avec d'autres techniques de vérification. D'autre part, la logique temporelle linéaire, en tant que formalisme de spécification commune de la vérification de l'exécution, est présentée avec la syntaxe, les opérateurs et la sémantique. Enfin, quelques cadres de la vérification de l'exécution bien connus sont présents, ainsi que d'une simple comparaison entre eux.

2.1 VÉRIFICATION DE L'EXÉCUTION

La vérification et la validation de logiciels, étant considéré comme un aspect important de la gestion de projet et de l'ingénierie du logiciel, est le processus pour utiliser diverses techniques nécessaires pour détecter les violations ou les satisfactions et d'évaluer la qualité des logiciels et la performance d'un système (IEEE, 2012).

2.1.1 CONCEPTS FONDAMENTAUX

Il y a normalement deux types de techniques de vérification : analyse statique et dynamique. Certains techniques traditionnelles bien connues d'analyse statique sont la vérification de modèles (Clarke et al., 1999) et la démonstration de théorèmes (Heisel et al., 1990). L'analyse statique est généralement appliquée pour vérifier les comportements d'un système avant son exécution, mais ces techniques ont des lacunes naturelles. Par exemple, la vérification de modèles ne peut pas fonctionner sur un système dont la taille ou le nombre d'états pourraient se développer au-delà de la capacité de puissance de calcul en raison de la "state-explosion problem".

L'analyse dynamique est destinée de surveiller les systèmes en cours d'exécution. Bien que parfois le résultat pourrait être de faux négatifs à cause de son incomplétude, celle-ci permet aux techniques d'analyse dynamique de briser la limitation de méthodes statiques et devenir ainsi les méthodes complémentaires de vérification. (Falcone et al., 2013)

la vérification de l'exécution est une sorte de technique de vérification sur la base de l'analyse dynamique. En 2001, le Runtime Verification workshop [1] a été fondée, comme la terminologie "runtime verification" qui a été officiellement établie (Wikipedia, 2016). Elle est une technique relativement nouvelle qui est légère et qui vise à compléter d'autres techniques traditionnelles de vérification comme "model checking" et "testing".

Leucker et Schallhart (2009) définissent la vérification de l'exécution comme suit :

> "Runtime verification is the discipline of computer science that deals with the study, development, and application of those verification techniques that allow checking whether a run of a system under scrutiny satisfies or violates a given

1. http ://www.runtime-verification.org/

correctness property."

Normalement, lorsqu'une violation est observée, la vérification de l'exécution ne résout pas le programme détecté en soi, mais son résultat est un guide et une base importante pour d'autre composants dans le même système pour régler le problème.

Leucker et Schallhart (2009) définissent également un *run* d'un système en tant qu'une séquence de traces infinies du système, et une *exécution* d'un système comme une trace finie et aussi un *préfixe fini* d'un *run*. Le travail de la vérification de l'exécution se concentre principalement sur l'analyse d'*exécutions* qui sont effectuées par les *moniteurs*.

Un *moniteur* est une procédure de décision générée (ou "synthétisée") à partir de l'une des propriétés formelles qui doit être vérifiée par rapport à l'exécution du système donné. Lors de la vérification, un *moniteur* énumère les traces finies d'un *exécution*, vérifie si elles satisfont aux propriétés d'exactitude et produit un *verdict* comme le résultat. Un *verdict*, qui est normalement une valeur de vérité, indique la satisfaction de la propriété par rapport aux événements recueillis.

Un verdict dans la plupart des cas simples peut être normalement exprimé comme vrai/faux, oui / non ou 1/0, selon le contexte. Mais en réalité, de nombreux systèmes de la vérification de l'exécution doivent introduire d'autres valeurs pour fournir un résultat plus précis. Par exemple, grâce à l'incomplétude du système de la vérification de l'exécution, un verdict ne peut pas être facilement émis lorsque le moniteur a besoin de plus d'événements successifs, donc une valeur *inconcluante* (écrite comme " ?") est introduite pour indiquer l'état présent du système surveillé. (Falcone et al., 2013)

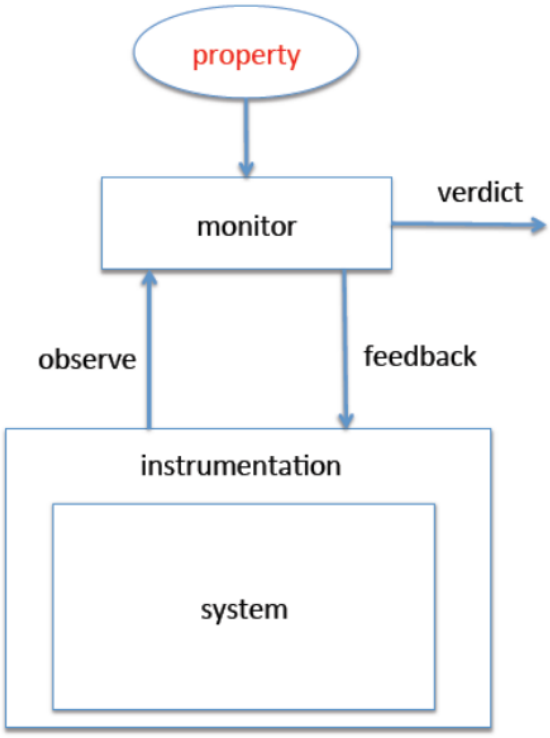


Figure 2.1: Processus de la vérification de l'exécution (de Falcone et al. (2013))

2.1.2 PROCÉDURE

La figure 2.1 décrit le processus d'un système typique de la vérification de l'exécution qui contient les quatre étapes suivantes (Falcone et al., 2013) :

1. *Synthèse du moniteur* : Un moniteur est synthétisé à partir d'une propriété.
2. *Instrumentation du système* : Les instruments supplémentaires sont intégrés au système sous surveillance afin de générer les événements pour le *moniteur*.
3. *Exécution* : Le système est exécuté et commence à générer les événements et les envoie au moniteur.
4. *Analyse et réponse* : Le moniteur analyse les événements collectés, émet un *verdict* et envoie des informations supplémentaires, c.-à-d. des *commentaires*, au système si

nécessaire.

Les moniteurs peuvent être classés en plusieurs modes d'aspects différents (Chen et Roşu, 2007) :

— *online/offline* dépend du moment où les moniteurs et le système fonctionnent : *Online* s'ils travaillent en même temps, et *offline* si les moniteurs commencent à travailler après la fin de l'exécution du système.

— *inline/outline* dépend de l'endroit où les moniteurs sont exécutés : *Inline* si les moniteurs sont embarqués dans le système et *outline* si les moniteurs fonctionnent tout seuls en recevant les traces d'événements du système par certaines méthodes, par exemple par un système de fichiers ou par un signal sans fil.

— *violation/validation* est déterminé par la spécification du verdict.

D'après les définitions des modes, on peut voir qu'un moniteur travaillant en mode *offline* travaille également en mode *outline* et le mode *inline* implique le mode *online*.

2.1.3 COMPARAISON AVEC D'AUTRES TECHNIQUES

En comparant avec la vérification de modèle (Clarke et al., 1999) qui vise à vérifier les systèmes d'états finis, on voit que les méthodes de génération de moniteurs dans la vérification de l'exécution et de la génération d'automates dans la vérification de modèle ont des similitudes. Cependant, alors que la vérification de modèle traite principalement les traces infinies, la vérification de l'exécution ne traite que les traces finies, c.-à-d. les *exécutions*. Pour cette raison, les moniteurs de la vérification de l'exécution travaillant en mode *online* doivent être en mesure d'accepter les traces supplémentaires.

Une autre différence importante entre la vérification de modèle et la vérification de l'exécution est que, contrairement à la vérification de modèle qui vérifie si toutes les *exécutions* d'un

système satisfont une propriété d'exactitude, la vérification de l'exécution est intéressée uniquement par le fait qu'il y a ou non une *exécution* qui appartient à un ensemble d'exécutions valides. En outre, la vérification de l'exécution requiert seulement l'analyse des événements observés d'un système donné, sans avoir à regarder ses informations internes, mais dans la vérification de modèle, le modèle approprié du système doit être reconnu afin de préparer chaque exécution possible avant l'exécution du système. (Leucker et Schallhart, 2009)

Le test du logiciel (Broy et al., 2005) est une autre technique de vérification. Elle est un processus de l'exécution d'un programme avec un ensemble fini de séquences entrée-sortie qui est nommé "la suite de tests". En comparant avec la vérification de l'exécution, les suites de test sont définies directement, à la différence des propriétés de la vérification de l'exécution qui sont générées à partir des spécifications de formalisme. En outre, "le test exhaustif" qui est une méthode courante dans le test du logiciel, n'est normalement pas une option de la vérification de l'exécution.

2.2 LOGIQUE TEMPORELLE LINÉAIRE

Dans la vérification de l'exécution, un moniteur est traduit à partir d'une propriété d'exactitude, et les propriétés d'exactitude sont spécifiées dans les logiques temporelles en temps linéaire, telles que la logique temporelle linéaire.

La logique temporelle est une extension de la logique classique, et elle fournit un langage pratique avec les expressions des propriétés pour raisonner sur le changement des états en termes de temps. Bien qu'il y ait beaucoup de logiques temporelles différentes qui sont inventées pour satisfaire aux diverses exigences, les logiques temporelles sont normalement classées par le temps qu'il soit linéaire ou de branchement. La logique temporelle avec le temps linéaire est appelé *Logique Temporelle Linéaire* (LTL), qui a d'abord été proposée par

Pnueli (1977). le temps dans la LTL est transformé en une séquence d'états qui s'étendent vers le futur infini. La séquence d'états est un *chemin* de calcul. (Clarke et al., 1999; Huth et Ryan, 2004)

Leucker et Schallhart (2009) résument la LTL comme une logique temporelle en temps linéaire qui est bien acceptée et utilisée pour spécifier les propriétés de traces infinies. Néanmoins, dans la vérification de l'exécution, la LTL est employée pour vérifier les exécutions finies.

2.2.1 SYNTAXE

Une formule LTL bien formée consiste en un ensemble fini de propositions atomiques, des opérateurs booléens $\neg, \wedge, \vee, \rightarrow$ et des opérateurs de logiques temporelles **F**(future), **G**(global), **X**(next), **U**(until), **W**(weak-until) et **R**(release). Elle peut être représentée sous la forme Backus Naur comme suit (Huth et Ryan, 2004) :

$$\begin{aligned}\phi ::= &\top|\bot|p|(\neg\phi)|(\phi\wedge\phi)|(\phi\vee\phi)|(\phi\rightarrow\phi)\\ &|(\mathbf{X}\,\phi)|(\mathbf{F}\,\phi)|(\mathbf{G}\,\phi)|(\phi\ \mathbf{U}\ \phi)|(\phi\ \mathbf{W}\ \phi)|(\phi\ \mathbf{R}\ \phi)\end{aligned} \tag{2.1}$$

2.2.2 SÉMANTIQUES

Pour une séquence d'états $s_0, s_1, s_2, ..., s_i, s_{i+1}, ...$ où s_{i+1} est un état futur de s_i, on définit un chemin avec $\pi^i = s_i \rightarrow s_{i+1} \rightarrow ...$ où i est le premier état dans ce chemin. Étant donné que $\pi(i)$ est l'ensemble de propositions atomiques qui sont vraies au i-ème état, le fait qu'un chemin π^i satisfait ou non une formule LTL est définie comme suit (Rozier, 2011) :

— $\pi^i \models \top$ (2.2)

— $\pi^i \not\models \bot$ (2.3)

— $\pi^i \vDash p \iff p \in \pi(i)$ (2.4)

— $\pi^i \vDash \neg\psi \iff \pi^i \nvDash \psi$ (2.5)

— $\pi^i \vDash \psi \wedge \varphi \iff \pi^i \vDash \psi$ et $\pi^i \vDash \varphi$ (2.6)

— $\pi^i \vDash \psi \vee \varphi \iff \pi^i \vDash \psi$ ou $\pi^i \vDash \varphi$ (2.7)

— $\pi^i \vDash \psi \rightarrow \varphi \iff \pi^i \vDash \varphi$ chaque fois que $\pi^i \vDash \psi$ (2.8)

— $\pi^i \vDash \mathbf{X}\ \psi \iff \pi^{i+1} \vDash \psi$ (2.9)

— $\pi^i \vDash \mathbf{G}\ \psi \iff \forall j \geq i, \pi^j \vDash \psi$ (2.10)

— $\pi^i \vDash \mathbf{F}\ \psi \iff \exists j \geq i, \pi^j \vDash \psi$ (2.11)

— $\pi^i \vDash \psi\ \mathbf{U}\ \varphi \iff \exists j \geq i, \pi^j \vDash \varphi$ et $\forall k, i \leq k < j, \pi^k \vDash \psi$ (2.12)

— $\pi^i \vDash \psi\ \mathbf{W}\ \varphi \iff$ soit $\exists j \geq i, \pi^j \vDash \varphi$ et $\forall k, i \leq k < j, \pi^k \vDash \psi$,
soit $\forall k \geq i, \pi^k \vDash \psi$ (2.13)

— $\pi^i \vDash \psi\ \mathbf{R}\ \varphi \iff$ soit $\exists j \geq i, \pi^j \vDash \psi$ et $\forall k, i \leq k \leq j, \pi^k \vDash \varphi$,
soit $\forall k \geq i, \pi^k \vDash \varphi$ (2.14)

Les formules 2.2 et 2.3 suggèrent que les états dans le chemin π^i devraient toujours être vrais ou faux.

Dans la formule 2.4, p est une proposition atomique appartenant à l'ensemble fini de propositions atomiques de la LTL, et cette formule demande de vérifier seulement le i-ème état.

Les formules 2.5—2.8 sont les opérateurs booléens de la logique propositionnelle qui respectent les règles du tableau 2.1.

Les formules 2.9, 2.11 et 2.10 sont les conjonctions unaires de logiques temporelles . L'opérateur **X** signifie "la prochaine fois" et il saute le i-ème état et évalue le chemin π^{i+1}. L'opérateur **F** signifie "parfois dans l'avenir" qui exige qu'à partir du i-ème état, une propriété reste valide

ψ	φ	$\neg\psi$	$\psi \wedge \varphi$	$\psi \vee \varphi$	$\psi \rightarrow \varphi$
Vrai	Vrai	Faux	Vrai	Vrai	Vrai
Vrai	Faux	Faux	Faux	Vrai	Faux
Faux	Vrai	Vrai	Faux	Vrai	Vrai
Faux	Faux	Vrai	Faux	Faux	Vrai

Tableau 2.1: La table de vérité des opérateurs booléens de la logique propositionnelle

dans un état futur sur le chemin. Et l'opérateur **G** ("globalement" ou "toujours") indique qu'une propriété devrait reste valide dans chaque état depuis la *i*-ème état jusqu'à la fin ou le futur infini.

Les formules 2.12, 2.13 et 2.14 sont les opérateurs binaires de logiques temporelles. L'opérateur **U** est l'abréviation de "until". La formule ψ **U** φ reste valide si φ reste valide à un état futur sur le chemin, et avant cet état, la propriété ψ reste valide à chaque état. L'opérateur **W** est une version faible de l'opérateur **U**, sauf que pour la formule ψ **W** φ, φ n'a pas besoin de rester valide à terme dans un état futur. L'opérateur **R**, qui signifie "libérer", est en fait une logique duale de l'opérateur **U**, c.-à-d. ψ **U** $\varphi \equiv \neg(\neg\psi$ **R** $\neg\varphi)$. L'opérateur **R** exige que pour la formule ψ **R** φ, la propriété φ devrait continuellement rester valide jusqu'à ce que ψ devienne valide et ψ n'a pas besoin de rester valide à terme.

Il convient de noter que dans la LTL, les logiques à deux valeurs pourraient donner un résultat prématuré qui soit vrai ou faux. Tel que mentionné précédemment, la LTL est définie pour travailler avec les traces infinies tandis que le monitoring de la vérification de l'exécution est seulement capable de traiter les traces finies, ce qui pourrait conduire à un conflit, en particulier dans un système en cours d'exécution. Par exemple, **F** ψ indique que ψ devraient rester valide dans un état futur. Dans un système actif, tant que ψ ne reste pas valide dans les états observés, les résultats de la formule sont toujours $faux$, mais si ψ devient valide dans l'observation suivante, les résultats précédents deviendront corrompus et obsolètes. Par conséquent, Bauer et al. (2006) ont proposé la logique à trois valeurs (LTL_3) qui introduit une nouvelle valeur

inconcluante pour les cas où la propriété ne peut pas être évaluée immédiatement.

2.2.3 DIVERSES LOGIQUES TEMPORELLES

Metric Temporal Logic

Metric Temporal Logic (MTL) (Chang et al., 1994) a été inventée pour raisonner sur les propriétés en temps réel. Pour préciser le temps avec exactitude, MTL coupe le temps en morceaux numérotés qui sont également appelés les modules de transition chronométrés, et emploie des opérateurs de limites pour contraindre les opérateurs de logiques temporelles. Ses formules sont définies comme suit :

$$\phi ::= \bot | p | (\phi \rightarrow \phi) | (\bigcirc_{\sim c} \phi) | (\ominus_{\sim c} \phi) | (\phi \; U_{\sim c} \; \phi) | (\phi \; S_{\sim c} \; \phi)$$
$$\text{où } \sim \in \{<, =, >, \equiv_d\} \text{ et } c \geq 0, d \geq 2$$

$\bigcirc_{\sim c}\phi$ signifie “Suivant”, $\ominus_{\sim c}\phi$ signifie “Précédent”, $\phi \; U_{\sim c} \; \phi$ signifie “Jusqu’à” et $\phi \; S_{\sim c} \; \phi$ signifie “Depuis” (Chang et al., 1994). Étant donné que T_i indique le temps du i-ème état du chemin π^i, le fait qu’une formule reste ou non valide à la position j du chemin π est défini

comme suit (on ignore les opérateurs propositionnels ici) :

$$\begin{aligned}
\pi^i \vDash \bigcirc_{\sim c} \psi &\iff \pi^{i+1} \vDash \psi \text{ et } T_{i+1} - T_i \sim c \\
\pi^i \vDash \ominus_{\sim c} \psi &\iff i \geq 1 \text{ et } \pi^{i-1} \vDash \psi \text{ et } T_i - T_{i-1} \sim c \\
\pi^i \vDash \psi \, U_{\sim c} \, \varphi &\iff \exists j \text{ où } i \leq j, \pi^j \vDash \varphi \\
&\qquad \text{et } T_k - T_j \sim c, et \forall k \text{ où } i \leq k < j, \pi^k \vDash \psi \\
\pi^i \vDash \psi \, S_{\sim c} \, \varphi &\iff \exists j \text{ où } 0 \leq j \leq i, \pi^j \vDash \varphi \\
&\qquad \text{et } T_j - T_k \sim c, et \forall k \text{ où } j < k \leq j, \pi^k \vDash \psi
\end{aligned}$$

Past Time LTL

Considérant que dans la dernière partie la LTL est définie pour vérifier les états futurs, Past Time LTL (ptLTL) vise à vérifier les états dans le passé. Les formules de la ptLTL sont définies comme suit (Havelund et Roşu, 2004) :

$$\begin{aligned}
\phi ::= &\top | \bot | p | (\neg \phi) | (\phi \wedge \phi) | (\phi \vee \phi) | (\phi \rightarrow \phi) \\
&| (\odot \phi) | (\diamond \phi) | (\boxdot \phi) | (\phi \; S_s \; \phi) | (\phi \; S_w \; \phi) \\
&| (\uparrow \phi) | (\downarrow \phi) | [\phi, \phi)_s | [\phi, \phi)_w
\end{aligned}$$

Comme on le voit dans la définition de formules, la ptLTL conserve plusieurs opérateurs fondamentaux comme la LTL et introduit cinq opérateurs du temps passé et quatre opérateurs de monitoring.

Les cinq opérateurs du temps passé sont $\odot$ qui signifie “précédent”, $\diamond$ “finalement dans le

passé", $\boxdot$ "toujours dans le passé", S_s "intervalle fort" et S_w "intervalle faible".

Les opérateurs de monitoring $\uparrow\downarrow [,)_s[,)_w$ désignent respectivement "le début", "la fin", " intervalles fort" et "intervalle faible".

Les sémantiques des opérateurs temporels sont décrites comme suit, dans la même forme que la dernière section :

$$
\begin{aligned}
\pi^i \vDash \odot\psi &\iff \text{si } i > 0, \pi^{i-1} \vDash \psi, \textit{ou} \text{ si } i = 0, \pi^0 \vDash \psi \\
\pi^i \vDash \diamond\psi &\iff i > 0 \text{ et } \exists j \text{ où } 0 \leq j \leq i, \pi^j \vDash \psi \\
\pi^i \vDash \boxdot\psi &\iff i > 0 \text{ et } \forall j \text{ où } 0 \leq j \leq i, \pi^j \vDash \psi \\
\pi^i \vDash \psi\ S_s\ \varphi &\iff \exists 0 \leq j \leq i, \pi^j \vDash \varphi \text{ et } \forall k, j < k \leq i, \pi^k \vDash \psi \\
\pi^i \vDash \psi\ S_w\ \varphi &\iff \pi^i \vDash \psi\ S_s\ \varphi \text{ ou } \pi^i \vDash \boxdot\psi \\
\pi^i \vDash \uparrow \psi &\iff \pi^i \vDash \psi \text{ et } \pi^{i-1} \nvDash \psi \\
\pi^i \vDash \downarrow \psi &\iff \pi^i \nvDash \psi \text{ et } \pi^{i-1} \vDash \psi \\
\pi^i \vDash [\psi, \varphi)_s &\iff \exists j \text{ où } 0 \leq j \leq i, \pi^j \vDash \psi, \text{ et } \forall k \text{ où } j \leq k \leq i, \pi^k \nvDash \varphi \\
\pi^i \vDash [\psi, \varphi)_w &\iff \pi^i \vDash [\psi, \varphi)_s \text{ et } \pi^i \vDash \boxdot\neg\varphi
\end{aligned}
$$

EAGLE

EAGLE (Barringer et al., 2004) est une logique temporelle de monitoring de traces finies supportant les équations paramétrées en combinant les sémantiques de points fixes minimales/maximales avec des opérateurs temporels.

La vérification de l'exécution en mode *online* requiert l'acceptation de traces incrémentales,

ce qui signifie qu'il y a des limites possibles entre les traces. Les règles de points fixes minimales/maximales ont été conçues pour résoudre ce problème. Avant d'évaluer la prochaine trace, les équations avec les règles maximales sont nécessaires pour être toujours valides et celles avec les règles minimales nécessitent seulement d'être éventuellement valides.

2.3 CADRE DE LA VÉRIFICATION DE L'EXÉCUTION

Dans la vérification de l'exécution, les moniteurs sont générés à partir de spécifications formelles par les cadres de la vérification de l'exécution. Il y a quatre modes de monitoring : *offline*, *online*, *inline*, et *outline* comme nous en avons discuté antérieurement dans cette section. De nombreux cadres et systèmes utilisant des variantes ou des extension de la LTL ont été proposés, comme le montre le tableau 2.2.

Non	Logique	Mode
JPax(Havelund et Rosu, 2001)	LTL & Past-time LTL	outline
JavaMaC(Kim et al., 2004)	Past-time LTL	outline
Hawk (d'Amorim et Havelund, 2005)	Hawk	outline
Temporal Rover(Drusinsky, 2000)	LTL & MTL	outline
MOP (Chen et Roşu, 2007)	Divers	inline/outline/offline

Tableau 2.2: Cadre de la vérification de l'exécution

Java PathExplorer (JPaX) (Havelund et Rosu, 2001) est un système de la vérification de l'exécution en ligne en vue de surveiller les traces d'exécution de programmes Java. Il dispose de trois modules (montré dans la figure 2.2) : un module d'instrumentation, un module d'observation et un module d'interconnexion. Le programme travaillant avec le module d'instrumentation envoie les traces nécessaires d'événements au module d'interconnexion, qui transmet ensuite les traces au module d'observation qui fonctionne éventuellement sur un autre ordinateur. Le module d'instrumentation est mis en fonction par un script spécifié par l'utilisateur en Java ou en Maude qui est destiné pour la spécification du monitoring de

l'exécution.

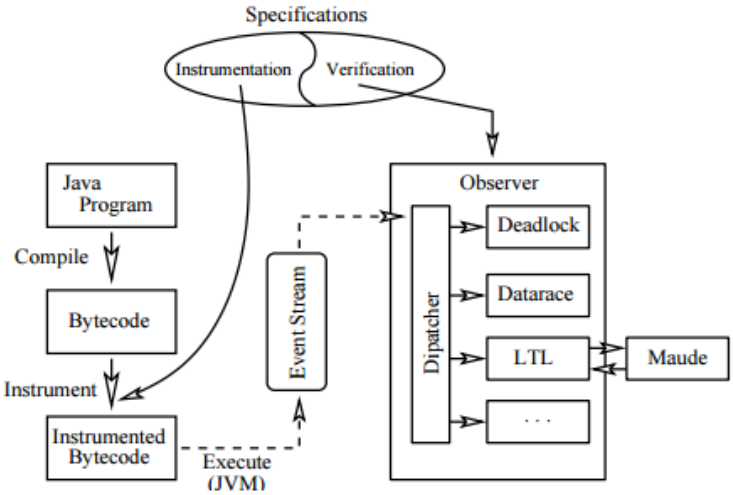


Figure 2.2: L'architecture de JPaX (Havelund et Rosu, 2001)

JavaMaC (Kim et al., 2004) est un "système d'assurance d'exécution" pour les programmes Java tandis que Mac signifie monitoring et vérification. Son architecture est représentée dans la figure 2.3. Deux langues de définition sont proposées : l'une pour les spécifications de haut niveau qui spécifie les propriétés requises, l'autre pour la spécification de bas niveau qui définit les événements et les conditions. Pendant la préparation, un "filtre" et un "reconnaisseur d'événement" qui sont utilisés pour recueillir les traces d'événements nécessaires, sont générés à partir de la spécification de bas niveau, et un "vérificateur d'exécution" est généré à partir de la spécification de haut niveau. Lors de l'exécution du programme cible, les événements collectés par le "filtre" et le "reconnaisseur d'événement" sont envoyés au "vérificateur d'exécution" qui est alors responsable pour les travaux de la vérification de l'exécution.

d'Amorim et Havelund (2005) présentent une logique nommée HAWK et ses outils de programmes Java pour la vérification de l'exécution. HAWK est en fait construite sur EAGLE, une autre logique temporelle qui est considérée comme plus expressive (Barringer et al., 2004).

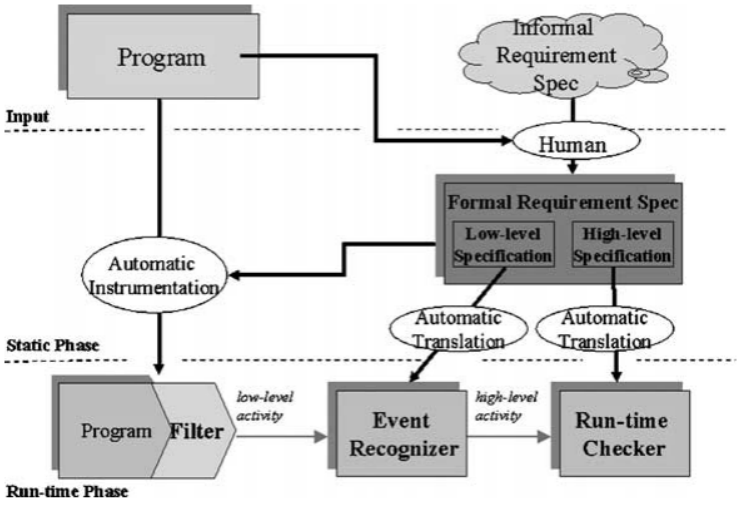


Figure 2.3: L'architecture de MaC (Kim et al., 2004)

Bien que HAWK soit basée sur les événements en contraste avec EAGLE qui est basée sur les états, les spécifications de HAWK sont converties aux moniteurs d'EAGLE. Comme le décrit la figure 2.4, pendant l'exécution du programme, l'état d'EAGLE est mis à jour par le programme instrumenté qui notifie alors l'observateur d'EAGLE. Après cela, l'observateur évalue la formule dans l'état actuel pour produire un résultat.

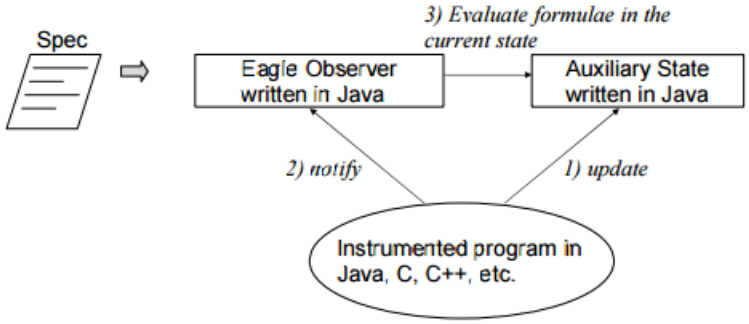


Figure 2.4: L'architecture d'Eagle (d'Amorim et Havelund, 2005)

Temporal Rover (Drusinsky, 2000) est un outil commercial de la vérification de l'exécution basé sur LTL et MTL. Le code de spécification de Temporal Rover est inséré dans le code source Java, C, C ++ ou HDL, puis converti en un fichier source compilable du langage

correspondant. Un système de la vérification de l'exécution de Temporal-Rover a normalement deux parties : l'hôte et la cible. L'hôte est responsable de la vérification alors que la cible effectue le calcul de formules propositionnelles et renvoie les résultats à l'hôte via le port série, RPC ou un autre protocole configurable.

Chacun des cadres évoqués ci-dessus utilise une spécification formalisme différente, qui suggère qu'il n'existe pas une seule spécification formalisme général pouvant servir tous les objectifs. Pour être plus expressif et générique, Chen et Roşu (2007) ont introduit les "logic-plugins" personnalisables et extensibles dans leur cadre de l'exécution MOP et ont conçu son architecture qui est représentée à la figure 2.5 avec deux couches : l'une est appelée "language clients" qui soutient différents langages de programmation, tandis que l'autre est nommée "logic repository" qui comprend et gère divers "logic-plugins" pour soutenir différents formalismes de spécification, tels que : Linear Temporal Logic (LTL), Finite State Machines (FSM), Extended Regular Expressions (ERE), Context Free Grammars (CFG) and String Rewriting Systems (SRS).

Outre les cadres présentés ci-dessus, il y a aussi beaucoup d'autres cadres inventés pour leurs exigences correspondantes ou leur logiques temporelles spécifiques. En comparant ces cadres, on peut voir qu'ils ont leurs spécialités aussi bien qu'ils partagent des caractéristiques communes. Par exemple, presque tous les cadres appuient le mode *online* de monitoring, le langage de programmation Java et la communication du réseau, peut-être parce que ces caractéristiques sont les plus populaires exigées par les industries. Comme Temporal Rover est un cadre commercial, il doit donc prendre en charge plus de langages de programmation et fournir plus d'options de collecte de données pour son expansion commerciale. MOP est conçue pour être extrêmement générale, et ainsi la plupart des composants peuvent être échangés ou séparément optimisés.

