



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 professeur **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	vii
Liste des tableaux	viii
1 Introduction	1
1.1 Contexte	1
1.2 Objectifs du mémoire	4
1.3 Méthodologie	5
1.4 Organisation du mémoire	6
2 Revue de la vérification de l'exécution et des travaux connexes	9
2.1 Vérification de l'exécution	9
2.2 Logique Temporelle Linéaire	14
2.3 Cadre de la vérification de l'exécution	21
3 Communication en streaming et en temps réel avec les codes optiques	27
3.1 Introduction	27
3.2 Communications sans fil	30
3.3 Flux de codes QR	33
3.4 Expériences	39
3.5 Un protocole de canaux de communication unidirectionnelle avec perte	52
3.6 Conclusion	74
4 Offline Evaluation of LTL Formulae with Bitmap Manipulations	77
4.1 Introduction	77
4.2 Bitmaps and Compression	79
4.3 Evaluating LTL formulae with Bitmap	83
4.4 Implementation and Experiments	92
4.5 Related Work	98
4.6 Conclusion and future work	100
5 Conclusion and future work	105

Annexe A Experiment results of calculating LTL formulæes with bitmap compression	111
Bibliographie	125

TABLE DES FIGURES

2.1	Processus de la vérification de l'exécution (de Falcone et al. (2013))	12
2.2	L'architecture de JPaX (Havelund et Rosu, 2001)	22
2.3	L'architecture de MaC (Kim et al., 2004)	23
2.4	L'architecture d'Eagle (d'Amorim et Havelund, 2005)	23
2.5	L'architecture de MOP (Chen et Roşu, 2007)	24
3.1	Un code QR avec le texte "Hello world !"	34
3.2	Installation expérimental pour lire les codes QR.	41
3.3	Un code QR généré par ZXing que ZXing en soi ne peut pas décoder dans l'expérience.	43
3.4	Bande passante et taux de décodage dans la première expérience	46
3.5	Bande passante et taux de décodage dans la deuxième expérience, où chaque code est affiché deux fois	48
3.6	Exemples de deux codes avec les données légèrement différentes, mais très différentes formes de points. Le code à gauche contient la chaîne "abcdefg", tandis que celui à droite contient "abcdefg".	49
3.7	Troisième expérience : affichage en trois fois avec bourrage aléatoire	50
3.8	Quatrième expérience : affichage en deux fois et les fréquences plus élevées de codes	51
3.9	Une partie de l'interface texte du récepteur de codes QR qui fonctionne en mode Lake	66
3.10	Le schéma d'événements produits par un jeu vidéo instrumenté.	68
3.11	Temps de l'envoi des données en mode Lake	70
3.12	Temps de l'envoi des données en mode Lake Stream	71
3.13	Passer la caméra au-dessus d'un ensemble de codes QR pour reconstituer le contenu d'un fichier plus grand.	73
4.1	A graphical representation of the computation of three temporal operators on bitmaps	87
4.2	Bitmap generation with compression algorithms	97
4.3	Comparison of compression ratio and processing rate for LTL formula 1, for various bitmap compression libraries and various values of <i>slen</i>	98
4.4	Comparison of compression ratio and processing rate for LTL formula 14, for various bitmap compression libraries and various values of <i>slen</i>	98

LISTE DES TABLEAUX

2.1	La table de vérité des opérateurs booléens de la logique propositionnelle . . .	17
2.2	Cadre de la vérification de l'exécution	21
3.1	Résumé de protocoles WiFi (Theng, 2008; Perahia et Stacey, 2013)	30
3.2	Spécifications de Bluetooth (Gupta, 2013)	31
3.3	Spécifications de ZigBee (Lee et al., 2007)	31
3.4	Débits de données de schémas de couche physique de IrDA (Millar et al., 1998)	32
3.5	Capacités	34
3.6	Tailles d'échantillons	44
4.1	Parameters of RLE-model algorithms	82
4.2	The semantics of LTL. Here \bar{s}^i denotes the subtrace of \bar{s} that starts at event i . .	84
4.3	Derivative bitmap functions	85
4.4	Bitmap libraries	93
4.5	Running time for evaluating each LTL operator on a bit vector, without the use of a compression library.	95
4.6	The complex LTL formulæ evaluated experimentally.	102
4.7	Running time for the evaluation of LTL formulæ of Table 4.6, without the use of a compression library.	103
A.1	Bitmap generation with compression algorithms	111
A.2	Calculation of $\neg s_0$ with compression algorithms	112
A.3	Calculation of $s_0 \wedge s_1$ with compression algorithms	112
A.4	Calculation of $s_0 \vee s_1$ with compression algorithms	113
A.5	Calculation of $s_0 \vee s_1$ with compression algorithms	113
A.6	Calculation of $\mathbf{X} s_0$ with compression algorithms	114
A.7	Calculation of $\mathbf{G} s_0$ with compression algorithms	114
A.8	Calculation of $\mathbf{F} s_0$ with compression algorithms	115
A.9	Calculation of $s_0 \mathbf{U} s_1$ with compression algorithms	115
A.10	Calculation of $s_0 \mathbf{W} s_1$ with compression algorithms	116
A.11	Calculation of $s_0 \mathbf{R} s_1$ with compression algorithms	116
A.12	Formulæ 1 calculation with compression algorithms	117
A.13	Formulæ 2 calculation with compression algorithms	117
A.14	Formulæ 3 calculation with compression algorithms	118
A.15	Formulæ 4 calculation with compression algorithms	118

A.16 Formulæ 5 calculation with compression algorithms	119
A.17 Formulæ 6 calculation with compression algorithms	119
A.18 Formulæ 7 calculation with compression algorithms	120
A.19 Formulæ 8 calculation with compression algorithms	120
A.20 Formulæ 9 calculation with compression algorithms	121
A.21 Formulæ 10 calculation with compression algorithms	121
A.22 Formulæ 11 calculation with compression algorithms	122
A.23 Formulæ 12 calculation with compression algorithms	122
A.24 Formulæ 13 calculation with compression algorithms	123
A.25 Formulæ 14 calculation with compression algorithms	123

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 (Avizienis 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 certaines 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 le nombre 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. Zijze-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 faible 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 de 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 de 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 de 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 de 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 bibliothèques 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 workshop¹ 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’autres 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’une *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)

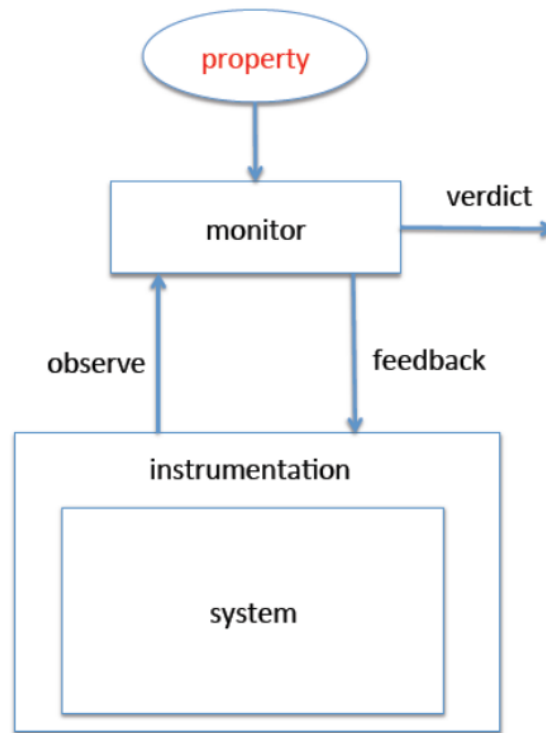


Figure 2.1: Processus de la vérification de l'exécution (de Falcone et al. (2013))

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 satisfait 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 bien formée de la LTL 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) :

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

2.2.2 SÉMANTIQUES

Pour une séquence d'états $s_0, s_1, s_2, \dots, s_i, s_{i+1}, \dots$ où s_{i+1} est un état futur de s_i , on définit un chemin avec $\pi^i = s_i \rightarrow s_{i+1} \rightarrow \dots$ 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 de la LTL est définie comme suit (Rozier, 2011) :

$$\text{— } \pi^i \models \top \quad (2.2)$$

$$\text{— } \pi^i \not\models \perp \quad (2.3)$$

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

$$— \pi^i \models \neg \psi \iff \pi^i \not\models \psi \quad (2.5)$$

$$— \pi^i \models \psi \wedge \varphi \iff \pi^i \models \psi \text{ et } \pi^i \models \varphi \quad (2.6)$$

$$— \pi^i \models \psi \vee \varphi \iff \pi^i \models \psi \text{ ou } \pi^i \models \varphi \quad (2.7)$$

$$— \pi^i \models \psi \rightarrow \varphi \iff \pi^i \models \varphi \text{ chaque fois que } \pi^i \models \psi \quad (2.8)$$

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

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

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

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

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

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

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 \mathbf{X} signifie “la prochaine fois” et il saute le i ème état et évalue le chemin π^{i+1} . L'opérateur \mathbf{F} signifie “parfois dans l'avenir” qui exige qu'à partir du i ème état, une propriété reste valide dans un état futur sur le chemin. Et l'opérateur \mathbf{G} (“globalement” ou “toujours”) indique

ψ	ϕ	$\neg\psi$	$\psi \wedge \phi$	$\psi \vee \phi$	$\psi \rightarrow \phi$
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

qu'une propriété devrait rester 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 $\psi \text{ U } \phi$ 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 $\psi \text{ W } \phi$, 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. $\psi \text{ U } \phi \equiv \neg(\neg\psi \text{ R } \neg\phi)$. L'opérateur **R** exige que pour la formule $\psi \text{ R } \phi$, 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, $\mathbf{F} \psi$ 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 (LTL₃) 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 :

$$\phi ::= \perp \mid p \mid (\phi \rightarrow \phi) \mid (\bigcirc_{\sim c} \phi) \mid (\ominus_{\sim c} \phi) \mid (\phi U_{\sim c} \phi) \mid (\phi S_{\sim c} \phi)$$

où $\sim \in \{<, =, >, \equiv_d\}$ 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) :

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

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) :

$$\begin{aligned} \phi ::= & \top \mid \perp \mid p \mid (\neg \phi) \mid (\phi \wedge \phi) \mid (\phi \vee \phi) \mid (\phi \rightarrow \phi) \\ & \mid (\odot \phi) \mid (\diamond \phi) \mid (\Box \phi) \mid (\phi S_s \phi) \mid (\phi S_w \phi) \\ & \mid (\uparrow \phi) \mid (\downarrow \phi) \mid [\phi, \phi]_s \mid [\phi, \phi]_w \end{aligned}$$

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”, \Box “toujours dans le passé”, S_s “intervalle fort” et S_w “intervalle faible”.

Les opérateurs de monitoring $\uparrow \downarrow$, $[\cdot]_s$, $[\cdot]_w$ 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 :

$$\begin{aligned}
\pi^i \models \odot \psi &\iff \text{si } i > 0, \pi^{i-1} \models \psi, \text{ ou si } i = 0, \pi^0 \models \psi \\
\pi^i \models \diamond \psi &\iff i > 0 \text{ et } \exists j \text{ où } 0 \leq j \leq i, \pi^j \models \psi \\
\pi^i \models \Box \psi &\iff i > 0 \text{ et } \forall j \text{ où } 0 \leq j \leq i, \pi^j \models \psi \\
\pi^i \models \psi S_s \varphi &\iff \exists 0 \leq j \leq i, \pi^j \models \varphi \text{ et } \forall k, j < k \leq i, \pi^k \models \psi \\
\pi^i \models \psi S_w \varphi &\iff \pi^i \models \psi S_s \varphi \text{ ou } \pi^i \models \Box \psi \\
\pi^i \models \uparrow \psi &\iff \pi^i \models \psi \text{ et } \pi^{i-1} \not\models \psi \\
\pi^i \models \downarrow \psi &\iff \pi^i \not\models \psi \text{ et } \pi^{i-1} \models \psi \\
\pi^i \models [\psi, \varphi)_s &\iff \exists j \text{ où } 0 \leq j \leq i, \pi^j \models \psi, \text{ et } \forall k \text{ où } j \leq k \leq i, \pi^k \not\models \varphi \\
\pi^i \models [\psi, \varphi)_w &\iff \pi^i \models [\psi, \varphi)_s \text{ et } \pi^i \models \Box \neg \varphi
\end{aligned}$$

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 extensions 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.

