UQAC
Université du Québec
à Chicoutimi

MÉMOIRE

PRÉSENTÉ À

L'UNIVERSITÉ DU QUÉBEC À CHICOUTIMI

COMME EXIGENCE PARTIELLE

DE LA MAÎTRISE EN INFORMATIQUE

PAR

KUN XIE

IMPROVEMENTS TO DATA COLLECTION AND FORMULÆEVALUATION OF RUNTIME VERIFICATION

MAY 2016

Abstract

Information technology has become an important part of our lives. Although lots of great techniques make us live easy and comfortable, accidents and disasters caused by software malfunctioning cost much losses of lives and wealth which actually can be avoided. Software verification and validation is a set of techniques aiming to verify the functionality and evaluate the software quality. Runtime verification is one of the techniques widely used in the industry. It has its origin from other verification techniques but it also has its own features and characteristics.

The goal of this research is to explore methods and solutions to improve two aspects of runtime verification: data collection and formulæ evaluation. In the first part, we present an one-way communication channel based on optical codes which is applicable for data transmission in some specific environment. Then in the second part, we introduce our solution of offline evaluation of temporal logics based on bitmap manipulation and bitmap compression. Both parts were written in papers for publication, one of which has been published, while another one is under review.

Keywords: runtime verification, linear temporal logic, bitmap compression, optical communication protocols, QR code

Acknowledgements

I would like to thank everyone who helped me complete my degree.

First I would like to express my sincere appreciation to my director **Professor Sylvain Hallé** for his supervision, support, patience and generosity. I was very lucky to have a supervisor who cares so much about students and who is always patient for every kind of questions and problems. Before I entered the department, I had never imagined that as a foreign MSc. student, I could have the chance of receiving scholarships and taking part in the work of paper writing for real journals and conferences. It was **Professor Sylvain Hallé** who granted me the opportunities and confidence. Thanks to him, I have a fantastic memory in UQAC.

I would like to express my gratitude to **Professor Sylvain Boivin** for his guidance, support and help. He motivated me to return to university for a Master degree, encouraged me for my French learning and introduced me to **Professor Sylvain Hallé**.

I would like also to thank **Mme. Marjolaine Hénault** for her kindness, generosity, encouragement and great help for my French learning, friends living in Chicoutimi: **Ran Wei, Ping Li, Jian Qin et al.** for their friendship, encouragement, support and help, all staff of **l'École de langue française** for teaching me French, friends and students working in LIF: **Edmond la Chance, Francis Guérin, Daehli Nadeau-Otis et al.** for their friendship and support.

At last, I would like to express my warmest thanks to my wife **Moon Ji Hyun**, whose companionship, selfless dedication, encouraging words and best cooking technique on earth were the essential motivation of the completion of my Master.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

In recent decades, a huge number of hardware and software systems have been introduced in almost every area of our life (Clarke et al., 1999). While people enjoy the facilities brought by these systems, there is always the risk of failure in the systems. Some failure like some buggy computer game is annoying but tolerable, but some failure is fatal and unacceptable, e.g. the medical instruments, automated vehicle control system and aerospace system. One recent example is the Japanese astronomical satellite Hitomi which cost the Japan Aerospace Exploration Agency (JAXA) 286 million dollars. It was launched on February 17, 2016 and officially declared lost on April 28 of the same year because of a software error (Witze, 2016).

1.1 CONTEXT

Obviously, the dependability of a system is critical, and a dependable system should have the ability to function strictly according to its specification in the stated period (Avižienis et al., 2004). Software verification and validation is the process of measuring this ability and evaluating the software quality (iee, 2012).

Runtime verification (Leucker et Schallhart, 2009) is an approach of software verification

and validation which is applicable for checking whether the behavior of a computing system satisfies or violates certain property. Normally runtime verification does not influence the execution of the running system, even if it detects a violation of the properties. For this purpose, a monitor is employed to analyze the collected finite trace and then produce a verdict which is generally a truth value. Therefore a runtime verification system should have at least two crucial components: data collection and formulæ evaluation.

Nowadays in order to meet the demand of analyzing rapidly growing amount of the trace data, a variety of solutions have been proposed. Some researchers like Barre et al. (2012) ported existing methodologies to distributed computing frameworks. However, no matter how many processors and memory a cloud system has, there is always a limit of their usages. Thus some other researchers managed to optimize the algorithms, such as Havelund et Roşu (2001).

Bitmap is an efficient method to reduce the space overhead of data thanks to the concise structure, and the bit-level parallel and cache friendly feature is able to boost the performance of operations (Culpepper et Moffat, 2010). It is widely applied in the applications which have serious demand of space and efficiency, e.g. database and search engines (Kaser et Lemire, 2014). If a bitmap is sparse, i.e. the fraction of used bits is low, the bitmap may use less storage space with the aid of the bitmap compression algorithms (Antoshenkov, 1995).