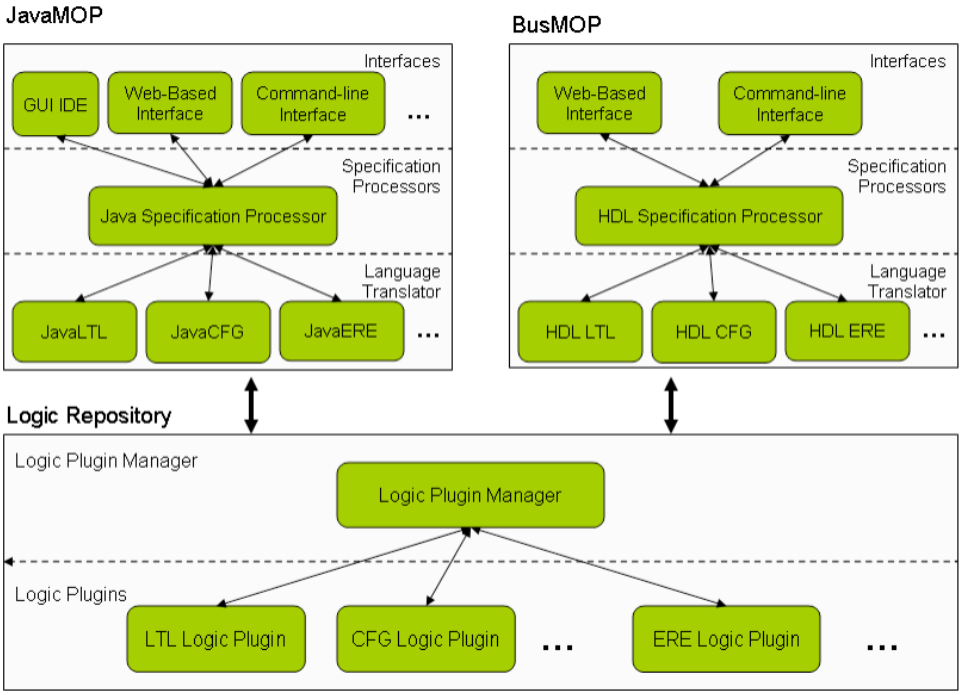


Figure 2.5: L'architecture de MOP (Chen et Roşu, 2007)

CHAPITRE 3

COMMUNICATION EN STREAMING ET EN TEMPS RÉEL AVEC LES CODES OPTIQUES

Ce chapitre présente une version reformatée et traduite d'un article publié en 2016 dans IEEE Access, v.4, p.284-298 par K. Xie, S. Gaboury et S. Hallé.

3.1 INTRODUCTION

La communication sans fil est une technologie qui permet à deux ou plusieurs pairs de communiquer sans câbles électriques ou conducteurs (Tse et Viswanath, 2005). Alors que la majorité des technologies de communication sans fil utilise les ondes radio comme leurs milieux de transmission, quelques autres utilisent la lumière, en particulier dans les situations où la technologie radio est difficile à fonctionner. La communication optique en elle-même remonte à l'utilisation de drapeaux, les signaux de fumée et les lampes signalétiques pour communiquer de l'information entre deux points avec l'utilisation d'un code spécifique (Burns, 2004).

Récemment, une forme simple de communication optique, appelée *Quick Response Code* (code QR) (Denso Wave Inc., 2015), a émergé comme un raffinement de la technologie

existante de code-barres unidimensionnels. En raison de leur exactitude et leur grande capacité, les codes QR ont été utilisés dans de nombreux domaines ; les applications du traitement de tels codes ont également été portés à une variété de dispositifs, y compris les ordinateurs de bureau, les smartphones, et même les télévisions.

Cependant, jusqu'à présent les codes QR ont été utilisés pour la transmission de données statiques. En général, un code est imprimé sur un milieu physique, tel qu'une feuille de papier, et il est lu par un appareil optique (généralement une caméra) pour le décodage à un moment ultérieur. Dans ce chapitre, nous explorons l'idée de l'expansion de codes QR et les transformons en un canal dynamique de communication unidirectionnelle. Dans un tel canal, un flux de données est transmis par une *séquence de codes* ; généralement, ces codes sont continuellement générés et affichés sur un appareil, et simultanément capturés et décodés par un autre, ce qui est similaire aux autres types de technologies de communication.

Après avoir brièvement décrit dans la Section 3.2 les bases de codes QR et d'autres technologies de communication sans fil, nous nous concentrons dans la Section 3.3 sur le principe de communication de codes QR bruts. Particulièrement, nous essayons de trouver les limites intrinsèques d'un tel canal de communication, par l'analyse de l'influence de divers facteurs, tels que la densité de code, le nombre de codes affichés par seconde, etc. Les résultats d'un benchmark qui couvrent plus de cent combinaisons différentes de paramètres permettent d'extraire les conditions optimales qui minimisent le taux d'erreur dans le décodage des images, tout en maximisant la quantité de données qui peuvent être transmises par unité de temps.

Ces premiers résultats indiquent que les flux de codes QR peuvent en effet être utilisés comme un canal simple et unidirectionnel, mais que la communication sans erreur et la bande passante élevée sont plus ou moins impossibles. Par conséquent, en tant qu'un deuxième temps, nous concevons un protocole qui est approprié pour la nature spécifique de communication de

codes QR. Ce protocole, appelé BufferTannen, est décrit dans la section 3.5. Il est capable d'encapsuler les données brutes, de fournir diverses capacités de signalisation, de pouvoir représenter les données semi-structurées (telles que JSON) sous une forme binaire compacte, et prend en charge le cadrage/décadrage et le streaming de données.

Une seconde expérience révèle la robustesse de ce schéma de transmission : en utilisant notre protocole spécialement conçu, le canal de communication créé par une personne qui pointe un smartphone à bout de bras vers un écran avec les codes QR des codes QR vacillants, produit une bande passante suffisante pour transmettre l'audio à usage vocal en temps réel. La Section 3.4 présente l'environnement et les résultats des expériences de cette deuxième étape. Nous montrons aussi comment un morceau de données, coupé en plusieurs codes avec l'utilisation de BufferTannen, peut être reconstruit automatiquement par un utilisateur qui passe sa caméra sur une feuille de ces codes imprimés dans aucun ordre particulier.

Ce travail a été motivé par une application pratique dans le domaine de la vérification de l'exécution. Dans les recherches antérieures, nous avons proposé et officieusement expérimenté l'utilisation de codes optiques comme une forme de communication à couplage lâche entre un système logiciel et un moniteur externe qui reçoit les événements produits par ce système-là (Lavoie et al., 2014). Dans ce contexte, la communication par les milieux optiques assure un isolement complet entre le système et son moniteur.

Bien que l'utilisation de séquences de codes QR a été officieusement suggéré dans le passé, au mieux de notre connaissance, notre recherche est la première enquête systématique du potentiel de codes QR pour envoyer les flux de données en temps réel.

La solution que nous proposons fournit une méthode pour distribuer les données en streaming sans dispositifs dédiés de communication. Les appareils requis sont seulement une caméra commune (comme une webcam ou la caméra dans un téléphone portable) et une petite surface

plane pour afficher les codes QR séquentiels — par exemple, un écran d'ordinateur, une télévision ou un smartphone, ou même une feuille de papier.

3.2 COMMUNICATIONS SANS FIL

Cette section rappelle quelques technologies communes de communication sans fil. Malgré leur popularité et leur performance, chacune a ses propres limites et ses scénarios d'application.

3.2.1 ONDES RADIO

Le premier milieu évident de communication sans fil est grâce à l'utilisation d'ondes radio, dont le meilleur exemple est WiFi (Comer, 2008), utilisé pour la mise en réseau sans fil de la zone locale. Ses variantes sont basées sur la famille de standards IEEE 802.11, et prennent en charge la mise en réseau centralisée (routage) et décentralisée (*ad hoc*). Le Tableau 3.1 montre les standards populaires de 802.11 et une partie de leurs spécifications.

Protocole	Fréquence	Débit maximal de données physiques	Portée intérieure
802.11a	5 GHz	54 Mbps	35 m
802.11b	2.4 GHz	11 Mbps	35 m
802.11g	2.4 GHz	54 Mbps	38 m
802.11n	2.4/5 GHz	150 Mbps	70 m
802.11ac	5 GHz	866.7 Mbps	35 m

Tableau 3.1: Résumé des protocoles WiFi (Theng, 2008; Perahia et Stacey, 2013)

Un deuxième concurrent dans cette famille est Bluetooth (Comer, 2008), qui est utilisé à courte portée et généralement comme communication point à point entre les appareils. Sa portée varie d'environ 1 à 100 mètres en fonction de la classe d'énergie. Le Tableau 3.2 montre différentes versions de Bluetooth et leurs débits de données spécifiques.

Version	Débit de données	Portée
Version 1.2	1 Mbps	Classe 1 : 100 m ; Classe 2 : 10 m ; Classe 3 : 1 m
Version 2.0 + EDR	3 Mbps	
Version 3.0 + HS	24 Mbps	
Version 4.0	24 Mbps	

Tableau 3.2: Spécifications de Bluetooth (Gupta, 2013)

Enfin, ZigBee (Farahani, 2011), basé sur le standard IEEE 802.15.4, vise à mettre en œuvre une communication sans fil à courte portée avec une faible énergie et une batterie de longue durée. Le Tableau 3.3 montre ses performances dans différentes bandes de fréquences.

Bande de fréquence	Débit de données	Portée
868–870 MHz	20 kbps	10–100 m, en fonction de la puissance de sortie et de l'environnement
902–928 MHz	40 kbps	
2.4–2.4835 GHz	250 kbps	

Tableau 3.3: Spécifications de ZigBee (Lee et al., 2007)

Tous ces protocoles partagent un point commun : avant d'autoriser toute forme de communication entre deux extrémités, une certaine forme de *découverte* ou *configuration* de dispositifs est nécessaire. Ce processus est généralement achevé à travers l'établissement d'une *connexion* avec états à long terme entre les extrémités.

3.2.2 IRDA

Dans une autre famille, on trouve des technologies utilisant des ondes infrarouges au lieu du signal radio (Sarkar et al., 2007). Outre les longueurs d'onde différentes, ces technologies assouplissent généralement les exigences de l'établissement d'une connexion, et permettent une communication plus "on-the-fly" entre deux appareils. En règle générale, une extrémité d'une liaison infrarouge attend les données entrantes, tandis que périodiquement, un autre

appareil pointe au récepteur et émet les rayons infrarouges transformés à partir de petits morceaux de données sans nécessité de préavis.

Une importance particulière est le standard IrDA (Infrared Data Association) ; ses émetteurs envoient des impulsions infrarouges avec un angle de cône et une irradiance modérée alors que ses récepteurs peuvent être à moins d'un mètre ou plusieurs mètres de ceux-là, en fonction de l'énergie des émetteurs et de la position dans le cône. La communication IrDA est semi-duplex et fournit CRC de base. Le Tableau 3.4 montre plusieurs schémas IrDA et leurs débits de données dans la portée spécifique.

Schéma	Débit de données	Portée
SIR	2.4–115.2 kbps	jusqu'à un mètre
MIR	0.576–1.152 Mbps	
FIR	4 Mbps	
GigaIR	512 Mbps–1 Gbps	

Tableau 3.4: Débits de données de schémas de couche physique de IrDA (Millar et al., 1998)

3.2.3 VISIBLE LIGHT COMMUNICATION

Comme l'indique son nom, Visible Light Communication (VLC) (Komine et Nakagawa, 2004) utilise des longueurs d'onde dans la gamme visible (400-700 nm) pour communiquer les données entre les pairs — ceci est généralement réalisé par allumer et fermer rapidement une source de lumière, ce qui permet une forme de codage des données comme les codes Morse. Un récepteur (par exemple une cellule photo-électrique) pointé par la source de lumière détecte ce vacillement et le convertit en données numériques. Avec les lampes fluorescentes comme la source lumineuse, le débit de données peut atteindre 10 kbps, tandis qu'avec la technologie LED, le débit de données peut être aussi élevé que 500 Mbps. La gamme dépend surtout des spécifications différentes, mais comme la lumière ne peut pas traverser les murs et

peut également être affectée par le mauvais temps ou d'autres sources de lumière, sa portée et sa fiabilité sont limitées (Arnon, 2015).

Ce mode de communication est intrinsèquement unidirectionnel, car on ne peut pas répondre à la source de lumière, reconnaître la réception de l'information, ou demander d'une retransmission en cas de la perte de données. Par conséquent, cette technologie est aussi celle qui nécessite le moins de couplage entre un émetteur et un récepteur ; selon tous les moyens pratiques, l'émetteur n'a pas connaissance de la présence d'un récepteur, qui, de son côté, peut choisir de commencer à recevoir à tout moment. Nous verrons plus loin que cette caractéristique est également partagée avec le canal de communication de codes QR que nous essayons de concevoir.

3.3 FLUX DE CODES QR

Dans le court sondage précédent de technologies de communication sans fil sont mentionnés les codes optiques, qui sont aussi un moyen de transport de données sans nécessité d'un milieu physique. Dans cette section, nous passons en revue le concept de codes QR, et discutons de l'idée de produire les flux de données par les séquences de tels codes.

3.3.1 APERÇU DE CODES QR

Un code QR (officiellement appelé "Quick Response Code") (Denso Wave Inc., 2015) est un code-barre à deux dimensions qui stocke des données, comme le montre la figure 3.1. En comparaison avec le code-barre bien connu UPC, qui est linéaire (i.e. unidimensionnel), un code QR peut stocker plus d'informations dans une impression plus petite. [1] Le standard

1. `http://www.qrcode.com/en/`

Figure 3.1: Un code QR avec le texte “Hello world !”

Niveau de correction d’erreur	Bits de données	Caractère numérique	Caractère alphanumérique	Caractère binaire	Kanji
L	23,648	7,089	4,296	2,953	1,817
M	18,672	5,596	3,391	2,331	1,435
Q	13,328	3,993	2,420	1,663	1,024
H	10,208	3,057	1,852	1,273	784

Tableau 3.5: Capacités maximales de stockage de codes de la version 40

du code QR stipule que ces codes peuvent avoir une capacité aussi élevée que 7089 caractères numériques ou 2953 caractères à 8 bits, et qu’un code est représenté dans un tableau carré d’un maximum de 177 $\times$ 177 “pixels”, appelé *modules* (International Organization for Standardization, 2006).

La capacité d’un code QR dépend principalement de son type de données, de sa version et de son niveau de correction d’erreur. Le type de données peut être *uniquement numérique*, *alphanumérique*, *binaire*, ou *Kanji*. La version de 1 à 40, détermine les dimensions d’un code, qui varient de 21 $\times$ 21 à 177 $\times$ 177 modules. Les codes QR utilisent une forme de codage de correction d’erreur qui agit à quatre niveaux : *Low* (L), *Medium* (M), *Quartile* (Q), et *High* (H), comme il est indiqué dans le Tableau 3.5. Évidemment, comme le niveau augmente, plus de redondance est introduite dans le contenu du code, ce qui diminue sa capacité de stockage ; cependant, plus de données peuvent être restaurées si le code est sale ou endommagé. Avec les formes de détection de position incluses dans le symbole, un code QR peut être décodé en 360 degrés.

Avant de générer un code QR à partir d'un morceau de données, un générateur de code doit analyser les données d'entrée pour décider du mode et de la version les plus efficaces. Dans le codage de données, les caractères sont convertis en un flux de bits, et dans ce progrès, certains *indicateurs de mode* et *terminateurs* sont insérés pour les changements de mode. Le flux de bits est ensuite divisé en mots de code à 8-bits, et les caractères de remplissage sont nécessaires pour combler le nombre de mots de code de la version choisie. La séquence de mots de code générée est divisée en blocs selon le niveau de correction d'erreur spécifique et un mot de code de correction d'erreur est généré pour chaque bloc. Ensuite, les mots de code de chaque bloc sont entrelacés et quelques bits restants sont ajoutés selon le besoin.

Dans l'étape suivante, le générateur met les modules de mots de code dans une matrice en noir et blanc avec la forme de recherche, les séparateurs, la forme de synchronisation et les formes d'alignement ; il applique les formes de masquage, évalue et sélectionne ensuite la forme appropriée. Enfin, il génère le *format* et *l'information de version* et complète le code QR (International Organization for Standardization, 2006).

Les étapes de décodage ne sont que tout simplement l'inverse de la procédure de codage. Dans un premier temps, le code QR doit être situé et les modules noirs et blancs sont reconnus comme 0s et 1s qui forment un tableau binaire. De ce tableau binaire, le décodeur reçoit l'information du format et de la version. Avec ces informations, il peut commencer à lire les caractères et les mots de code de correction d'erreurs, puis tente de détecter et de corriger les erreurs avec les mots de code de correction d'erreurs selon le niveau de correction d'erreur approprié. Dans l'étape suivante, les mots de code de données sont divisés selon le *indicateurs de mode* et les *indicateurs de nombre de caractères*, et les caractères de données sont finalement décodés et sortis.

3.3.2 LE CAS DE COMMUNICATION DE CODES QR

Le processus précité s'applique au codage et au décodage d'un code unique contenant des données statiques. Nous enquêtons maintenant sur l'idée d'utiliser les codes QR en tant qu'un canal de communication, où les données en temps réel seraient transformées en situation réelle comme une *séquence* de codes QR, qui pourraient ensuite être optiquement capturées par un dispositif, et reconverties en flux de données d'origine à l'extrémité de réception.

L'utilisation de communication de codes QR présente plusieurs avantages dans une poignée de scénarios. Par exemple, la Marine américaine a enquêté sur l'utilisation de codes QR comme un "sémaphore numérique". La technologie proposée se concentre sur la détection de codes à basse résolution à partir de très longues distances, et souligne l'intérêt et les cas possibles d'utilisation de cette technologie dans un contexte militaire :

> "Arguably the most significant advantage of QR code LOS [line of sight] communications is the fact that they can be conducted without emitting energy in the RF spectrum. In an emissions controlled (EMCON) environment, this will provide a critical ability to communicate between ships without increasing the possibility of position detection." (Richter, 2013, p. 46)

Cependant, dans l'ouvrage cité, les codes sont considérés comme *statiques*, c'est-à-dire qu'ils ne changent pas au fil du temps pour former un flux de données, et agissent plus ou moins comme un substitut de drapeaux ou de signes. Néanmoins, l'absence de toute émission d'ondes radio dans la communication de codes QR s'avère un avantage attrayant dans certains scénarios.

Nous avons également vu dans la section précédente comment toutes les autres technologies, telles que Bluetooth ou IrDA, nécessitent un matériel spécialisé. En revanche, la communica-

tion de codes QR peut être réalisée à travers les codes imprimés sur une surface dure, ou par un dispositif capable d'afficher des images à une résolution suffisante : les écrans de télévision, les écrans d'ordinateur, les tablettes et les téléphones cellulaires. De même, la réception peut être faite par un appareil équipé d'une caméra commerciale normale. Cela peut convertir les appareils équipés de ce matériel courant en dispositifs de communication, même s'ils ne sont pas conçus à cet effet en premier lieu. On peut même imaginer les situations d'urgence dans lesquelles tous les moyens numériques de communication entre deux points ne fonctionnent pas. Si la ligne de vision peut être établie et un affichage et une caméra sont disponibles, l'utilisation de codes QR permet néanmoins de transmettre les données numériques — sans doute beaucoup plus rapidement que le manuel écriture ou transcription.

Enfin, nous avons mentionné au début comment l'utilisation d'un canal de communication optique et strictement unidirectionnelle peut également être souhaitable, même dans les situations où la communication radio ou câble est disponible. Par exemple, dans le contexte de la vérification de l'exécution, l'exécution d'un système est actuellement observée par un processus externe appelé *moniteur*. Pour empêcher le moniteur d'interférer avec l'exécution du système, il est souvent placé sur une machine séparée, avec un canal de communication qui transporte les événements du premier au second. Cependant, dans les protocoles traditionnels tels que TCP, la nature bidirectionnelle d'une connexion présente un risque trop élevé d'attaques contre le programme de monitoring. En outre, certaines configurations de logiciels sont nécessaires pour brancher le moniteur au programme : les adresses IP, les noms de tube, les ports, etc., ce qui représente trop de couplage dans de nombreux scénarios. Nous avons discuté dans le travail précédent (Lavoie et al., 2014) comment l'utilisation d'un canal de communication optique peut atténuer ces problèmes en fournissant un plus grand isolement entre le système et son moniteur.

3.3.3 ESTIMATION DE LA BANDE PASSANTE ET DU TAUX D'ERREUR

Cependant, la transmission unidirectionnelle introduit la possibilité de perte d'images pendant le procédure, en raison de la limitation des dispositifs physiques ou la vulnérabilité du logiciel. De plus, c'est impossible que l'émetteur puisse être conscient d'images manquantes à l'extrémité de réception et les renvoyer. Par conséquent, nous avons besoin d'analyser profondément cette approche pour estimer le *taux de reconnaissance* et la *bande passante de transmission* d'un tel canal de communication.

La transmission de codes utilise plusieurs paramètres : la taille de données de chaque image, le nombre d'images générées par seconde (fps) et le niveau de correction d'erreur. Tous les trois peuvent avoir un effet important de la génération de codes QR et de la bande passante résultante. Une plus grande taille de données mène à une plus haute version de symbole de codes QR et plus de modules de symboles, et avec la même taille d'image, un niveau plus élevé de correction exige plus de modules de symboles qu'un niveau moins élevé.

Comme l'émetteur ne peut pas détecter le fait que des codes soient manqués ou non par le récepteur, ce canal de communication unidirectionnel est en fait un canal avec perte, dont la bande passante actuelle peut être calculée avec le taux mesuré de reconnaissance d'images. Cela représente le nombre de bits qui sont reçus correctement.

$$bande\_passante = fps \times bits\_de\_image \times taux\_de\_reconnaissance$$

Si le récepteur craint que les images ne sont pas tout reçues, la seule façon est de s'assurer que l'émetteur envoie toutes les images plusieurs fois jusqu'à ce que le récepteur reçoit toutes les images ; alors la bande passante réelle est la suivante :

$$bande\_passante = fps \times bits\_de\_image \div nombre\_de\_fois\_de\_envoi$$

Le taux de reconnaissance est normalement déterminé par la capacité de la caméra et de l'écran, la précision de l'algorithme de reconnaissance et la complexité de codes (c.-à-d. le nombre des modules affichés). Cependant, dans la capacité de la caméra et de l'écran, si nous pouvons envoyer la même image plus d'une fois, et entre-temps la valeur de *fps* n'a pas à changer, le taux pratique de reconnaissance peut être amélioré.

$$taux\_pratique\_de\_reconnaissance = 1 - (1 - taux\_de\_reconnaissance)^{nombre\_de\_fois}$$

3.4 EXPÉRIENCES

Dans cette section, nous décrivons les expériences où nous mesurons la précision de reconnaissance de séquences de codes QR dans diverses conditions. Le but de ces expériences est triple :

1. évaluer si les données peuvent être transmises avec succès à travers la reconnaissance de séquences de codes optiques ;
2. trouver les paramètres qui maximisent la vitesse de décodage et la bande passante des données transmises ;
3. à partir de ces résultats, déterminer les caractéristiques d'un canal typique de communication de flux de codes QR.

3.4.1 PRÉPARATION DES EXPÉRIENCES

Notre installation expérimentale implique la production et l'affichage de séquences de codes QR à une extrémité, et la capture et le décodage de ces séquences à l'autre extrémité. Dans notre environnement expérimental, nous avons utilisé un écran LED de 19 pouces de Samsung comme émetteur et une webcam Logitech à haute définition comme récepteur. La caméra était mise à une distance fixe de 50 cm de l'écran. La résolution de l'écran est de 1280×1024 pixels.

La caméra était mise sur une surface stable, avec la zone de code optique correctement mise au point et couvrant tout le champ de vision. L'ordinateur utilisé dans les expériences est un ordinateur portable avec le processeur Intel Core i7-3632QM et 16 Go de mémoire. La Figure 3.2 montre l'installation utilisée dans les expériences.

Dans le développement, nous avons choisi OpenCV [2] pour capturer les images de la caméra et ZXing [3] pour générer et décoder les codes QR. Pour réduire la pression du CPU et de la mémoire de capture et de décodage, les images capturées ont été transformées en 16 niveaux de gris. Les données utilisées pour générer les codes QR étaient les caractères alphanumériques aléatoirement générés. Tout le code du benchmark est implémenté en Java et disponible gratuitement. [4]

Le décodage de codes dépend de la qualité et de la complexité de l'image capturée. Si l'image est trouble ou cassée, elle sera difficile à décoder. Et les algorithmes de reconnaissance d'images peuvent avoir la probabilité de défaillance (Adelmann et al., 2006). Par conséquent, notre première étape a pour but de mesurer la capacité des librairies de reconnaissance optique

2. `http://opencv.org/`
3. `https://github.com/zxing/zxing`
4. `http://github.com/sylvainhalle/GyroGearloose`

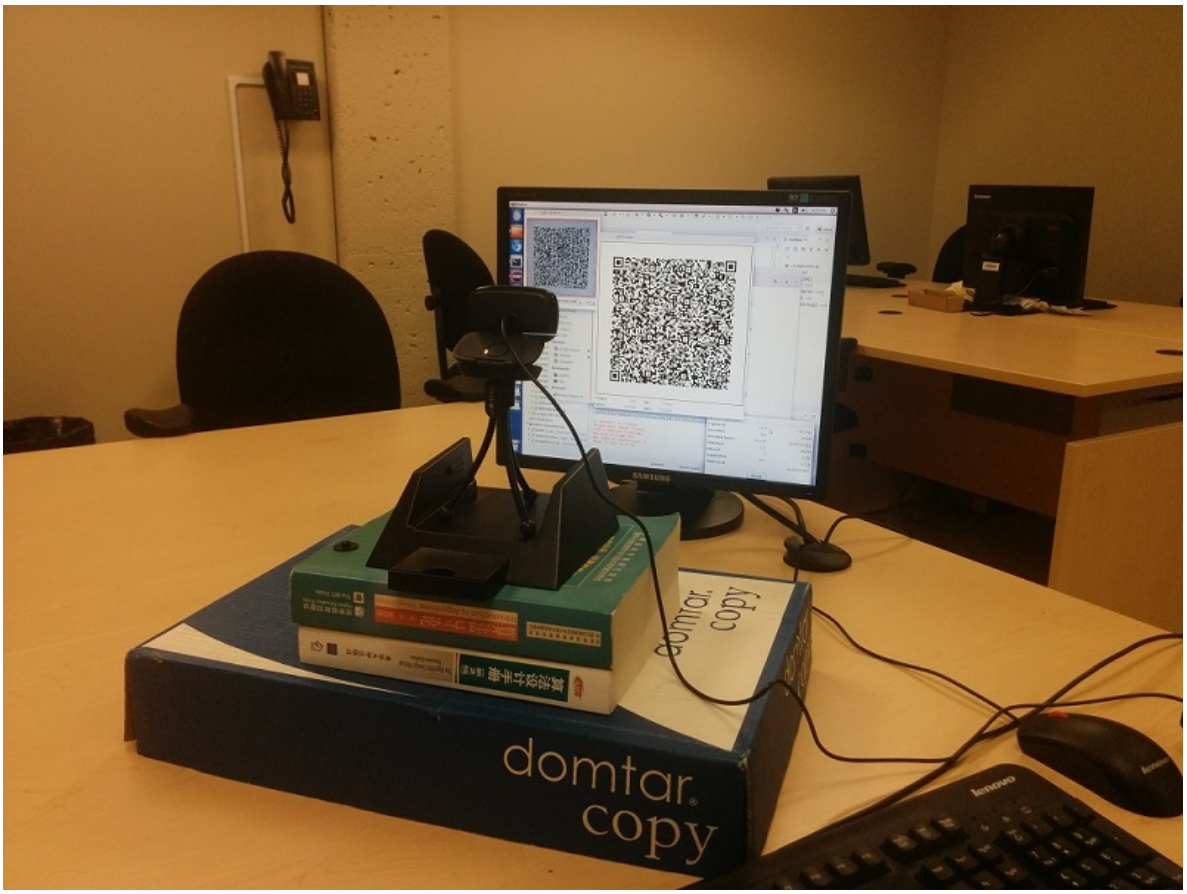


Figure 3.2: Installation expérimental pour lire les codes QR.

pour reconnaître correctement les séquences de codes, quelles que soient les données actuelles contenues dans ces codes. Les séquences de codes ont été générées par la production d'une chaîne de caractères de la forme `dddd#rrrr...`, où `dddd` est un numéro séquentiel à partir de zéro et incrémenté par un dans chaque code successifs, et `rrrr...` est une chaîne de caractères aléatoires (différent dans chaque code) assez longue pour combler le code jusqu'à sa taille maximale. Chaque test consistait à filmer la séquence de tels codes et stocker le numéro séquentiel de chaque image correctement décodée dans un fichier. Cela nous permet de calculer la fraction de tous les codes qui ont été correctement reconnus ; compte tenu de la taille de chaque code et du nombre de codes envoyés, cela permet de calculer la bande passante et le taux d'erreur de décodage.

Nos expériences ont rapidement trébuché par ce qui semble être un bogue dans de la librairie de décodage d'images ZXing. Lors de l'analyse de séquences d'images capturées par la caméra pour rechercher des erreurs de décodage, nous avons découvert que pour un nombre de fois, le décodage a échoué pendant que l'image correspondante semblait avoir aucun problème apparent (aucun trouble, cadrage juste, etc.). L'essai d'afficher encore les codes échoués à l'écran et puis d'essayer de les décoder avec la caméra n'a donné aucun succès, même après avoir changé la taille des codes, la position de la caméra, les conditions d'éclairage, etc. Le fait le plus curieux est que les codes immédiatement avant et après le code problématique ont été correctement décodées, tout en étant capturés dans les mêmes conditions. Même l'envoi de l'image "pure" du code directement à l'algorithme de décodage, sans aide de la caméra, produit une erreur de décodage.

Il semble donc la librairie ne peut pas reconnaître certains des codes qu'elle produit en soi (La Figure 3.3 montre un tel exemple). Cela indique très probablement un bogue de la librairie, qui a persisté jusqu'à la dernière version disponible au moment où ce chapitre a été écrit. Par conséquent, dans ce qui suit, le lecteur doit garder à l'esprit qu'une proportion inconnue d'erreurs de reconnaissance sont causées par ce soi-disant bug, et non par les conditions expérimentales particulières. Tel est le cas, par exemple, pour les écarts dans le taux de correction que nous observerons dans les Figures 3.4 et 3.5.

3.4.2 PARAMÈTRES DES EXPÉRIENCES

L'expérience vise à identifier la combinaison de paramètres qui permettraient de maximiser la bande passante et de minimiser le taux d'erreur de transmission de codes. Les paramètres qui ont été considérés sont les suivants.

Figure 3.3: Un code QR généré par ZXing que ZXing en soi ne peut pas décoder dans l'expérience.

Résolution de codes

Le premier paramètre est la taille de données de codes (c.-à-d. le nombre de bits de données contenues dans chaque code) et la taille physique (le nombre de pixels utilisés pour afficher le code sur l'écran). Nous avons varié la taille des données par incréments de 500 bits, de 500 à 4500 bits. Comme le montre le tableau 3.6, le plus grand code QR, qui contient 4500 bits de données en utilisant le niveau le plus élevé de correction d'erreur, occupe 101 $\times$ 101 modules. Nous avons aussi fixé la taille physique de codes à 700 $\times$ 700 pixels sur l'écran, ce qui rend

chaque module un carré d'au moins 6 × 6 pixels.