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

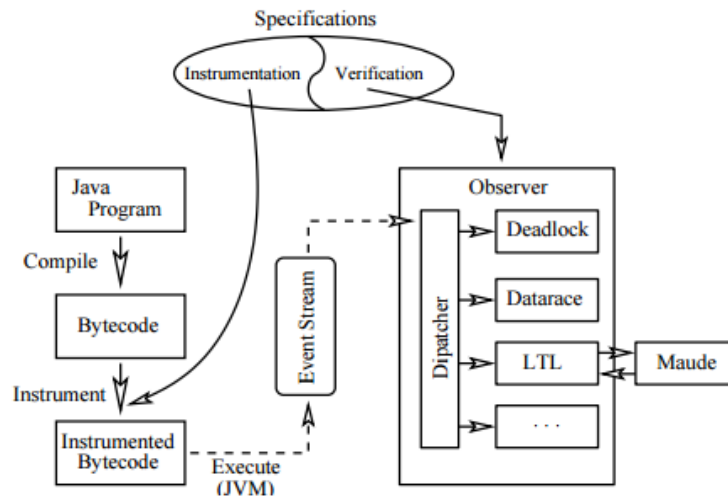


Figure 2.2: L'architecture de JPaX (Havelund et Rosu, 2001)

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). 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

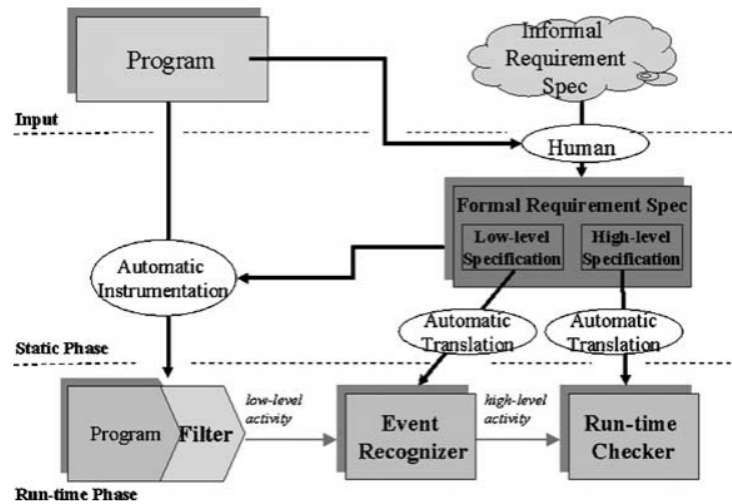


Figure 2.3: L'architecture de MaC (Kim et al., 2004)

évalue la formule dans l'état actuel pour produire un résultat.

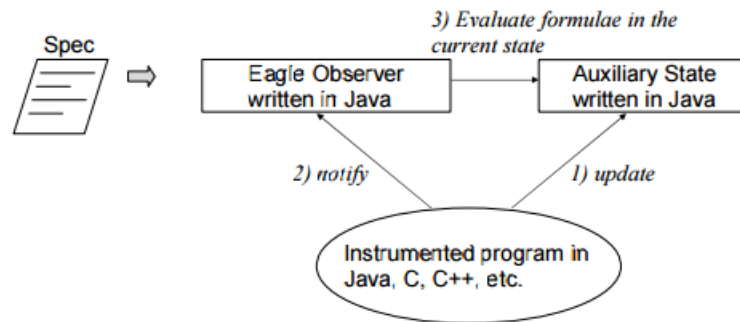


Figure 2.4: L'architecture d'Eagle (d'Amorim et Havelund, 2005)

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).

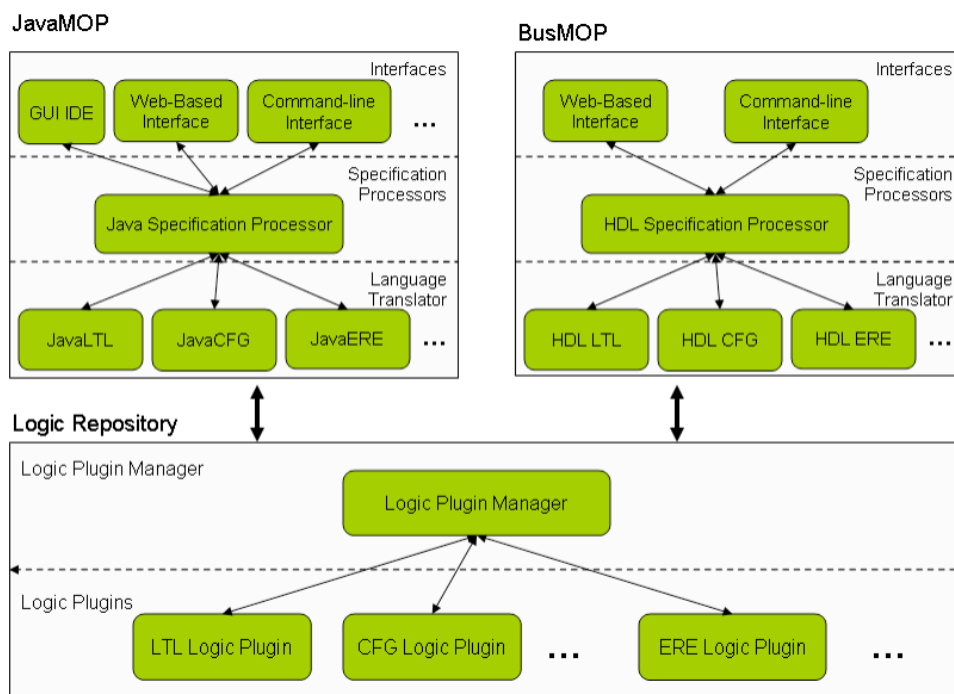


Figure 2.5: L'architecture de MOP (Chen et Roşu, 2007)

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.

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 de 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, appelé *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 portées à une variété de dispositifs, y compris les ordinateurs de bureau, les smartphones, et même les télévisions.

Cependant, jusqu’au présent les codes QR ont été utilisés pour la transmission de données statique. En général, un code est imprimée 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 ?? 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 couvre 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 code 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 impossible. 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 ?? . 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 du bras vers un écran avec les code 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 de 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 séquentielles — 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, chacun a ses propres limites et les 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 physique	Portée intérieur
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 de 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 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 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 commun point : 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 les technologies utilisant les ondes infrarouges au lieu du signal radio (Sarkar et al., 2007). Outre les longueurs d’onde différentes, ces technologies se détendent 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 des petit morceaux de données sans nécessité de préavis.

Une importance particulière est le standard IrDA (Infrared Data Association) ; ses émetteurs envoient les 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 Communication Light (VLC) (Komine et Nakagawa, 2004) utilise les 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 parce que 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 la 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 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 mentionné 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 discute 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 petit.¹ Le standard du code QR

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

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 177×177 “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 21×21 à 177×177 modules. Les codes QR utilisent une forme de codage à correction d’erreur qui peut choisir parmi 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 inclus 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 le mode et la version la plus efficace. Dans le codage de données, les caractères sont convertis en un flux de bits, et dans ce progrès, certains *indicateurs de mode* et *termateurs* 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'erreurs 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 forme 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'erreur 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 DU CODE QR COMMUNICATION

Le processus précité s'applique au codage au décodage d'un code unique contenant les données statiques. Nous enquêtons maintenant l'idée de l'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 capturés par un dispositif, et reconverti 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 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 *statique*, c'est-à-dire qu'ils ne changent pas au fil du temps pour former un flux de données, et plus ou moins agissent 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 dédié. En revanche, la communication de codes QR peut être réalisée à travers les codes imprimés sur une surface dure, ou par un

dispositif capable d'afficher les 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 d'une en faveur de l'autre. 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 représentent trop de couplage dans de nombreux scénarios. Nous avons discuté dans le travail passé (?) 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 des trames 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 en résulte. 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 plus élevé niveau de correction exige plus de modules de symboles qu'un moins élevé niveau.

Parce que l'émetteur ne peut pas détecter le fait que des codes sont 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 déterminée 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 le même image plus d'une fois, et en ce moment 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 expérimental 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 de 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 vue. 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 OpenCV² pour capturer les images de la caméra et ZXing³ pour générer et décoder les codes QR. Pour réduire la pression du CPU et du 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. Tous les codes du benchmark est implémentés 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 bibliothèques de reconnaissance optique

2. <http://opencv.org/>

3. <https://github.com/zxing/zxing>

4. <http://github.com/sylvainhalle/GyroGearloose>

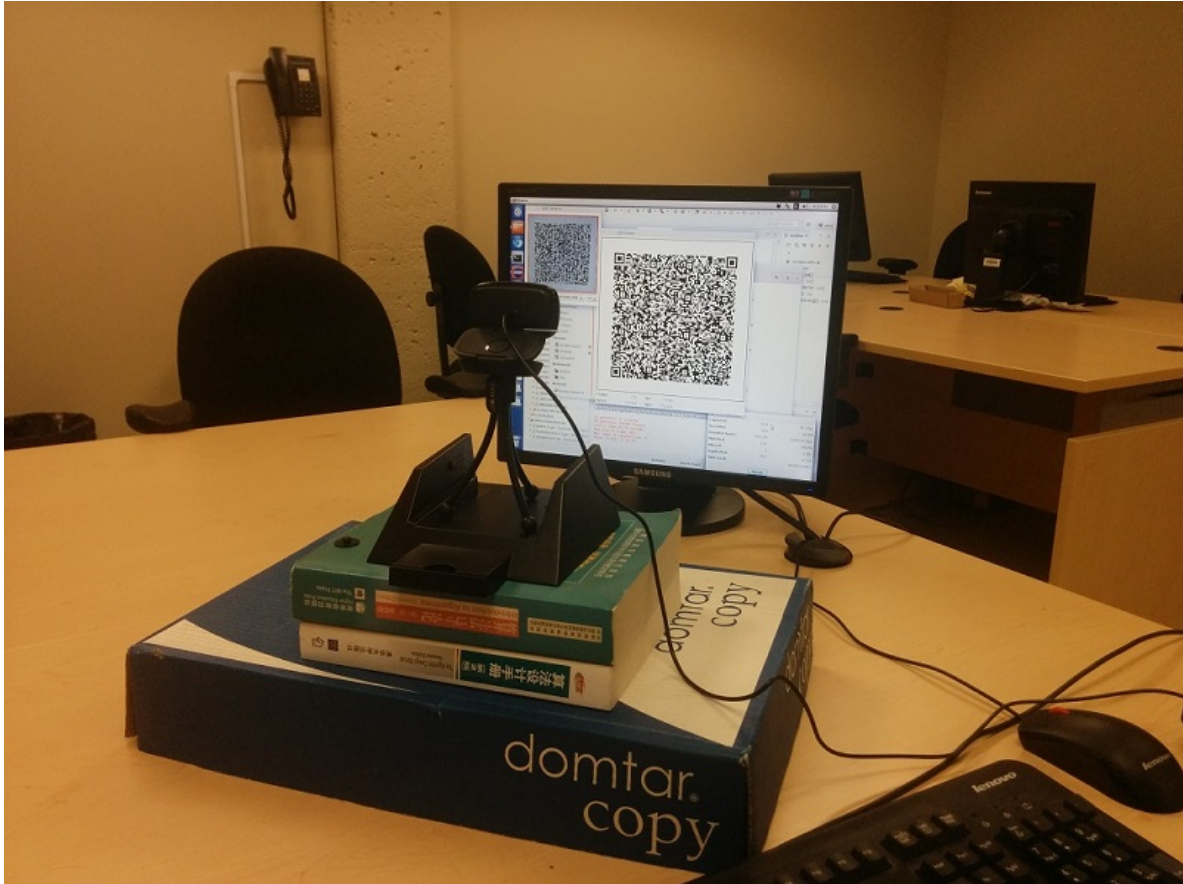


Figure 3.2: Installation expérimental pour lire les codes QR.

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 $dddd\#rrrr\dots$, où $dddd$ est un numéro séquentiel à partir de zéro et incrémenté par un dans chaque code successifs, et $rrrr\dots$ est une chaîne de caractères aléatoires (différent dans chaque code) assez longue pour combler le code jusqu'à sa taille maximale. Chaque test a consisté à filmer la séquence des 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 (non trouble, correct image, 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 plus curieux fait est que les codes immédiatement avant et après le code problématique ont été correctement décodées, tout en étant capturé 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 persiste 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 soient causées être par ce bug prétendu, 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'expérimentation 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 plus élevé niveau de correction d'erreur, occupe 101×101 modules. Nous avons aussi fixé la taille physique de codes à 700×700 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 leur tailles de données et les niveaux de correction d'erreur (International Organization for Standardization, 2006)

Fréquence de codes

Le deuxième paramètre expérimental 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 ultérieure phase 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'erreur. 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 base résolutions (en baisse à 640×480), qui étaient mondialement concluants, mais qui n'étaient pas considérées pertinentes 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