Nombre de bits de données d'entrées	Niveau de correction d'erreur	Version de symbole	Taille de symbole
500	L	3	29×29
	H	5	37×37
1000	L	5	37×37
	H	9	53×53
1500	L	6	41×41
	H	11	61×61
2000	L	8	49×49
	H	13	69×69
2500	L	9	53×53
	H	15	77×77
3000	L	10	57×57
	H	17	85×85
3500	L	11	61×61
	H	18	89×89
4000	L	12	65×65
	H	20	97×97
4500	L	13	69×69
	H	21	101×101
5800	L	19	93×93
	H	30	137×137

Tableau 3.6: Tailles d'échantillons de codes QR, selon leurs tailles de données et les niveaux de correction d'erreurs (International Organization for Standardization, 2006)

Fréquence de codes

Le deuxième paramètre expérimental que nous avons considéré est la fréquence de codes, c'est-à-dire le nombre de codes affichés par unité de temps. Nous avons d'abord choisi 2, 4, 6, 8 et 10 codes par seconde (cps), et également considéré jusqu'à 16 cps dans une phase ultérieure de l'expérience.

Niveau de correction d'erreur

Comme nous l'avons vu, les codes QR comprennent des données supplémentaires destinées à la correction d'erreurs. Nous avons donc aussi varié le niveau de correction d'erreur dans chaque expérience, en utilisant soit son réglage le plus élevé (H) ou le plus bas (L).

Résolution et fréquence de la caméra

La résolution de la caméra n'a pas été considérée comme un paramètre expérimental. Elle était fixe à sa valeur maximale, 1920×1080 pixels. De même, la fréquence d'image était fixe à 30 images par seconde. Cela correspond à la vidéo à haute définition 1080p, un réglage qui devrait être trouvé dans la plupart d'appareils récents et futurs de capture vidéo. Nous avons effectué quelques tests informels avec des basses résolutions (en baisse à 640×480), qui étaient mondialement concluants, mais qui n'ont pas été considérés pertinents de les inclure dans notre analyse détaillée.

3.4.3 RÉSULTATS DES EXPÉRIENCES

Le produit de toutes les combinaisons de tailles de codes, de niveaux d'erreur de correction et de fréquences de codes mène à un total de 90 expériences différentes. Ces expériences ont été répétées dans les trois ensembles, qui diffèrent par la façon dont les codes ont été affichés.

Affichage en une fois

Dans la première expérience, chaque code a été affiché en séquence pour une durée de $1/f$ seconde, où f est la fréquence de codes. La bande passante et le taux de décodage sont

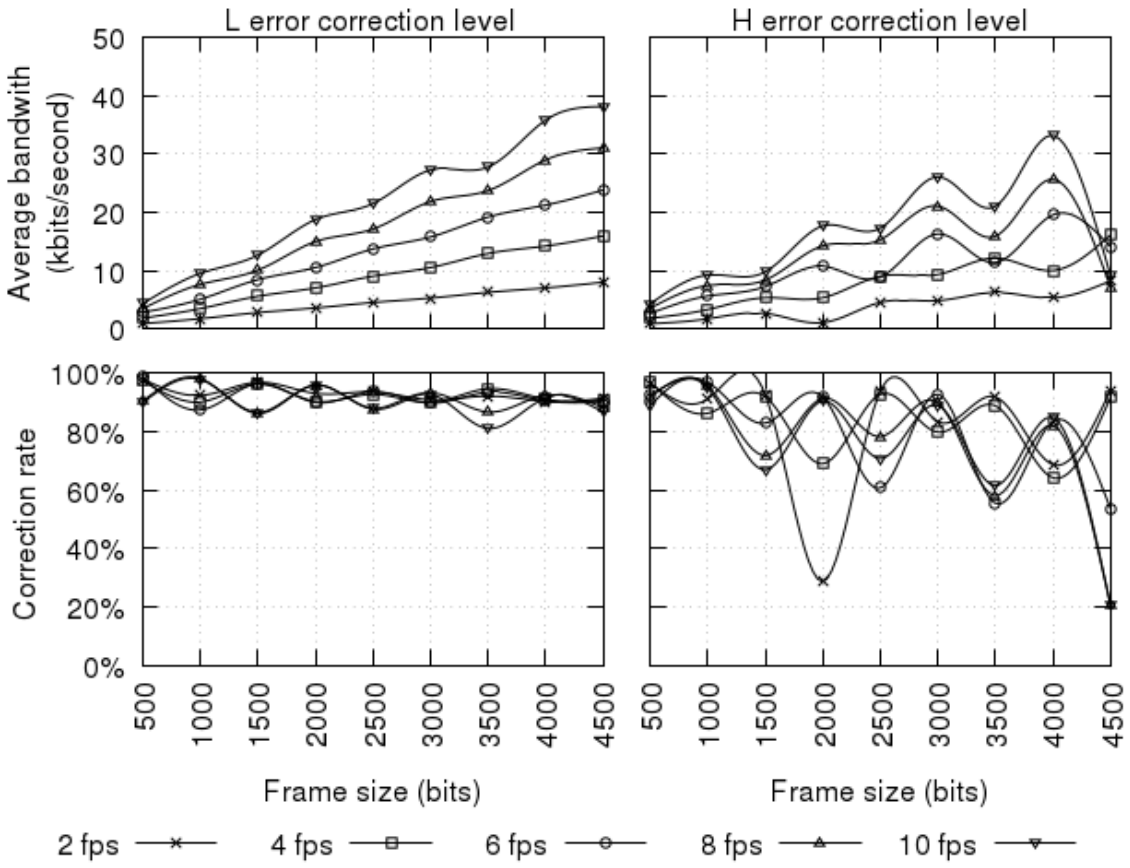


Figure 3.4: Bande passante et taux de décodage dans la première expérience

présentés dans la Figure 3.4 pour les combinaisons de tous les paramètres.

Comme on peut le voir, les taux de reconnaissance de niveaux plus élevé de correction étaient inférieurs à ceux de niveaux plus bas de correction, avec tous les autres paramètres étant égaux. Ceci peut être expliqué par le fait que la même quantité de données, portées dans un code avec un niveau plus élevé de correction, doit afficher plus de modules. Par exemple, selon le Tableau 3.6, les modules d'un code 2000 bits en niveau H, sont aussi petits que ceux d'un code de 4500 bits en niveau L. Les modules plus petits, à leur tour, augmentent la difficulté de reconnaissance par la caméra. Par conséquent, une première conclusion que l'on peut tirer est que, de façon surprenante, la bande passante effective semble être améliorée en utilisant un

bas niveau de correction d'erreur.

Avec les mêmes tailles de données et les mêmes niveaux de correction, la figure montre que le taux de reconnaissance diminue alors que la fréquence de codes augmente. Ceci peut être expliqué par le fait que, dans une fréquence plus élevée de codes, le même code occupe moins d'images de la caméra, et a donc moins de chances d'être correctement décodé dans l'une des images. En outre, la probabilité qu'un changement de code se produise au moment où une image soit prise (entraînant une image trouble qui montre une partie de deux différents codes) est également augmentée. En niveau L, la diminution est légère, tandis qu'en niveau H, la diminution est dramatique lorsque la taille du code atteint 3000 bits. Alors que la taille de données augmente, le taux de reconnaissance diminue constamment et considérablement.

Ces chiffres semblent indiquer que la configuration idéale pour le niveau L est 4500 bits et 10 fps, ce qui donne une bande passante effective de 39,0 kbps ; pour le niveau H, 4000 bits et 10 fps mène à une bande passante de 24,6 kbps.

Affichage en deux fois

Considérant que la caméra pourrait avoir manqué plusieurs images, nous avons réalisé une seconde expérience dans laquelle chaque QR code est affiché deux fois dans une petite fenêtre du temps. Par conséquent, au lieu d'afficher chaque code une fois en $1/f$ seconde, chaque code a été entrelacé avec ses codes voisins et affiché deux fois en $1/2f$ seconde chaque fois. Ceci a pour résultat le même temps total d'exposition pour chaque code, mais augmente la diversité des images capturées par la caméra.

Les résultats sont tracés sur la Figure 3.5. Ils montrent une augmentation de tous les taux de reconnaissance, qui sont maintenant tous supérieurs à 90%. Ceci, à son tour, augmente la

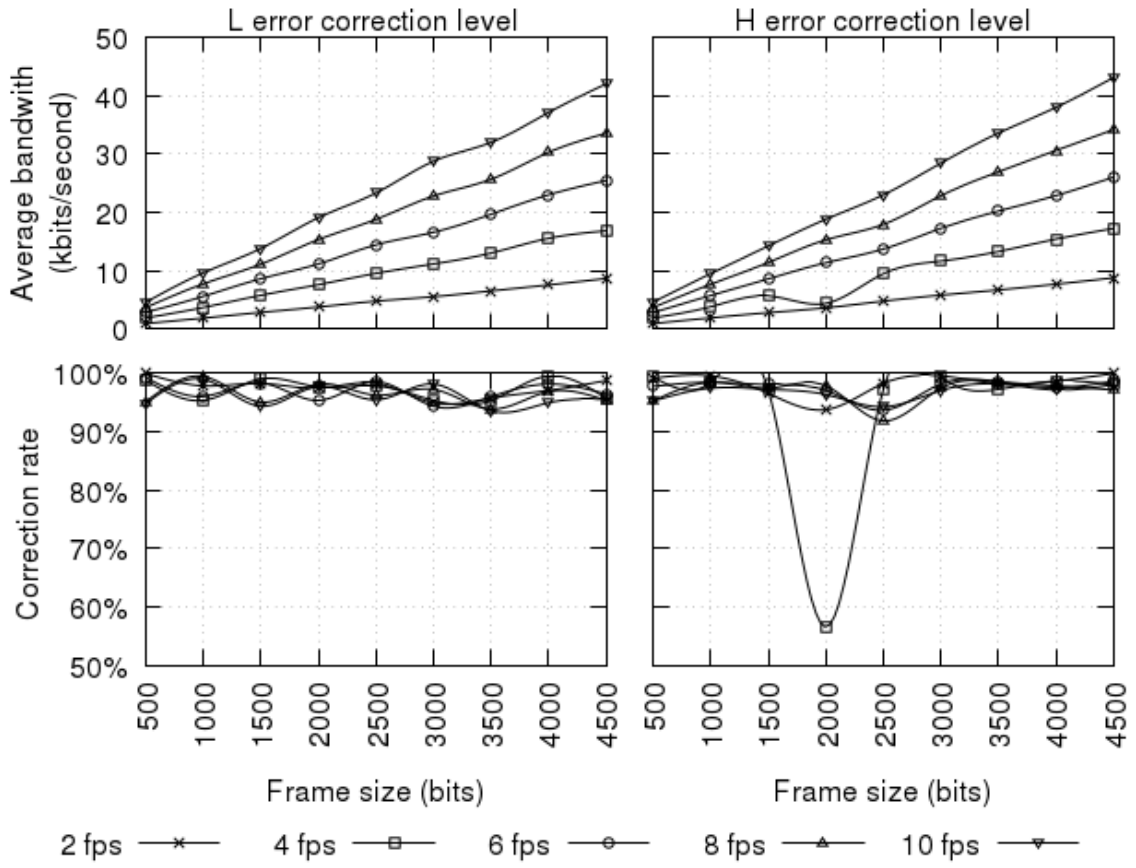


Figure 3.5: Bande passante et taux de décodage dans la deuxième expérience, où chaque code est affiché deux fois

bande passante effective ; en utilisant les mêmes paramètres que ci-dessus, on peut obtenir une bande passante de 43,0 kbps en utilisant le niveau L, et 44,1 kbps en utilisant le niveau H.

Bourrage aléatoire

Cependant, comme nous avons discuté plus tôt, les codes QR ne sont pas tous créés égaux ; Avec la même résolution et le même niveau de correction d'erreur, les résultats expérimentaux indiquent que certains codes semblent être plus difficiles à reconnaître que d'autres. Par conséquent, la simple répétition de la même image en plusieurs fois n'a aucun impact sur cette

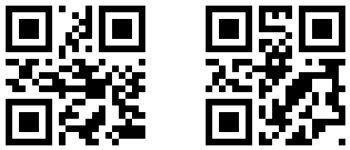


Figure 3.6: Exemples de deux codes avec les données légèrement différentes, mais avec de très différentes formes de points. Le code à gauche contient la chaîne “abcdefg”, tandis que celui à droite contient “abcdefg”.

intrinsèque “dureté”. Notre troisième expérience présente encore un autre mécanisme pour augmenter le taux de reconnaissance.

Cette fois, nous avons essayé de générer les codes à partir des mêmes données d’entrée différentes en ajoutant, à la fin des données, une petite chaîne de caractères aléatoires qui est destinée à changer chaque fois où le code doit être affiché. De ce fait, les mêmes données originales, si elles sont affichées deux fois, sont préfixées à un différent bourrage aléatoire chaque fois, ce qui donne un peu différent tableau de bits. Toutefois, en vertu du schéma de codage QR, même un petit changement à la fin d’un tableau produit une forme complètement différente de points dans le code QR généré. La Figure 3.6 montre un exemple de ce phénomène. Par conséquent, si un code est plus difficile à reconnaître, les mêmes données sont également affichées dans une forme largement différente de points, ce qui augmente les chances d’être correctement ramassées au moins une fois.

Bien que la raison objective que certains codes sont plus difficiles à reconnaître soit inconnue et hors sujet de ce chapitre, les résultats expérimentaux semblent confirmer cette hypothèse. Nous avons effectué une troisième expérience où chaque donnée d’entrée a été affichée trois fois avec différents codes QR générés. Le taux de reconnaissance est meilleure qu’avant lorsque la fréquence de codes est inférieure à 10 fps, comme le montre la Figure 3.7.

Ces résultats nous ont amenés à expérimenter avec les fréquences plus élevées de codes ; nous

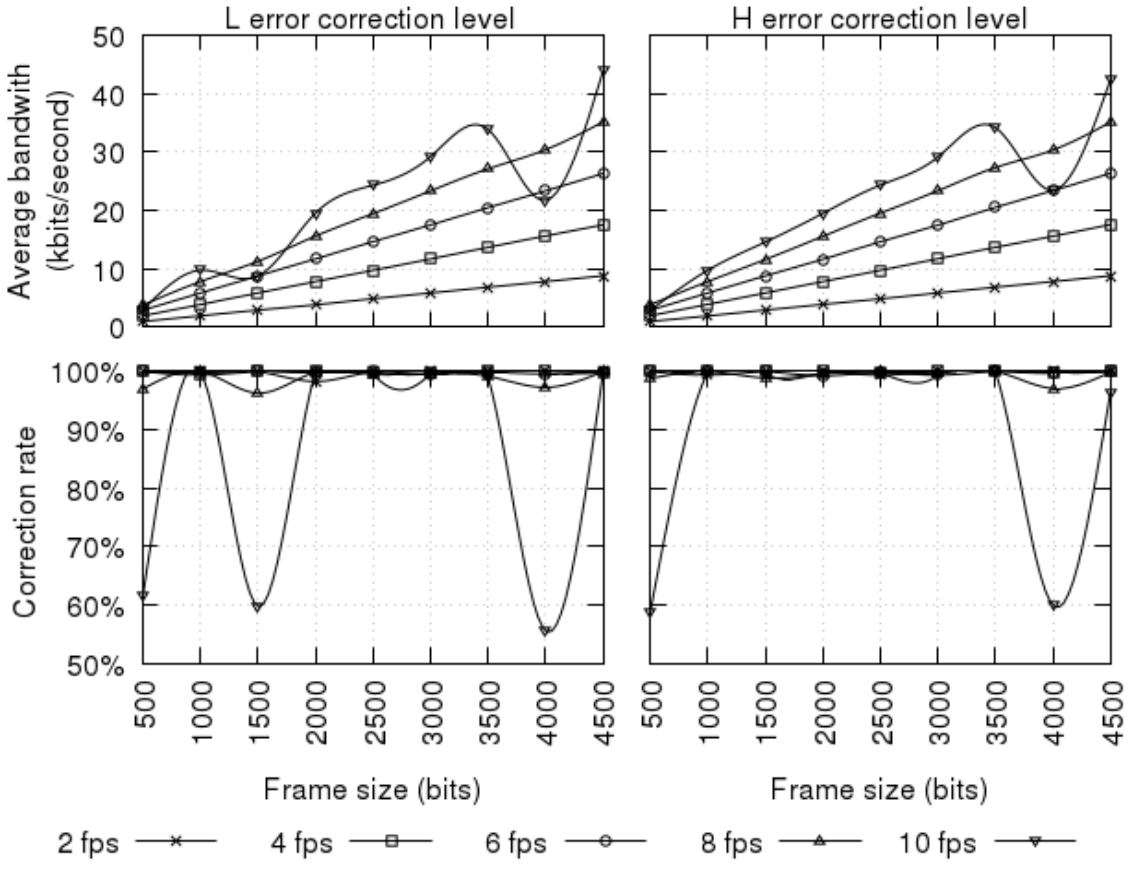


Figure 3.7: Troisième expérience : affichage en trois fois avec bourrage aléatoire

avons ajouté 12 cps, 14 cps et 16 cps. Les codes ont été affichés deux fois. Comme la figure 3.8 le montre, la bande passante maximale du résultat est 65,5 kbps en niveau L, et 68,3 kbps en niveau H, en utilisant 16 cps et les codes de 4500 bits.

3.4.4 CONCLUSIONS PARTIELLES

Ces expériences initiales nous permettent de tirer quelques conclusions sur la nature d'un canal de communication de codes QR. Premièrement, bien que la fréquence plus élevée de codes et la taille plus élevée de codes aient un impact négatif sur le taux de reconnaissance, les données accrues qui peuvent être globalement portées compensent le taux plus élevé d'erreurs

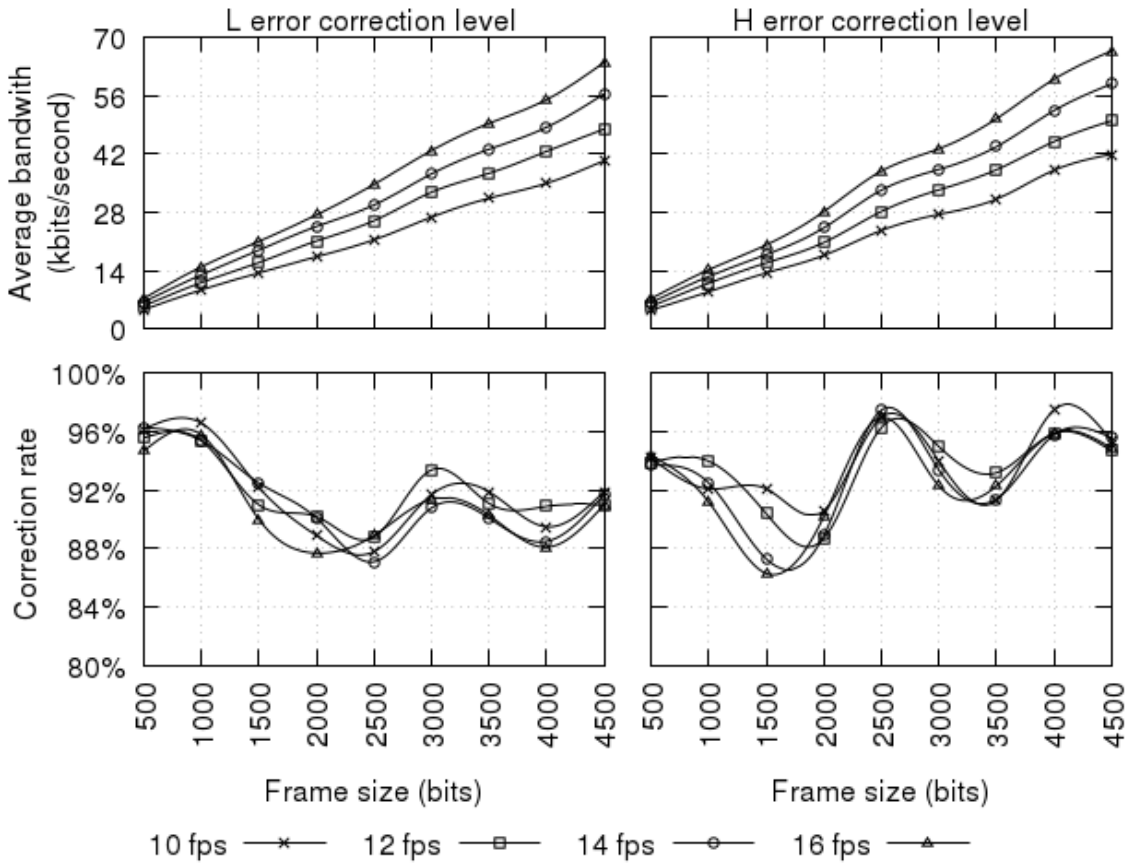


Figure 3.8: Quatrième expérience : affichage en deux fois et les fréquences plus élevées de codes

en termes de la bande passante effective. Deuxièmement, l'introduction de répétition et la variation de formes de points pour les mêmes données augmentent la bande passante effective ; cela montre que deux codes différents pour la moitié du temps est plus efficace qu'un seul code pour le même intervalle. Troisièmement, même pour les plus petites tailles de codes, le taux d'erreur du canal est jamais zéro, ce qui indique que le canal est intrinsèquement avec perte.

A partir de ces constatations, on peut raisonnablement s'attendre qu'un flux de QR code fournisse un canal avec une bande passante effective d'environ 40 kbps, lors de l'affichage de 10 codes de 4000 bits par seconde en utilisant la technique de bourrage aléatoire et le niveau L

de correction d'erreur. Le taux de décodage du canal avec ces paramètres doit être au moins 95%. Évidement, ces constatations sont applicables à un réglage d'une caméra fixe. Elle ne prennent pas en considération la gigue potentielle, le flou ou d'autres effets qui peuvent se produire dans d'autres contextes —bien qu'une expérience informelle décrite dans la section 3.5.8 tende à indiquer que la technologie est relativement robuste.

3.5 UN PROTOCOLE DE CANAUX DE COMMUNICATION UNIDIRECTIONNELLE AVEC PERTE

Dans cette section, nous proposons une approche qui utilise les codes QR continus en tant qu moyen pour réaliser une transmission unidirectionnelle de données.

3.5.1 OBJECTIFS DE LA CONCEPTION

Afin de mettre en œuvre un canal de communication, un protocole spécifique est essentiel ; il doit être bien conçu de telle sorte que les données peuvent être sérialisées et transférées sans coût important. En outre, le protocole doit avoir la capacité de diviser les données transférées dans des images pour générer les codes QR. Le résultat est BufferTannen, un logiciel Java dédié à la sérialisation et à la transmission de données structurées sur des canaux limités de communication. [5] Il fournit un ensemble de classes permettant la représentation de données structurées sous une forme binaire compacte. Contrairement à d'autres systèmes, comme Protocol Buffers de Google [6], la définition de nouveaux types de messages peuvent être effectués lors de l'exécution et ne requiert pas la compilation de nouvelles classes utilisées. De plus, les messages dans BufferTannen ne peuvent pas être codés et décodés sans connaissance

5. `https://github.com/sylvainhalle/BufferTannen`
6. `https://github.com/google/protobuf`

préalable de leur structure. Toutefois, étant donné que les messages ne contiennent pas d'information de leur structure, ils utilisent beaucoup moins d'espace.

BufferTannen définit aussi un protocole permettant la transmission de messages. Malgré que tout canal (e.g. connexion TCP, etc.) puisse être utilisé, BufferTannen a été conçu pour l'opération sur un canal avec les spécifications suivantes, qui sont basées sur nos premiers résultats expérimentaux :

— Le canal est point à point. Le but est d'envoyer de l'information directement de A à B ; il ne fournit pas d'adressage, de routage, etc.
— Le canal a une bande passante faible qui est en mesure de transmettre que quelques centaines d'octets à la fois, peut-être moins de 10 fois par seconde).
— Le canal est unidirectionnel : généralement, un côté de la communication envoie des données qui doivent être ramassées par un récepteur. Cela implique que le récepteur ne peut pas détecter la réception de données ou demander à l'expéditeur de transmettre une autre fois, comme dans des protocoles, p.ex. TCP.
— Le canal entraîne une perte. Cependant, nous supposons que le canal fournit un mécanisme (comme une forme de somme de contrôle) pour détecter et jeter des morceaux de données corrompues.
— Un récepteur peut commencer à écouter sur le canal à tout moment, et être capable de recevoir correctement des messages à partir de ce moment-là. Ainsi, la communication n'a pas de "début" formel qui pourrait être appliqué, par exemple, pour annoncer des paramètres utilisés pour l'échange.

Par conséquent, le canal de communication envisagé comme le milieu de transmission pour les messages de BufferTannen peut être similaire, à bien des égards, à un signal de diffusion lente, comme Hellschreiber (Evers, 1979), télévision à balayage lente (Bretz, 1984), Télétexte

(tel, 1976) ou RBDS (rbd, 2011).

Le protocole BufferTannen vise à transmettre des messages de manière aussi fiable que possible avec ces conditions, tout en préservant l'intégrité de données et l'ordonnancement de messages. La nature de la bande passante faible du canal explique l'accent de sérialisation de messages sous une forme binaire compacte. Puisque le récepteur ne peut pas demander de aucune forme de re-transmission, le protocole doit fournir le mécanisme qui retransmet automatiquement les messages afin de maximiser leurs chances d'être ramassés, alors qu'en même temps il ne doit pas confondre une re-transmission avec un nouveau message ayant un contenu identique. En outre, parce que le récepteur peut commencer à écouter à tout moment, et que le schéma de messages doit être connu afin de les décoder, les schémas de la communication doivent également être transmis à intervalles périodiques.

3.5.2 SCHÉMAS

La déclaration d'une structure de données est appelée *schéma*. L'information peut être représentée sous trois formes différentes :

— Smallscii : Une chaîne de caractères de longueur variable. Puisque BufferTannen vise à limiter autant que possible le nombre de bits requis pour représenter de l'information, ces chaînes sont restreintes à un sous-ensemble de 63 caractères ASCII (lettres, chiffres et ponctuations). Chaque caractère dans une chaîne Smallscii occupe 6 bits, et chaque chaîne se termine avec la chaîne de 6-bit `000000`.

— Integer : Le seul type numérique disponible en BufferTannen. Lorsqu'il est déclaré, les entiers, il leur est donné une "largeur", c.-à-d. le nombre de bits utilisés pour encoder. La largeur peut être une valeur entre 1 et 16 bits.

— Enum : Une liste de constantes prédéfinies de Smallscii. Une énumération est applicable

pour réduire davantage la quantité d'espace occupé par un élément de données lorsque son ensemble de valeurs possibles est connu à l'avance.

Ces blocs de construction de base peuvent être utilisés pour écrire des schémas en les combinant avec l'aide de structures de données composées :

— List : une séquence d'éléments de longueur variable, qui doivent tous être du même type (ou schéma). Les éléments dans la liste sont accessibles par leur index, en commençant par l'index 0.

— FixedMap : une table qui associe des chaînes à des valeurs. La structure est fixe et les chaînes de caractères exacts qui peuvent être utilisés en tant que clés doit être déclarée. Cependant, chaque clé peut être associée à une valeur de type différent.

Ces constructions peuvent être mélangées librement. Ce qui suit représente la déclaration d'un schéma de messages complexes :

```
FixedMap {
  "titre" : Smallscii,
  "prix" : Integer(5),
  "chapitres" : List [
     FixedMap {
       "nom" : Smallscii,
       "longueur" : Integer(8),
       "type" : Enum {"normal", "appendice"}
     }
  ]
}
```

La structure de haut niveau pour ce message est un map (délimité par `{...}`). Ce map a trois clés : `titre`, dont la valeur associée est une chaîne Smallscii, `prix`, dont la valeur associée est un entier dans la gamme 0-32 (qui occupe 5 bits), et `chapitres`, dont la valeur n'est pas un type primitif, mais est une liste en soi (délimitée par `[...]`). Chaque élément de cette liste est un map avec trois clés : une chaîne `nom`, un entier `longueur`, et une énumération `type` dont les valeurs possibles sont `normal` ou `appendice`.

Les schémas peuvent être représentés dans une représentation binaire compacte et sans équivoque comme suit.

Integer La déclaration d'un entier est encodée comme la séquence de bits suivants :

```
ttt wwwww ddddd s
```

La séquence `ttt` représente le type d'élément, encodé sur 3 bits. Un entier contient la valeur décimale 6. La séquence de `w` indique la largeur de l'entier en bits. La largeur elle-même est encodée en 5 bits. La séquence de `d` indique la largeur de l'entier en bits, s'il est exprimé comme une valeur delta, c.-à-d. comme la différence par rapport à un entier d'un message précédent. La largeur elle-même est encodée en 5 bits. Le seul bit `s` est le signe drapeau. Si la valeur est 0, l'entier est non signé ; si elle est 1, l'entier est signé. Notez que des entiers exprimés en valeurs delta sont toujours codés comme les entiers signés ; par conséquent, ce drapeau s'applique uniquement aux entiers qui se produisent en tant que valeurs complètes.

Smallscii La déclaration d'une chaîne Smallscii est simplement encodée comme trois bits représentant le type d'élément ; une chaîne contient la valeur décimale 2.

Enum Une énumération doit fournir la liste de toutes les valeurs possibles qu'elle peut prendre. Elle est formellement représentée comme suit :

```
ttt llll [ssssss ssssss ... 000000 ...
 ssssss ssssss ... 000000]
```

Le type d'élément est la valeur décimale 1, et la séquence `llll` est le nombre d'éléments dans l'énumération, encodé sur 4 bits. Ce qui suit est une concaténation de chaînes Smallscii qui définissent les valeurs possibles de l'énumération. Chaque caractère est encodé sur 6 bits, et la fin d'une chaîne est signalée par la séquence de 6 bits `000000`.

List La déclaration d'une liste est la suivante :

```
ttt llllllll ...
```

Le type d'élément est la valeur décimale 3 ; le 8-bit séquence `llllllll` définit le nombre maximal d'éléments dans la liste. Ce qui suit est la déclaration du type d'élément des éléments dans cette liste.

FixedMap Le dernier type d'élément est le map fixe, déclaré comme suit :

```
ttt [ssssss ssssss ... 000000 ddd...]
```

Le type d'élément est la valeur décimale 4 ; ce qui suit est une chaîne Smallscii qui définit le nom d'une clé, suivie par la déclaration du type d'élément pour cette clé ; ceci est répété pour autant de clés que le map déclare.

3.5.3 MESSAGES

Un *message* est une instance d'un schéma. Par exemple, ce qui suit est un message possible en respectant le schéma précédent :

```
{
  "titre" : "hello world",
  "prix" : 21,
  "chapitres" : [
    {
      "nom" : "chapitre 1",
      "longueur" : 3,
      "type" : "normal"
    },
    {
      "nom" : "chapitre 2",
      "longueur" : 7,
      "type" : "normal"
    },
    {
      "nom" : "conclusion",
      "longueur" : 2,
      "type" : "chapitre"
    }
  ]
}
```

Le lecteur qui est familier avec JSON ou des notations similaires remarquera des fortes similitudes entre BufferTannen et ces langages. Effectivement, les éléments d'un message peuvent être interrogés en utilisant une syntaxe similaire à JavaScript. Par exemple, en supposant que `m` est un objet qui représente le message ci-dessus, la récupération de la longueur du deuxième chapitre serait écrite comme l'expression :

```
m[chapitres][1][longueur]
```

Ceci amène la valeur `chapitres` de la structure de haut niveau (une liste), puis le deuxième élément de cette liste (index 1), puis la valeur `longueur` de l'élément de map correspondant.

Avec les schémas, les messages peuvent être représentés sous une forme binaire compacte.

Smallscii Les chaînes sont représentées comme une séquence de caractères de 6 bits, terminée par la fin du délimiteur de chaîne `000000`.

Integer Les nombres sont représentés par la séquence de bits qui encode leur valeur, sans aucune séquence de terminaison : le nombre de bits à lire est dicté par la taille de l'entier, tel que spécifié par l'élément de schéma correspondant. Si l'entier est signé, le premier bit représente le signe (0 = positif, 1 = négatif) et le reste de la séquence représente la valeur absolue.

Enum Une énumération est simplement fabriquée en bits de séquence correspondant à la valeur appropriée. Encore, le nombre de bits à lire est dicté par la taille de l'énumération, tel que spécifié dans le schéma du message à lire. Par exemple, si l'énumération définit 4 valeurs, alors 2 bits seront lus. La valeur numérique *i* correspond à la *i*-ème chaîne déclarée dans l'énumération.

List Une liste commençant par 8 bits enregistre le nombre d'éléments de la liste. Le reste de la liste est la concaténation de la représentation binaire de chaque élément de la liste. Puisque le type de chaque élément et le nombre de ces éléments à lire sont tous deux connus, aucun délimiteur n'est nécessaire entre chaque élément ou à la fin de la liste.

Fixed Map Le contenu d'un map fixe est simplement la concaténation de la représentation binaire de valeur de chaque map. La clé auquel chaque valeur est associée, et le type de valeur à lire, sont précisés dans le schéma du message à lire, et devraient apparaître exactement dans l'ordre où ils ont été déclarés. Cela nous évite de répéter les clés de maps dans chaque message.

3.5.4 LIRE ET ÉCRIRE DES MESSAGES

En BufferTannen, les deux schémas et des instances de schémas sont représentés par le même objet, appelé `SchemaElement`. Un SchemaElement vide doit d'abord être instancié en utilisant certains schémas ; cela peut être effectué soit par :

— Lire une chaîne de caractères formatée comme ci-dessus ; soit par
— Lire une chaîne binaire qui contient un codage du schéma. En effet, en BufferTannen les messages et les schémas peuvent être transmis sous une forme binaire sur un canal de communication, et une méthode est fournie pour exporter le schéma d'un message dans une séquence de bits.

Une fois qu'un SchemaElement vide est obtenu, il peut être rempli avec des données, encore de deux façons :

— Lire une chaîne de caractères formatée comme ci-dessus ; ou

— Lire une chaîne binaire qui contient un codage des données.

Il y a des méthodes similaires pour fonctionner en sens inverse, et pour *écrire* le schéma ou le contenu de données d'un message comme une chaîne de caractères ou une chaîne binaire. De cette façon, les messages et les schémas peuvent être librement encodés/décodés en utilisant des chaînes de textes lisibles ou des chaînes binaires compactes.

Comme on peut le voir, pour lire ou écrire un message, il faut d'abord instancier un objet avec un schéma. En fait, l'essai de décodage d'un flux de données sans publicité du schéma sous-jacent provoquera une erreur, même si le flux contient correctement des données formatées. De même, l'essai de lecture de données qui utilise un schéma avec un objet instancié avec un autre schéma provoquera également une erreur. En d'autres termes, aucune donnée ne peut être lue ou écrite sans connaissance du schéma correct.

Cela peut sembler restrictif, mais il permet BufferTannen d'optimiser fortement la représentation binaire de messages. En l'absence d'un schéma connu, chaque message aurait besoin de porter, en plus de ses données réelles, de l'information de sa propre structure.

En pratique, grâce à lui, les répétitions de la description de son schéma dans chaque message se trouvent toutes dans les données du message. Au contraire, si le schéma est connu, toutes ces informations de signalisation peuvent être jetées : lors de la réception d'une séquence de bits, un lecteur qui possède le schéma connaît exactement le nombre de bits à lire, les données qu'il représente et la position où les données sont placées dans le structure de message. Cela implique toutefois qu'un récepteur qui ne connaît pas le schéma à appliquer n'a aucune idée sur la façon de traiter une chaîne binaire.

Pour illustrer l'intérêt de BufferTannen comme un schéma de codage de messages, nous considérons l'exemple de transmettre de événements à partir d'un jeu vidéo à un moniteur

externe.

3.5.5 SEGMENTS

Les messages et les schémas sont encapsulés dans une structure appelée *segment*. Un segment peut être de quatre types :

Segments de message contiennent la représentation binaire d'un message, avec un numéro séquentiel (utilisé pour préserver l'ordre des messages reçus), ainsi que le numéro se référant au schéma qui doit être utilisé pour décoder le message. Un segment de message est constitué d'un en-tête structuré comme suit :

```
tt nnnnnnnnnnnn wwwwwwwwwwww ssss ...
```

L'en-tête commence par deux bits décrivant le type du segment ; un segment de message contient la valeur décimale 1. Les sections `n` et `w` décrivent le numéro séquentiel et la longueur totale du segment, qui sont tous deux encodées sur 12 bits. Les quatre bits de `s` fournissent le numéro de schéma dans la banque de schéma qui devrait être utilisée pour lire ce segment. Le reste du segment est composé d'un map, d'une liste, d'une chaîne Smallscii ou d'un nombre, dont la représentation binaire a été décrite ci-dessus.

Segments de schéma contiennent la représentation binaire d'un schéma, qui est associé à un numéro. Plusieurs schémas peuvent être utilisés dans la même communication, donc une banque de schémas identifiés par leurs numéros est crée. Un segment de schéma consiste en un en-tête structuré comme suit :

```
tt nnnnnnnnnnnn ssss ...
```

L'en-tête commence par deux bits décrivant le type du segment ; un segment de schéma contient la valeur décimale 2. La section `n` décrit le numéro séquentiel du segment, et la section `s` donne le numéro du schéma dans la banque de schémas auquel ce segment devrait être attribué. Le reste du segment est constitué d'une chaîne binaire décrivant le schéma, dont la représentation a été décrite ci-dessus.

Segments de blob sont destinés à transporter des données binaires brutes selon le protocole BufferTannen.

Segments deltas contiennent la représentation binaire d'un message exprimé comme la différence ("delta") entre ce message et le précédent utilisé comme référence. Les segments deltas sont utilisés pour comprimer davantage la représentation d'un message, dans le cas où les messages ne changent pas beaucoup pour un intervalle de temps.

```
tt nnnnnnnnnnnn wwwwwwwwwwww rrrrrrrrrrr...
```

L'en-tête commence par deux bits décrivant le type du segment ; un segment delta contient la valeur décimale 1. Les sections `n` et `w` décrivent le numéro séquentiel et la longueur totale du segment, qui sont tous deux encodées sur 12 bits. La section `r` donne le numéro séquentiel d'un autre segment, en rapport avec laquelle le delta du segment actuel est exprimé. Ce qui suit est une chaîne binaire qui décrit le contenu de la "différence", et il doit être calculé par rapport à ce segment pour obtenir le contenu de l'actuel.

Le calcul du delta est récursivement effectué sur chaque élément des deux messages pour comparer dans l'ordre où ils se produisent. Chaque type d'élément est défini comme suit.

— Smallscii strings : si les chaînes correspondantes sont identiques, émettre le seul bit `0`.

Sinon, émettre le bit `1` qui est suivi par la chaîne Smallscii du message cible.

— Integers : si les numéros correspondants sont identiques, émettre le seul bit `0`. Dans le cas contraire, émettre le bit `1` suivi par la différence entre la source et le nombre entier cible.

— Enumerations : si la valeur correspondante du type énuméré est la même, émettre le seul bit `0`. Autrement, émettre le bit `1` suivi par la valeur de l'entier correspondant à l'index de la valeur dans le message cible.

— Lists : si les deux listes ont les mêmes éléments dans le même ordre, émettre le seul bit `0`. Sinon, émettre le bit `1` suivi par la représentation binaire de la liste des cibles.

— FixedMaps : appliquer récursivement les règles précédentes pour chaque clé du map.

On peut voir que les segments delta appliquent uniquement une forme grossière de comparaison. Par exemple, on ne tente pas de détecter si les deux listes diffèrent par l'addition ou la suppression d'un élément ; le contenu de la liste est retransmis en intégralité chaque fois qu'il n'est pas identique à l'original. Néanmoins, cette technique permet de réaliser des économies substantielles chaque fois qu'une partie d'une structure de données reste identique d'un message à l'autre.

3.5.6 IMAGES

Le canal de communication envoie des données binaires en unités appelées *images*. Une image est simplement un ensemble de segments concaténés sous une forme binaire, précédés d'un en-tête contenant le numéro de version du protocole (actuellement "1") et la longueur (en bits) du contenu de l'image. Formellement, la structure binaire d'une image est la suivante :

```
vvvv nnnnnnnnnnnnnn ffff... ffff...
```

La section v est constituée du numéro de version de 4 bits du protocole, suivi de 14 bits indiquant la longueur (en bits) de l'image. Chaque segment est ajouté directement à cet en-tête de 18 bits. Parce que l'en-tête de chaque segment contient sa propre longueur, aucune autre transformation n'est nécessaire pour décoder correctement les données de segment.

Lorsque de nombreux segments sont en attente d'être transmis, le protocole tente d'adapter autant de segments que possible (dans un ordre séquentiel) au sein de la taille maximale d'une image avant de l'envoyer. Cette taille maximale peut être modifiée pour correspondre aux spécificités du canal de communication qui est utilisé. Dans l'incarnation actuelle du protocole, les segments ne peuvent pas être fragmentés en plusieurs images. Ainsi, un segment ne peut pas dépasser la taille maximale d'une image.

Chaque image est ensuite convertie en un code QR, avec son contenu binaire base64-encodé comme le texte de codes. Ce code QR peut alors être reconnu sur le site de réception, converti en une séquence binaire, et analysé et transmis aux images, aux segments et aux messages à travers l'application des transformations inverses.

3.5.7 MODES DE STREAMING

BufferTannen est conçu avec deux modes d'envoi, respectivement appelés le mode "Lake" et le mode "Stream".

Le mode Lake est destiné à l'envoi d'une pièce finie de données, comme un fichier, ou une séquence de messages BufferTannen dont le contenu complet est connu à l'avance. Les données à envoyer sont divisées en un ensemble fini de segments, et toute la séquence de segments est émise à plusieurs reprises à travers des codes QR. Si des images sont manquantes ou mal décodées, la répétition infinie de tous les segments permet d'attraper les données

```
------------------------------------------------------------------------
Sending mode:        lake
Buffer state:        [||>       ||:.::||:|||] 59% (130/219)
Progress:            0408/0000 (13.8 sec @30 fps)
Link quality:        22/30 [*******   ] (73%)  Global:   339/454 (74%)
Data stream index:   0
Resource ident.:     myfile.jpg
Processing rate:     35 ms/frame (27 fps)
------------------------------------------------------------------------
```

Figure 3.9: Une partie de l'interface texte du récepteur de codes QR qui fonctionne en mode Lake

manquantes à la boucle suivante. Finalement, les erreurs de décodage peuvent indiquer que les données doivent être lues pour plus d'une boucle avant qu'elles soient complètement reçues.

L'utilisation du mode Lake peut être détectée par des images transportant une valeur non nulle à leur champ d'en-tête de "segments totaux". Ainsi, un récepteur qui commence à lire à tout moment la séquence d'images sait combien de segments doivent au total être reçus, et la position relative de chaque segment dans les données pour être reconstruite. Cela rend le mode Lake un schéma de transmission optique de données relativement lente, mais très robuste.

En mode Stream, les données sont constamment lues en segments qui forment alors un flux d'images, et les images sont immédiatement envoyées. Le processus de lecture arrête seulement quand il n'y a plus de données à lire. Les images déjà envoyées sont supprimées de la mémoire, donc il n'y a aucun moyen de renvoyer les données à plusieurs reprises. Cependant, pour le bien de la cohérence des données, nous avons fait un tampon pour les clones des images envoyées, et après avoir envoyé une quantité spécifique d'images récupérées à partir des données originales, les images dans la mémoire tampon sont renvoyées à nouveau, puis enlevées de la mémoire tampon . Par conséquent, le mode Stream est destiné à envoyer des données en temps réel, généralement là où la perte de données est acceptable et la consistance peut être légèrement sacrifiée (par exemple audio ou vidéo).

La Figure 3.9 montre une partie de l'interface texte de notre mise en œuvre de récepteur de codes QR. L'interface montre que les images étant reçues sont en mode Lake. Le champ d'état de tampon indique la progression de la réception. Dans cet exemple, il montre que les segments 130 sur 219 ont été correctement reçus ; la barre de texte à gauche indique à quelles parties de la séquence totale ces segments correspondent. Une section de la séquence qui n'a pas été reçue du tout est indiquée par un espace vide ; les parties de plus en plus complètes de la séquence sont représentées respectivement par les symboles `.`, `:` et `|`. Le symbole > indique la position relative du dernier segment qui a été lu correctement.

Le champ "Link quality" donne une indication en temps réel du taux de décodage. Il montre que 22 des 30 dernières images capturées par la caméra ont été correctement décodées, et que globalement, 339 images ont été décodées sur 454 images capturées. L'identifiant de ressource et l'index de flux de données, portés par chaque image, sont également affichés.

3.5.8 RÉSULTATS EXPÉRIMENTAUX

Les expériences de la section 3.4 ont confirmé notre intuition que les flux de codes optiques sont un canal de communication intrinsèquement peu fiable et à faible bande passante. La compacité du protocole BufferTannen peut être motivée par un exemple de la vérification de l'exécution. Un jeu vidéo particulier, appelé Pingus, a été instrumenté pour produire des événements qui contiennent l'état de chaque personnage dans le jeu. Le schéma de ces événements est illustré à la figure 3.10.

```
FixedMap {
  "pingus" : List [
    FixedMap [
      "id" : Integer(6),
      "x" : Integer(10),
      "y" : Integer(10),
      "velocity-x" : Integer(4),
      "velocity-y" : Integer(4),
      "state" : Enum {"floater",
         "basher", "builder",
         "athlete", "normal"}
    ]
  ]
}
```

Figure 3.10: Le schéma d'événements produits par un jeu vidéo instrumenté.

Transfert de fichiers

Un événement contient généralement des données pour 50 caractères, ainsi la structure map est répétée autant de fois. L'envoi d'un tel événement en format texte clair, sans aucun espace, prend environ 3750 octets. À un fréquence de 30 événements par seconde, il prend 879 kbps de bande passante pour transmettre le flux d'événements. Le même événement dans BufferTannen prend 1.856 bits, ou 232 octets. Cela divise par plus de 16 les exigences de bande passante pour envoyer un flux de ces événements, ce qui donne une bande passante de 54 kbps. [7] Dès cet instant, les segments de delta peuvent être utilisés pour réduire davantage la bande passante du flux, et transmettre les événements restants en utilisant un peu plus de 100 octets chacun, qui consomme une bande passante d'environ 24 kbps. Nos expériences précédentes montrent que cela est dans la gamme de ce que l'on peut raisonnablement s'attendre à transmettre à travers les codes QR.

7. Envoi de la même chaîne de caractères en Gzip se rétrécit vers à 716 octets, ce qui rend la compression standard une alternative moins attrayante dans ce contexte.

Nous avons ensuite testé la capacité du protocole BufferTannen pour atténuer ces défauts, à travers son utilisation de répétition et sa représentation binaire compacte [8]

Nous avons choisi d'encoder des données en codes de 4000 bits, qui, après l'encapsulation de BufferTannen, équivaut à un code QR d'environ 5800 bits. Avec la combinaison de 5800 bits et du niveau L de correction, selon le Tableau 3.6, la taille de symboles est d'environ 93×93 qui est entre les combinaisons de 3500 bits, du niveau H et de 4000 bits, du niveau H. D'après le résultat de la dernière expérience, les combinaisons de 4000 bits, du niveau H et tous les fps ont un taux de plus 95% de correction qui est fiable. D'autre part, les fréquences de codage sont 4, 6, 8, 10, 12 images par seconde. Selon la dernière expérience, cette configuration est fiable et censée être capable de fournir, respectivement 23.2–69.6 kbps et 16.0–48.0 kbps de bande passante. Étant donné l'application pratique, nous avons choisi de transférer un fichier d'exemple dont la taille est de 37,656 octets, et nous avons effectué chaque expérience 20 fois.

Le résultat de l'expérience du mode Lake dans la Figure 3.11 montre que la meilleure valeur de fps est de 10, et dans ce cas, le fichier d'exemple avait besoin d'être transféré en moyenne 2,4 fois pour s'assurer que le récepteur pourra obtenir toutes les images. Le temps passé en moyenne est de 17,27 secondes, et la bande passante est donc environ 17,0 kbps.

En mode Stream, le pourcentage de codes reçus est important. Dans l'expérience, selon la Figure 3.12, les taux moyens d'achèvement de toutes les configurations sont plus de 99%, et la configuration de 12 fps a besoin d'environ 13,11 secondes pour envoyer toutes les images avec un canal de streaming de données de 22,4 kbps .

8. Une vidéo de BufferTannen en action est disponible en ligne : `https://www.youtube.com/watch?v=GSL0md0TlY8`

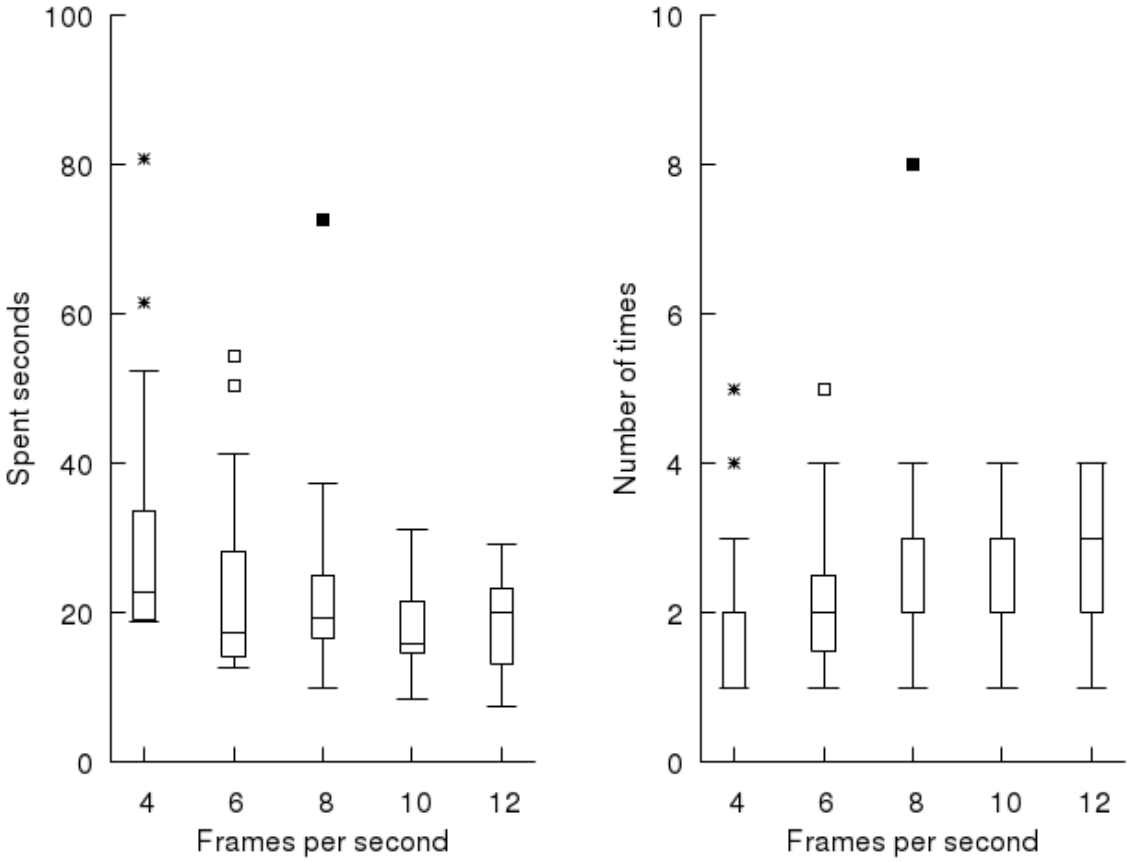


Figure 3.11: Temps de l'envoi des données en mode Lake

Glisser du papier

La capacité d'envoi des flux de données en mode Lake peut également être utilisée pour compléter la capacité intrinsèquement limitée de codes QR. Lors de l'affichage d'un code imprimé sur une feuille de papier, de grandes quantités de données peuvent être seulement apportées par l'augmentation de la résolution de codes ; cependant, la résolution ne peut être augmentée jusqu'à une limite certaine et prédéfinie.[9] En outre, pour des résolutions plus élevées, le code peut devenir difficile à reconnaître en utilisant des caméras en niveau d'entrée et en basse résolution. Par conséquent, c'est prudent de supposer que, avec l'aide de

9. 4296 caractères alphanumériques ou 3222 octets en codage Base-64.

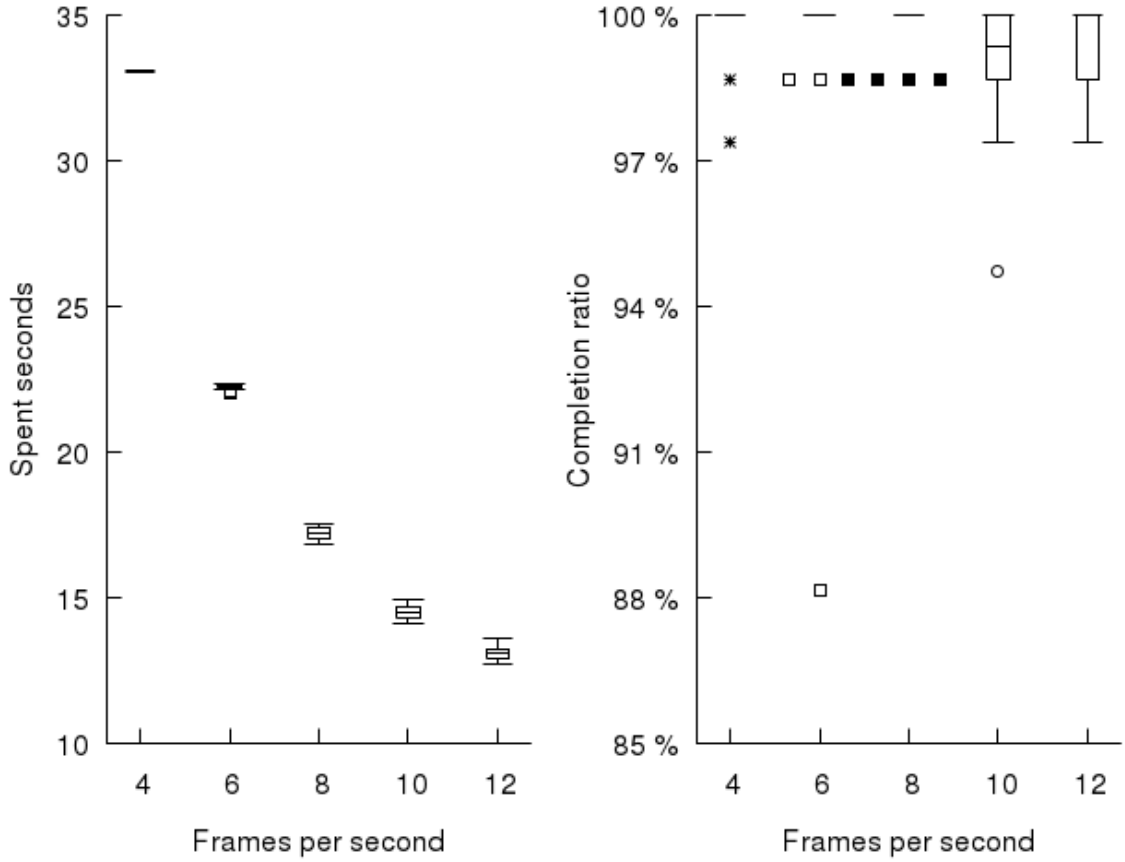


Figure 3.12: Temps de l'envoi des données en mode Lake Stream

la technologie existante de codes QR, un maximum de 3000 octets de données peuvent être transférées à travers un code QR.

Cette limitation peut être surmontée par l'utilisation du protocole BufferTannen. Bien qu'aucun code dont la taille est supérieure à environ 4000 octets ne puisse être créé, plusieurs de ces codes peuvent être alignés sur un morceau de papier. Chacun de ces codes peut être formaté pour contenir une image unique de données envoyées par le protocole BufferTannen en mode Lake. Il suffit à l'utilisateur de glisser la caméra sur ces codes ; en vertu du mode Lake, l'ordre dans lequel les codes sont scannés n'est pas pertinent, et les pièces complètes de données peuvent être correctement reconstruites à partir des images individuelles. c'est donc possible

de théoriquement transmettre des quantités illimitées de données, tout en utilisant des codes d'une résolution inférieure (cette résolution inférieure étant compensée par la présence de plus d'un code).

Pour vérifier cette affirmation, nous avons imprimé sur une feuille de papier le contenu d'un fichier de 37 ko en tant qu'une séquence de codes QR, traités comme des images par BufferTannen en mode Lake. Nous avons ensuite passé la caméra au-dessus de cette feuille de papier à une distance d'une longueur de bras (montré dans la Figure 3.13). L'interface utilisateur du logiciel affiche en temps réel le nombre d'images restantes à décoder et leurs positions dans le flux complet, qui donne à l'utilisateur des indications de codes que passe la caméra. Il convient de noter que la caméra fonctionnait en mode "film", et non en mode "snapshot". En d'autres termes, les images ont été capturées continuellement par la caméra lorsque elle était déplacée au-dessus de la feuille ; l'utilisateur n'a pas besoin de pointer et de cliquer sur chaque code QR individuel (ce qui serait fastidieux).

Le niveau de correction d'erreur choisi était L et les tailles de données brutes par code que nous avons testées étaient 500, 750, 1000, 1250, 1500, 1750, et 2000 octets. Avec 500 octets, il y avait 76 codes tandis qu'avec 2000 octets, il y en avait seulement 19. Après avoir été encapsulées par le protocole BufferTannen, les tailles finales de données utilisées pour générer les codes QR sont devenues en conséquence 723, 1055, 1391, 1724, 2054, 2389 et 2722 octets. Les codes ont été imprimés à 300 ppp (points par pouces) et 600 ppp sur des papiers de bureau , et nous avons programmés pour donner 500 points à chaque bord de tous les codes dans le papier.

À 300 ppp, les codes dont la taille était de 500 à 1250 octets étaient tous décodés avec succès et en douceur. Cependant, quand la taille a atteint 1500 octets, le décodage est devenu plus problématique. Pour certains codes, nous avons dû recommencer plusieurs fois et coller la

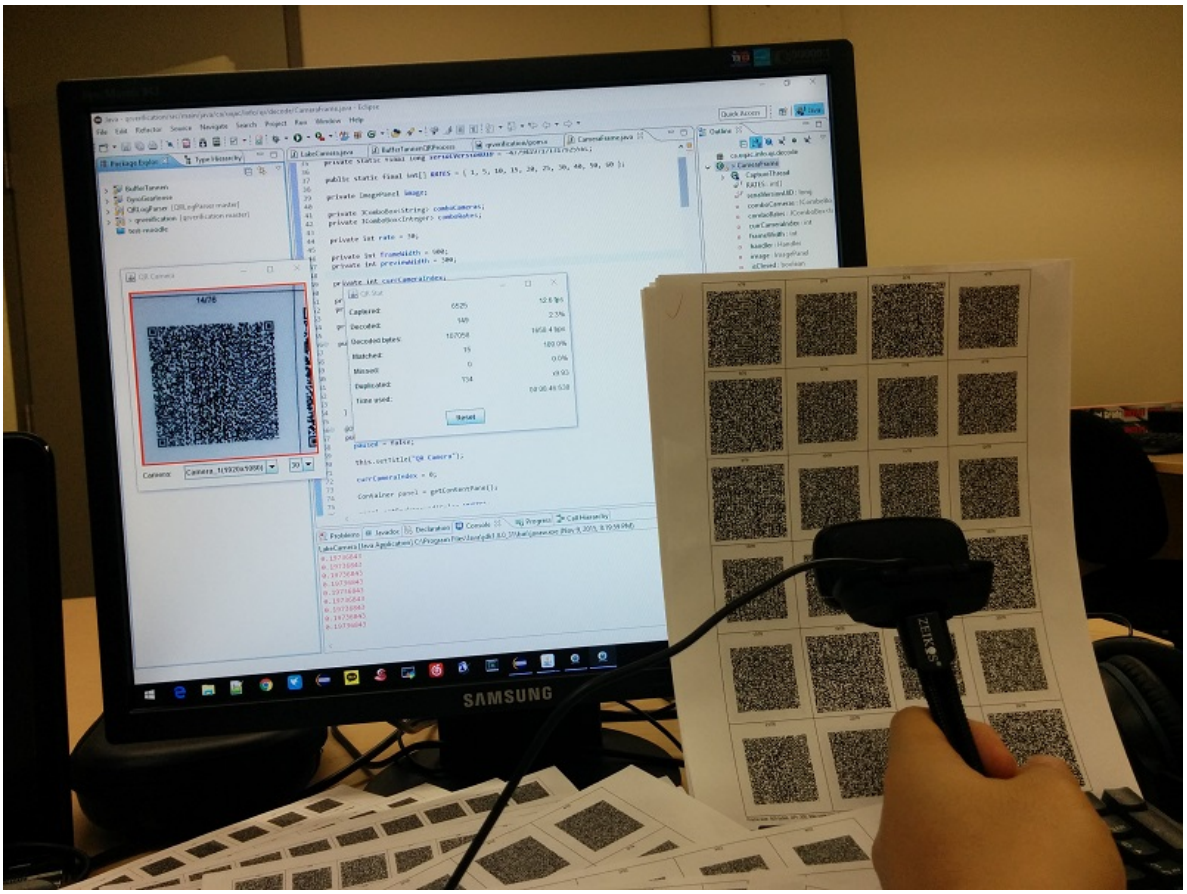


Figure 3.13: Passer la caméra au-dessus d'un ensemble de codes QR pour reconstituer le contenu d'un fichier plus grand.

caméra près du papier pendant quelques secondes, mais parmi le total de 31 codes, trois restaient impossible à décoder. Dans le cas des codes de 1.750 et de 2.000 octets, aucun des codes n'aurait pu être décodé. Ceci peut être expliqué par le fait que les codes de densité plus élevée ont pour effet que chaque module de code est plus petit et ainsi plus difficile à capturer. Par exemple, le bord d'un code de 500-octets et du niveau L est d'environ 77 modules, de sorte que chaque module peut avoir environ 6 points dans le papier, tandis que le bord d'un code de 2000-octets et du niveau L est 149 modules, donc chaque module ne peut avoir que 3 points pour imprimer.

À 600 ppp, aucun des codes ne pourrait pu être décodé, peu importe le nombre d'octets qu'ils

ont apporté. La raison en est que les codes imprimés à 600 ppp sont trop petits et la caméra doit approcher de très près au papier, mais les images capturées étaient toutes troubles et hors de foyer. Ceci semble donc que les codes imprimés à 600 ppp sont au-delà de la capacité d'une standard caméra web.

Néanmoins, cette expérience démontre la viabilité du concept de passer une caméra à travers un tableau de codes QR. Nos découvertes empiriques indiquent qu'un flux de données peut être divisé en un ensemble de codes QR d'environ 1000 octets chacun, dont le contenu correspond à des images individuelles du protocole BufferTannen qui contiennent les données de transmission. Il suffit de passer la caméra au-dessus de cet ensemble de codes, dans aucun ordre particulier, pour reconstruire le contenu complet de données du côté de l'appareil.