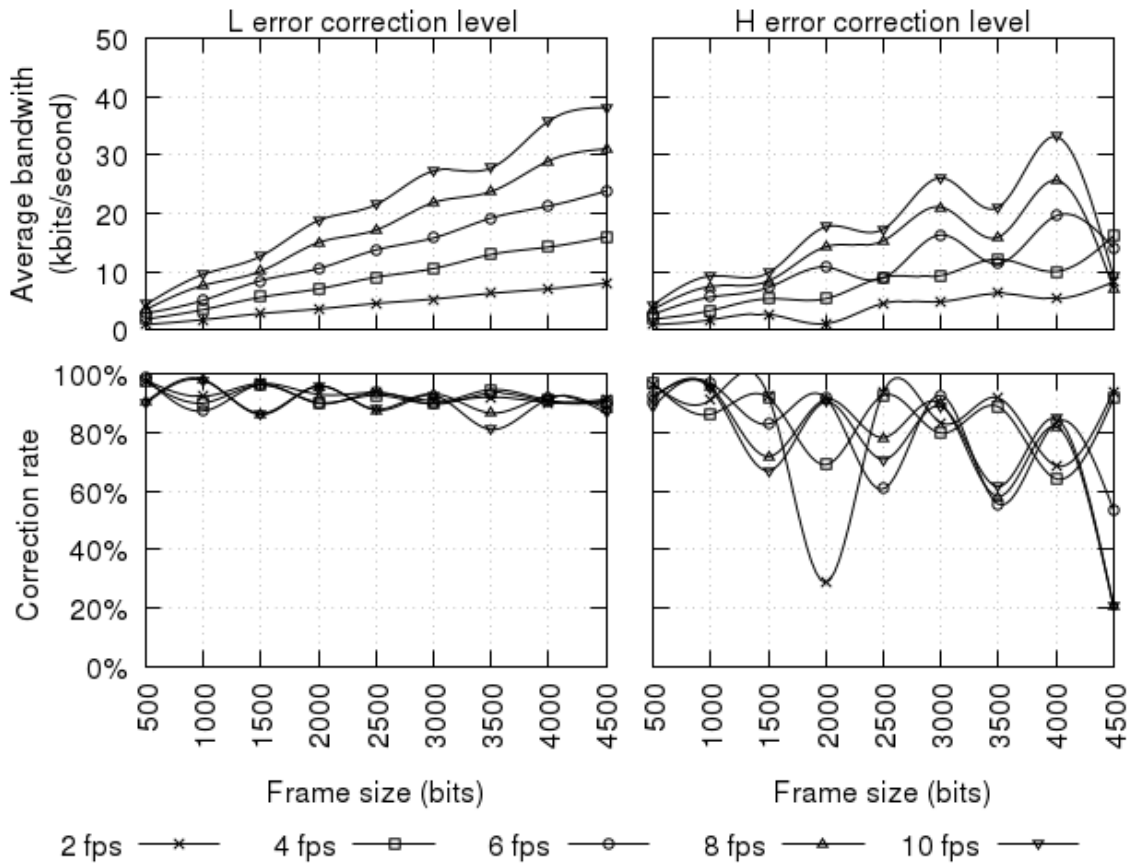


Figure 3.4: Bande passante et taux de décodage dans la première expérience

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

Comme on peut le voir, les taux de reconnaissance de plus élevé niveaux de correction étaient inférieurs à ceux de plus bas niveaux de correction, avec tous les autres paramètres étant égaux. Ceci peut être expliqué par le fait que la même quantité de données, portées dans d'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 son 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 d'une image trouble qui montre une partie de deux différents codes) est également augmentée. En niveau L, la diminution est légère, tandis que en niveau H, la diminution est dramatique lorsque la taille du codes 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 voisins codes 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

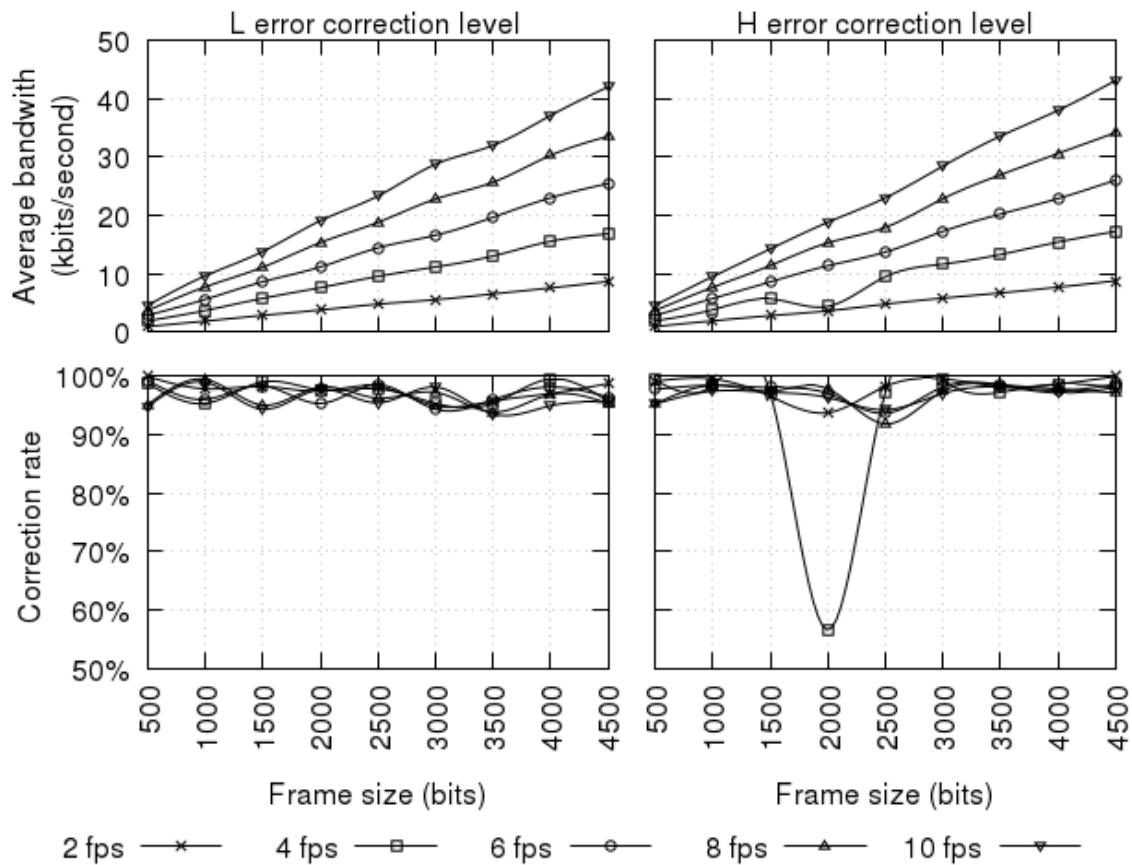


Figure 3.5: Bande passante et taux de décodage dans la deuxième expérience, où chaque code est affiché deux fois

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

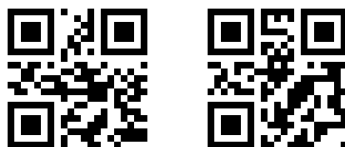


Figure 3.6: Exemples de deux codes avec les données légèrement différentes, mais très différentes formes de points. Le code à gauche contient la chaîne “abcdefg”, tandis que celui à droite contient “abcdefg”.

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 difficile à reconnaître soit inconnue et hors du point 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

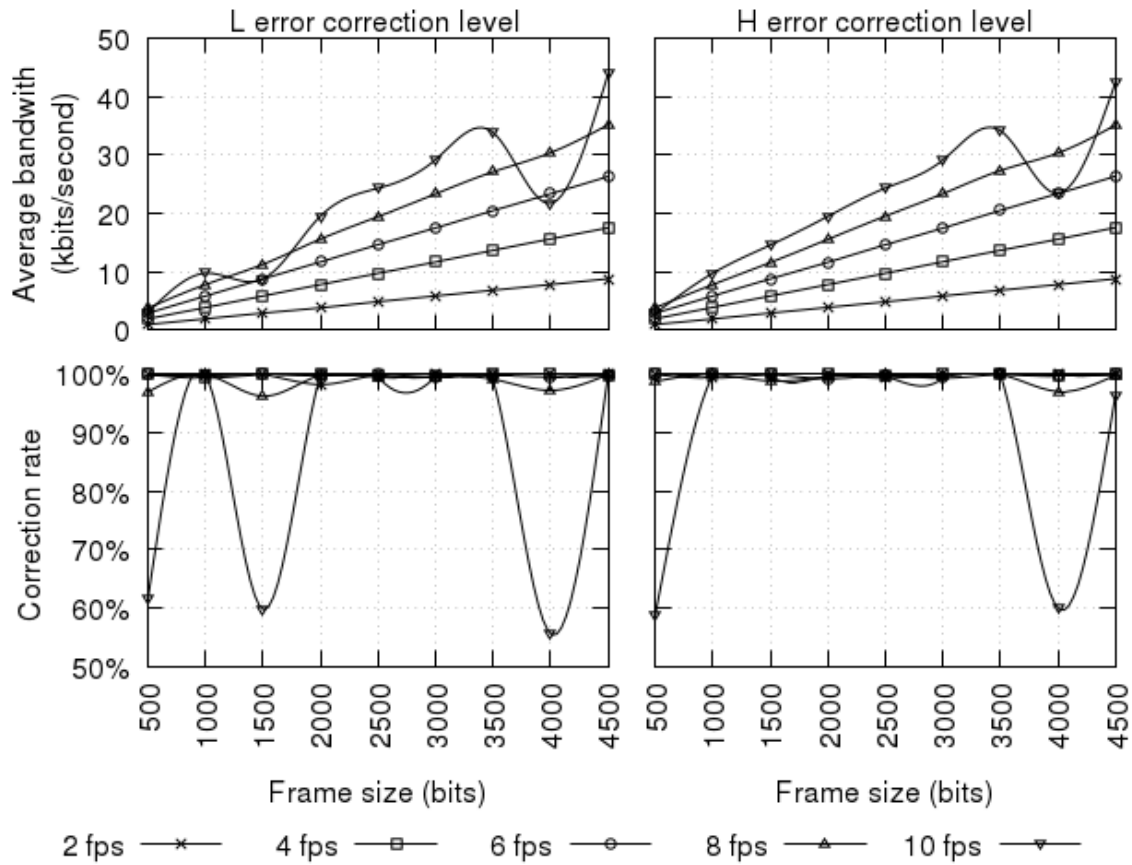


Figure 3.7: Troisième expérience : affichage en trois fois avec bourrage aléatoire

avons ajouté 12 cps, 14 cps et 16 cps. Les codes ont été affichés deux fois. Comme la figure 3.8 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'erreur

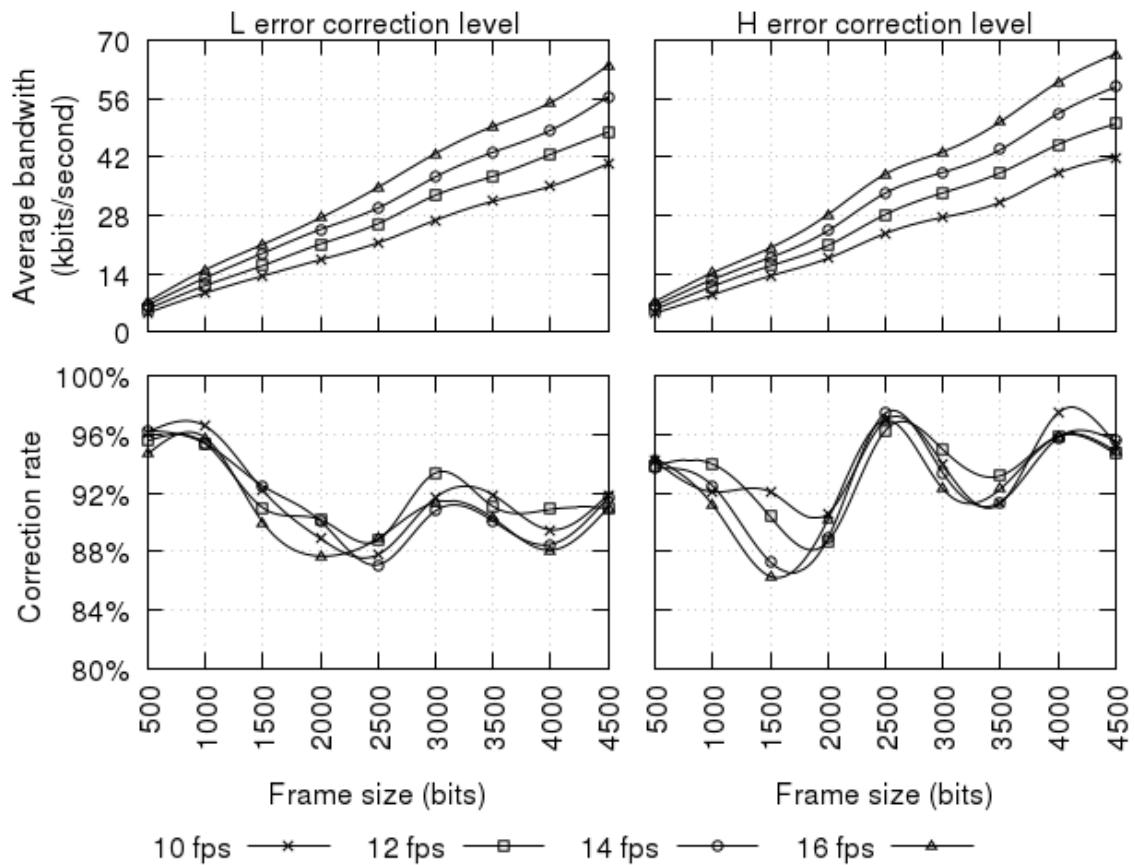


Figure 3.8: Quatrième expérience : affichage en deux fois et les fréquences plus élevées de codes

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 attendre d'un flux de QR code qui fournit 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%. Évidemment, 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 une expérience informelle décrite dans la section 3.5.8 tend 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 continus codes QR en tant qu'un 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 essentielle ; 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 Google⁶, la définition de nouveaux types de messages peuvent être effectuées 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

5. <https://github.com/sylvainhalle/BufferTannen>

6. <https://github.com/google/protobuf>

sans connaissance 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.) peut 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 quelques centaines d'octets à un moment, peut-être moins de 10 fois par seconde).
- Le canal est unidirectionnelle : 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, e.g. 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 formel "début" 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), Télétex

(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 toute 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 pas confondre une re-transmission avec un nouveau message qui ont le identique contenu. 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 avec la 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 sont donnés 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 avec la 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 exactes qui peuvent être utilisés en tant que clés doit être déclarée. Cependant, chaque clé peut être associée à une valeur d'un 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 de 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 `chapters`, dont la valeur n'est pas un type primitif, mais en soi est une liste (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 applique uniquement aux entiers qui se produisent en tant que valeurs complètes.

Smallscii La déclaration d'une chaîne Smallscii est simplement encodé 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 1111 [ssssss ssssss ... 000000 ...
      ssssss ssssss ... 000000]
```

Le type d'élément est la valeur décimale 1, et la séquence 1111 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 11111111 ...
```

Le type d'élément est la valeur décimale 3 ; le 8-bit séquence 11111111 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 correspondant de `map`.

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 commence par 8 bits qui 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.

FixedMap 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 utilisent 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 restrictive, 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 celui, les répétitions de la description de son schéma dans chaque message se trouvent tout 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 que d'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 wwwwwwwwww 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 créée. 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 wwwwwwwwwww rrrrrrrrrrrr...
```

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 tout encodées sur 12 bits. La section *r* donne le numéro séquentiel d'un autre segment, par rapport à 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 pleine 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 pour plusieurs images. Ainsi, un segment ne peut pas dépasser la taille maximale d'une image.

Chaque image est ensuite converti 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 manquantes

```

-----
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

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 reconstruire. 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 pour 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 de 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 original, 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 reçus 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 quelles parties de la séquence totale correspondent ces segments. 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 code 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 une 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 pour chacun, qui consomme une bande passante d'environ 24 kbps. Nos expériences précédentes montrent que cela est dans le gamme de ce que l'on peut raisonnablement 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 compacte⁸

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 le taux plus de 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 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 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 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=GSL0mdOT1Y8>

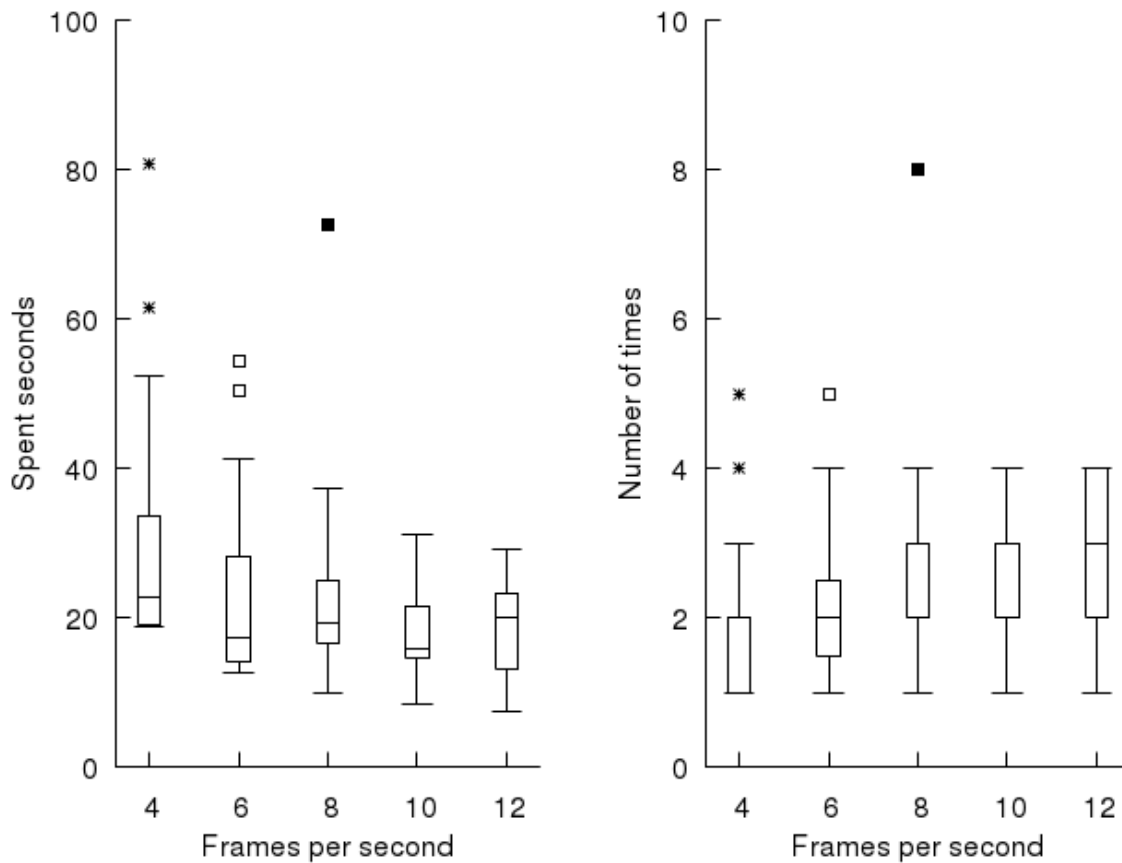


Figure 3.11: Temps de l'envoi des données en mode Lake

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 un limite certain et prédéfini.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 la technologie

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

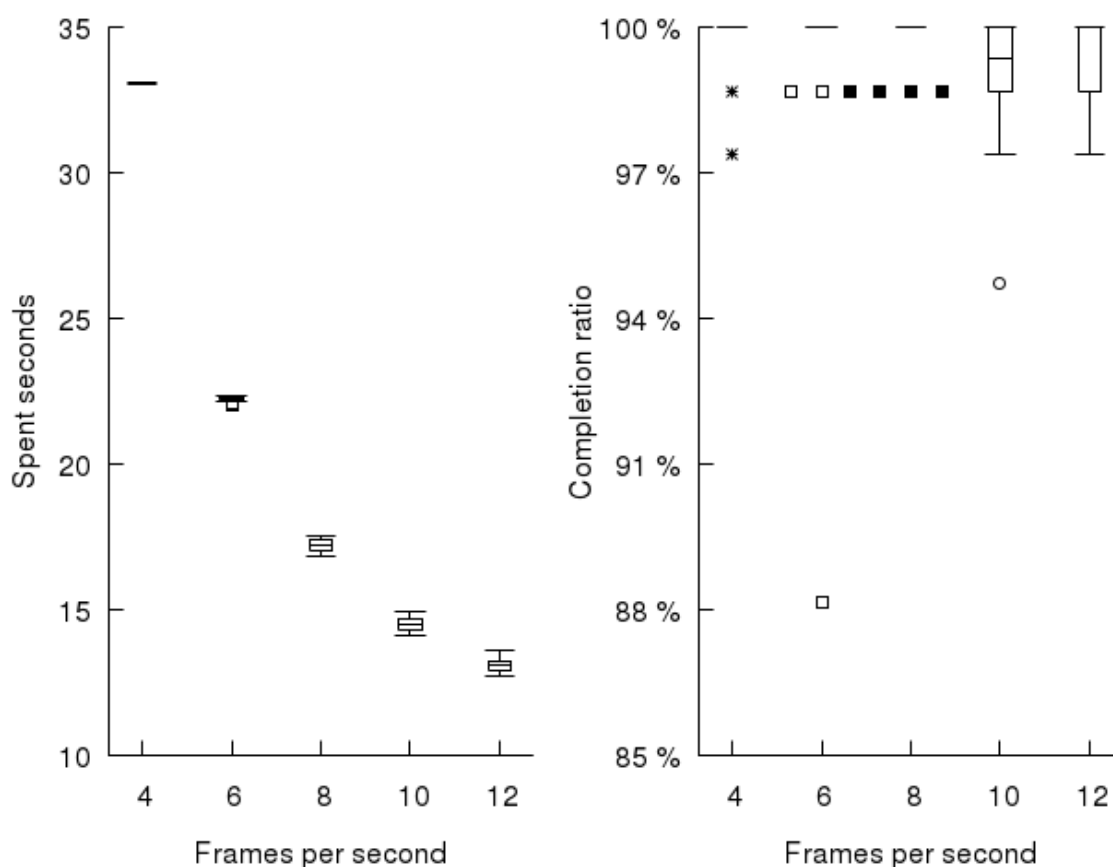


Figure 3.12: Temps de l'envoi des données en mode Lake Stream

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. Chaque tel code peut être formaté pour contenir une unique image 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 testé 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

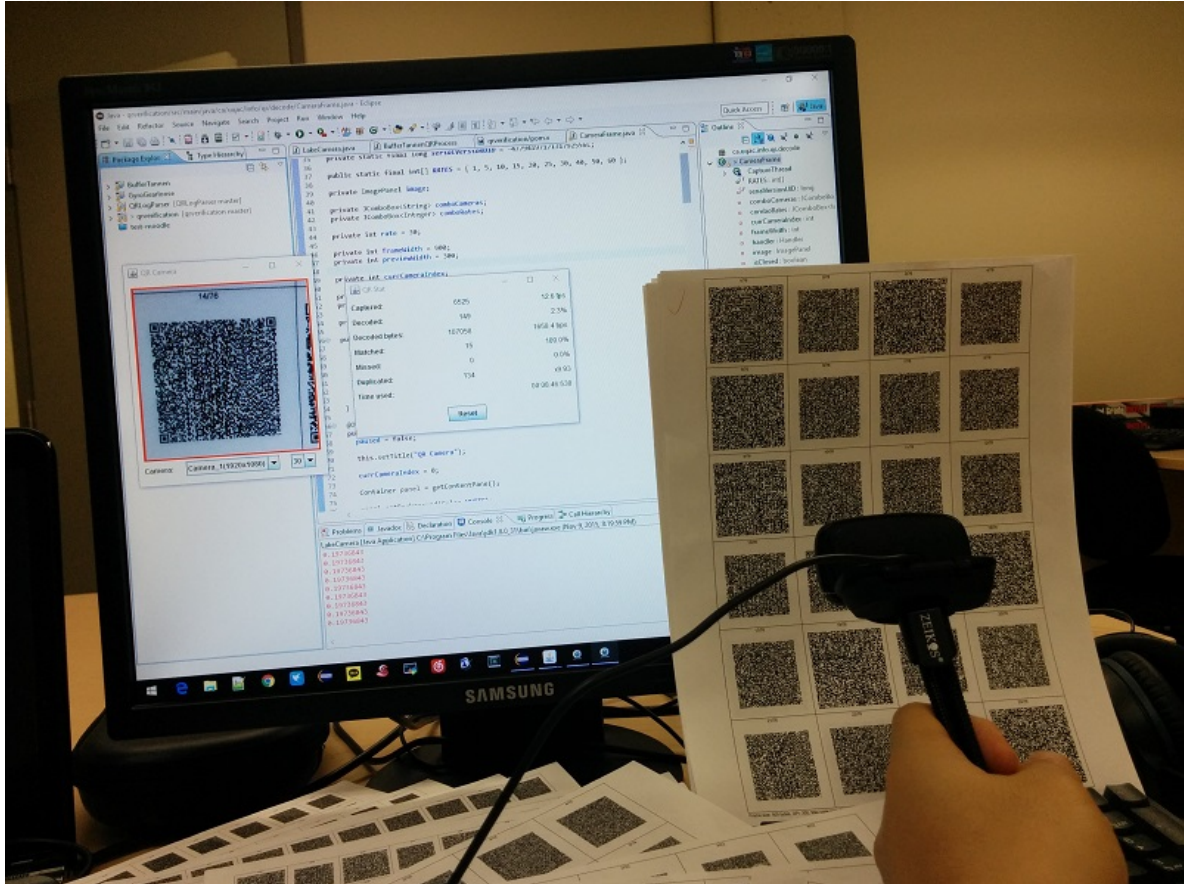


Figure 3.13: Passer la caméra au-dessus d'un ensemble de codes QR pour reconstituer le contenu d'un fichier plus grand.

caméra près du papier pendant quelques secondes, mais parmi les totaux 31 codes, trois restaient impossible de décoder du tout. Dans le cas des codes de 1.750 et de 2.000 octets, aucun des codes ne pourra 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 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 pourra 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 du focus. 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 dans le 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 les 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 de 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 de 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 difficiles, 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

OFFLINE EVALUATION OF LTL FORMULÆ WITH BITMAP MANIPULATIONS

This chapter represents a modified version of a paper which is written by K. Xie and S. Hallé and still under review for publication in the proceedings of the International Conference : Runtime Verification 2016 (RV'16) in Madrid, Spain in 2016.

4.1 INTRODUCTION

Temporal logic Huth et Ryan (2004) is a logistic system which uses rules and symbols to describe and reason about the change of a system's state in terms of time. It is based on the idea that one state may not be constantly true or false as time goes. *Linear Temporal Logic (LTL)* (Pnueli, 1977) is a temporal logic, and as its name entails, *LTL* can denote only one sequence of states and for each state there is only one future state.

A *bitmap*, also known as a bit array or bitset, is a compact data structure storing a sequence of binary values. As will be shown in Section 4.2, it can be used to express a set of numbers, or an array where each bit represents a 2-valued option. Bitmaps present several advantages as a data structure : they can concisely represent information, and provide very efficient functions to manipulate them, taking advantage of the fact that multiple bits can be processed in parallel

in a single CPU instruction.