3.6 CONCLUSION

Dans ce chapitre, nous avons présenté une solution d'un canal de communication unidirectionnelle sur la base de codes QR, et réalisé des expériences pour mesurer sa performance. Nous avons d'abord testé expérimentalement les caractéristiques d'un flux de données de codes QR sous diverses conditions, et extrait les paramètres qui maximisent la bande passante effective du canal. Toutefois, étant donné que le canal est intrinsèquement enclin à erreur et à faible bande passante, nous avons ensuite introduit BufferTannen, un protocole conçu spécialement pour ce type de canal. BufferTannen prend soin du fractionnement, de la transformation, et dans une certaine mesure avec la compression des données de transmission afin de maximiser l'efficacité du flux de codes QR. La faisabilité de cette approche a ensuite été empiriquement observée lors d'une nouvelle série d'expériences.

Même avec les limites du protocole et du canal de communication, les résultats présentés peuvent être utilisés à bon escient dans une variété de situations. Dans les environnements

limités où l'utilisation du signal de radio ou de câbles est interdite ou difficile, notre approche peut fournir un moyen facile pour communiquer entre pairs, soit comme sauvegarde d'urgence ou comme moyen principal. Par ailleurs, en raison de l'évolution de la qualité des deux dispositifs d'affichage et de capture d'images, c'est possible de prévoir des vitesses de transmission accrues dans l'avenir.

Enfin, les techniques susmentionnées pourraient être transformées en une liaison de communication bidirectionnelle dans le cas de terminaux équipés d'une caméra et d'un écran. Dans un tel cas, les reconnaissances d'images correctement décodées pourraient être échangées, ce qui permettrait de renvoyer des données sur demande et d'augmenter la bande passante effective.

CHAPITRE 4

ÉVALUATION HORS LIGNE DE FORMULES LTL AVEC LES MANIPULATIONS DE BITMAPS

Ce chapitre présente une version modifiée et traduite d'un article qui est écrit par K. Xie et S. Hallé et qui est encore en cours de révision pour sa publication dans les actes de la conférence internationale : Runtime Verification 2016 (RV'16) à Madrid, Espagne en 2016.

4.1 INTRODUCTION

Une *logique temporelle* (Huth et Ryan, 2004) est un système logistique qui utilise des règles et des symboles pour décrire et raisonner sur le changement de l'état d'un système en termes de temps. Elle est basée sur l'idée qu'un état ne peut pas rester constamment vrai ou faux avec le temps. Une *Logique Temporelle Linéaire (LTL)* (Pnueli, 1977) est une logique temporelle, et comme son nom l'implique, une *LTL* peut désigner une seule séquence d'états et pour chaque état il n'y a qu'un état futur.

Un *bitmap*, qui est également connu sous forme de tableau de bites ou de bitset, est une structure compacte de données stockant une séquence de valeurs binaires. Comme on le verra dans la section 4.2, il peut être utilisé pour exprimer un ensemble de nombres, ou un tableau

dont chaque bit représente une option de 2 valeurs. Les bitmaps présentent plusieurs avantages en tant qu'une structure de données : ils peuvent représenter brièvement de l'information et fournir des fonctions très efficaces pour les manipuler, grâce au fait que les bits multiples peuvent être traités en parallèle à travers une seule instruction du processeur.

Dans ce chapitre, nous explorons l'idée de l'utilisation des manipulations de bitmaps pour l'évaluation hors ligne de formules LTL d'un journal d'événements. A cet effet, dans la section 4.3, nous introduisons une solution qui, pour une trace d'événements données σ et une formule LTL φ, convertit d'abord des termes de base en autant de bitmaps ; intuitivement, le bitmap correspondant à une proposition atomique p décrit les événements de σ qui satisfont p. Les algorithmes sont ensuite détaillés pour chaque opérateur LTL qui prend des bitmaps comme leur entrée et qui retourne un bitmap comme leur sortie. L'application récursive de ces algorithmes peut être utilisée pour évaluer toute formule LTL.

Cette solution présente plusieurs avantages. Tout d'abord, l'utilisation de bitmaps peut être considérée comme une forme d'*indexation* (dans le sens du terme de base de données) du contenu d'une trace. Plutôt que d'être un algorithme en ligne qui lit simplement une trace pré-enregistrée, notre solution exploite le fait que la trace est complètement connue à l'avance, et profite largement de cet indice pour accéder directement à des endroits spécifiques dans la trace afin d'accélérer son processus. Deuxièmement, un bitmap ayant des 0s ou 1s consécutifs peut être compressé, ce qui réduit le coût de l'espace et accélère profondément l'exécution de nombreuses opérations (Kaser et Lemire, 2014).

À cette fin, la Section 4.4 décrit une installation expérimentale utilisée pour tester notre solution. Elle révèle que, pour des formules LTL complexes qui contiennent près de 20 opérateurs temporels et conjonctifs, des grandes traces d'événements peuvent être évaluées à un débit de plusieurs dizaines de millions d'événements par seconde. Ces expériences montrent que les

bitmaps sont une structure de données compact et rapide, et sont particulièrement appropriés pour le type de manipulations nécessaires de monitoring hors ligne.

4.2 BITMAPS ET COMPRESSION

Un bitmap (ou bitset) est un tableau binaire que l'on peut considérer comme une représentation efficace et compacte d'un ensemble entier. Étant donné un bitmap de n bits, le i-ème bit est mis à 1 si le i-ème entier dans la gamme $[0, n-1]$ existe dans l'ensemble.

On a reconnu que des bitmaps pourraient fournir des moyens efficaces de manipulation de ces ensembles, en vertu de leur représentation binaire. Par exemple, Une union et une intersection entre les ensembles d'entiers peuvent être calculées avec les opérations au niveau du bit (OR, AND) sur leurs bitmaps correspondants ; à leur tour, de telles opérations au niveau du bit peuvent être effectuées très rapidement par les microprocesseurs, même dans une seule opération du processeur de larges morceaux de 32 ou de 64 bits qui dépend de l'architecture.

Par ailleurs, un bitmap peut être utilisé pour mapper n blocs de données à n bits. Si la taille de chaque bloc est supérieur à 1, le bitmap peut réduire considérablement la taille du stockage. De plus, avec sa capacité de l'exploitation du parallélisme au niveau des bits du matériel, les opérations standard de bitmaps peuvent être très efficaces. Sans surprise, les bitmaps ont été utilisés dans de nombreuses applications où les exigences d'espace ou de vitesse sont essentielles, telles que la recherche d'information (Chan et Ioannidis, 1998), les bases de données (Burdick et al., 2001), et l'exploration de données (Ayres et al., 2002; Uno et al., 2005).

Un bitmap avec une faible fraction de bits mis à la valeur 1 peut être considéré comme *creux* (Kaser et Lemire, 2014). Un tel bitmap creux est une perte de temps et surtout d'espace. Par

conséquent, de nombreux algorithmes ont été développés pour *compresser* ces bitmaps ; la plupart d'entre eux sont basés sur le modèle Run-Length Encoding (RLE) dérivé du système de compression BBC (Antoshenkov, 1995). Dans ce qui suit, nous décrivons brièvement quelques-unes de ces techniques. En particulier, nous détaillons les algorithmes WAH (Wu et al., 2006), Concise (Colantonio et Di Pietro, 2010) et EWAH (Lemire et al., 2010), parce qu'ils ont les librairies open source bien implémentées en Java que nous allons évaluer expérimentalement plus tard dans ce chapitre.

4.2.1 WAH

L'algorithme WAH (Wu et al., 2006) divise un bitmap de n bits en $\lceil \frac{n}{w-1} \rceil$ mots de $w-1$ bits où w est une pratique longueur de mot (par exemple, 32). WAH fait la distinction entre deux types de mots : les mots avec seulement les $w-1$ uns $(11\ldots1)$ ou avec seulement $w-1$ zéros $(00\ldots0)$, sont les *mots pleins*, alors que les mots contenant un mélange de zéros et d'uns sont les *mots littéraux*. les mots littéraux sont stockés avec w bits : le bit le plus significatif est mis à zéro et les bits restants stockent les $w-1$ bits hétérogènes. Des séquences de mots pleins homogènes (tous uns ou tous zéros) sont également stockées avec w bits : le bit le plus significatif est mis à 1, le bit le deuxième plus significatif indique la valeur de bits de la séquence de bloc homogène, tandis que les restants $w-2$ bits stockent la longueur de série de la séquence de bloc homogène.

4.2.2 CONCISE

L'algorithme Concise (Colantonio et Di Pietro, 2010) est un algorithme de compression bitmap sur la base de WAH. En comparant avec WAH, dont la longueur de série est de $w-2$ bits, Concise utilise $w-2-\lceil \log_2 w \rceil$bits pour la longueur de série et $\lceil \log_2 w \rceil$ bits pour stocker une

valeur entière qui indique de retourner un bit d'un seul mot de $w-1$ bits. Cette fonction peut améliorer le taux de compression dans le pire cas.

4.2.3 EWAH

L'algorithme EWAH (Lemire et al., 2010) est aussi une variante de WAH mais il n'utilise pas son premier bit pour indiquer le type de mot comme WAH et Concise. EWAH définit plutôt un *mot de marqueur* de w bits . Les $w/2$ bits les plus significatifs du mot sont utilisés pour stocker le nombre des mots pleins suivants (tous uns ou tous zéros) et les restants $w/2$ bits encodent pour le nombre des *mots sales*. Ces mots sont exactement comme les mots littéraux de WAH, mais utilisent tous w bits.

En ce qui concerne WAH et Concise, la structure utilisée pour EWAH nous rend difficile de reconnaître un seul mot dans la séquence comme un mot de marqueur ou un mot sale, sans lire la séquence depuis le début. De ce fait, en dehors des situations exceptionnelles, une énumération inverse des bits de la séquence est presque impossible.

4.2.4 ROARING

Dans tous les modèles précédents, l'accès aléatoire rapide aux bits dans une séquence arbitraire est relativement difficile. À tout le moins, le mot qui contient le bit à lire doit être identifié, et la position de ce mot dans le flux requiert une connaissance du nombre de mots littéraux ou pleins qui apparaissent plus tôt. Outre les algorithmes de modèle RLE, il existe d'autres modèles de compression bitmap qui prennent en charge un accès aléatoire rapide similaire à des bitmaps sans compression. L'un d'eux est appelé "Roaring bitmap" (Chambi et al., 2015), que nous allons décrire brièvement.

Roaring bitmap a une structure compacte et efficace de données d'indexation en deux niveaux qui divise les indices de 32 bits en blocs, dont chacun stocke les 16 bits les plus significatifs d'un nombre entier de 32 bits et pointe à un conteneur spécialisé stockant les 16 bits les moins significatifs. Il existe deux types de conteneurs : un tableur d'entiers de 16 bits triés pour les *creux* morceaux, qui stockent au maximum 4096 entiers, et un bitmap pour les *denses* morceaux qui stockent 2^{16} entiers. Cette structure de données hybride permet l'accès aléatoire rapide alors que tous les algorithmes de modèles RLE mentionnés ne peuvent pas en raison des caractéristiques mentionnées plus tôt.

4.2.5 DISCUSSION

Les algorithmes de modèles RLE partagent certaines fonctions communes et ont également leurs propres caractéristiques. Tout d'abord, ils ont tout deux types de mots, dont l'un est de stocker le mot non compressé brut (mot littéral) et l'autre est le mot compressé (mot de séquence) qui a un bit et un nombre. Le nombre représente le nombre de mots consécutifs qui sont pleins de bits de zéros ou d'uns qui sont déterminés par le bit.

Nous utilisons une variable *wlen* pour représenter le nombre de bits dans un mot, une variable *ulen* pour le nombre de bits disponibles dans un mot littéral et une variable *wcap* pour le nombre maximal de bits stockés dans un mot de séquence. Le Tableau 4.1 liste les paramètres des trois algorithmes de modèles RLE.

	ulen	wlen	wcap
WAH	31 bits	32 bits	$2^{30}-1$
Concise	31 bits	32 bits	$2^{25}-1$
EWAH	32 or 64 bits	32 or 64 bits	$2^{16}-1$ or $2^{32}-1$

Tableau 4.1: Paramètres d'algorithmes de modèles RLE

Étant donné qu'un bitmap de n bits a m séquences de bits consécutifs comme (0...1...) :

$$c_0^1 c_1^0 c_1^1 c_1^0 c_2^1 c_2^0 ... c_{m-1}^1 c_{m-1}^0, c_j^i \text{ est le nombre de}$$
$$i \text{ bits consécutifs et } i \in (0,1), 0 \leq j \leq m.$$

Alors le nombre de bits au total, c.-à-d. la taille du bitmap non compressé est :

$$bits\_totaux = \sum_{j=0}^{m-1} \sum_{i=0}^{1} c_j^i = \sum_{j=0}^{m-1} \sum_{i=0}^{1} l_j^i + s_j^i,$$
$$l_j^i = c_j^i \bmod ulen, s_j^i = c_j^i - l_j^i$$

S'il y a un nombre entier positif $slen$, $\forall c_j^i = slen$, alors

$$m = n \div (2 \times slen) \tag{4.1}$$

Lorsque $1 \leq slen < wlen$, alors $\forall l_j^i > 0, \forall s_j^i = 0$, ce qui est considéré comme le pire cas, la taille du bitmap compressé est :

$$bits\_compressés = \lceil \frac{bits\_totaux}{ulen} \rceil \times wlen$$

Aucun des trois algorithmes de modèles RLE ne peut bien compresser ce genre de bitmaps. *WAH* et *Concise* perdent un bit pour l'identification de types et *EWAH* semble coûter le moins grâce à $ulen = wlen$, mais sa taille actuelle devrait être un peu plus de *bits_totaux* parce qu'au moins un mot de séquence est nécessaire pour stocker le nombre de mots littéraux.

En outre, lorsque $wlen \leq slen$, alors $\forall s_j^i > 0$, la séquence peut être bien compressée avec tout

algorithme de modèles RLE. Supposons $\forall l_j^i > 0$, la taille du bitmap compressé est :

$$\begin{aligned} bits\_compressés &= \sum_{j=0}^{m-1} \sum_{i=0}^{1} \lceil \frac{slen}{wcap} \rceil \times wlen + wlen \\ &= 2 \times m \times wlen \times (1 + \lceil \frac{slen}{wcap} \rceil) \end{aligned}$$

De cette discussion, nous pouvons savoir que la variable *slen*, c.-à-d. le nombre de bits consécutifs d'uns ou de zéros dans une séquence, est un argument crucial et capable de décider le taux de compression d'un algorithme de modèle RLE. Une optimisation comme Concise est seulement un essai de l'amélioration de la performance du pire cas.

4.3 ÉVALUATION DE FORMULES LTL AVEC BITMAPS

Comme il a été montré que les bitmaps sont très efficaces pour stocker et manipuler des ensembles d'entiers encodés, dans cette section, nous décrivons une technique d'évaluation des formules arbitraires exprimées en Logique Temporelle Linéaire sur une trace donnée d'événements par des manipulations bitmap.

4.3.1 FONCTIONS BITMAP

On suppose qu'une structure bien conçue de données bitmap met en œuvre un certain nombre de fonctions de base. Compte tenu des bitmaps a, b, nous noterons $|a|$ en tant que la fonction qui calcule la longueur de a. La notation $a \otimes b$ désignera la logique au niveau du bit ET de a et b, $a \oplus b$ au niveau du bit OU, et $!a$ au niveau du bit NON.

Fonction	Description
addMany(bitmap, val, len)	Elle ajoute une séquence de *len* bits de la même valeur *val* à la fin du bitmap dont la taille augmente alors par *len*.
copyTo(bitmapDest, bitmapSrc, start, len)	Elle copie la séquence de *len* bits à partir de l'index *start* dans le bitmap *bitmapSrc* à la fin d'un autre bitmap *bitmapDest* dont la taille augmente alors par *len*.
removeFirstBit(bitmap)	Elle supprime le premier bit du bitmap et la taille du bitmap diminue par 1.
next(b, bitmap, start)	Elle retourne la position de la prochaine apparition du bit de la valeur *b* de la position inclusive *start* du bitmap, ou -1 s'il n'y a pas de tel bit.
last(b, bitmap)	Elle retourne la position de la dernière apparition du bit de la valeur *b* dans le bitmap, ou -1 s'il n'y a pas de tel bit.

Tableau 4.2: Fonctions bitmap dérivés

Ces fonctions bitmap seraient suffisantes pour évaluer les opérateurs LTL, mais dans le but d'optimiser notre solution et d'intégrer plus étroitement avec les algorithmes de compression bitmap montrés dans la Section 4.2, nous avons besoin de manipuler la structure de données interne du bitmap et donc d'introduire sept fonctions bitmap dérivés (montrés dans le Tableau 4.2).

4.3.2 MANIPULATION DE BITMAPS POUR METTRE EN ŒUVRE LES OPÉRATEURS LTL

Nous sommes maintenant prêts à définir une procédure d'évaluation des formules LTL arbitraires avec l'aide de bitmaps. Étant donnée une séquence finie d'états $(s_0, s_1, ..., s_{n-1})$ et une formule LTL φ, le principe est de calculer un bitmap $(b_0 b_1 ... b_i b_{i+1} ... b_{n-1})$ de n bits, a noté

B_φ, dont le contenu est défini comme suit :

$$b_i = \begin{cases} 1 & \text{si } \overline{s}^i \models \varphi \\ 0 & \text{autrement} \end{cases} \tag{4.2}$$

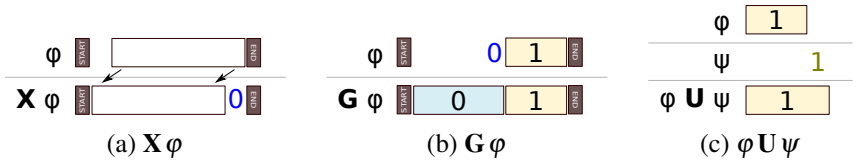
L'ensemble fini de propositions atomiques constituent les bitmaps initiaux. Ces bitmaps de base sont créés par lire la trace originale, et mettre le i-ème bit de B_p à 1 si la proposition atomique est vraie à l'état correspondant s_i, et le mettre autrement à 0. On peut voir que cette construction respecte la Définition 4.2 dans le cas de termes atomiques.

De ces bitmaps initiaux, des bitmaps correspondants à des formules de plus en plus complexes peuvent maintenant être récursivement calculés. Les cas de conjonction, de disjonction et de négation sont faciles à traiter, puisque ces opérateur ont leurs équivalents directs comme les opérateurs au niveau du bit. Par exemple, étant donnés les bitmaps B_φ et B_ψ, le bitmap $B_{\varphi \wedge \psi}$ peut être obtenue en calculant $B_\varphi \otimes B_\psi$. Les autres opérateurs propositionnels peuvent être facilement convertis à leurs opérateurs au niveau du bit correspondants à travers les identités standards.

Les opérateurs logiques temporelles sont un peu plus compliqué, car ils concernent le changement des états en termes de temps, ce qui requiert potentiellement l'énumération des états actuels et des bits dans les bitmaps.

Quelques-uns d'entre eux peuvent encore être traités facilement. L'expression $\mathbf{X}\ \varphi$ indique que φ doit être valide à l'état suivant de la trace. Pour le calcul du bitmap $B_{\mathbf{X}\ \varphi}$, il suffit de supprimer le premier état de B_φ, déplacer les bits restants une position vers la gauche, et mettre le dernier bit à 0. Ceci est illustré dans la Figure 4.1 (a), et formalisé dans l'Algorithme 1.

Pour calculer le vecteur de $\mathbf{G}\ \psi$, il suffit de trouver la plus petite position i de telle sorte que



(a) $\mathbf{X}\,\varphi$ (b) $\mathbf{G}\,\varphi$ (c) $\varphi\,\mathbf{U}\,\psi$

Figure 4.1: Une représentation graphique du calcul des trois opérateurs temporels sur bitmaps

Algorithme 1 Calcul de $\mathbf{X}\ a$

Entrée: Bitmap a

```
1: out ← removeFirstBit(a)
2: addMany(out, 0, 1)
3: retourner out
```

tous les bits suivants sont 1. Dans $B_{\mathbf{G}\ \psi}$, tous les bits devant i sont mis à 0, et tous les bits d'après (inclusivement) i sont mis à 1. Ainsi, pour mettre en œuvre cet opérateur avec des bitmaps, nous avons besoin de faire une recherche dans le bitmap B_ψ de l'arrière vers l'avant pour trouver la dernière apparition du bit 0, comme on peut le voir de l'Algorithme 2.

L'opérateur **F** (montré dans l'Algorithme 3) est le double de **G** ; son algorithme correspondant fonctionne de la même manière que pour **G**, en échangeant 0 et 1.

Algorithme 2 Calcul de $\mathbf{G}\ a$

Entrée: Bitmap a

```
1: p ← last(0, a)
2: si p = −1 alors
3:     retourner a
4: sinon
5:     out ← ⟨ ⟩
6:     addMany(out, 0, p + 1)
7:     addMany(out, 1, |a| − p − 1)
8:     retourner out
9: fin si
```

D'après la Définition (2.12), s'il y a un indice j avec qui $\overline{s}^j \models \psi$ et $\overline{s}^i$ pour tout $i < j$, alors $\overline{s} \models \varphi\ \mathbf{U}\ \psi$. En termes d'opérations bitmap, nous avons besoin de continuer à vérifier s'il y a du bit mis à 1 dans le bitmap B_φ devant chaque apparition de bit 1 dans B_ψ (montré dans

Algorithme 3 Calcul de **F** *a*

Entrée: Bitmap *a*
1: $pos \leftarrow$ last(1, *a*)
2: **si** $pos = -1$ **alors**
3: **retourner** *a*
4: **sinon**
5: $out \leftarrow$ empty Bitmap
6: addMany(*out*, 1, $pos + 1$)
7: addMany(*out*, 0, $|a| - pos - 1$)
8: **retourner** *out*
9: **fin si**

l'Algorithme 4).

L'opération ψ **W** φ (la Définition (2.13)) est tout à fait semblable à ψ **U** φ, sauf que comme l'implique l'équation (4.3) (Huth et Ryan, 2004), l'opération de la première comprend également l'opération $\mathbf{G}\psi$. L'Algorithme 5 explique son opération.

$$\psi \mathbf{W} \varphi \equiv \psi \mathbf{U} \varphi \vee \mathbf{G} \psi \tag{4.3}$$

Comme le double de l'opérateur **U**, l'opérateur **R** défini dans (2.14) a besoin de faire une union de deux parties pour la formule ψ **R** φ : la première partie vise à détecter si existent $i, j (0 \leq i < j)$ qui rendent la séquence $\pi^i, \pi^{i+1}, ...\pi^j$ satisfaite à φ lorsque π^j satisfait ψ ; et la seconde est tout simplement $G\varphi$. L'Algorithme 6 décrit cette procédure.

4.3.3 DISCUSSION

Un point intéressant des derniers trois algorithmes est que les bitmaps *a* et *b* ne sont pas toujours traversés de façon linéaire. Au contraire, des blocs entiers de chaque bitmap peuvent être contournés pour atteindre directement le prochain bit de 0 ou 1, selon le cas. Il faut

Algorithme 4 Calcul de a **U** b

Entrée: Bitmaps a et b

1: $out \leftarrow \langle\,\rangle$
2: $p, a_0, a_1, b_0, b_1 \leftarrow 0$
3: **tant que** $p < |a|$ **faire**
4: **si** $a_1 \leq p$ **alors**
5: $a_1 \leftarrow \text{next}(1, a, p)$
6: **fin si**
7: **si** $b_1 \leq p$ **alors**
8: $b_1 \leftarrow \text{next}(1, b, p)$
9: **fin si**
10: **si** $a_1 = -1$ or $b_1 = -1$ **alors**
11: **sortir tant que**
12: **fin si**
13: $nearest1 \leftarrow \min(a_1, b_1)$
14: **si** $nearest1 > p$ **alors**
15: addMany(out, 0, $nearest1 - p$)
16: $p \leftarrow nearest1$
17: **continuer**
18: **fin si**
19: **si** $p = b_1$ **alors**
20: **si** $b_0 \leq b_1$ **alors**
21: $b_0 \leftarrow \text{next}(0, b, b_1)$
22: **si** $b_0 = -1$ **alors**
23: $b_0 \leftarrow |a|$
24: **fin si**
25: **fin si**
26: addMany(out, 1, $b_0 - p$)
27: $p \leftarrow b_0$
28: **continuer**
29: **fin si**
30: **si** $a_0 \leq a_1$ **alors**
31: $a_0 \leftarrow \text{next}(0, a, a_1)$
32: **si** $a_0 = -1$ **alors**
33: $a_0 \leftarrow |a|$
34: **fin si**
35: **fin si**
36: **si** $a_0 \geq b_1$ **alors**
37: addMany(out, 1, $b_1 - p + 1$)
38: $p \leftarrow b_1 + 1$
39: **sinon**
40: addMany(out, 0, $a_0 - p + 1$)
41: $p \leftarrow a_0 + 1$
42: **fin si**
43: **fin tant que**
44: **si** $b_1 = -1$ **alors**
45: addMany(out, 0, $|a| - |out|$))
46: **sinon si** $a_1 = -1$ **alors**
47: copyTo(out, b, p, $|a| - p$)
48: **fin si**
49: **retourner** out

remarquer que ceci est possible uniquement si la trace est complètement connue à l'avance avant de commencer à évaluer une formule (et de plus, la trace est traversée vers l'arrière). Par conséquent, la solution proposée est un exemple d'un moniteur en mode hors ligne qui n'est pas simplement un moniteur en ligne qui est fourni des événements d'une trace pré-enregistrée un par un : il exploite la possibilité de *accès aléatoire* à des parties de la trace qui est seulement possible dans un réglage hors ligne.

Cet exemple montre l'un des avantages de notre technique proposée en termes de complexité.

Algorithme 5 Calcul de a **W** b

Entrée: Bitmaps a et b

$out \leftarrow \langle\,\rangle$
$p, a_0, a_1, b_0, b_1 \leftarrow 0$
tant que p < |a| **faire**
  si $a_1 \leq p$ **alors**
    $a_1 \leftarrow$ next(1, a, p)
  fin si
  si $b_1 \leq p$ **alors**
    $b_1 \leftarrow$ next(1, b, p)
  fin si
  si $a_1 = -1$ or $b_1 = -1$ **alors**
    sortir tant que
  fin si
  $nearest1 \leftarrow \min(a_1, b_1)$
  si $nearest1 > p$ **alors**
    addMany(out, 0, $nearest1 - p$)
    $p \leftarrow nearest1$
    continuer
  fin si
  si $p = b_1$ **alors**
    si $b_0 \leq b_1$ **alors**
      $b_0 \leftarrow$ next(0, b, b_1)
      si $b_0 = -1$ **alors**
        $b_0 \leftarrow |a|$
      fin si
    fin si
    addMany(out, 1, $b_0 - p$)
    $p \leftarrow b_0$
    continuer
  fin si
  si $a_0 \leq a_1$ **alors**
    $a_0 \leftarrow$ next(0, a, a_1)
    si $a_0 = -1$ **alors**
      $a_0 \leftarrow |a|$
    fin si
  fin si
  si $a_0 \geq b_1$ **alors**
    addMany(out, 1, $b_1 - p + 1$)
    $p \leftarrow b_1 + 1$
  sinon
    addMany(out, 0, $a_0 - p + 1$)
    $p \leftarrow a_0 + 1$
  fin si
fin tant que
si $b_1 = -1$ **alors**
  si $a_1 = -1$ **alors**
    addMany(out, 0, $|a| - |out|$)
  sinon
    $last0 \leftarrow$ last(0, a)
    si $last0 = -1$ or $last0 < p$ **alors**
      addMany(out, 1, $|a| - |out|$)
    sinon
      addMany(out, 0, $last0 - p + 1$)
      addMany(out, 1, $|a| - |out|$)
    fin si
  fin si
sinon si $a_1 = -1$ **alors**
  copyTo(out, b, $|b| - b_1 - 1$, $|b| - b_1$)
fin si
retourner out

Algorithme 6 Calcul de a **R** b

Entrée: Bitmaps a et b

1: $out \leftarrow \langle\,\rangle$
2: $p, a_0, a_1, b_0, b_1 \leftarrow 0$
3: **tant que** $p < |a|$ **faire**
4: **si** $b_1 \leq p$ **alors**
5: $b_1 \leftarrow \text{next}(1, b, p)$
6: **fin si**
7: **si** $b_1 = -1$ **alors**
8: **sortir tant que**
9: **fin si**
10: **si** $b_1 > p$ **alors**
11: addMany(out, 0, $b_1 - p$)
12: $p \leftarrow b_1$
13: **continuer**
14: **fin si**
15: **si** $a_1 \leq p$ **alors**
16: $a_1 \leftarrow \text{next}(1, a, p)$
17: **fin si**
18: **si** $a_1 = -1$ **alors**
19: **sortir tant que**
20: **fin si**
21: **si** $b_0 \leq b_1$ **alors**
22: $b_0 \leftarrow \text{next}(0, b, b_1)$
23: **si** $b_0 = -1$ **alors**
24: $b_0 \leftarrow |a|$
25: **fin si**
26: **fin si**
27: **si** $a_1 \geq b_0$ **alors**
28: addMany(out, 0, $b_0 - p + 1$)
29: $p \leftarrow b_0 + 1$
30: **continuer**
31: **fin si**
32: **si** $a_0 \leq a_1$ **alors**
33: $a_0 \leftarrow \text{next}(0, b, a_1)$
34: **si** $a_0 = -1$ **alors**
35: $a_0 \leftarrow |a|$
36: **fin si**
37: **fin si**
38: $nearest0 \leftarrow \min(a_0, b_0)$
39: addMany(out, 1, $nearest0 - p$)
40: $p \leftarrow nearest0$
41: **fin tant que**
42: **si** $a_1 = -1$ and $b_1 \neq -1$ **alors**
43: $last0 \leftarrow \text{last}(0, b)$
44: **si** $last0 = -1$ and $last0 < p$ **alors**
45: addMany(out, 1, $|a| - |out|$)
46: **sinon**
47: addMany(out, 0, $last0 - p + 1$)
48: addMany(out, 1, $|a| - |out$)
49: **fin si**
50: **sinon**
51: addMany(out, 0, $|a| - |out|$)
52: **fin si**
53: **retourner** out

En effet, la lecture du journal d'origine pour la création des bitmaps fragmentés peut être faite en temps linéaire (et en une seule passe pour tous les symboles propositionnels à la fois). Cependant, une fois que ces bitmaps initiaux sont calculés, un grand nombre des opérations nécessaires ne requièrent plus un traitement linéaire de la trace. Par exemple, l'évaluation $\mathbf{X}\,\varphi$ nécessite un simple décalage de bits, qui peut être fait dans une seule opération de CPU pour 64 bits à la fois, et potentiellement beaucoup plus si la compression est appliquée. [1] De la même façon, la recherche du prochain bit de 0 ou 1 requiert rarement la recherche linéaire, car l'utilisation de la compression permet de contourner des mots pleins avec une opération. Le calcul du bitmap pour un opérateur **F** ou **G** exige une seule telle recherche pour toute la trace.

Un autre point intéressant est le fait que les opérateurs **F** et **G** sont monotones. Comme on peut le voir dans la Figure 4.1, le bitmap résultant est sous forme de 0^*1^* (ou l'inverse). Ainsi, un bitmap très simple se propage vers d'autres algorithmes ; il peut être fortement compressé, et rend toute recherche du prochaine 0 ou 1 trivial. Bien que les bitmaps résultant des applications de **U**, de **W** et de **R** ne produisent pas de vecteurs tellement simples, ils ont toujours une structure relativement régulière qui est aussi souple pour la compression raisonnable.