In this paper, we explore the idea of using bitmap manipulations for the offline evaluation of LTL formulæ on an event log. For this purpose, in Section 4.3, we introduce a solution which, for a given event trace σ and an LTL formula φ , first converts each ground term into as many bitmaps ; intuitively, the bitmap for atomic proposition p describes which events of σ satisfy p . Algorithms are then detailed for each LTL operator, taking bitmaps as their input and returning a bitmap as their output. The recursive application of these algorithms can be used to evaluate any LTL formula.

This solution presents several advantages. First, the use of bitmaps can be seen as a form of *indexing* (in the database sense of the term) of a trace’s content. Rather than being an online algorithm merely reading a pre-recorded trace, our solution exploits the fact that the trace is completely known in advance, and makes extensive use of this index to jump to specific locations in the trace to speed up its process. Second, a bitmap having consecutive 0s or 1s can be compressed, which reduces the space cost and speeds up the execution of many operations even further (Kaser et Lemire, 2014).

To this end, Section 4.4 describes an experimental setup used to test our solution. It reveals that, that, for complex LTL formulæ containing close to 20 temporal operators and connectives, large event traces can be evaluated at a throughput ranging in the tens of millions of events per second. These experiments show that the bitmaps are a compact and fast data structure, and are particularly appropriate for the kind of manipulations required for offline monitoring.

4.2 BITMAPS AND COMPRESSION

A bitmap (or bitset) is a binary array that we can view as an efficient and compact representation of an integer set. Given a bitmap of n bits, the i -th bit is set to 1 if the i -th integer in the range $[0, n - 1]$ exists in the set.

It was recognized early on that bitmaps could provide efficient ways of manipulating these sets, by virtue of their binary representation. For example, union and intersection between sets of integers can be computed using bitwise operations (OR, AND) on their corresponding bitmaps ; in turn, such bitwise operations can be performed very quickly by microprocessors, and even in a single CPU operation for 32 or 64-bit wide chunks, depending on the architecture

Furthermore, a bitmap can be used to map n chunks of data to n bits. If the size of each chunk is greater than 1, the bitmap can greatly reduce the size of the storage. In addition, with its capacity of exploiting bit-level parallelism in hardware, standard operations on bitmaps can be very efficient. Unsurprisingly, bitmaps have been used in a lot of applications where the space or speed requirements are essential, such as information retrieval Chan et Ioannidis (1998), databases Burdick et al. (2001), and data mining Ayres et al. (2002); Uno et al. (2005).

A bitmap with low fraction of bits set to value 1 can be considered *sparse* Kaser et Lemire (2014). Such a sparse bitmap, stored as is, is a waste of both time and especially space. Consequently, many algorithms have been developed to *compress* these bitmaps ; most of them there are based on the Run-Length Encoding (RLE) model derived from the BBC compression scheme Antoshenkov (1995). In the following, we briefly describe a few of these techniques. In particular, we detail the WAH Wu et al. (2006), Concise Colantonio et Di Pietro (2010) and EWAHLemire et al. (2010) algorithms, because they have well-implemented open source libraries in Java that we will evaluate experimentally later in this paper.

4.2.1 WAH

WAH (Wu et al., 2006) divides a bitmap of n bits into $\lceil \frac{n}{w-1} \rceil$ words of $w-1$ bits, where w is a convenient word length (for example, 32). WAH distinguishes between two types of words : words made of just $w-1$ ones ($11 \dots 1$) or just $w-1$ zeros ($00 \dots 0$), are *fill words*, whereas words containing a mix of zeros and ones are *literal words*. Literal words are stored using w bits : the most significant bit is set to zero and the remaining bits store the heterogeneous $w-1$ bits. Sequences of homogeneous fill words (all ones or all zeros) are also stored using w bits : the most significant bit is set to 1, the second most significant bit indicates the bit value of the homogeneous block sequence, while the remaining $w-2$ bits store the run length of the homogeneous block sequence.

4.2.2 CONCISE

Concise (Colantonio et Di Pietro, 2010) is a bitmap compression algorithm based on WAH. Comparing with WAH, for which the run length is $w-2$ bits, Concise uses $w-2-\lceil \log_2 w \rceil$ for the run length and $\lceil \log_2 w \rceil$ bits to store an integer value indicating to flip a bit of a single word of $w-1$ bits. This feature can improve the compression ratio in the worst case.

4.2.3 EWAH

EWAH Lemire et al. (2010) is also a variant of WAH but it does not use its first bit to indicate the type of the word like WAH and Concise. EWAH rather defines a w -bits marker word. The most significant $w/2$ bits of the word are used to store the number of the following fill words (all ones or all zeros) and the rest $w/2$ bits encodes the number of *dirty words*. These words are exactly like the literal words of WAH, but utilize all w bits.

With respect to WAH and Concise, the structure used for EWAH makes it difficult to recognize a single word in the sequence as a marker word or a dirty word without reading the sequence from the beginning. Hence, apart from exceptional situations, a reverse enumeration of the bits in the sequence is nearly impossible.

4.2.4 ROARING

In all the previous models, fast random access to the bits in an arbitrary sequence is relatively difficult. At the very least, the word that contains the bit to read must be identified, and the position of this word in the stream requires a knowledge of how many literal or fill words are present before. Besides the RLE-model algorithms, there exist other bitmap compression models that support fast random access similar to uncompressed bitmaps. One of them is called “Roaring bitmap” Chambi et al. (2015), which we shall briefly describe.

Roaring bitmap has a compact and efficient two-level indexing data structure that splits 32-bit indexes into chunks, each of which stores the 16 most significant bits of a 32-bit integer and points to a specialized container storing the 16 least significant bits. There are two types of containers : a sorted 16-bit integer array for *sparse* chunks, which store at most 4,096 integers, and a bitmap for *dense* chunks that stores 2^{16} integers. This hybrid data structure allows fast random access whereas all RLE-model algorithms mentioned cannot because of the characteristics mentioned earlier.

4.2.5 DISCUSSION

The RLE-model algorithms share some common features and also have their own characteristics. First, all of them have two different kinds of words, one of which is to store the raw

uncompressed word (literal word) and the other is compressed word (sequence word) having a bit and a number. The number represents the number of consecutive words which are full of 0s or 1s determined by the bit.

We use $wlen$ to represent the number of bits in a word, $ulen$ the number of available in a literal word and $wcap$ the maximum number of bits stored in a sequence word. Table 4.1 lists the parameters of the three RLE-model algorithms.

	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: Parameters of RLE-model algorithms

Considering that a n -bits bitmap has m sequences of consecutive (0...1...) bits :

$c_0^1 c_1^0 c_1^1 c_1^0 c_2^1 c_2^0 \dots c_{m-1}^1 c_{m-1}^0, c_j^i$ is the number of consecutive i bits and $i \in (0, 1), 0 \leq j \leq m$.

Then the number of total bits, i.e. the size of the uncompressed bitmap is :

$$total_bits = \sum_{j=0}^{m-1} \sum_{i=0}^1 c_j^i = \sum_{j=0}^{m-1} \sum_{i=0}^1 l_j^i + s_j^i,$$

$$l_j^i = c_j^i \bmod ulen, s_j^i = c_j^i - l_j^i$$

If exists a positive integer $slen$, $\forall c_j^i = slen$, then

$$m = n \div (2 \times slen) \quad (4.1)$$

When $1 \leq slen < wlen$, then $\forall l_j^i > 0, \forall s_j^i = 0$, which is considered the worst case, the size of the compressed bitmap is :

$$compressed_bits = \lceil \frac{total_bits}{ulen} \rceil \times wlen$$

None of the three RLE-model algorithms can well compress this kind of bitmap. Both *WAH* and *Concise* waste one bit for the type identification and *EWAH* seems to cost the least for its $ulen = wlen$ but its actual size should be a little more than $total_bits$ because some empty sequence word is needed to store the number of the literal words.

Furthermore, when $wlen \leq slen$, then $\forall s_j^i > 0$, the sequence can be well compressed with any RLE-model algorithm. Suppose $\forall l_j^i > 0$, the size of the compressed bitmap is :

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

From this discussion, we can see that $slen$, i.e. the number of consecutive 1 or 0 bits in a sequence, is a crucial argument and is able to decide the compression rate of a RLE bitmap compression algorithm. Some optimization like *Concise* is merely to try to improve the performance of the worst case.

4.3 EVALUATING LTL FORMULÆ WITH BITMAP

Since bitmaps have been shown to be very efficient for storing manipulating encoded sets of integers, in this section we describe a technique for evaluating arbitrary formulæ expressed in

$\bar{s} \models p$	\iff	$p \in \pi(0)$
$\bar{s} \models \neg \psi$	\iff	$\bar{s} \not\models \psi$
$\bar{s} \models \psi \wedge \phi$	\iff	$\bar{s} \models \psi$ and $\bar{s} \models \phi$
$p^i \models \psi \vee \phi$	\iff	$\bar{s} \models \psi$ or $\bar{s} \models \phi$
$\bar{s} \models \psi \rightarrow \phi$	\iff	$\bar{s} \models \phi$ whenever $\bar{s} \models \psi$
$\bar{s} \models \mathbf{X} \psi$	\iff	$\pi^1 \models \psi$
$\bar{s} \models \mathbf{G} \psi$	\iff	$\forall j \geq 0, \bar{s}^j \models \psi$
$\bar{s} \models \mathbf{F} \psi$	\iff	$\exists j \geq 0, \bar{s}^j \models \psi$
$\bar{s} \models \psi \mathbf{U} \phi$	\iff	$\exists j \geq i, \pi^j \models \phi$ and $\forall k, i \leq k < j, \pi^k \models \psi$
$\bar{s} \models \psi \mathbf{W} \phi$	\iff	either $\exists j \geq i, \pi^j \models \phi$ and $\forall k, i \leq k < j, \pi^k \models \psi$, or $\forall k \geq i, \pi^k \models \psi$
$\bar{s} \models \psi \mathbf{R} \phi$	\iff	either $\exists j \geq i, \pi^j \models \psi$ and $\forall k, i \leq k \leq j, \pi^k \models \phi$, or $\forall k \geq i, \pi^k \models \phi$

Tableau 4.2: The semantics of LTL. Here \bar{s}^i denotes the subtrace of \bar{s} that starts at event i .

Linear Temporal Logic on a given trace of events through bitmap manipulations.

4.3.1 PRELIMINARIES

We shall first recall some basic background about Linear Temporal Logic (LTL). LTL formulæ are made of a finite set of atomic propositions, constituting the ground terms of any expression. These propositions can be combined using the Boolean connectives \neg , \wedge , \vee , \rightarrow and temporal logic operators \mathbf{F} (eventually), \mathbf{G} (globally), \mathbf{X} (next), and \mathbf{U} (until).

Let $\bar{s} = s_0, s_1, s_2, \dots, s_n$ be a finite sequence of *events*, and let $\pi(i)$ be the set of atomic propositions that are true in s_i . The trace \bar{s} is said to satisfy an LTL formula ϕ if the rules described in Table 4.2 apply recursively. We assume a finite-trace semantics where, if \bar{s} is the empty trace, $\bar{s} \not\models \mathbf{F} \phi$, $\bar{s} \not\models \mathbf{X} \phi$, $\bar{s} \not\models \phi \mathbf{U} \psi$, but $\bar{s} \models \mathbf{F} \phi$.

LTL is one of the notations that is widely used in the context of offline monitoring and

Function	Description
<code>addMany(bitmap, val, len)</code>	adds a <i>len</i> -bits sequence of the same value <i>val</i> to the end of the bitmap whose size then increases by <i>len</i> .
<code>copyTo(bitmapDest, bitmapSrc, start, len)</code>	copies the <i>len</i> -bits sequence from the index <i>start</i> in bitmap <i>bitmapSrc</i> to the end of another bitmap <i>bitmapDest</i> whose size then increases by <i>len</i> .
<code>removeFirstBit(bitmap)</code>	removes the first bit of the bitmap, and the size of the bitmap decreases by 1.
<code>next(b, bitmap, start)</code>	gets the position of the next occurrence of the bit with value <i>b</i> from the inclusive position <i>start</i> of the bitmap, or -1 if there is no more.
<code>last(b, bitmap)</code>	gets the position of last occurrence of the bit with value <i>b</i> in the bitmap, or -1 if the bitmap does not have a bit with value <i>b</i> .

Tableau 4.3: Derivative bitmap functions

runtime verification. Depending on the context, LTL formulæ can represent security policies, constraints on sequences of method calls in an object-oriented program, correct interaction between a user and some interface, etc.

We suppose that a well-designed bitmap data structure implements a number of basic functions. Given bitmaps a , b , we will note $|a|$ the function that computes the length of a . The notation $a \otimes b$ will denote the bitwise logical AND of a and b , $a \oplus b$ the bitwise logical OR, and $!a$ its bitwise inverse.

These bitmap functions would be enough to evaluate the LTL operators, but in order to optimize our solution and integrate more closely with bitmap compression algorithms shown in Section 4.2, we need to manipulate the internal data structure of the bitmap and thus introduce seven derivative bitmap functions see Table 4.3).