4.4 IMPLÉMENTATION ET EXPÉRIENCES

Alors que la complexité du pire cas de chaque algorithme présenté dans la section précédente est encore $O(n)$ (où n est la taille du bitmap d'entrée), nous soupçonnons que la performance pratique devrait être beaucoup mieux. Par conséquent, dans cette section, nous décrivons des expériences en vue d'atteindre les objectifs suivants :

1. Tester la performance des algorithmes LTL fondamentaux ;

1. Le décalage à gauche de bits d'un bloc compressé est le bloc lui-même, aussi longtemps que le premier bit du prochain bloc à droite a la même valeur.

2. Tester la performance de l'application récursive de ces algorithmes sur des formules LTL complexes ;
3. Évaluer la performance et l'espace économique occasionnés grâces à l'utilisation de la compression.

4.4.1 PRÉPARATION DES EXPÉRIENCES

Comme un moyen d'éviter les coûts d'entrées ou de sorties de disques à l'exécution, nous chargeons tous les fichiers pertinents dans la mémoire avant les calculs. Ainsi, bien que l'utilisation de bitmaps peut considérablement réduire l'exigence de mémoire, nous avons préparé un poste de travail qui a un processeur d'Intel Xeon E5-2630 v3 et 48 Go de mémoire.

Tous les codes sont mis en œvre en Java qui prend en soi la responsabilité de la gestion de la mémoire et de la collecte des ordures. En ce qui concerne le délai causé par la collecte des ordures (GC) et surtout Full-GC, nous avons manuellement effectué *System.gc()* avant et après chaque calcul de formules pour fournir un environnement d'exécution qui était aussi "propre" que possible.

Le Tableau 4.3 montre les librairies utilisées pour différents types de bitmap. Afin de mettre en œuvre toutes les opérations LTL, nous avons modifié les codes des librairies pour ajouter les fonctions nécessaires énumérées dans le Tableau 4.2 et optimiser les fonctions de sorte que les complexités en temps des opérateurs deviennent $O(m)$ où m est le nombre de séquences de bits consécutifs de 0 ou de 1.

En raison du manque de soutien de l'accès aléatoire pour les algorithmes de modèles RLE de compression bitmap, nous ne pouvons pas énumérer les bits de la même manière qu'un bitmap non compressé. Par conséquent, nous avons conçu une structure de données *itérateur*

Librairie bitmap	Source
Non compressée	`java.util.BitSet` à partir de SDK Java
WAH	Originale : `https://github.com/metamx/extendedset` Modifiée : `https://github.com/phoenixxie/extendedset`
Concise	Originale : `https://github.com/metamx/extendedset` Modifiée : `https://github.com/phoenixxie/extendedset`
EWAH	Originale : `https://github.com/lemire/javaewah` Modifiée : `https://github.com/phoenixxie/javaewah`
Roaring	`https://github.com/lemire/RoaringBitmap`

Tableau 4.3: Librairies bitmap

pour stocker non seulement l'indice absolu du bit actuel dans le correspondant bitmap non compressé, mais aussi l'indice relatif dans le bitmap compressé. Prenant l'exemple de la fonction **next(1, x)**, si l'indice relatif actuel est dans un mot de séquence de 0, la recherche dans ce mot est inutile, et nous passons simplement au mot suivant ; si l'indice est dans un mot de séquence de 1, on retourne l'index actuel ; cependant, si l'indice est dans un mot littéral, nous devons chercher le bit 1 dans le mot de $ulen$ bits.

Pour les expériences, nous avons développé un générateur de données aléatoires. Chaque fois qu'il génère 5×10^7 tuples, et chaque tuple contient trois nombres aléatoires (a, b, c) liés à trois inégalités simples : $a > 0$, $b > 0$ et $c \leq 0$, qui seront étiquetées comme s_0, s_1 et s_2, respectivement. Selon (2.4), les valeurs vrai/faux de ces trois déclarations sont composées des propositions atomiques. Quand un tuple a été passé aux trois déclarations, nous avons obtenu trois valeurs booléennes dont chacune a ensuite été transformée en bit de 1 ou de 0 dans le bitmap correspondant à l'un des trois états. Lorsque tous les tuples ont été traités, nous avions trois bitmaps ayant 50 millions de bits chacun.

4.4.2 OPÉRATEURS LTL FONDAMENTAUX

Une première expérience est formée de l'évaluation de la performance, en termes de temps de calcul, pour évaluer un vecteur de bits sur chaque opérateur propositionnel et temporel.

Dans la première expérience, nous avons effectué 100 passes d'un benchmark sur les opérateurs fondamentaux avec des bitmaps non compressés. Dans chaque passe, les données d'expérience ont été régénérées et transmises aux déclarations relationnelles à partir desquelles les bitmaps ont été créés. Puis les formules ont été exécutées avec les bitmaps. Dans la dernière étape, nous avons calculé le temps moyen d'exécution d'une passe pour chaque opérateur LTL, et le nombre de bits traités par seconde.

Le Tableau 4.4 montre que les opérateurs logiques propositionnels étaient plus rapides que la plupart des opérateurs logiques temporels. Parmi les opérateurs logiques temporels, les opérateurs binaires sont plus lents que celles unaire parce que les premiers requièrent plus d'opérations que les seconds, particulièrement dans la situation où plusieurs séquences de 0s et de 1s sont mélangées dans le bitmap. Les duals opérateurs **G** et **F** ont des algorithmes similaires, mais **F** a pris étonnamment trois fois plus longtemps que **G**. Ceci peut être expliqué par le fait que pour un bitmap d'entrée assez randomisé, **F** ajoute plus de bits de 1s que de 0s à son bitmap de sortie, tandis que **G** ajoute plus de bits de 0s que de 1s. Bien que l'implémentation de `BitSet` en Java puisse mettre un bit à 1 et à 0[2], elle ne fait actuellement rien quand elle est demandée de mettre à 0 un nouveau bit dont l'indice est au-delà de sa taille, c'est-à-dire l'ajout d'un bit 0. Il en résulte un traitement asymétrique de bits de 0s et de 1s dans le bitmap.

2. `https://docs.oracle.com/javase/8/docs/api/java/util/BitSet.html`

Formule	Temps Min. (ms)	Temps Max. (ms)	Temps Moyen (ms)	Débit (b/s)
$\neg s_0$	0	15	6.18	8.09×10^9
$s_0 \wedge s_1$	0	16	5.86	8.53×10^9
$s_0 \vee s_1$	0	16	5.8	8.62×10^9
$s_0 \rightarrow s_1$	0	16	4.66	1.07×10^{10}
$\mathbf{X}\, s_0$	0	16	8.93	5.60×10^9
$\mathbf{G}\, s_0$	46	63	51.3	9.75×10^8
$\mathbf{F}\, s_0$	140	174	150.55	3.32×10^8
$s_0\ \mathbf{U}\ s_1$	1562	2017	1747.05	5.72×10^7
$s_0\ \mathbf{W}\ s_1$	1531	1957	1685.71	5.93×10^7
$s_0\ \mathbf{R}\ s_1$	1735	2188	1961.37	5.10×10^7

Tableau 4.4: Temps d'exécution d'évaluation de chaque opérateur LTL sur un vecteur de bits, sans l'utilisation de librairie de compression.

4.4.3 FORMULES COMPLEXES

Les résultats de cette première expérience suggèrent que les opérateurs logiques propositionnels, les opérateurs logiques temporels unaires et les opérateurs logiques temporels binaires ont des magnitudes différentes de vitesse de traitement ; par conséquent, nous pouvons diviser les opérateurs en trois groupes.

Au début de cette deuxième expérience, nous avons composé diverses combinaisons d'opérateurs en 14 formules LTL avec l'aide de l'outil *randltl* de la librairie *Spot* [3] ; les formules sont présentées dans le Tableau 4.5. Ensuite, nous avons également effectué un benchmark de 50 passes sur ces formules avec des bitmaps non compressés. Dans chaque cycle les données ont été régénérées et ré-exécutées avec les 14 formules. Nous avons mesuré le temps d'exécution de chaque cycle et calculé le coût du temps moyen et la vitesse de traitement comme avant.

Comme il est indiqué dans le Tableau 4.6, trois groupes d'opérateurs ont différentes échelles de vitesse de traitement. Les combinaisons ayant les opérateurs logiques temporels et binaires

3. `https://spot.lrde.epita.fr/index.html`

$$\mathbf{G}\,((s_2 \rightarrow \mathbf{F}\,(\neg(s_1\ \mathbf{U}\ s_2)\ \mathbf{W}\ (s_2 \vee \mathbf{G}\,s_1)))\ \mathbf{W}\ (\neg\mathbf{F}\,(s_0\ \mathbf{R}\ \mathbf{X}\,s_2)\ \mathbf{W}\ ((s_0 \wedge s_2 \wedge \mathbf{F}\,s_2)\ \mathbf{U}\ s_0))) \quad \text{(F1)}$$

$$\mathbf{F}\,(\neg(s_2 \rightarrow \mathbf{X}\,(s_0\ \mathbf{U}\ s_1))\ \mathbf{U}\ (\neg(s_0 \vee \mathbf{F}\,\mathbf{X}\,(s_0\ \mathbf{U}\ (\mathbf{X}\,(\mathbf{F}\,s_1\ \mathbf{W}\ s_1)\ \mathbf{R}\ s_1)))\ \mathbf{U}\ (s_0\ \mathbf{R}\ \mathbf{G}\,s_2))) \quad \text{(F2)}$$

$$\mathbf{X}\,\mathbf{F}\,((s_1 \vee s_2 \vee (\mathbf{G}\,(s_0 \vee s_1 \vee \neg s_1) \wedge \mathbf{X}\,\neg s_0)) \rightarrow ((\neg s_0 \rightarrow (s_0 \wedge \neg s_1)) \wedge \mathbf{G}\,s_0)) \quad \text{(F3)}$$

$$\mathbf{X}\,(\neg\mathbf{G}\,(s_0 \rightarrow s_2) \rightarrow \mathbf{F}\,(s_1 \wedge ((\mathbf{F}\,(s_0 \wedge s_2) \rightarrow s_1) \rightarrow \mathbf{X}\,\neg s_2) \wedge \mathbf{G}\,(s_2 \rightarrow (s_2 \wedge \mathbf{F}\,s_1)))) \quad \text{(F4)}$$

$$\neg((s_0\ \mathbf{U}\ (\neg(\neg s_0 \wedge s_2) \vee (\neg s_0\ \mathbf{W}\ (s_2 \rightarrow s_0))))\ \mathbf{W}\ \neg s_0) \vee (s_1\ \mathbf{R}\ ((s_1 \vee (s_0\ \mathbf{W}\ s_2))\ \mathbf{W}\ (\neg s_0\ \mathbf{W}\ s_2))) \quad \text{(F5)}$$

$$(s_1\ \mathbf{W}\ ((s_2 \rightarrow (\neg s_2\ \mathbf{R}\ \neg(\neg s_1\ \mathbf{W}\ s_0)))\ \mathbf{W}\ (\neg s_1 \vee \neg((\neg s_2 \rightarrow s_1) \rightarrow \neg s_0))))\ \mathbf{W}\ (s_0\ \mathbf{R}\ \neg s_2) \quad \text{(F6)}$$

$$\mathbf{X}\,(((\mathbf{F}\,s_2\ \mathbf{R}\ s_0)\ \mathbf{U}\ \mathbf{F}\,s_0)\ \mathbf{R}\ \mathbf{G}\,((s_2\ \mathbf{W}\ s_1)\ \mathbf{W}\ (((\mathbf{G}\,s_2\ \mathbf{U}\ s_1)\ \mathbf{R}\ \mathbf{X}\,s_0)\ \mathbf{R}\ (s_2\ \mathbf{W}\ ((s_2\ \mathbf{R}\ \mathbf{X}\,s_2)\ \mathbf{W}\ s_1))))) \quad \text{(F7)}$$

$$(\mathbf{G}\,(s_0\ \mathbf{R}\ \mathbf{F}\,s_1)\ \mathbf{U}\ \mathbf{F}\,s_2)\ \mathbf{W}\ \mathbf{G}\,((s_1\ \mathbf{U}\ s_2)\ \mathbf{R}\ ((\mathbf{G}\,\mathbf{X}\,s_0\ \mathbf{U}\ (s_2\ \mathbf{W}\ s_0))\ \mathbf{W}\ \mathbf{F}\,((\mathbf{G}\,s_1\ \mathbf{U}\ s_2)\ \mathbf{R}\ s_2))) \quad \text{(F8)}$$

$$\mathbf{G}\,\mathbf{F}\,(\mathbf{G}\,\mathbf{F}\,s_0 \wedge \mathbf{F}\,\mathbf{X}\,\mathbf{G}\,s_1 \wedge \mathbf{G}\,\mathbf{F}\,\mathbf{X}\,\mathbf{X}\,\mathbf{X}\,\mathbf{G}\,\mathbf{X}\,\mathbf{F}\,\mathbf{G}\,s_2) \quad \text{(F9)}$$

$$\mathbf{F}\,\mathbf{G}\,\mathbf{F}\,\mathbf{X}\,(\mathbf{X}\,s_2 \wedge \mathbf{X}\,\mathbf{G}\,\mathbf{X}\,\mathbf{X}\,\mathbf{G}\,\mathbf{F}\,(\mathbf{G}\,\mathbf{X}\,\mathbf{F}\,s_1 \wedge \mathbf{X}\,\mathbf{G}\,s_0)) \quad \text{(F10)}$$

$$\neg(((s_0 \vee s_2) \rightarrow (\neg(s_2 \wedge (\neg s_2 \rightarrow \neg(s_0 \wedge (s_0 \vee \neg s_1))))) \vee (s_0 \wedge \neg s_0))) \vee (\neg s_0 \wedge (s_0 \vee s_2))) \quad \text{(F11)}$$

$$(s_1 \wedge \neg s_2 \wedge (s_2 \rightarrow s_0)) \vee \neg((s_0 \wedge \neg s_0) \rightarrow s_1) \vee ((s_2 \vee (s_1 \rightarrow s_0)) \wedge ((s_0 \wedge \neg s_2 \wedge (s_1 \rightarrow s_0)) \rightarrow s_0)) \quad \text{(F12)}$$

$$(((s_0\ \mathbf{W}\ s_2)\ \mathbf{W}\ s_0)\ \mathbf{U}\ ((s_1\ \mathbf{U}\ (((s_1\ \mathbf{W}\ s_2)\ \mathbf{W}\ (s_1\ \mathbf{R}\ (s_1\ \mathbf{R}\ s_0)))\ \mathbf{W}\ s_2))\ \mathbf{W}\ s_2))\ \mathbf{W}\ ((s_0\ \mathbf{R}\ s_1)\ \mathbf{R}\ (((s_2\ \mathbf{U}\ s_1)\ \mathbf{U}\ s_1)\ \mathbf{R}\ ((s_0\ \mathbf{W}\ s_2)\ \mathbf{W}\ s_1))) \quad \text{(F13)}$$

$$((((s_1\ \mathbf{U}\ s_2)\ \mathbf{U}\ (s_2\ \mathbf{U}\ s_1))\ \mathbf{U}\ s_1)\ \mathbf{R}\ (s_1\ \mathbf{R}\ s_2))\ \mathbf{U}\ (((s_2\ \mathbf{W}\ ((s_0\ \mathbf{W}\ ((s_2\ \mathbf{R}\ s_0)\ \mathbf{R}\ s_1))\ \mathbf{U}\ s_1))\ \mathbf{W}\ s_0)\ \mathbf{W}\ (((s_0\ \mathbf{R}\ s_1)\ \mathbf{R}\ (s_0\ \mathbf{W}\ s_1))\ \mathbf{U}\ s_0)) \quad \text{(F14)}$$

Tableau 4.5: Les formules LTL complexes évaluées expérimentalement

ont toujours pris plus de temps que d'autres, et les formules F13 et F14 sont les plus lentes. Ce résultat montre également que notre solution peut manipuler un assez grand nombre de bits (événements de la trace) par seconde, allant de millions à des milliards.

Formule No.	Opérateurs Logiques Props.	Opérateurs Temporels Unaires	Opérateurs Temporels. Binaires	Temps Min. (ms)	Temps Max. (ms)	Temps Avg. (ms)	Bits/second Approx.
F1	6	6	6	10454	14205	11483.02	1.31×10^7
F2	4	7	7	7728	10673	8937.59	1.68×10^7
F3	13	5	0	281	422	326.63	4.59×10^8
F4	11	7	0	422	704	560.58	2.68×10^8
F5	11	0	7	8532	10496	9374.5	1.60×10^7
F6	12	0	6	7280	9357	7934.6	1.89×10^7
F7	0	7	11	12330	15004	13413.91	1.18×10^7
F8	0	8	10	9442	11833	10428.37	1.44×10^7
F9	2	16	0	431	1155	682.68	2.20×10^8
F10	2	16	0	375	857	472.76	3.17×10^8
F11	18	0	0	31	56	45.18	3.32×10^9
F12	18	0	0	46	68	51.58	2.91×10^9
F13	0	0	18	22768	27308	24825.21	6.04×10^6
F14	0	0	18	22800	27481	24877.67	6.03×10^6

Tableau 4.6: Temps d'exécution de l'évaluation des formules LTL du Tableau 4.5, sans l'utilisation de librairie de compression.

4.4.4 UTILISATION DE COMPRESSION BITMAP

Selon les algorithmes de modèles RLE, le taux de compression dépend principalement de la longueur de bits consécutifs de 0s ou de 1s. De ce fait, dans cette expérience, nous avons modifié le générateur pour lui permettre de répéter le même tuple un certain nombre de fois : 1, 32 et 64. Ce nouveau mécanisme est en mesure d'assurer l'existence de séquences continues d'une longueur minimum (*slen*) dans les bitmaps générés. Intuitivement, lorsque la valeur de *slen* augmente, le nombre de séquences diminue ; par conséquent, les algorithmes de modèles RLE devraient avoir de meilleures performances qu'un bitmap non compressé.

Dans la première partie de l'expérience, nous avons généré les bitmaps avec les algorithmes différents et les valeurs différentes de *slen*, puis calculé les taux de compression. Le résultat dans la Figure 4.2 confirme l'hypothèse selon laquelle quand $slen < wlen$ (où *wlen* est la longueur d'un mot), le bitmap ne peut pas être bien compressé par un algorithme de modèle

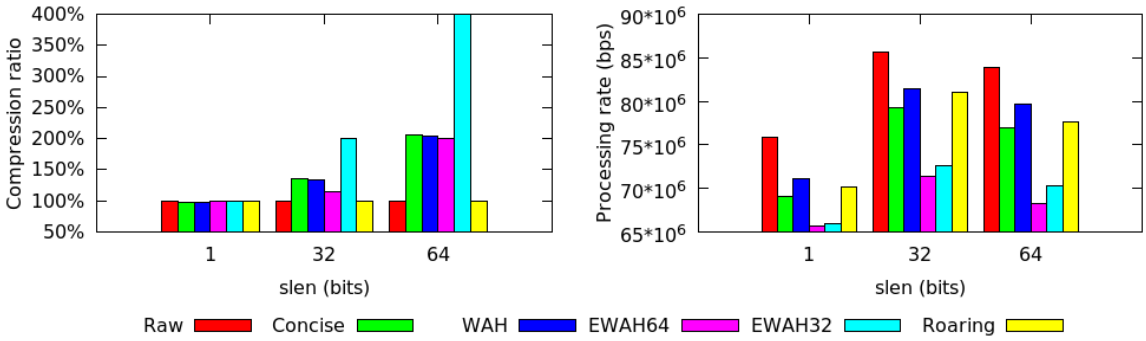


Figure 4.2: Génération de bitmaps avec les algorithmes de compression

RLE, et dans ce cas, l'algorithme EWAH se comporte un peu mieux que les autres en raison de son coût structurel plus petit. Lorsque $slen$ augmente à 32 et à 64, c.-à-d. $slen \geq wlen$, les algorithmes RLE commencent à bien fonctionner et le taux de compression lors de $slen = 64$ est évidemment meilleure que celui lors de $slen = 32$. De la Figure 4.2, nous pouvons également voir que quand $slen$ est égal à 1, à 32 et à 64, EWAHs sont plus lents que WAH, Concise et Roaring.

Dans la deuxième partie de l'expérience, nous avons mesuré les performances des bitmaps compressés lors de l'application des algorithmes pour tous les opérateurs fondamentaux et tous les formules LTL dans les expériences précédentes. Les résultats détaillés couvrant tous les opérateurs et les formules peuvent être trouvés dans l'Annexe A.

À cette fin, nous avons choisi les formules F1 et F14 de l'expérience précédente, car F1 contient tous les opérateurs et les conjonctions de LTL et F14 est la plus lente de toutes les formules dans l'expérience précédente. Nous avons à nouveau effectué le benchmark 100 fois ; dans chaque cycle, la formule a été évalué par un groupe de bitmaps d'entrée de la dernière étape et nous avons enregistré le coût de temps de chaque algorithme bitmap et de chaque longueur de bits consécutifs.

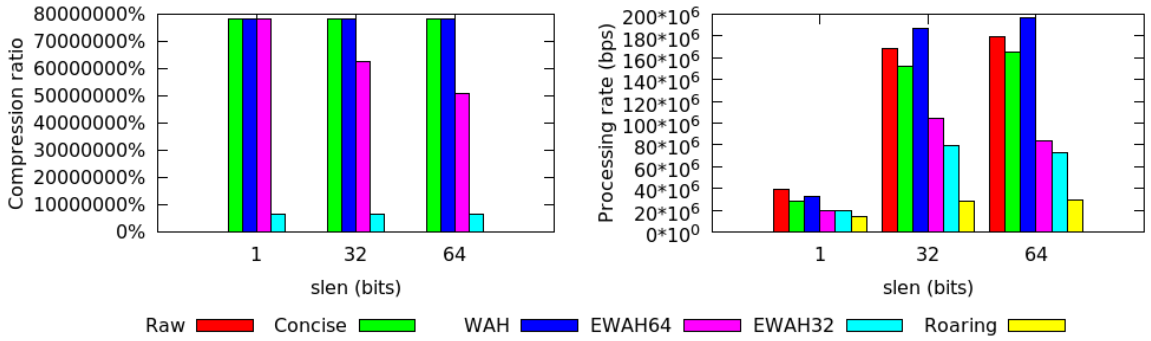


Figure 4.3: Comparaison du taux de compression et de la vitesse de traitement de la formule F1, avec diverses librairies de compression bitmap et différentes valeurs de *slen*

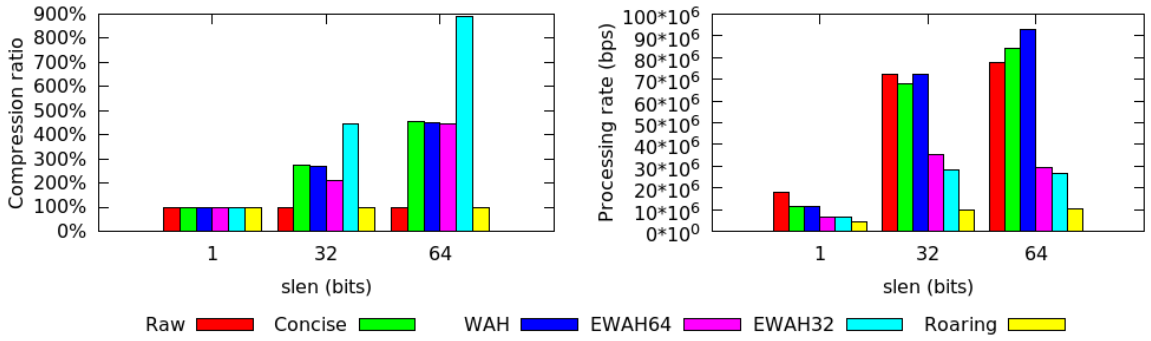


Figure 4.4: Comparaison du taux de compression et de la vitesse de traitement de la formule F14, avec diverses librairies de compression bitmap et différentes valeurs de *slen*

Selon les Figures 4.3 et 4.4, la performance des algorithmes de modèles RLE, WAH, EWAH et Concise est évidemment liée à la valeur de *slen*. La Figure 4.3 suggère également que la présence des opérateurs **G** et **F** peut grandement augmenter la longueur de bits consécutifs de même valeur, qui à leur tour peuvent être bien compressés par des algorithmes de modèles RLE. Dans un tel cas, plusieurs algorithmes ont de meilleures performances que le bitmap non compressé lors de l'augmentation de *slen*.

4.5 TRAVAUX CONNEXES

La perspective de l'utilisation de propriétés physiques de matériel pour améliorer les performances de la vérification de l'exécution a déjà été étudiée dans le passé récent. Par exemple, Pellizzoni et al. (Pellizzoni et al., 2008) ont utilisé du dédié matériel commercial-off-the-shelf (COTS) (Emerson, 1990) pour faciliter le monitoring de l'exécution de systèmes embarqués critiques dont les propriétés ont été exprimées en logique temporelle linéaire en temps passé (ptLTL).

Comme le nombre de cœurs (GPU ou CPU multi-cœurs) dans le matériel de base ne cesse de croître, la recherche de l'exploitation des processeurs disponibles ou des cœur disponibles pour paralléliser les tâches et les calculs apporte un défi et aussi une occasion d'amélioration de l'architecture de la vérification de l'exécution. Par exemple, Ha et al. (Ha et al., 2009) ont présenté une conception de tamponnage de *Cache-friendly Asymmetric Buffering (CAB)* pour améliorer les communications entre l'application et le moniteur d'exécution en utilisant le cache partagé de l'architecture multi-cœurs ; Berkovich et al. (Berkovich et al., 2015) a proposé une solution à base de GPU qui utilise efficacement les cœurs disponibles du GPU, de sorte que le moniteur conçu et implémenté avec leur méthode peut fonctionner en parallèle avec le programme cible et évaluer des propriétés LTL.

Le travail antérieur de l'un des auteurs (Hallé et Soucy-Boivin, 2015) introduit un algorithme pour la vérification automatisée de formules de logiques temporelles linéaires sur des traces d'événements, en utilisant un plus en plus populaire cadre de cloud computing appelé MapReduce. L'algorithme peut traiter plusieurs fragments arbitraires de la trace en parallèle, et calculer le résultat final à travers un cycle d'exécution d'instances de MapReduce. La technique proposée manipule des objets appelés *tuples*, qui sont de la forme $\langle \phi, (n,i) \rangle$, et sont interprétés comme la déclaration "the process is at iteration i, and LTL formula ϕ is true for

the suffix of the current trace starting at its n-th event". On peut voir que cette déclaration correspond exactement au fait, dans la présente solution, que la n-ème position du bitmap généré par l'évaluation de ϕ contient la valeur 1.

Outre cette similitude, toutefois, les deux techniques sont radicalement différentes. Puisque la méthode de MapReduce fonctionne sur les tuples un par un, alors que la présente solution manipule les bitmaps entiers, les algorithmes pour chaque opérateur LTL ont peu en commun (en particulier celui de **U**). Lorsque la méthode de MapReduce obtient sa vitesse du traitement de plusieurs sous-formules sur des machines différentes, notre solution présente est efficace parce que certaines opérations (comme conjonction) peuvent être calculées simultanément pour de nombreux événements adjacents dans un seul cycle de CPU. En plus, un inconvénient de la solution MapReduce est le grand nombre de tuples générées, et l'impossibilité de compression du volume de données.

Comme on peut le voir, il y avait plusieurs tentatives de s'appuyer le parallélisme et les propriétés de matériel pour évaluer les expressions temporelles sur des traces. Cependant, pour autant que l'on le sache, notre travail est le premier à obtenir l'amélioration de performance au niveau de la *structures de données* pour évaluer ces expressions.

4.6 CONCLUSION ET PERSPECTIVES

Nous avons proposé une solution pour l'évaluation hors ligne de formules LTL au moyen de manipulation de bitmaps. Dans un tel contexte, les prédicats propositionnel d'événements individuels d'états d'une trace sont mappés aux bits d'un vecteur ("bitmap") qui sont ensuite manipulés pour mettre en œuvre chaque opérateur LTL. En plus du fait que les manipulations de bitmaps sont en soi très efficaces, nos algorithmes profitent du fait que la trace est complètement connue à l'avance, et que l'accès aléatoire à une position quelconque de cette trace

permet de contourner de grands blocs d'événements pour accélérer l'évaluation.

Pour cette raison, notre solution est un important exemple d'un algorithme d'évaluation hors ligne qui exploite le fait que cela fonctionne bien en mode hors ligne — il n'est pas un algorithme en ligne qui lit les événements à partir d'une trace pré-enregistré un par un. En fait, dans certains cas (comme l'opérateur **U**), la trace est même évaluée à partir de la fin, plutôt que à partir du début. Un approfondi benchmark de performance pour les opérateurs fondamentaux et les formules LTL complexes a prouvé la faisabilité de la solution, et a montré comment les événements d'une trace peuvent être traités à une vitesse allant de millions à des milliards d'événements par seconde.

Pour exploiter davantage le potentiel de bitmap, nous avons présenté des algorithmes de compression bitmap dans notre solution et les avons intégrés dans notre benchmark. Dans les expériences, comme nous l'espérions, notre solution a démontré sa capacité de compresser facilement les creux bitmaps et d'accélérer les opérations LTL quand il y a un certain nombre de bits consécutifs avec la même valeur. Nous avons expliqué, comment de nombreux opérateurs LTL augmentent naturellement la régularité des bitmaps qu'ils traitent.

De toute évidence, cette solution ne convient que à l'évaluation hors ligne. Cependant, les résultats prometteurs obtenus dans notre implémentation conduisent à un grand nombre d'extensions et d'améliorations potentielles basées sur la méthode actuelle. Tout d'abord, l'algorithme peut être réutilisé comme base pour d'autres langages temporels qui se croisent avec LTL, tels que PSL (Eisner et Fisman, 2006). D'autre part, cette technique pourrait être étendue pour prendre en considération des paramètres et de la quantification de données. Enfin, on pourrait aussi envisager la parallélisation de l'évaluation de grands segments de bitmaps sur plusieurs machines.

CHAPITRE 5

CONCLUSION ET PERSPECTIVES

La vérification et la validation du logiciel est un élément critique du génie logiciel et de la gestion de projet. Les gens ont appris cela de plusieurs leçons dans le passé allant des jeux écrasés à la catastrophe fatale. Comparée avec les techniques traditionnelles de la vérification du logiciel, la vérification de l'exécution est relativement nouvelle. Elle a sa racine dans d'autres techniques et elle a ses propres caractéristiques. Ces dernières années, une grande quantité de travaux et de temps ont été investis dans divers aspects de ce domaine. Des chercheurs se concentrent sur l'amélioration et l'application de différentes variantes de LTL, et les autres s'efforcent d'inventer de plus en plus cadres génériques.

Bien que le réseau ait déjà couvert beaucoup de place sur la terre, de nombreux milieux du réseau ont été exploités et divers protocoles réseau ont été proposées, il y a encore de l'endroit où le câble et la radio sans fil ne sont pas parvenus, comme les déserts ou les lieux sous-marines, ou de l'environnement où le câble et la radio sans fil sont indésirables ou même interdits, par exemple, dans les avions, les hôpitaux, les mines ou les usines pétrochimiques. Même dans ces limites, la vérification du logiciel est aussi essentielle que celle dans les lieux de réseautage couvertes. Visible light communication (VLC) est une solution efficace dans ces situations et elle a de nombreux cas avec succès, qui nous ont éclairés de penser à une méthode

de l'utilisation de codes optiques pour le travail de communication de données. Un code QR est une étiquette encodée optique qui est capable de stocker a une quantité considérable de données et d'encoder et de décoder efficacement les données, ce qui nous a donné la confiance pour l'appliquer dans notre solution.