4.3.2 MANIPULATING BITMAPS TO IMPLEMENT LTL OPERATORS

We are now ready to define a procedure for evaluating arbitrary LTL formulæ with the help of bitmaps. Given a finite sequence of states $(s_0, s_1, \dots, s_{n-1})$ and an LTL formula φ , the principle is to compute a bitmap $(b_0 b_1 \dots b_i b_{i+1} \dots b_{n-1})$ of length n , noted B_φ , whose content is defined follows :

$$b_i = \begin{cases} 1 & \text{if } \bar{s}^i \models \varphi \\ 0 & \text{otherwise} \end{cases} \quad (4.2)$$

The finite set of atomic propositions constitute the initial bitmaps. These basic bitmaps are created by reading the original trace, and setting bit i of B_p to 1 if the atomic proposition is true at the corresponding state s_i , and otherwise 0. One can see that this construction respects Definition 4.2 in the case of ground terms.

From these initial bitmaps, bitmaps corresponding to increasingly complex formulæ can now be recursively computed. The cases of conjunction, disjunction and negation are easy to deal with, since these connectives have their direct equivalents as bitwise operators. For example, given bitmaps B_φ and B_ψ , the bitmap $B_{\varphi \wedge \psi}$ can be obtained by computing $B_\varphi \otimes B_\psi$. The remaining propositional connectives can be easily reduced to these three through standard identities. Temporal logic operators are a little more complicated because they concern the change of the states in terms of time, potentially requiring to enumerate the actual states and the bits in the bitmaps.

A few of them can still be handled easily. The expression $\mathbf{X} \varphi$ states that φ must hold in the next state of the trace. To compute the bitmap $B_{\mathbf{X} \varphi}$, it suffices to remove the first state of B_φ , shift the remaining bits one position to the left, and fill the last bit with 0. This is illustrated in

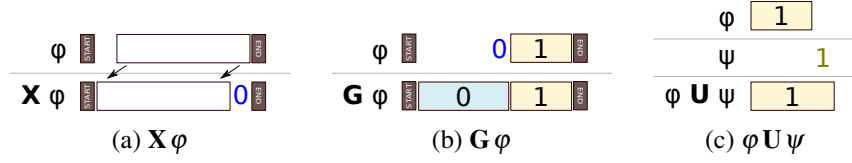


Figure 4.1: A graphical representation of the computation of three temporal operators on bitmaps

Figure 4.1a, and formalized in Algorithm 1.

Algorithm 1 Computing $X a$

Require: Bitmap a

- 1: $out \leftarrow \text{removeFirstBit}(a)$
 - 2: $\text{addMany}(out, 0, 1)$
 - 3: **return** out
-

To compute the vector for $G \psi$, it suffices to find the smallest position i such that all subsequent bits are 1. In $B_{G \psi}$, all bits before i are set to 0, and all bits after (and including) i are set to 1. Thus to implement this operator using bitmaps, we need to do a search in the bitmap B_ψ from back to front to find the last occurrence of the bit 0, as can be seen from Algorithm 2.

Operator **F** (See Algorithm 3) is the dual of **G**; its corresponding algorithm works in the same way as for **G**, swapping 0 and 1.

Algorithm 2 Computing $G a$

Require: Bitmap a

- 1: $p \leftarrow \text{last}(0, a)$
 - 2: **if** $p = -1$ **then**
 - 3: **return** a
 - 4: **else**
 - 5: $out \leftarrow \langle \rangle$
 - 6: $\text{addMany}(out, 0, p + 1)$
 - 7: $\text{addMany}(out, 1, |a| - p - 1)$
 - 8: **return** out
 - 9: **end if**
-

According to Definition (2.12), if there is an index j such that $\bar{s}^j \models \psi$ and \bar{s}^i for all $i < j$, then

Algorithm 3 Future

Require: Bitmap a

```

1:  $pos \leftarrow \text{last}(1, a)$ 
2: if  $pos = -1$  then
3:   return  $a$ 
4: else
5:    $out \leftarrow \text{empty Bitmap}$ 
6:    $\text{addMany}(out, 1, pos + 1)$ 
7:    $\text{addMany}(out, 0, |a| - pos - 1)$ 
8:   return  $out$ 
9: end if

```

$\bar{s} \models \varphi \mathbf{U} \psi$. In terms of bitmap operations, we need to keep checking if there is any bit set as 1 in B_φ before every occurrence of bit 1 in B_ψ (see Algorithm 4).

The operation $\psi \mathbf{W} \varphi$ (Definition (2.13)) is quite like $\psi \mathbf{U} \varphi$, except that as the equation (4.3) (Huth et Ryan, 2004) entails, the operation of the former also includes the operation $\mathbf{G}\psi$. Algorithm 5 explains its operation.

$$\psi \mathbf{W} \varphi \equiv \psi \mathbf{U} \varphi \vee \mathbf{G}\psi \quad (4.3)$$

As the dual of the operator \mathbf{U} , the operator \mathbf{R} defined in (2.14) need to union two parts for the formula $\psi \mathbf{R} \varphi$: the first part aims to find whether exist $i, j (0 \leq i < j)$ which make $\pi^i, \pi^{i+1}, \dots, \pi^j$ satisfy φ when π^j satisfies ψ ; and the second is simply $\mathbf{G}\varphi$. Algorithm 6 describes this procedure.

4.3.3 DISCUSSION

An interesting point of this last algorithm is that the bitmaps a and b are not traversed in a linear fashion. Rather, entire blocks of each bitmap can be skipped to reach directly the next 0

Algorithm 4 Computing $a \cup b$

<p>Require: Bitmaps a and b</p> <p>1: $out \leftarrow \langle \rangle$</p> <p>2: $p, a_0, a_1, b_0, b_1 \leftarrow 0$</p> <p>3: while $p < a$ do</p> <p>4: if $a_1 \leq p$ then</p> <p>5: $a_1 \leftarrow \text{next}(1, a, p)$</p> <p>6: end if</p> <p>7: if $b_1 \leq p$ then</p> <p>8: $b_1 \leftarrow \text{next}(1, b, p)$</p> <p>9: end if</p> <p>10: if $a_1 = -1$ or $b_1 = -1$ then</p> <p>11: break</p> <p>12: end if</p> <p>13: $\text{nearest1} \leftarrow \min(a_1, b_1)$</p> <p>14: if $\text{nearest1} > p$ then</p> <p>15: $\text{addMany}(out, 0, \text{nearest1} - p)$</p> <p>16: $p \leftarrow \text{nearest1}$</p> <p>17: continue</p> <p>18: end if</p> <p>19: if $p = b_1$ then</p> <p>20: if $b_0 \leq b_1$ then</p> <p>21: $b_0 \leftarrow \text{next}(0, b, b_1)$</p> <p>22: if $b_0 = -1$ then</p> <p>23: $b_0 \leftarrow a$</p> <p>24: end if</p>	<p>25: end if</p> <p>26: $\text{addMany}(out, 1, b_0 - p)$</p> <p>27: $p \leftarrow b_0$</p> <p>28: continue</p> <p>29: end if</p> <p>30: if $a_0 \leq a_1$ then</p> <p>31: $a_0 \leftarrow \text{next}(0, a, a_1)$</p> <p>32: if $a_0 = -1$ then</p> <p>33: $a_0 \leftarrow a$</p> <p>34: end if</p> <p>35: end if</p> <p>36: if $a_0 \geq b_1$ then</p> <p>37: $\text{addMany}(out, 1, b_1 - p + 1)$</p> <p>38: $p \leftarrow b_1 + 1$</p> <p>39: else</p> <p>40: $\text{addMany}(out, 0, a_0 - p + 1)$</p> <p>41: $p \leftarrow a_0 + 1$</p> <p>42: end if</p> <p>43: end while</p> <p>44: if $b_1 = -1$ then</p> <p>45: $\text{addMany}(out, 0, a - out)$</p> <p>46: else if $a_1 = -1$ then</p> <p>47: $\text{copyTo}(out, b, p, a - p)$</p> <p>48: end if</p> <p>49: return out</p>
--	---

or the next 1, depending on the case. Note that this is only possible if the trace is completely known in advance before starting to evaluate a formula (and moreover, the trace is traversed backwards). Therefore, our proposed solution is an example of an offline monitor that is not simply an online monitor that is fed events of a pre-recorded trace one by one : it exploits the possibility of *random access* to parts of the trace that is only possible in an offline setting.

This example shows one of the advantages of our proposed technique in terms of complexity. Indeed, reading the original log to create the ground bitmaps can be done in linear time (and

Algorithm 5 Computing $a \mathbf{W} b$

Require: Bitmaps a and b

```

1:  $out \leftarrow \langle \rangle$ 
2:  $p, a_0, a_1, b_0, b_1 \leftarrow 0$ 
3: while  $p < |a|$  do
4:   if  $a_1 \leq p$  then
5:      $a_1 \leftarrow \text{next}(1, a, p)$ 
6:   end if
7:   if  $b_1 \leq p$  then
8:      $b_1 \leftarrow \text{next}(1, b, p)$ 
9:   end if
10:  if  $a_1 = -1$  or  $b_1 = -1$  then
11:    break
12:  end if
13:   $\text{nearest1} \leftarrow \min(a_1, b_1)$ 
14:  if  $\text{nearest1} > p$  then
15:     $\text{addMany}(out, 0, \text{nearest1} - p)$ 
16:     $p \leftarrow \text{nearest1}$ 
17:    continue
18:  end if
19:  if  $p = b_1$  then
20:    if  $b_0 \leq b_1$  then
21:       $b_0 \leftarrow \text{next}(0, b, b_1)$ 
22:      if  $b_0 = -1$  then
23:         $b_0 \leftarrow |a|$ 
24:      end if
25:    end if
26:     $\text{addMany}(out, 1, b_0 - p)$ 
27:     $p \leftarrow b_0$ 
28:    continue
29:  end if
30:  if  $a_0 \leq a_1$  then
31:     $a_0 \leftarrow \text{next}(0, a, a_1)$ 
32:    if  $a_0 = -1$  then
33:       $a_0 \leftarrow |a|$ 
34:    end if
35:  end if
36:  if  $a_0 \geq b_1$  then
37:     $\text{addMany}(out, 1, b_1 - p + 1)$ 
38:     $p \leftarrow b_1 + 1$ 
39:  else
40:     $\text{addMany}(out, 0, a_0 - p + 1)$ 
41:     $p \leftarrow a_0 + 1$ 
42:  end if
43: end while
44: if  $b_1 = -1$  then
45:   if  $a_1 = -1$  then
46:      $\text{addMany}(out, 0, |a| - |out|)$ 
47:   else
48:      $\text{last0} \leftarrow \text{last}(0, a)$ 
49:     if  $\text{last0} = -1$  or  $\text{last0} < p$  then
50:        $\text{addMany}(out, 1, |a| - |out|)$ 
51:     else
52:        $\text{addMany}(out, 0, \text{last0} - p +$ 
53:         1)
54:        $\text{addMany}(out, 1, |a| - |out|)$ 
55:     end if
56:   end if
57: else if  $a_1 = -1$  then
58:    $\text{copyTo}(out, b, |b| - b_1 - 1, |b| - b_1)$ 
59: end if
60: return  $out$ 

```

Algorithm 6 Computing $a \mathbf{R} b$

Require: Bitmaps a and b

```

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

```

in a single pass for all propositional symbols at once). However, once these initial bitmaps are computed, many of the required operations do not require a linear processing of the trace anymore. For example, evaluating $\mathbf{X} \varphi$ requires a simple bit shift, which can be done in a single CPU operation for 64 bits at a time, and potentially much more if compression is used.¹ Similarly, looking for the next 0 or 1 seldom requires linear searching, as the use of compression makes it possible to skip over full words in one operation. Computing the bitmap for a \mathbf{F} or \mathbf{G} operator requires a single such lookup for the entire trace.

Another interesting point is the fact that operators \mathbf{F} and \mathbf{G} are monotonous. As can be seen in Figure 4.1, the resulting bitmap is of the form 0^*1^* (or the reverse). Hence a very simple bitmap is propagated upwards to further algorithms ; it can be heavily compressed, and makes any lookup for the next 0 or the next 1 trivial. While not producing such simple vectors, bitmaps resulting from the application of \mathbf{U} still have a relatively regular structure that is again amenable to reasonable compression.

4.4 IMPLEMENTATION AND EXPERIMENTS

While the worst-case complexity of every algorithm presented in the previous section is still $O(n)$ (where n is the size of the input bitmap), we suspect that performance in practice should be much better. Therefore, in this section, we describe experiments in order to achieve the following purposes :

1. Test the performance of fundamental LTL algorithms
2. Test the performance of the recursive application of these algorithms on complex LTL formulæ

1. The left bit shift of a compressed block is the block itself, as long as the next bit to the right has the same value.

Bitmap	Source
Uncompressed	java.util.BitSet from Java SDK
WAH	Original : https://github.com/metamx/extendedset Modified : https://github.com/phoenixxie/extendedset
Concise	Original : https://github.com/metamx/extendedset Modified : https://github.com/phoenixxie/extendedset
EWAH	Original : https://github.com/lemire/javaewah Modified : https://github.com/phoenixxie/javaewah
Roaring	https://github.com/lemire/RoaringBitmap

Tableau 4.4: Bitmap libraries

3. Evaluate the performance and space savings incurred by the use of compression

4.4.1 EXPERIMENTAL SETUP

As a means to avoid the runtime disk I/O cost we load all relevant files into memory before the calculations. Thus although using bitmap can considerably reduce the requirement of memory, we prepared a workstation with an Intel Xeon E5-2630 v3 Processor and 48 GB of memory.