Dans la première partie de notre recherche, l'objectif était de concevoir et de mettre en œuvre un canal de communication unidirectionnel de codes QR. Nous avons utilisé le langage de programmation informatique Java et la librairie bien connue ZXing. Dans les précoces expériences, nous avons testé les caractéristiques d'un flux de données QR et trouvé qu'en raison de la limitation des matériels que nous avons utilisés, le taux de perte de données était impossiblement zéro et il a augmenté vite alors que la taille de données de chaque image croissait. Ainsi, d'une part nous avons réussi à améliorer le taux de reconnaissance de codes QR en ajustant les options de la librairie ZXing et de la caméra. D'autre part, nous avons proposé le protocole BufferTannen qui est en charge de la division, de la transformation, et dans une certaine mesure de la compression de données à transmettre. Le résultat final présenté dans la Section 3.4 prouve la faisabilité de notre solution. Cette partie de notre recherche a été publiée au journal *IEEE Access* en 2016.

Chaque système du logiciel, que ce soit des systèmes de téléphones mobiles, des applications web ou des systèmes de cloud computing, doit garantir son bon fonctionnement et répondre en temps afin de réduire la perte ou d'éviter la catastrophe. L'exigence de la vérification du logiciel pour chaque système pourrait être similaire, comme le suggère la Section 2.3 qui a présenté plusieurs cadres de la vérification de l'exécution qui partagent les mêmes fonctionnalités. Toutefois, l'échelle de chaque système du logiciel varie beaucoup. Par exemple, un smartphone dispose d'une mémoire beaucoup plus petite et d'un processeur beaucoup plus lent qu'un poste de travail populaire, sans compter un système de cloud computing comme Amazon EC2. Lors du développement d'un système de la vérification de l'exécution pour un smartphone, l'usage

de la mémoire et du processeur est toujours la principale préoccupation. D'une autre manière, même dans un système informatique qui a une énorme mémoire et un puissant processeur, il y a toujours une limite de la mémoire et du processeur. Par conséquent, il existe beaucoup de travaux sur l'amélioration de systèmes de la vérification de l'exécution. Certains travaux visent à distribuer le calcul à un cluster de serveurs, tandis que certains réussissent à optimiser les algorithmes pour trouver une meilleure solution.

Notre deuxième objectif était de trouver un moyen d'amélioration de la vitesse d'évaluation de formules LTL. Un bitmap est une structure de données compacte et efficace qui existe dans un grand nombre d'applications. Un verdict émis par le moniteur en mode hors ligne de la vérification de l'exécution est une valeur de vérité qui n'a que deux valeurs, qui peut être facilement mise en correspondance avec un bit d'un bitmap. Nous avons également conçu quelques algorithmes pour mettre en œuvre les opérateurs LTL avec des bitmaps mappés. Kaser et Lemire (2014) suggère qu'un creux bitmap peut être bien compressé, et comme nous l'avons discuté dans la Section 4.3, le bitmap de sortie des algorithmes a des séquences plus longue de bits consécutifs de 1 ou de 0 que les bitmaps d'entrée. Par conséquent, nous avons intégré des algorithmes de compression bitmap dans notre solution. Les résultats expérimentaux démontrent la performance et la faisabilité de notre solution, et prouvent que l'utilisation de la compression bitmap peut rendre notre solution plus rapide et plus espace-économique avec certains algorithmes. Cette partie de notre recherche est également écrite dans un article qui est encore en cours de révision pour sa publication dans les actes de la conférence internationale : Runtime Verification 2016 (RV'16) à Madrid, Espagne en 2016.

Notre canal de communication de codes QR appuie seulement le transfert de données unidirectionnel, ce qui entraîne que même notre programme renvoie la même image de données à plusieurs fois, nous ne pouvons pas nous assurer que le récepteur a reçu toutes les données. Une communication bidirectionnelle semble une réponse raisonnable pour ce problème, et

elle permet aussi de renvoyer des données sur la demande et augmente donc la bande passante effective.

Un bitmap a les bits de 1 ou de 0, donc la valeur *inconcluante* de LTL_3 ne peut pas être facilement mise en œuvre. Par conséquent, notre solution ne fonctionne que pour le monitoring en mode hors ligne. L'Évaluation en ligne de formules LTL avec l'aide de bitmaps serait un grand mais intéressant défi. En outre, la parallélisation de notre solution dans un système de cloud computing est également très intéressante.

Appendices

CHAPITRE A

RÉSULTATS EXPÉRIMENTAUX D'ÉVALUATION DE FORMULES LTL

Dans cette annexe, nous présentons les résultats des expériences de la troisième partie de la Section 4.4. Pour obtenir une meilleure mise en forme, le format de données est différente de celui dans la Section 4.4. "Taux" d'ici n'a pas de marque de pourcentage parce qu'il est le taux de compression calculé par la division de la taille de données d'origine et de la taille compressée.

	slen	1			32			64		
Type		Min.	Max.	Moy.	Min.	Max.	Moy.	Min.	Max.	Moy.
Brut	Taux	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	bps	6.44E+7	8.32E+7	7.59E+7	7.64E+7	9.18E+7	8.57E+7	7.40E+7	9.27E+7	8.39E+7
Concise	Taux	0.97	0.97	0.97	1.35	1.35	1.35	2.06	2.07	2.07
	bps	6.18E+7	7.60E+7	6.91E+7	6.99E+7	8.56E+7	7.93E+7	6.67E+7	8.47E+7	7.70E+7
WAH	Taux	0.97	0.97	0.97	1.33	1.34	1.33	2.03	2.04	2.03
	bps	6.28E+7	7.84E+7	7.12E+7	7.23E+7	8.86E+7	8.15E+7	6.58E+7	8.95E+7	7.97E+7
EWAH64	Taux	1.00	1.00	1.00	1.14	1.14	1.14	1.99	2.01	2.00
	bps	5.66E+7	7.29E+7	6.56E+7	6.24E+7	7.62E+7	7.14E+7	5.99E+7	7.58E+7	6.83E+7
EWAH32	Taux	1.00	1.00	1.00	2.00	2.00	2.00	3.99	4.01	4.00
	bps	5.82E+7	7.34E+7	6.59E+7	6.48E+7	7.74E+7	7.26E+7	6.09E+7	7.73E+7	7.03E+7
Roaring	Taux	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	bps	6.11E+7	7.78E+7	7.01E+7	7.04E+7	8.72E+7	8.10E+7	6.78E+7	8.88E+7	7.76E+7

Tableau A.1: Génération de bitmaps avec les algorithmes de compression

	slen	1			32			64		
Type		Min.	Max.	Moy.	Min.	Max.	Moy.	Min.	Max.	Moy.
Brut	Taux	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	bps	3.00E+10	4.50E+11	7.28E+10	4.50E+10	7.50E+10	5.79E+10	2.65E+10	6.43E+10	5.17E+10
Concise	Taux	0.97	0.97	0.97	1.35	1.35	1.35	2.06	2.07	2.07
	bps	7.37E+9	3.32E+10	1.24E+10	1.45E+10	2.09E+10	1.84E+10	1.04E+10	1.68E+10	1.35E+10
WAH	Taux	0.97	0.97	0.97	1.33	1.34	1.33	2.03	2.04	2.03
	bps	9.48E+9	4.65E+11	1.80E+10	3.75E+10	4.82E+10	4.34E+10	3.16E+10	4.43E+10	3.84E+10
EWAH64	Taux	1.00	1.00	1.00	1.14	1.14	1.14	1.99	2.00	2.00
	bps	2.65E+10	4.50E+11	3.03E+10	1.57E+10	2.07E+10	1.85E+10	1.13E+10	1.41E+10	1.27E+10
EWAH32	Taux	1.00	1.00	1.00	2.00	2.00	2.00	3.99	4.01	4.00
	bps	1.41E+10	3.00E+10	1.67E+10	1.12E+10	1.25E+10	1.19E+10	5.63E+9	7.03E+9	6.45E+9
Roaring	Taux	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	bps	2.81E+10	4.50E+11	6.48E+10	5.00E+10	6.43E+10	5.76E+10	4.50E+10	6.43E+10	5.66E+10

Tableau A.2: Évaluation de $\neg s_0$ avec les algorithmes de compression

	slen	1			32			64		
Type		Min.	Max.	Moy.	Min.	Max.	Moy.	Min.	Max.	Moy.
Brut	Taux	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	bps	2.81E+10	4.50E+11	7.68E+10	6.43E+10	9.00E+10	7.48E+10	5.00E+10	9.00E+10	7.20E+10
Concise	Taux	0.97	0.97	0.97	1.79	1.80	1.80	2.74	2.77	2.76
	bps	5.96E+9	1.50E+10	8.93E+9	3.84E+9	4.77E+9	4.23E+9	2.72E+9	3.35E+9	2.98E+9
WAH	Taux	0.97	0.97	0.97	1.77	1.78	1.78	2.70	2.72	2.71
	bps	9.29E+9	2.90E+10	1.34E+10	3.92E+9	5.92E+9	5.54E+9	3.88E+9	4.61E+9	4.34E+9
EWAH64	Taux	1.00	1.00	1.00	1.47	1.47	1.47	2.66	2.68	2.67
	bps	1.41E+10	3.00E+10	1.83E+10	4.92E+9	6.67E+9	6.02E+9	3.81E+9	4.79E+9	4.33E+9
EWAH32	Taux	1.00	1.00	1.00	2.66	2.67	2.67	5.31	5.35	5.33
	bps	9.18E+9	1.55E+10	1.29E+10	3.04E+9	3.57E+9	3.26E+9	1.94E+9	2.34E+9	2.21E+9
Roaring	Taux	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	bps	1.41E+10	3.00E+10	2.37E+10	2.37E+10	2.81E+10	2.54E+10	2.25E+10	2.81E+10	2.52E+10

Tableau A.3: Évaluation de $s_0 \wedge s_1$ avec les algorithmes de compression

	slen	1			32			64		
Type		Min.	Max.	Moy.	Min.	Max.	Moy.	Min.	Max.	Moy.
Brut	Taux	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	bps	2.81E+10	4.50E+11	7.76E+10	5.00E+10	9.00E+10	7.13E+10	5.00E+10	9.00E+10	6.63E+10
Concise	Taux	0.97	0.97	0.97	1.79	1.80	1.80	2.74	2.77	2.76
	bps	7.74E+9	1.50E+10	1.25E+10	3.93E+9	6.42E+9	5.04E+9	2.59E+9	4.84E+9	3.52E+9
WAH	Taux	0.97	0.97	0.97	1.77	1.78	1.78	2.70	2.72	2.71
	bps	1.26E+10	3.10E+10	1.87E+10	6.14E+9	1.05E+10	9.07E+9	6.92E+9	9.22E+9	8.25E+9
EWAH64	Taux	1.00	1.00	1.00	1.47	1.47	1.47	2.65	2.68	2.67
	bps	1.41E+10	3.00E+10	2.01E+10	5.55E+9	7.43E+9	6.66E+9	4.41E+9	5.49E+9	4.88E+9
EWAH32	Taux	1.00	1.00	1.00	2.66	2.67	2.67	5.31	5.35	5.33
	bps	9.57E+9	1.50E+10	1.30E+10	3.46E+9	4.50E+9	3.86E+9	2.30E+9	2.96E+9	2.54E+9
Roaring	Taux	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	bps	2.81E+10	4.50E+11	3.94E+10	3.22E+10	3.75E+10	3.52E+10	3.22E+10	3.75E+10	3.47E+10

Tableau A.4: Évaluation de $s_0 \vee s_1$ avec les algorithmes de compression

	slen	1			32			64		
Type		Min.	Max.	Moy.	Min.	Max.	Moy.	Min.	Max.	Moy.
Brut	Taux	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	bps	2.81E+10	4.50E+11	9.66E+10	5.00E+10	1.12E+11	7.39E+10	2.81E+10	9.00E+10	5.78E+10
Concise	Taux	0.97	0.97	0.97	1.79	1.80	1.80	2.74	2.77	2.75
	bps	7.37E+9	2.90E+10	1.17E+10	6.30E+9	8.35E+9	7.22E+9	4.84E+9	6.40E+9	5.63E+9
WAH	Taux	0.97	0.97	0.97	1.77	1.78	1.78	2.70	2.72	2.71
	bps	9.88E+9	1.94E+10	1.36E+10	6.75E+9	9.64E+9	8.07E+9	6.15E+9	7.63E+9	6.92E+9
EWAH64	Taux	1.00	1.00	1.00	1.47	1.48	1.47	2.65	2.68	2.67
	bps	1.29E+10	3.00E+10	1.50E+10	5.18E+9	1.16E+10	8.60E+9	4.33E+9	7.76E+9	6.58E+9
EWAH32	Taux	1.00	1.00	1.00	2.66	2.68	2.67	5.30	5.36	5.33
	bps	1.41E+10	3.00E+10	1.87E+10	3.26E+9	4.41E+9	3.76E+9	2.45E+9	4.17E+9	3.50E+9
Roaring	Taux	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	bps	2.81E+10	4.50E+11	4.69E+10	3.75E+10	6.43E+10	5.05E+10	3.75E+10	6.43E+10	4.95E+10

Tableau A.5: Évaluation de $s_0 \vee s_1$ avec les algorithmes de compression

	slen	1			32			64		
Type		Min.	Max.	Moy.	Min.	Max.	Moy.	Min.	Max.	Moy.
Brut	Taux	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	bps	2.81E+10	4.50E+11	5.04E+10	4.50E+10	5.62E+10	5.06E+10	4.09E+10	5.62E+10	4.95E+10
Concise	Taux	0.97	0.97	0.97	1.35	1.35	1.35	2.06	2.07	2.07
	bps	1.50E+10	3.10E+10	1.80E+10	8.14E+9	1.04E+10	9.44E+9	4.84E+9	6.80E+9	6.22E+9
WAH	Taux	0.97	0.97	0.97	1.33	1.34	1.33	2.03	2.04	2.03
	bps	1.45E+10	3.10E+10	1.86E+10	7.34E+9	1.16E+10	1.04E+10	5.40E+9	7.91E+9	7.21E+9
EWAH64	Taux	1.00	1.00	1.00	1.07	1.07	1.07	1.33	1.34	1.33
	bps	1.41E+10	3.00E+10	2.23E+10	1.01E+10	1.41E+10	1.28E+10	5.11E+9	6.82E+9	6.09E+9
EWAH32	Taux	1.00	1.00	1.00	1.33	1.34	1.33	1.99	2.00	2.00
	bps	1.25E+10	3.00E+10	1.51E+10	4.17E+9	5.36E+9	4.73E+9	2.34E+9	3.13E+9	2.77E+9
Roaring	Taux	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	bps	9.72E+8	1.15E+9	1.07E+9	9.60E+8	1.14E+9	1.09E+9	9.38E+8	1.29E+9	1.05E+9

Tableau A.6: Évaluation de X s_0 avec les algorithmes de compression

	slen	1			32			64		
Type		Min.	Max.	Moy.	Min.	Max.	Moy.	Min.	Max.	Moy.
Brut	Taux	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	bps	7.14E+9	9.78E+9	8.77E+9	3.38E+9	9.38E+9	5.19E+9	9.38E+9	1.25E+10	1.12E+10
Concise	Taux	7.81E+5	1.56E+6	1.56E+6	3.91E+5	1.56E+6	9.89E+5	3.91E+5	1.56E+6	9.59E+5
	bps	4.65E+11	4.65E+11	4.65E+11	3.34E+11	3.34E+11	3.34E+11	2.18E+11	2.18E+11	2.18E+11
WAH	Taux	7.81E+5	1.56E+6	1.56E+6	3.91E+5	1.56E+6	9.77E+5	3.91E+5	1.56E+6	9.30E+5
	bps	4.65E+11	4.65E+11	4.65E+11	3.37E+11	3.37E+11	3.37E+11	2.21E+11	2.21E+11	2.21E+11
EWAH64	Taux	3.91E+5	7.81E+5	5.58E+5	2.60E+5	7.81E+5	5.11E+5	3.91E+5	7.81E+5	5.50E+5
	bps	4.50E+11	4.50E+11	4.50E+11	1.97E+10	3.03E+10	2.70E+10	1.32E+10	1.73E+10	1.58E+10
EWAH32	Taux	6.25E+4	6.51E+4	6.40E+4	6.25E+4	6.51E+4	6.39E+4	6.25E+4	6.51E+4	6.40E+4
	bps	4.50E+11	4.50E+11	4.50E+11	1.41E+10	1.87E+10	1.67E+10	7.03E+9	1.02E+10	8.78E+9
Roaring	Taux	2.60E+5	7.81E+5	5.23E+5	1.57E+4	7.81E+5	9.59E+4	6.87E+3	7.81E+5	5.18E+4
	bps	2.25E+11	4.50E+11	4.50E+11	2.25E+11	4.50E+11	3.91E+11	2.25E+11	4.50E+11	3.75E+11

Tableau A.7: Évaluation de G s_0 avec les algorithmes de compression

	slen	1			32			64		
Type		Min.	Max.	Moy.	Min.	Max.	Moy.	Min.	Max.	Moy.
Brut	Taux	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	bps	2.59E+9	3.21E+9	2.99E+9	2.59E+9	3.15E+9	2.95E+9	2.56E+9	3.17E+9	2.88E+9
Concise	Taux	7.81E+5	7.81E+5	7.81E+5	7.81E+5	7.81E+5	7.81E+5	7.81E+5	7.81E+5	7.81E+5
	bps	4.65E+11	4.65E+11	4.65E+11	3.34E+11	3.34E+11	3.34E+11	2.18E+11	2.18E+11	2.18E+11
WAH	Taux	7.81E+5	7.81E+5	7.81E+5	7.81E+5	7.81E+5	7.81E+5	7.81E+5	7.81E+5	7.81E+5
	bps	4.65E+11	4.65E+11	4.65E+11	3.37E+11	3.37E+11	3.37E+11	2.21E+11	2.21E+11	2.21E+11
EWAH64	Taux	3.91E+5	7.81E+5	4.88E+5	2.60E+5	7.81E+5	4.76E+5	3.91E+5	7.81E+5	4.94E+5
	bps	4.50E+11	4.50E+11	4.50E+11	1.97E+10	3.03E+10	2.65E+10	1.41E+10	1.73E+10	1.62E+10
EWAH32	Taux	6.25E+4	6.51E+4	6.35E+4	6.25E+4	6.51E+4	6.37E+4	6.25E+4	6.51E+4	6.36E+4
	bps	4.50E+11	4.50E+11	4.50E+11	1.50E+10	1.87E+10	1.70E+10	8.04E+9	1.02E+10	8.90E+9
Roaring	Taux	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	bps	7.11E+8	1.03E+9	9.36E+8	8.27E+8	1.03E+9	9.38E+8	7.48E+8	9.87E+8	8.91E+8

Tableau A.8: Évaluation de F s_0 avec les algorithmes de compression

	slen	1			32			64		
Type		Min.	Max.	Moy.	Min.	Max.	Moy.	Min.	Max.	Moy.
Brut	Taux	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	bps	2.23E+8	2.88E+8	2.58E+8	9.72E+8	1.27E+9	1.13E+9	1.12E+9	1.49E+9	1.34E+9
Concise	Taux	0.97	0.97	0.97	1.89	1.90	1.89	3.12	3.15	3.14
	bps	1.22E+8	1.52E+8	1.36E+8	5.52E+8	7.89E+8	6.43E+8	4.81E+8	9.31E+8	6.48E+8
WAH	Taux	0.97	0.97	0.97	1.86	1.87	1.86	3.05	3.08	3.07
	bps	1.47E+8	1.86E+8	1.68E+8	5.77E+8	7.83E+8	7.19E+8	7.74E+8	9.63E+8	8.61E+8
EWAH64	Taux	1.00	1.00	1.00	1.47	1.48	1.48	2.99	3.01	3.00
	bps	8.69E+7	1.04E+8	9.74E+7	4.82E+8	6.21E+8	5.51E+8	3.54E+8	4.88E+8	4.31E+8
EWAH32	Taux	1.00	1.00	1.00	2.99	3.01	3.00	5.97	6.03	6.00
	bps	8.77E+7	1.10E+8	1.00E+8	3.04E+8	4.48E+8	4.21E+8	3.07E+8	4.25E+8	3.90E+8
Roaring	Taux	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	bps	6.74E+7	8.35E+7	7.62E+7	1.71E+8	2.11E+8	1.93E+8	1.78E+8	2.27E+8	2.02E+8

Tableau A.9: Évaluation de s_0Us_1 avec les algorithmes de compression

	slen	1			32			64		
Type		Min.	Max.	Moy.	Min.	Max.	Moy.	Min.	Max.	Moy.
Brut	Taux	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	bps	2.30E+8	2.94E+8	2.67E+8	1.00E+9	1.32E+9	1.15E+9	1.15E+9	1.51E+9	1.33E+9
Concise	Taux	0.97	0.97	0.97	1.89	1.90	1.89	3.12	3.15	3.14
	bps	1.24E+8	1.56E+8	1.40E+8	5.71E+8	6.98E+8	6.47E+8	5.42E+8	7.56E+8	6.09E+8
WAH	Taux	0.97	0.97	0.97	1.86	1.87	1.86	3.05	3.08	3.07
	bps	1.44E+8	1.85E+8	1.64E+8	4.05E+8	8.19E+8	7.26E+8	8.20E+8	9.80E+8	8.96E+8
EWAH64	Taux	1.00	1.00	1.00	1.47	1.48	1.48	2.99	3.01	3.00
	bps	8.64E+7	1.06E+8	9.77E+7	4.72E+8	6.36E+8	5.63E+8	3.46E+8	5.06E+8	4.45E+8
EWAH32	Taux	1.00	1.00	1.00	2.99	3.01	3.00	5.97	6.03	6.00
	bps	9.30E+7	1.09E+8	1.01E+8	3.74E+8	4.63E+8	4.32E+8	3.38E+8	4.48E+8	4.03E+8
Roaring	Taux	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	bps	6.29E+7	9.32E+7	7.93E+7	1.65E+8	2.13E+8	1.91E+8	1.81E+8	2.25E+8	2.02E+8

Tableau A.10: Évaluation de $s_0\mathbf{W}s_1$ avec les algorithmes de compression

	slen	1			32			64		
Type		Min.	Max.	Moy.	Min.	Max.	Moy.	Min.	Max.	Moy.
Brut	Taux	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	bps	2.06E+8	2.59E+8	2.29E+8	1.19E+9	1.63E+9	1.41E+9	1.49E+9	1.92E+9	1.71E+9
Concise	Taux	0.97	0.97	0.97	1.92	1.93	1.92	3.18	3.21	3.20
	bps	1.34E+8	1.62E+8	1.48E+8	7.26E+8	1.21E+9	8.33E+8	6.52E+8	1.39E+9	7.60E+8
WAH	Taux	0.97	0.97	0.97	1.87	1.88	1.88	3.08	3.11	3.09
	bps	1.54E+8	2.10E+8	1.88E+8	5.35E+8	1.03E+9	8.74E+8	1.14E+9	1.33E+9	1.25E+9
EWAH64	Taux	1.00	1.00	1.00	1.47	1.48	1.48	2.99	3.01	3.00
	bps	8.99E+7	1.09E+8	1.01E+8	4.50E+8	5.80E+8	5.27E+8	3.67E+8	4.93E+8	4.35E+8
EWAH32	Taux	1.00	1.00	1.00	2.99	3.01	3.00	5.97	6.02	6.00
	bps	8.70E+7	1.11E+8	1.02E+8	3.80E+8	4.72E+8	4.33E+8	3.38E+8	4.59E+8	4.10E+8
Roaring	Taux	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	bps	6.65E+7	9.90E+7	8.38E+7	1.86E+8	2.43E+8	2.17E+8	2.00E+8	2.57E+8	2.31E+8

Tableau A.11: Évaluation de $s_0\mathbf{R}s_1$ avec les algorithmes de compression

	slen	1			32			64		
Type		Min.	Max.	Moy.	Min.	Max.	Moy.	Min.	Max.	Moy.
Brut	Taux	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	bps	3.17E+7	4.30E+7	3.92E+7	1.42E+8	1.91E+8	1.68E+8	1.60E+8	1.98E+8	1.79E+8
Concise	Taux	7.81E+5	7.81E+5	7.81E+5	7.81E+5	7.81E+5	7.81E+5	7.81E+5	7.81E+5	7.81E+5
	bps	2.52E+7	3.03E+7	2.78E+7	1.30E+8	1.81E+8	1.53E+8	1.38E+8	2.20E+8	1.65E+8
WAH	Taux	7.81E+5	7.81E+5	7.81E+5	7.81E+5	7.81E+5	7.81E+5	7.81E+5	7.81E+5	7.81E+5
	bps	2.91E+7	3.60E+7	3.30E+7	1.44E+8	2.20E+8	1.87E+8	1.78E+8	2.41E+8	1.97E+8
EWAH64	Taux	7.81E+5	7.81E+5	7.81E+5	3.91E+5	7.81E+5	6.25E+5	3.91E+5	7.81E+5	5.11E+5
	bps	1.71E+7	2.08E+7	1.92E+7	8.92E+7	1.17E+8	1.04E+8	7.45E+7	9.67E+7	8.40E+7
EWAH32	Taux	6.51E+4	6.51E+4	6.51E+4	6.25E+4	6.51E+4	6.36E+4	6.25E+4	6.51E+4	6.37E+4
	bps	1.71E+7	2.17E+7	2.00E+7	7.20E+7	8.44E+7	7.97E+7	6.47E+7	8.06E+7	7.30E+7
Roaring	Taux	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	bps	1.18E+7	1.58E+7	1.38E+7	2.50E+7	3.14E+7	2.82E+7	2.53E+7	3.33E+7	2.94E+7

Tableau A.12: Évaluation de la formule F1 avec les algorithmes de compression

	slen	1			32			64		
Type		Min.	Max.	Moy.	Min.	Max.	Moy.	Min.	Max.	Moy.
Brut	Taux	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	bps	4.22E+7	5.82E+7	5.03E+7	1.58E+8	2.35E+8	1.92E+8	1.56E+8	2.08E+8	1.77E+8
Concise	Taux	7.81E+5	1.56E+6	1.53E+6	7.81E+5	1.56E+6	1.56E+6	7.81E+5	1.56E+6	1.56E+6
	bps	3.57E+7	4.87E+7	4.16E+7	2.00E+8	2.87E+8	2.53E+8	2.07E+8	3.48E+8	2.92E+8
WAH	Taux	7.81E+5	1.56E+6	1.53E+6	7.81E+5	1.56E+6	1.56E+6	7.81E+5	1.56E+6	1.56E+6
	bps	3.94E+7	5.12E+7	4.51E+7	2.00E+8	2.87E+8	2.44E+8	2.54E+8	3.41E+8	2.93E+8
EWAH64	Taux	7.81E+5	7.81E+5	7.81E+5	7.81E+5	7.81E+5	7.81E+5	7.81E+5	7.81E+5	7.81E+5
	bps	2.22E+7	2.99E+7	2.61E+7	1.16E+8	1.77E+8	1.44E+8	9.22E+7	1.49E+8	1.14E+8
EWAH32	Taux	6.51E+4	6.51E+4	6.51E+4	6.51E+4	6.51E+4	6.51E+4	6.51E+4	6.51E+4	6.51E+4
	bps	2.37E+7	3.18E+7	2.74E+7	9.13E+7	1.27E+8	1.09E+8	7.95E+7	1.18E+8	1.01E+8
Roaring	Taux	1.00	7.81E+5	1.96	1.00	7.81E+5	2.22	1.00	7.81E+5	2.13
	bps	1.39E+7	1.92E+7	1.60E+7	2.61E+7	3.51E+7	3.03E+7	2.59E+7	3.89E+7	3.12E+7

Tableau A.13: Évaluation de la formule F2 avec les algorithmes de compression

	slen	1			32			64		
Type		Min.	Max.	Moy.	Min.	Max.	Moy.	Min.	Max.	Moy.
Brut	Taux	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	bps	1.07E+9	1.60E+9	1.38E+9	1.07E+9	1.81E+9	1.41E+9	9.26E+8	1.19E+9	1.09E+9
Concise	Taux	7.81E+5	1.56E+6	8.01E+5	7.81E+5	1.56E+6	7.85E+5	7.81E+5	1.56E+6	7.97E+5
	bps	1.16E+9	2.11E+9	1.66E+9	9.54E+8	1.29E+9	1.15E+9	7.64E+8	1.11E+9	9.36E+8
WAH	Taux	7.81E+5	1.56E+6	8.01E+5	7.81E+5	1.56E+6	7.85E+5	7.81E+5	1.56E+6	7.97E+5
	bps	1.43E+9	2.28E+9	1.85E+9	1.01E+9	1.35E+9	1.25E+9	9.42E+8	1.20E+9	1.07E+9
EWAH64	Taux	3.91E+5	3.91E+5	3.91E+5	2.60E+5	3.91E+5	3.30E+5	2.60E+5	3.91E+5	3.24E+5
	bps	4.05E+9	5.77E+9	5.19E+9	1.44E+9	1.87E+9	1.69E+9	1.08E+9	1.46E+9	1.26E+9
EWAH32	Taux	6.01E+4	6.25E+4	6.25E+4	6.01E+4	6.25E+4	6.15E+4	6.01E+4	6.25E+4	6.15E+4
	bps	2.56E+9	3.60E+9	3.06E+9	8.04E+8	9.74E+8	9.00E+8	5.38E+8	7.92E+8	6.56E+8
Roaring	Taux	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	bps	2.34E+8	2.97E+8	2.72E+8	2.48E+8	2.97E+8	2.73E+8	2.19E+8	2.89E+8	2.58E+8

Tableau A.14: Évaluation de la formule F3 avec les algorithmes de compression

	slen	1			32			64		
Type		Min.	Max.	Moy.	Min.	Max.	Moy.	Min.	Max.	Moy.
Brut	Taux	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	bps	6.39E+8	1.07E+9	8.03E+8	6.07E+8	1.19E+9	8.03E+8	5.21E+8	7.28E+8	6.14E+8
Concise	Taux	5.21E+5	1.56E+6	7.66E+5	2.60E+5	1.56E+6	5.92E+5	2.60E+5	1.56E+6	5.70E+5
	bps	1.42E+9	2.98E+9	1.99E+9	1.12E+9	1.60E+9	1.35E+9	9.23E+8	1.31E+9	1.10E+9
WAH	Taux	5.21E+5	1.56E+6	7.66E+5	2.60E+5	1.56E+6	5.81E+5	2.60E+5	1.56E+6	5.62E+5
	bps	1.62E+9	2.96E+9	2.16E+9	1.14E+9	1.67E+9	1.44E+9	1.02E+9	1.46E+9	1.22E+9
EWAH64	Taux	3.91E+5	7.81E+5	3.97E+5	1.56E+5	7.81E+5	3.08E+5	1.30E+5	7.81E+5	2.80E+5
	bps	5.06E+9	7.26E+9	6.17E+9	1.52E+9	2.04E+9	1.80E+9	1.24E+9	1.69E+9	1.48E+9
EWAH32	Taux	6.25E+4	6.51E+4	6.26E+4	5.39E+4	6.51E+4	6.03E+4	5.39E+4	6.51E+4	6.01E+4
	bps	3.08E+9	4.79E+9	3.89E+9	8.04E+8	1.02E+9	9.20E+8	6.94E+8	9.53E+8	8.21E+8
Roaring	Taux	1.00	7.81E+5	1.59	1.00	7.81E+5	1.59	1.00	7.81E+5	1.54
	bps	1.45E+8	3.66E+8	2.09E+8	1.55E+8	3.60E+8	2.10E+8	1.38E+8	3.58E+8	1.97E+8