All codes are implemented in Java which self takes responsibility of the memory management and garbage collection. Concerning the delay caused by garbage collection (GC) and especially Full-GC, we called *System.gc()* before and after every formula calculation to provide a runtime environment that was as “clean” as possible.

Table 4.4 shows the libraries used for different types of bitmap. In order to implement all the LTL operations, we modified the codes of the libraries to add the necessary functions listed in Table 4.3 and to optimize the functions so that the time complexities of the operators become $O(m)$ where m is the number of sequences of consecutive 0/1 bits.

Because of lack of the support of random access for the RLE-model bitmap compression algorithms, we cannot enumerate the bits in the same way as for an uncompressed bitmap.

Therefore we designed an *iterator* data structure to store not only the absolute index of current bit in the uncompressed bitmap but also the relative index in the compressed bitmap. Taking the function **next(1,x)** as an example, if the current relative index is in a sequence word of 0, the search in this word is unnecessary, and we just jump to the next word ; if the index is in a sequence word of 1, we return the current index ; however, if the index is in a literal word, we have to look for the bit 1 in the *ulen*-bits word.

For the experiments, we developed a random data generator. Every time it generates 5×10^7 tuples, and each tuple contains 3 random numbers (a, b, c) related with 3 simple inequalities : $a > 0$, $b > 0$ and $c \leq 0$, which will be labelled as s_0 , s_1 and s_2 , respectively. According to (2.4), the true/false values of these 3 statements consist of the atomic propositions. When a tuple was passed to the 3 statements, we got 3 boolean values each of which was then turned into a 1/0 bit in the bitmap corresponding to one of the 3 statements. When all tuples were processed, we had 3 bitmaps having 50 million bits each.

4.4.2 BASIC LTL OPERATORS

A first experiment consisted of evaluating the performance, in terms of computation time, for evaluating a bit vector on each propositional and temporal operator taken separately.

In the first experiment, we ran 100 passes of a benchmark on the fundamental operators with uncompressed bitmaps. In every pass, the experiment data was regenerated and passed to the relational statements from which the bitmaps were created. Then the formulæ were executed with the bitmaps. In the final step we calculated the average running time of a pass for each LTL operator, and the number of bits processed per second.

Table 4.5 shows that the propositional logic operators were faster than most temporal logic

Formula	Min. time (ms)	Max. time (ms)	Avg. time (ms)	Throughput (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 4.5: Running time for evaluating each LTL operator on a bit vector, without the use of a compression library.

operators. Among the temporal logic operators, the binary operators were slower than the unary ones because the former require more operations than the latter, especially in the situation that many 0s and 1s sequences are mixed in the bitmap. The dual operators **G** and **F** have similar algorithms but **F** surprisingly took three times longer than **G**. This can be explained by the fact that for a fairly-randomized input bitmap, **F** will append more 1s than 0s to its output bitmap, while **G** will append more 0s than 1s. Although the Java `BitSet` implementation supports both to set a bit to 1 and to clear a bit to 0², it actually does nothing when clearing a new bit of which the index is beyond its size, i.e. appending a bit 0. This results in an asymmetrical processing of 0s and 1s in the bitmap.

4.4.3 COMPLEX FORMULÆ

The results from this first experiment suggest that propositional logic operators, temporal logic unary operators and temporal logic binary operators have different magnitudes of processing speed ; therefore we can divide the operators into three groups.

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

At the beginning of this second experiment, we composed various combinations of operators into 14 LTL formulæ with the help of the tool *randltl* from the library *Spot*³; the formulæ are shown in Table 4.6. Then we also ran a 50-pass benchmark on these formulæ with uncompressed bitmaps. In each cycle the data was regenerated and re-executed with the 14 formulæ. We measured the running time of each cycle and calculated the average time cost and the processing speed as before.

As is indicated in Table 4.7, 3 groups of operators have different scales of processing speed. The combinations having temporal logic and binary operators always took more time than others, and formulæ 13 and 14 are the slowest. This result also shows that our solution can handle a fairly large number of bits (events from the trace) per second, ranging from millions to billions.

4.4.4 USE OF BITMAP COMPRESSION

According to the RLE-model algorithms, the compression ratio mostly depends on the length of consecutive 0s or 1s. Hence in this experiment we modified the generator to enable it to repeat the same tuple a specified number of times : 1, 32 and 64. This new mechanism is able to ensure the existence of continuous sequences with a minimum length (*slen*) in the generated bitmaps. Intuitively, when the value of *slen* increases, the number of sequences decreases; therefore the RLE-model algorithms can be expected to have better performance than an uncompressed bitmap.

In the first part of the experiment, we generated the bitmaps with different algorithms and different values of *slen*, and then calculated the compression ratios. The result in Figure 4.2 confirms the hypothesis that when $slen < wlen$ (where *wlen* is the length of a word),

3. <https://spot.lrde.epita.fr/index.html>

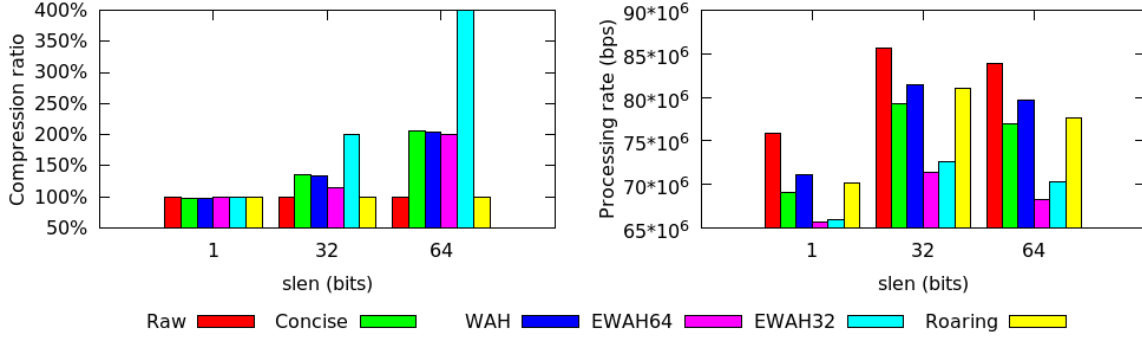


Figure 4.2: Bitmap generation with compression algorithms

the bitmap cannot be well compressed by any RLE-model algorithm, and in this case, the algorithm EWAH behaves a little better than the others due to its smaller structural cost. When $slen$ increases to 32 and 64, i.e. $slen \geq wlen$, the RLE algorithms start to work well and the compression ratio of $slen = 64$ is obviously better than the one of $slen = 32$. From the figure 4.2 we can also see that when $slen$ is 1, 32 and 64, EWAHs are much slower than WAH, Concise and Roaring.

In the second part of the experiment, we measured the performance of the compressed bitmaps when applying the algorithms for all fundamental operators and all LTL formulæ in the previous experiments. Detailed results covering all the operators and formulæ can be found in Appendix A.

To this end, we picked formula F1 and F14 from the previous experiment, as F1 contains all the operators and connectives of LTL and F14 is the slowest of all formulæ in the previous experiment. We again ran the benchmark 100 passes ; in each cycle, the formula was evaluated with one group of input bitmaps from the last step and we recorded the time cost of each bitmap algorithm and each length of consecutive bits.

According to Figure 4.3 and 4.4, the performance of the RLE-model algorithms, WAH, EWAH

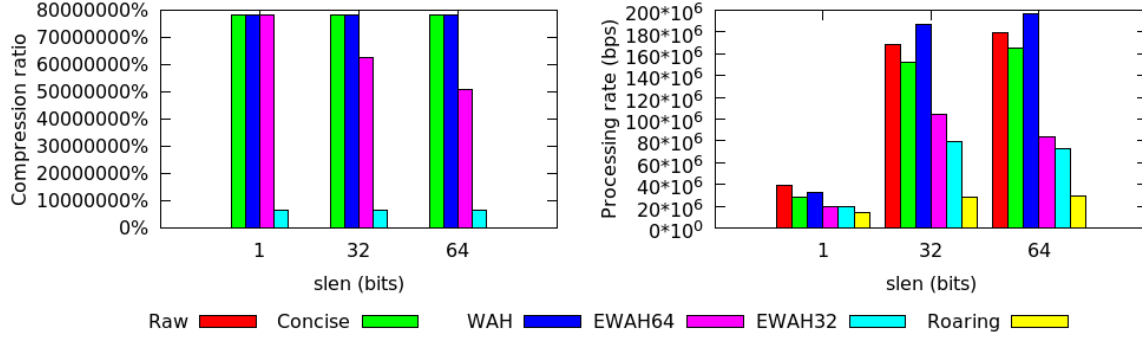


Figure 4.3: Comparison of compression ratio and processing rate for LTL formula 1, for various bitmap compression libraries and various values of $slen$

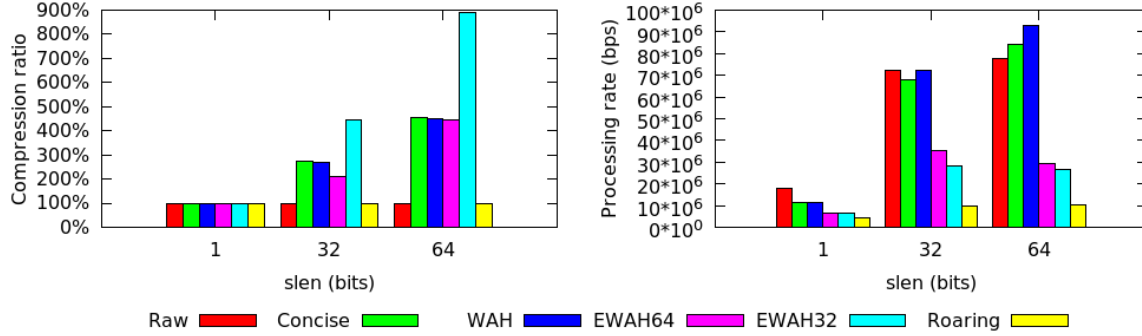


Figure 4.4: Comparison of compression ratio and processing rate for LTL formula 14, for various bitmap compression libraries and various values of $slen$

and Concise is obviously related to the value of $slen$. The Figure 4.3 also suggests that the presence of operators **G** and **F** can vastly increase the length of consecutive bits of same value, which in turn can be well compressed by RLE-model algorithms. In such a case, several algorithms have better performance than the uncompressed bitmap as $slen$ increases.

4.5 RELATED WORK

The prospect of using physical properties of hardware to boost the performance of runtime verification has already been studied in the recent past. For example, Pellizzoni et al. (Pellizzoni

et al., 2008) utilized dedicated commercial-off-the-shelf (COTS) hardware (Emerson, 1990) to facilitate the runtime monitoring of critical embedded systems whose properties were expressed in Past-time Temporal Linear Logic (PTLTL).

As the number of cores (GPU or multi-core CPUs) in the commodity hardware keeps increasing, the research of exploiting the available processors or cores to parallelize the tasks and the computing brings a challenge and also an opportunity to improve the architecture of runtime verification. For example, Ha et al. (Ha et al., 2009) introduced a buffering design of *Cache-friendly Asymmetric Buffering (CAB)* to improve the communications between application and runtime monitor by exploiting the shared cache of the muticore architecture ; Berkovich et al. (Berkovich et al., 2015) proposed a GPU-based solution that effectively utilizes the available cores of the GPU, so that the monitor designed and implemented with their method can run in parallel with the target program and evaluate LTL properties.

Previous work by one of the authors (Hallé et Soucy-Boivin, 2015) introduced an algorithm for the automated verification of Linear Temporal Logic formulæ on event traces, using an increasingly popular cloud computing framework called MapReduce. The algorithm can process multiple, arbitrary fragments of the trace in parallel, and compute its final result through a cycle of runs of MapReduce instances. The proposed technique manipulates objects called *tuples*, which are of the form $\langle \phi, (n, i) \rangle$, and are interpreted as the statement “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”. One can see that this statement corresponds exactly to the fact, in the present solution, that the n -th position of the bitmap generated by the evaluation of ϕ contains the value 1.

Apart from this similarity, however, the two techniques are radically different. Since the MapReduce approach operates on tuples one by one, while the present solution manipulates entire bitmaps, the algorithms for each LTL operator have little in common (especially that

for U). Where the MapReduce approach gets its speed from the processing of multiple subformulae on different machines, our present solution is efficient because some operations (such as conjunction) can be computed simultaneously for many adjacent events in a single CPU cycle. In addition, a downside of the MapReduce solution is the large number of tuples generated, and the impossibility of compressing that volume of data.

As one can see, there have been multiple attempts at leveraging parallelism and properties of hardware to evaluate temporal expressions on traces. However, As far as we know, our work is the first to get its performance boost at the level of the *data structures* used to evaluate these expressions.

4.6 CONCLUSION AND FUTURE WORK

We proposed a solution for the offline evaluation of LTL formulae by means of bitmap manipulations. In such a setting, propositional predicates on individual events of a trace states are mapped to bits of a vector (“bitmap”) that are then manipulated to implement each LTL operator. In addition to the fact that bitmap manipulations are in themselves very efficient, our algorithms take advantage of the fact that the trace is completely known in advance, and that random access to any position of that trace makes it possible to skip large blocks of events to speed up the evaluation.

For this reason, our solution is a prime example of an offline evaluation algorithm that exploits the fact that it indeed works offline—it is not merely an online algorithm that reads events from a pre-recorded trace one by one. As a matter of fact, in some cases (such as the U operator), the trace is even evaluated from the end, rather than from the beginning. A thorough performance benchmark for both fundamental operators and complex LTL formulae proved the feasibility of the approach, and showed how events from a trace can be processed at a rate

ranging from millions to billions of events per second.

To further exploit the potential of bitmaps, we introduced bitmap compression algorithms in our solution and integrated them in our benchmark. In the experiments, as we expected, compressed bitmaps demonstrated their ability to easily compress sparse bitmaps and accelerating the LTL operations when there is a certain amount of consecutive bits with the same value. We have explained, how many LTL operators naturally increase the regularity of the bitmaps they are processing.

Obviously, this solution is suitable only for offline evaluation. However, The promising results obtained in our implementation lead to a number of potential extensions and improvements over the current method. First, the algorithm can be reused as a basis for other temporal languages that intersect with LTL, such as PSL Eisner et Fisman (2006). Second, the technique could be expanded to take into account data parameters and quantification. Finally, one could also consider to parallelize the evaluation of large segments of bitmaps on multiple machines.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Tableau 4.6: The complex LTL formulæ evaluated experimentally.

Formula No.	Prop. Logic Ops.	Temp. Unary Ops.	Temp. Binary Ops.	Min Time (ms)	Max Time (ms)	Avg. Time (ms)	Approx. bits/second
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 4.7: Running time for the evaluation of LTL formulæ of Table 4.6, without the use of a compression library.

CHAPITRE 5

CONCLUSION AND FUTURE WORK

Software verification and validation is a critical part in software engineering and project management. People have learned this from lots of lessons in the past ranging from crashed games to fatal disaster. Comparing with traditional techniques of software verification, runtime verification is relatively new. It has root in other techniques and it has its own feature. In recent years, a great amount of work and time has been invested in various aspects of this area. Some researchers focus on the improvement and applications of various variant of LTL, and some others manage to develop more and more generic frameworks.

Although network has already covered much place on earth, many networking mediums have been exploited and various networking protocols have been proposed, there is still some place where cable and wireless radio have not reached, like deserts or underwater, or some environment where both cable and wireless radio are undesirable or even forbidden, for example, in planes, hospitals, mines or petro-chemical plants. Even under these limitations, software verification is as essential as in networking-covered places. Visible light communication (VLC) is an efficient solution in these situations and has many successful solutions, which enlightened us to think of a method of using optical codes for the data communication work. QR code is encoded optical label which is able to store considerable data and to efficiently encode and

decode data, which gave us the confidence to apply it in our solution.

In the first part of our research, the goal was to design and implement an one-way QR Code communication channel. In the development, we used the computer programming language Java and the well-known QR Code library ZXing. In early experiment, we tested the characteristics of a QR data stream and found that due to the limitation of the hardwares we used, the data loss rate was impossibly zero and it grew fast as the data size of each frame increased. Therefore on one hand we managed to improve the recognition rate of QR codes by adjusting the options of the ZXing library and the camera. On the other hand we proposed the protocol BufferTannen which is in charge of splitting, marshaling, and to some extend compressing the data to transmit. The final result presented in Section 3.4 proves the feasibility of our solution. This part of our research has been published in the journal IEEE Access in 2016.

Every software system, no matter operation systems of mobile phones, web applications, or cloud computing system, needs to guarantee its smooth running and response in time for the exceptions in order to reduce the loss or avoid the disaster. The requirement of software verification for each system could be similar, as is suggested in Section 2.3 which introduced several runtime verification frameworks sharing same functionalities. However, the scale of every software systems varies a lot. For example, a smartphone has much lower memory and slower processor than a mainstream workstation, not to mention a cloud computing system like Amazon EC2. When developing a RV system for a smartphone, the usage of memory and processor is always the main concern. In another way, even in a computer system which has huge memory and power processor, there is always a limit to the memory and the processor. Therefore, there are many researches on the improvement of RV systems. Some tried to distribute the computing to a cluster of servers, some managed to optimize the algorithms or find a better solution.

Our second objective was to find a way to improve the speed of LTL formulæ's calculation. Bitmap is a compact and efficient data structure which has a lot of applications and solutions. A verdict issued by offline runtime verification monitor is two-valued truth value, which can be easily mapped into a bit in a bitmap. We also designed a few algorithms to implement the LTL operators with mapped bitmaps. Kaser et Lemire (2014) suggests that a sparse bitmap can be well compressed, and as we discussed in Section 4.3, the output bitmap from the algorithms has longer consecutive 1/0 sequence than the input. Therefore we integrated bitmap compression algorithms into our solution. The experiment results demonstrate the performance and the feasibility of our solution, and prove that the usage of bitmap compression can make our solution faster and more space-efficient with certain algorithms. This part of our research is also written in a paper which is still under review for publication in the proceedings of the International Conference : Runtime Verification 2016 (RV'16) in Madrid, Spain in 2016.

FUTURE WORK

Our QR code communication channel supports only one-way data transfer, resulting that even our program resends the same data frame for several times, we still cannot ensure that the receiver has got all the data. A bidirectional communication seems an answer to this problem, and it can also allow to resend data on demand and thus increase the effective bandwidth.

A bitmap has bits of either 1 or 0, so the *inconclusive* value of LTL_3 cannot be easily implemented. As a result, our solution works only in the offline monitoring mode. To support the online mode is a big but interesting challenge. In addition, making our solution parallelized to be able to run in a computing cloud is also very intriguing.

Appendices

CHAPITRE A

EXPERIMENT RESULTS OF CALCULATING LTL FORMULÆS WITH BITMAP COMPRESSION

In this appendix, we present the experiment results of the third part of Section 4.4. To get better formatting, the data format is different than it in Section 4.4. “Ratio” here has not percentage mark because it is compression rate calculated by the division of original data size and compressed size.

	<i>slen</i>	1			32			64		
Type		Min	Max	Avg.	Min	Max	Avg.	Min	Max	Avg.
Raw	Ratio	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	Ratio	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	Ratio	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	Ratio	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	Ratio	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	Ratio	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: Bitmap generation with compression algorithms

	<i>slen</i>	1			32			64		
Type		Min	Max	Avg.	Min	Max	Avg.	Min	Max	Avg.
Raw	Ratio	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	Ratio	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	Ratio	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	Ratio	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	Ratio	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	Ratio	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: Calculation of $\neg s_0$ with compression algorithms

	<i>slen</i>	1			32			64		
Type		Min	Max	Avg.	Min	Max	Avg.	Min	Max	Avg.
Raw	Ratio	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	Ratio	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	Ratio	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	Ratio	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	Ratio	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	Ratio	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: Calculation of $s_0 \wedge s_1$ with compression algorithms

	<i>slen</i>	1			32			64		
Type		Min	Max	Avg.	Min	Max	Avg.	Min	Max	Avg.
Raw	Ratio	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	Ratio	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	Ratio	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	Ratio	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	Ratio	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	Ratio	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: Calculation of $s_0 \vee s_1$ with compression algorithms

	<i>slen</i>	1			32			64		
Type		Min	Max	Avg.	Min	Max	Avg.	Min	Max	Avg.
Raw	Ratio	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	Ratio	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	Ratio	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	Ratio	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	Ratio	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	Ratio	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: Calculation of $s_0 \vee s_1$ with compression algorithms

	<i>slen</i>	1			32			64		
Type		Min	Max	Avg.	Min	Max	Avg.	Min	Max	Avg.
Raw	Ratio	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	Ratio	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	Ratio	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	Ratio	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	Ratio	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	Ratio	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: Calculation of X_{s_0} with compression algorithms

	<i>slen</i>	1			32			64		
Type		Min	Max	Avg.	Min	Max	Avg.	Min	Max	Avg.
Raw	Ratio	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	Ratio	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	Ratio	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	Ratio	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	Ratio	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	Ratio	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: Calculation of G_{s_0} with compression algorithms

	<i>slen</i>	1			32			64		
Type		Min	Max	Avg.	Min	Max	Avg.	Min	Max	Avg.
Raw	Ratio	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	Ratio	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	Ratio	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	Ratio	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	Ratio	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	Ratio	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: Calculation of F_{s_0} with compression algorithms

	<i>slen</i>	1			32			64		
Type		Min	Max	Avg.	Min	Max	Avg.	Min	Max	Avg.
Raw	Ratio	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	Ratio	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	Ratio	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	Ratio	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	Ratio	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	Ratio	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: Calculation of $s_0U_{s_1}$ with compression algorithms

	<i>slen</i>	1			32			64		
Type		Min	Max	Avg.	Min	Max	Avg.	Min	Max	Avg.
Raw	Ratio	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	Ratio	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	Ratio	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	Ratio	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	Ratio	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	Ratio	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: Calculation of $s_0W_{s_1}$ with compression algorithms

	<i>slen</i>	1			32			64		
Type		Min	Max	Avg.	Min	Max	Avg.	Min	Max	Avg.
Raw	Ratio	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	Ratio	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	Ratio	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	Ratio	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	Ratio	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	Ratio	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: Calculation of $s_0R_{s_1}$ with compression algorithms

	<i>slen</i>	1			32			64		
Type		Min	Max	Avg.	Min	Max	Avg.	Min	Max	Avg.
Raw	Ratio	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	Ratio	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	Ratio	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	Ratio	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	Ratio	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	Ratio	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: Formulæ 1 calculation with compression algorithms

	<i>slen</i>	1			32			64		
Type		Min	Max	Avg.	Min	Max	Avg.	Min	Max	Avg.
Raw	Ratio	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	Ratio	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	Ratio	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	Ratio	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	Ratio	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	Ratio	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: Formulæ 2 calculation with compression algorithms

	<i>slen</i>	1			32			64		
Type		Min	Max	Avg.	Min	Max	Avg.	Min	Max	Avg.
Raw	Ratio	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	Ratio	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	Ratio	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	Ratio	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	Ratio	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	Ratio	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: Formulæ 3 calculation with compression algorithms

	<i>slen</i>	1			32			64		
Type		Min	Max	Avg.	Min	Max	Avg.	Min	Max	Avg.
Raw	Ratio	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	Ratio	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	Ratio	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	Ratio	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	Ratio	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	Ratio	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: Formulæ 4 calculation with compression algorithms

	<i>slen</i>	1			32			64		
Type		Min	Max	Avg.	Min	Max	Avg.	Min	Max	Avg.
Raw	Ratio	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	Ratio	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	Ratio	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	Ratio	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	Ratio	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	Ratio	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: Formulæ 5 calculation with compression algorithms

	<i>slen</i>	1			32			64		
Type		Min	Max	Avg.	Min	Max	Avg.	Min	Max	Avg.
Raw	Ratio	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	Ratio	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	Ratio	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	Ratio	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	Ratio	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	Ratio	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: Formulæ 6 calculation with compression algorithms

	<i>slen</i>	1			32			64		
Type		Min	Max	Avg.	Min	Max	Avg.	Min	Max	Avg.
Raw	Ratio	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	Ratio	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	Ratio	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	Ratio	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	Ratio	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	Ratio	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: Formulæ 7 calculation with compression algorithms

	<i>slen</i>	1			32			64		
Type		Min	Max	Avg.	Min	Max	Avg.	Min	Max	Avg.
Raw	Ratio	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	Ratio	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	Ratio	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	Ratio	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	Ratio	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	Ratio	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: Formulæ 8 calculation with compression algorithms

	<i>slen</i>	1			32			64		
Type		Min	Max	Avg.	Min	Max	Avg.	Min	Max	Avg.
Raw	Ratio	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	Ratio	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	Ratio	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	Ratio	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	Ratio	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	Ratio	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: Formulæ 9 calculation with compression algorithms

	<i>slen</i>	1			32			64		
Type		Min	Max	Avg.	Min	Max	Avg.	Min	Max	Avg.
Raw	Ratio	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	Ratio	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	Ratio	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	Ratio	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	Ratio	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	Ratio	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: Formulæ 10 calculation with compression algorithms

	<i>slen</i>	1			32			64		
Type		Min	Max	Avg.	Min	Max	Avg.	Min	Max	Avg.
Raw	Ratio	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	Ratio	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	Ratio	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	Ratio	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	Ratio	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	Ratio	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: Formulæ 11 calculation with compression algorithms

	<i>slen</i>	1			32			64		
Type		Min	Max	Avg.	Min	Max	Avg.	Min	Max	Avg.
Raw	Ratio	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	Ratio	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	Ratio	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	Ratio	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	Ratio	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	Ratio	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: Formulæ 12 calculation with compression algorithms

	<i>slen</i>	1			32			64		
Type		Min	Max	Avg.	Min	Max	Avg.	Min	Max	Avg.
Raw	Ratio	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	Ratio	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	Ratio	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	Ratio	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	Ratio	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	Ratio	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: Formulæ 13 calculation with compression algorithms

	<i>slen</i>	1			32			64		
Type		Min	Max	Avg.	Min	Max	Avg.	Min	Max	Avg.
Raw	Ratio	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	Ratio	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	Ratio	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	Ratio	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	Ratio	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	Ratio	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: Formulæ 14 calculation with compression algorithms

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 Semantics (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.