Tableau A.15: Évaluation de la formule F4 avec les algorithmes de compression

	slen	1			32			64		
Type		Min.	Max.	Moy.	Min.	Max.	Moy.	Min.	Max.	Moy.
Brut	Taux	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	bps	4.29E+7	5.27E+7	4.80E+7	1.48E+8	1.91E+8	1.71E+8	1.49E+8	1.98E+8	1.74E+8
Concise	Taux	0.98	1.03	0.99	5.01	5.10	5.06	8.04	8.22	8.13
	bps	2.59E+7	3.47E+7	3.02E+7	1.39E+8	2.01E+8	1.81E+8	1.48E+8	2.42E+8	1.96E+8
WAH	Taux	0.98	0.99	0.98	5.16	5.25	5.20	8.52	8.71	8.62
	bps	2.76E+7	3.56E+7	3.18E+7	1.50E+8	2.12E+8	1.79E+8	1.94E+8	2.54E+8	2.17E+8
EWAH64	Taux	1.00	1.00	1.00	3.59	3.63	3.61	9.17	9.33	9.24
	bps	1.82E+7	2.21E+7	2.03E+7	1.05E+8	1.42E+8	1.24E+8	9.43E+7	1.28E+8	1.07E+8
EWAH32	Taux	1.01	1.01	1.01	9.19	9.30	9.24	1.83E+1	1.87E+1	1.85E+1
	bps	1.87E+7	2.29E+7	2.10E+7	8.76E+7	1.06E+8	9.76E+7	8.04E+7	1.03E+8	9.24E+7
Roaring	Taux	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	bps	1.08E+7	1.46E+7	1.25E+7	2.01E+7	2.83E+7	2.42E+7	2.09E+7	3.11E+7	2.53E+7

Tableau A.16: Évaluation de la formule F5 avec les algorithmes de compression

	slen	1			32			64		
Type		Min.	Max.	Moy.	Min.	Max.	Moy.	Min.	Max.	Moy.
Brut	Taux	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	bps	4.81E+7	6.18E+7	5.67E+7	1.87E+8	2.36E+8	2.12E+8	1.82E+8	2.37E+8	2.14E+8
Concise	Taux	1.68	1.78	1.73	4.55	4.88	4.70	6.68	7.48	6.98
	bps	3.36E+7	4.18E+7	3.75E+7	1.89E+8	2.50E+8	2.33E+8	1.97E+8	2.94E+8	2.62E+8
WAH	Taux	1.49	2.10	1.74	4.08	4.47	4.26	6.30	7.37	6.90
	bps	3.24E+7	4.34E+7	3.86E+7	1.94E+8	2.47E+8	2.17E+8	2.36E+8	3.06E+8	2.66E+8
EWAH64	Taux	7.81E+5	7.81E+5	7.81E+5	7.81E+5	7.81E+5	7.81E+5	7.81E+5	7.81E+5	7.81E+5
	bps	2.07E+7	2.44E+7	2.26E+7	1.08E+8	1.45E+8	1.28E+8	9.67E+7	1.28E+8	1.10E+8
EWAH32	Taux	6.51E+4	6.51E+4	6.51E+4	6.51E+4	6.51E+4	6.51E+4	6.51E+4	6.51E+4	6.51E+4
	bps	2.08E+7	2.53E+7	2.33E+7	9.19E+7	1.10E+8	1.03E+8	8.25E+7	1.08E+8	9.73E+7
Roaring	Taux	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	bps	1.28E+7	1.71E+7	1.48E+7	2.47E+7	3.45E+7	2.99E+7	2.58E+7	3.86E+7	3.12E+7

Tableau A.17: Évaluation de la formule F6 avec les algorithmes de compression

	slen	1			32			64		
Type		Min.	Max.	Moy.	Min.	Max.	Moy.	Min.	Max.	Moy.
Brut	Taux	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	bps	3.90E+8	1.04E+9	6.59E+8	3.26E+8	1.15E+9	5.65E+8	3.05E+8	5.06E+8	3.93E+8
Concise	Taux	1.56E+6	1.56E+6	1.56E+6	1.56E+6	1.56E+6	1.56E+6	1.56E+6	1.56E+6	1.56E+6
	bps	6.83E+9	4.65E+11	1.54E+10	4.23E+9	3.34E+11	1.07E+10	2.69E+9	2.18E+11	6.48E+9
WAH	Taux	1.56E+6	1.56E+6	1.56E+6	1.56E+6	1.56E+6	1.56E+6	1.56E+6	1.56E+6	1.56E+6
	bps	7.04E+9	4.65E+11	1.59E+10	4.44E+9	3.37E+11	1.16E+10	2.77E+9	2.21E+11	6.96E+9
EWAH64	Taux	7.81E+5	7.81E+5	7.81E+5	7.81E+5	7.81E+5	7.81E+5	7.81E+5	7.81E+5	7.81E+5
	bps	2.81E+10	4.50E+11	4.50E+11	2.19E+10	2.81E+10	2.55E+10	3.21E+10	5.63E+10	4.36E+10
EWAH32	Taux	6.51E+4	6.51E+4	6.51E+4	6.51E+4	6.51E+4	6.51E+4	6.51E+4	6.51E+4	6.51E+4
	bps	4.50E+11	4.50E+11	4.50E+11	2.05E+10	2.81E+10	2.55E+10	1.88E+10	2.81E+10	2.42E+10
Roaring	Taux	7.81E+5	7.81E+5	7.81E+5	7.81E+5	7.81E+5	7.81E+5	7.81E+5	7.81E+5	7.81E+5
	bps	7.77E+7	9.60E+8	1.63E+8	7.68E+7	9.89E+8	1.80E+8	7.26E+7	9.81E+8	1.63E+8

Tableau A.18: Évaluation de la formule F7 avec les algorithmes de compression

	slen	1			32			64		
Type		Min.	Max.	Moy.	Min.	Max.	Moy.	Min.	Max.	Moy.
Brut	Taux	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	bps	5.25E+8	1.20E+9	9.52E+8	4.59E+8	1.29E+9	7.51E+8	4.10E+8	6.11E+8	5.19E+8
Concise	Taux	1.56E+6	1.56E+6	1.56E+6	1.56E+6	1.56E+6	1.56E+6	1.56E+6	1.56E+6	1.56E+6
	bps	3.57E+9	2.90E+10	1.06E+10	2.38E+9	1.08E+10	6.39E+9	1.60E+9	7.51E+9	4.11E+9
WAH	Taux	1.56E+6	1.56E+6	1.56E+6	1.56E+6	1.56E+6	1.56E+6	1.56E+6	1.56E+6	1.56E+6
	bps	4.18E+9	3.10E+10	1.10E+10	2.46E+9	1.12E+10	6.72E+9	1.83E+9	7.91E+9	4.50E+9
EWAH64	Taux	3.91E+5	3.91E+5	3.91E+5	3.91E+5	3.91E+5	3.91E+5	3.91E+5	3.91E+5	3.91E+5
	bps	1.45E+10	4.50E+11	3.08E+10	6.35E+9	1.46E+10	1.13E+10	2.50E+9	1.41E+10	7.92E+9
EWAH32	Taux	6.25E+4	6.25E+4	6.25E+4	6.25E+4	6.25E+4	6.25E+4	6.25E+4	6.25E+4	6.25E+4
	bps	1.02E+10	3.00E+10	2.13E+10	1.92E+9	9.37E+9	5.89E+9	9.53E+8	7.50E+9	3.61E+9
Roaring	Taux	7.81E+5	7.81E+5	7.81E+5	7.81E+5	7.81E+5	7.81E+5	7.81E+5	7.81E+5	7.81E+5
	bps	8.20E+7	3.40E+8	2.09E+8	7.87E+7	3.41E+8	2.16E+8	7.63E+7	3.33E+8	1.92E+8

Tableau A.19: Évaluation de la formule F8 avec les algorithmes de compression

	slen	1			32			64		
Type		Min.	Max.	Moy.	Min.	Max.	Moy.	Min.	Max.	Moy.
Brut	Taux	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	bps	8.04E+9	1.45E+10	9.96E+9	3.81E+9	1.18E+10	7.89E+9	5.42E+9	9.38E+9	7.04E+9
Concise	Taux	0.97	0.97	0.97	1.79	1.80	1.80	2.74	2.77	2.75
	bps	1.22E+9	2.04E+9	1.51E+9	8.35E+8	1.04E+9	9.72E+8	7.31E+8	9.15E+8	8.31E+8
WAH	Taux	0.97	0.97	0.97	1.77	1.78	1.78	2.70	2.72	2.71
	bps	1.25E+9	2.01E+9	1.53E+9	8.17E+8	1.06E+9	9.92E+8	7.85E+8	9.88E+8	9.00E+8
EWAH64	Taux	1.00	1.00	1.00	1.47	1.47	1.47	2.66	2.68	2.67
	bps	2.81E+9	4.79E+9	3.78E+9	1.09E+9	1.40E+9	1.28E+9	9.26E+8	1.17E+9	1.07E+9
EWAH32	Taux	1.00	1.00	1.00	2.66	2.67	2.67	5.31	5.35	5.33
	bps	2.15E+9	2.88E+9	2.62E+9	6.11E+8	7.28E+8	6.91E+8	5.74E+8	7.26E+8	6.59E+8
Roaring	Taux	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	bps	7.76E+9	1.45E+10	9.97E+9	8.18E+9	1.02E+10	9.64E+9	7.90E+9	1.05E+10	9.31E+9

Tableau A.20: Évaluation de la formule F9 avec les algorithmes de compression

	slen	1			32			64		
Type		Min.	Max.	Moy.	Min.	Max.	Moy.	Min.	Max.	Moy.
Brut	Taux	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	bps	6.62E+9	9.78E+9	8.72E+9	3.49E+9	1.05E+10	6.45E+9	5.17E+9	8.18E+9	6.63E+9
Concise	Taux	7.81E+5	7.81E+5	7.81E+5	7.81E+5	7.81E+5	7.81E+5	7.81E+5	7.81E+5	7.81E+5
	bps	1.01E+9	1.65E+9	1.26E+9	7.97E+8	1.00E+9	9.19E+8	6.78E+8	8.75E+8	7.88E+8
WAH	Taux	7.81E+5	7.81E+5	7.81E+5	7.81E+5	7.81E+5	7.81E+5	7.81E+5	7.81E+5	7.81E+5
	bps	1.09E+9	1.65E+9	1.31E+9	7.58E+8	1.02E+9	9.38E+8	7.16E+8	9.58E+8	8.38E+8
EWAH64	Taux	7.81E+5	7.81E+5	7.81E+5	7.81E+5	7.81E+5	7.81E+5	7.81E+5	7.81E+5	7.81E+5
	bps	2.88E+9	4.13E+9	3.58E+9	1.00E+9	1.44E+9	1.32E+9	9.78E+8	1.19E+9	1.11E+9
EWAH32	Taux	6.51E+4	6.51E+4	6.51E+4	6.51E+4	6.51E+4	6.51E+4	6.51E+4	6.51E+4	6.51E+4
	bps	2.04E+9	2.66E+9	2.41E+9	6.32E+8	7.35E+8	6.98E+8	5.92E+8	7.35E+8	6.69E+8
Roaring	Taux	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	bps	8.83E+9	1.45E+10	1.12E+10	9.79E+9	1.25E+10	1.10E+10	9.58E+9	1.22E+10	1.08E+10

Tableau A.21: Évaluation de la formule F10 avec les algorithmes de compression

	slen	1			32			64		
Type		Min.	Max.	Moy.	Min.	Max.	Moy.	Min.	Max.	Moy.
Brut	Taux	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	bps	3.00E+7	3.65E+7	3.35E+7	9.97E+7	1.28E+8	1.15E+8	9.94E+7	1.28E+8	1.16E+8
Concise	Taux	5.21E+5	1.56E+6	9.08E+5	3.91E+5	1.56E+6	6.82E+5	3.91E+5	1.56E+6	5.92E+5
	bps	2.19E+7	2.64E+7	2.41E+7	1.24E+8	1.68E+8	1.51E+8	1.35E+8	2.05E+8	1.74E+8
WAH	Taux	5.21E+5	1.56E+6	9.08E+5	3.91E+5	1.56E+6	6.73E+5	3.91E+5	1.56E+6	5.92E+5
	bps	2.17E+7	2.72E+7	2.43E+7	1.26E+8	1.62E+8	1.43E+8	1.58E+8	2.04E+8	1.75E+8
EWAH64	Taux	3.91E+5	7.81E+5	4.44E+5	1.95E+5	7.81E+5	4.27E+5	1.95E+5	7.81E+5	5.24E+5
	bps	1.22E+7	1.46E+7	1.35E+7	5.86E+7	7.72E+7	6.95E+7	4.83E+7	6.42E+7	5.54E+7
EWAH32	Taux	6.25E+4	6.51E+4	6.31E+4	5.79E+4	6.51E+4	6.38E+4	5.79E+4	6.51E+4	6.37E+4
	bps	1.21E+7	1.51E+7	1.39E+7	4.63E+7	5.69E+7	5.31E+7	4.21E+7	5.27E+7	4.80E+7
Roaring	Taux	8.45E+4	7.81E+5	3.10E+5	9.53E+3	7.81E+5	3.47E+4	3.46E+3	7.81E+5	2.00E+4
	bps	7.50E+6	1.01E+7	8.63E+6	1.48E+7	1.98E+7	1.74E+7	1.51E+7	2.20E+7	1.81E+7

Tableau A.22: Évaluation de la formule F11 avec les algorithmes de compression

	slen	1			32			64		
Type		Min.	Max.	Moy.	Min.	Max.	Moy.	Min.	Max.	Moy.
Brut	Taux	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	bps	3.80E+7	4.77E+7	4.32E+7	9.41E+7	1.31E+8	1.09E+8	8.88E+7	1.23E+8	1.04E+8
Concise	Taux	7.81E+5	1.56E+6	1.15E+6	3.91E+5	1.56E+6	9.95E+5	3.91E+5	1.56E+6	1.04E+6
	bps	2.95E+7	3.63E+7	3.31E+7	1.73E+8	2.46E+8	2.19E+8	2.04E+8	3.10E+8	2.72E+8
WAH	Taux	7.81E+5	1.56E+6	1.15E+6	3.91E+5	1.56E+6	9.95E+5	3.91E+5	1.56E+6	1.04E+6
	bps	2.90E+7	3.70E+7	3.34E+7	1.80E+8	2.44E+8	2.10E+8	2.30E+8	3.17E+8	2.64E+8
EWAH64	Taux	3.91E+5	7.81E+5	7.66E+5	3.91E+5	7.81E+5	6.25E+5	3.91E+5	7.81E+5	5.11E+5
	bps	1.76E+7	2.15E+7	1.96E+7	9.55E+7	1.29E+8	1.13E+8	8.28E+7	1.13E+8	9.47E+7
EWAH32	Taux	6.25E+4	6.51E+4	6.50E+4	6.25E+4	6.51E+4	6.36E+4	6.25E+4	6.51E+4	6.37E+4
	bps	1.80E+7	2.23E+7	2.01E+7	8.10E+7	9.79E+7	9.06E+7	7.58E+7	9.51E+7	8.59E+7
Roaring	Taux	1.00	7.81E+5	1.51	1.00	7.81E+5	1.51	1.00	7.81E+5	1.49
	bps	9.18E+6	1.26E+7	1.04E+7	1.63E+7	2.22E+7	1.86E+7	1.56E+7	2.41E+7	1.91E+7

Tableau A.23: Évaluation de la formule F12 avec les algorithmes de compression

	slen	1			32			64		
Type		Min.	Max.	Moy.	Min.	Max.	Moy.	Min.	Max.	Moy.
Brut	Taux	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	bps	1.65E+7	1.98E+7	1.81E+7	6.18E+7	7.36E+7	6.90E+7	6.35E+7	8.07E+7	7.39E+7
Concise	Taux	0.97	0.97	0.97	3.34	3.38	3.36	5.91	5.99	5.95
	bps	9.88E+6	1.20E+7	1.10E+7	5.49E+7	7.12E+7	6.62E+7	7.55E+7	1.02E+8	9.28E+7
WAH	Taux	0.97	0.97	0.97	3.23	3.26	3.24	5.79	5.88	5.84
	bps	9.88E+6	1.24E+7	1.11E+7	5.65E+7	7.88E+7	6.90E+7	7.94E+7	1.04E+8	8.89E+7
EWAH64	Taux	1.00	1.00	1.00	2.33	2.35	2.34	5.69	5.76	5.72
	bps	5.79E+6	6.95E+6	6.41E+6	3.14E+7	4.21E+7	3.75E+7	2.66E+7	3.56E+7	3.07E+7
EWAH32	Taux	1.00	1.00	1.00	5.70	5.75	5.72	1.14E+1	1.15E+1	1.14E+1
	bps	5.82E+6	7.10E+6	6.54E+6	2.57E+7	3.18E+7	2.92E+7	2.48E+7	3.15E+7	2.77E+7
Roaring	Taux	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	bps	3.85E+6	5.04E+6	4.37E+6	7.86E+6	1.10E+7	9.50E+6	8.31E+6	1.23E+7	9.98E+6

Tableau A.24: Évaluation de la formule F13 avec les algorithmes de compression

	slen	1			32			64		
Type		Min.	Max.	Moy.	Min.	Max.	Moy.	Min.	Max.	Moy.
Brut	Taux	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	bps	1.64E+7	1.97E+7	1.81E+7	6.45E+7	7.70E+7	7.21E+7	6.58E+7	8.51E+7	7.79E+7
Concise	Taux	0.97	0.97	0.97	2.74	2.76	2.75	4.52	4.57	4.54
	bps	9.64E+6	1.22E+7	1.12E+7	5.70E+7	7.14E+7	6.82E+7	7.13E+7	9.21E+7	8.44E+7
WAH	Taux	0.97	0.97	0.97	2.67	2.69	2.68	4.47	4.52	4.50
	bps	9.71E+6	1.25E+7	1.12E+7	6.47E+7	8.22E+7	7.23E+7	8.34E+7	1.08E+8	9.31E+7
EWAH64	Taux	1.00	1.00	1.00	2.08	2.10	2.09	4.43	4.49	4.46
	bps	5.77E+6	6.85E+6	6.33E+6	3.01E+7	3.98E+7	3.56E+7	2.50E+7	3.33E+7	2.91E+7
EWAH32	Taux	1.00	1.00	1.00	4.44	4.48	4.46	8.86	8.97	8.92
	bps	5.79E+6	6.98E+6	6.43E+6	2.50E+7	3.02E+7	2.82E+7	2.37E+7	3.03E+7	2.68E+7
Roaring	Taux	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	bps	3.80E+6	5.04E+6	4.36E+6	7.96E+6	1.12E+7	9.64E+6	8.44E+6	1.25E+7	1.01E+7

Tableau A.25: Évaluation de la formule F14 avec les algorithmes de compression

BIBLIOGRAPHIE

1976. Broadcast teletext specification. Rapport.

2011. United States RBDS standard, NRSC-4B. Rapport.

Adelmann, R., M. Langheinrich, et C. Flörkemeier. 2006. « Toolkit for bar code recognition and resolving on camera phones-jump starting the internet of things », *GI Jahrestagung (2)*, vol. 94, p. 366–373.

Antoshenkov, G. 1995. « Byte-aligned bitmap compression ». In *Data Compression Conference, 1995. DCC'95. Proceedings*, p. 476. IEEE.

Arnon, S. 2015. *Visible light communication*. Cambridge University Press.

Avižienis, A., J.-C. Laprie, B. Randell, et C. Landwehr. 2004. « Basic concepts and taxonomy of dependable and secure computing », *Dependable and Secure Computing, IEEE Transactions on*, vol. 1, no. 1, p. 11–33.

Ayres, J., J. Flannick, J. Gehrke, et T. Yiu. 2002. « Sequential pattern mining using a bitmap representation ». In *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. Coll. « KDD '02 », p. 429–435, New York, NY, USA. ACM.

Barre, B., M. Klein, M. Soucy-Boivin, P.-A. Ollivier, et S. Hallé. 2012. « Mapreduce for parallel trace validation of ltl properties ». In *Runtime Verification*, p. 184–198. Springer.

Barringer, H., A. Goldberg, K. Havelund, et K. Sen. 2004. « Rule-based runtime verification ». In *Verification, Model Checking, and Abstract Interpretation*, p. 44–57. Springer.

Bauer, A., M. Leucker, et C. Schallhart. 2006. *Monitoring of real-time properties*. Coll. « FSTTCS 2006 : Foundations of Software Technology and Theoretical Computer Science », p. 260–272. Springer.

Berkovich, S., B. Bonakdarpour, et S. Fischmeister. 2015. « Runtime verification with minimal intrusion through parallelism », *Formal Methods in System Design*, vol. 46, no. 3, p. 317–348.

Bretz, R. 1984. « Slow-scan television : Its nature and uses », *Educational Technology*, vol. 24, no. 7, p. 35–42.

Broy, M., B. Jonsson, J.-P. Katoen, M. Leucker, et A. Pretschner. 2005. *Model-based testing of reactive systems : advanced lectures*. T. 3472. Springer.

Burdick, D., M. Calimlim, et J. Gehrke. 2001. « Mafia : A maximal frequent itemset algorithm for transactional databases ». In *Data Engineering, 2001. Proceedings. 17th International Conference on*, p. 443–452. IEEE.

Burns, R. W. 2004. *Communications : an international history of the formative years*. T. 32. IET.

Calabrese, F., M. Colonna, P. Lovisolo, D. Parata, et C. Ratti. 2011. « Real-time urban monitoring using cell phones : A case study in rome », *Intelligent Transportation Systems, IEEE Transactions on*, vol. 12, no. 1, p. 141–151.

Casley, D. J. et K. Kumar. 1988. *The collection, analysis and use of monitoring and evaluation data*. The World Bank.

Chambi, S., D. Lemire, O. Kaser, et R. Godin. 2015. « Better bitmap performance with roaring bitmaps », *Software : Practice and Experience*. Available as a preprint.

Chan, C.-Y. et Y. E. Ioannidis. 1998. « Bitmap index design and evaluation », *SIGMOD Rec.*, vol. 27, no. 2, p. 355–366.

Chang, E., Z. Manna, et A. Pnueli. 1994. « Compositional verification of real-time systems ». In *Logic in Computer Science, 1994. LICS'94. Proceedings., Symposium on*, p. 458–465. IEEE.

Chen, F. et G. Roşu. 2007. « Mop : an efficient and generic runtime verification framework ». In *ACM SIGPLAN Notices*. T. 42, p. 569–588. ACM.

Clarke, E. M., O. Grumberg, et D. Peled. 1999. *Model checking*. MIT press.

Colantonio, A. et R. Di Pietro. 2010. « Concise : Compressed 'n' composable integer set », *Information Processing Letters*, vol. 110, no. 16, p. 644–650.

Comer, D. E. 2008. *Computer Networks and Internets*. Upper Saddle River, NJ, USA : Prentice Hall Press, 5th édition.

Culpepper, J. S. et A. Moffat. 2010. « Efficient set intersection for inverted indexing », *ACM Transactions on Information Systems (TOIS)*, vol. 29, no. 1, p. 1.

d'Amorim, M. et K. Havelund. 2005. « Event-based runtime verification of java programs ». In *ACM SIGSOFT Software Engineering Notes*. T. 30, p. 1–7. ACM.

Denso Wave Inc. 2015. What is a QR code ? Online ; accessed 11-June-2015.

Drusinsky, D. 2000. *The temporal rover and the ATG rover*. Coll. « SPIN Model Checking and Software Verification », p. 323–330. Springer.

Eisner, C. et D. Fisman. 2006. *A Practical Introduction to PSL*. Springer.

Emerson, E. A. 1990. « Temporal and modal logic. », *Handbook of Theoretical Computer Science, Volume B : Formal Models and Sematics (B)*, vol. 995, no. 1072, p. 5.

Evers, H. 1979. « The Hellschreiber : A rediscovery », *HAM Radio*, p. 28–32.

Falcone, Y., K. Havelund, et G. Reger. 2013. « A tutorial on runtime verification. », *Engineering Dependable Software Systems*, vol. 34, p. 141–175.

Farahani, S. 2011. *ZigBee wireless networks and transceivers*. newnes.

Gao, J. Z., L. Prakash, et R. Jagatesan. 2007. « Understanding 2d-barcode technology and applications in m-commerce-design and implementation of a 2d barcode processing solution ». In *Computer Software and Applications Conference, 2007. COMPSAC 2007. 31st Annual International*. T. 2, p. 49–56. IEEE.

Gupta, N. 2013. *Inside Bluetooth low energy*. Artech house.

Ha, J., M. Arnold, S. M. Blackburn, et K. S. McKinley. 2009. « A concurrent dynamic analysis framework for multicore hardware ». In *ACM SIGPLAN Notices*. T. 44, p. 155–174. ACM.

Hallé, S. et M. Soucy-Boivin. 2015. « MapReduce for parallel trace validation of LTL properties », *Journal of Cloud Computing*, vol. 4, no. 8, p. 1–16.

Havelund, K. et G. Rosu. 2001. « Java pathexplorer : A runtime verification tool ».

Havelund, K. et G. Roşu. 2001. « Monitoring programs using rewriting ». In *Automated Software Engineering, 2001.(ASE 2001). Proceedings. 16th Annual International Conference on*, p. 135–143. IEEE.

———. 2004. « Efficient monitoring of safety properties », *International Journal on Software Tools for Technology Transfer*, vol. 6, no. 2, p. 158–173.

Heisel, M., W. Reif, et W. Stephan. 1990. « Tactical theorem proving in program verification ». In *10th International Conference On Automated Deduction*, p. 117–131. Springer.

Huth, M. et M. Ryan. 2004. *Logic in Computer Science : Modelling and reasoning about systems*. Cambridge University Press.

IEEE. 2012. « Ieee standard for system and software verification and validation », *IEEE Std 1012-2012 (Revision of IEEE Std 1012-2004)*, p. 1–223.

International Organization for Standardization. 2006. Information technology – automatic identification and data capture techniques – QR Code 2005 bar code symbology specification, ISO standard 18004.

Kaser, O. et D. Lemire. 2014. Compressed bitmap indexes : beyond unions and intersections. Accepted for publication in Software : Practice and Experience on August 14th 2014. Note that arXiv :1402.4073 [cs :DB] is a companion to this paper ; while they share some text, each contains many results not in the other.

Kim, M., M. Viswanathan, S. Kannan, I. Lee, et O. Sokolsky. 2004. « Java-mac : A run-time assurance approach for java programs », *Formal methods in system design*, vol. 24, no. 2, p. 129–155.

Komine, T. et M. Nakagawa. 2004. « Fundamental analysis for visible-light communication system using led lights », *Consumer Electronics, IEEE Transactions on*, vol. 50, no. 1, p. 100–107.

Lavoie, K., C. Leplongeon, S. Varvaressos, S. Gaboury, et S. Hallé. 2014. « Portable runtime verification with smartphones and optical codes ». In Bonakdarpour, B. et S. A. Smolka, édi-

teurs, *Runtime Verification - 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings*. T. 8734, série *Lecture Notes in Computer Science*, p. 80–84. Springer.

Lee, J.-S., Y.-W. Su, et C.-C. Shen. 2007. « A comparative study of wireless protocols : Bluetooth, UWB, ZigBee, and Wi-Fi ». In *Industrial Electronics Society, 2007. IECON 2007. 33rd Annual Conference of the IEEE*, p. 46–51. IEEE.

Lemire, D., O. Kaser, et K. Aouiche. 2010. « Sorting improves word-aligned bitmap indexes », *Data & Knowledge Engineering*, vol. 69, no. 1, p. 3–28.

Leucker, M. et C. Schallhart. 2009. « A brief account of runtime verification », *The Journal of Logic and Algebraic Programming*, vol. 78, no. 5, p. 293–303.

Millar, I., M. Beale, B. J. Donoghue, K. W. Lindstrom, et S. Williams. 1998. « The IrDA standard for high-speed infrared communications », *HP Journal*, p. 2.

Okazaki, S., H. Li, et M. Hirose. 2012. « Benchmarking the use of qr code in mobile promotion », *Journal of Advertising Research*, vol. 52, no. 1, p. 102–117.

Pellizzoni, R., P. Meredith, M. Caccamo, et G. Rosu. 2008. « Hardware runtime monitoring for dependable cots-based real-time embedded systems ». In *Real-Time Systems Symposium, 2008*, p. 481–491. IEEE.

Perahia, E. et R. Stacey. 2013. *Next Generation Wireless LANS : 802.11 n and 802.11 ac*. Cambridge university press.

Pnueli, A. 1977. « The temporal logic of programs ». In *Foundations of Computer Science, 1977., 18th Annual Symposium on*, p. 46–57.

Richter, S. P. 2013. « Digital semaphore : Tactical implications of QR code optical signaling for fleet communications ». Mémoire de maîtrise, Naval Postgraduate School.

Rozier, K. Y. 2011. « Linear temporal logic symbolic model checking », *Computer Science Review*, vol. 5, no. 2, p. 163–203.

Sarkar, S. K., T. Basavaraju, et C. Puttamadappa. 2007. *Ad hoc mobile wireless networks : principles, protocols and applications*. CRC Press.

Shabtai, A. et Y. Elovici. 2010. *Applying behavioral detection on android-based devices*. Coll. « Mobile Wireless Middleware, Operating Systems, and Applications », p. 235–249. Springer.

Theng, Y.-L. 2008. *Ubiquitous Computing : Design, Implementation and Usability : Design, Implementation and Usability*. IGI Global.

Tse, D. et P. Viswanath. 2005. *Fundamentals of wireless communication*. Cambridge university press.

Uno, T., M. Kiyomi, et H. Arimura. 2005. « Lcm ver.3 : Collaboration of array, bitmap and prefix tree for frequent itemset mining ». In *Proceedings of the 1st International Workshop on Open Source Data Mining : Frequent Pattern Mining Implementations*. Coll. « OSDM '05 », p. 77–86, New York, NY, USA. ACM.

Vasilescu, I., K. Kotay, D. Rus, M. Dunbabin, et P. Corke. 2005. « Data collection, storage, and retrieval with an underwater sensor network ». In *Proceedings of the 3rd international conference on Embedded networked sensor systems*, p. 154–165. ACM.

Wikipedia. 2016. Runtime verification — wikipedia, the free encyclopedia. [Online ; accessed 10-May-2016].

Witze, A. 2016. « Software error doomed japanese hitomi spacecraft », *Nature*, vol. 533, p. 18–19.

Wu, K., E. J. Otoo, et A. Shoshani. 2006. « Optimizing bitmap indices with efficient compression », *ACM Transactions on Database Systems (TODS)*, vol. 31, no. 1, p. 1–38.

Xie, K., S. Gaboury, et S. Hallé. 2016. « Real-time streaming communication with optical codes », *IEEE Access*, vol. 4, p. 284–298.

Zwijze-Koning, K. H. et M. D. De Jong. 2005. « Auditing information structures in organizations : A review of data collection techniques for network analysis », *Organizational Research Methods*, vol. 8, no. 4, p. 429–453.