Before monitors analyzing the traces, data gathering plays an important role (Casley et Kumar, 1988). For different systems, there are corresponding solutions of data collection. Zwijze-Koning et De Jong (2005) reviewed the data collection techniques for network analysis. Calabrese et al. (2011) presented a real-time monitoring system with high-resolution and high-definition data collection of the cellphone usage of an Italian city. Shabtai et Elovici (2010) developed a host-based intrusion detection system for Android mobile devices by gathering the data of system events and user interaction. As is indicated in the examples,

various mediums are utilized to retrieve and transfer data to the location where the verification takes place. Visible light is also an efficient communication medium, as is suggested in Komine et Nakagawa (2004), especially in some restricted environment where cable or radio communication are inconvenient, like Vasilescu et al. (2005).

Various barcodes have been applied in diverse areas from traditional e-commerce systems to rapidly increasing mobile devices (Gao et al., 2007), for the digital barcodes provide a simple but accurate method with low cost of distribution and recognition. Compared with the well-known 1-D UPC barcode which can only encode numbers, 2-D barcodes which appeared at the end of 1980s are able to not only encode alphanumeric data and even binary data, but also supply much larger data capacity. Quick Response Code (QR code) (Denso Wave Inc., 2015) has become one of the most popular 2-D barcodes owing to its accuracy, considerable capacity, relatively small printout and high efficiency. It has been put on nearly every kind of visible surface, like paper, phone and computer screen, store windows (Okazaki et al., 2012).

1.2 THESIS OBJECTIVES

The objectives of this research center around the development of methods or techniques which is able to give assistance to the two mentioned aspects of runtime verification: data collection and formulæ evaluation.

The first principle objective and contribution was to present a new method of data collection and discuss about its feasibility and performance. QR Code is considered fast and large-capacity, and more important is that its utilization needs only a surface (e.g. screen) as the transmitter and a camera as the receiver, both of which have become mainstream configuration of nearly every laptop and mobile phone in recent years. If a QR code containing certain amount of data is considered as a network data packet, a sequence of QR codes is like a network

data stream. Our main concern here was the bandwidth of the one-way communication channel consisting of QR Codes and the critical factors which affect the performance, and also the method of applying this communication channel to our runtime verification practice.

The second objective was to propose a solution of improving the performance of runtime verification system. Bitmap has been proved by many solutions for its ability to improve the performance, and for the temporal logic states are often expressed with boolean values, i.e. true or false, Bitmap was anticipated to enhance the calculation of LTL formulæs. Therefore, one of our contributions was the solution of mapping temporal logic states to bits and design necessary algorithms to implement the operations of LTL. As Kaser et Lemire (2014) suggests, a sparse bitmap is a waste of space. An additional contribution was thus the observation of the impact of the bitmap compression algorithms on the calculation of LTL formulæs.

It is important to mention that our work and achievement of the QR Code communication channel has been published in the journal IEEE Access, vol. 4, pp. 284-298, 2016. Another part of our research, LTL Formulæ with Bitmap Manipulations, is under review for publication in the proceedings of the International Conference: Runtime Verification 2016 (RV'16) in Madrid, Spain in 2016.

1.3 METHODOLOGY

This research followed a three-steps methodology.

The first step was to develop the one-way QR-Codes communication channel which corresponds to our second principle objective. The data packet was encoded to and decoded from QR codes with an open source library, and a specific protocol dedicated to the serialization and transmission of structured data over limited communication channels was introduced. As

the experiment was running, we kept optimizing our solution based on the early experiment result to improve the correction rate and the recognition speed. To take well knowledge of the performance with general devices, a common webcam and a 19-inch LED monitor were used as the receiver and the transmitter in the experiment. In the last part of this step, QR codes were printed on the office papers and swiped before the webcam in order to verify a claim that the protocol can accept incoming data packet without order.

The second step had for goal to define the mapping relationship between temporal logic states and bitmap and also to design the algorithms of the temporal logic operators. The temporal sequence of states of an atomic proposition can be mapped into a bitmap where the value of each bit is either 0 or 1, which rightly corresponds to the boolean-type value of temporal logic states. We categorized the usual LTL operators defined in Huth et Ryan (2004) into three groups: propositional logic operators, unary temporal operators and binary temporal operators. Each group had its feature and difficult, especially the binary temporal operators which have to enumerate two bitmaps together and take care of more conditions than other groups.

In the last step, we implemented our solution with a popular computer programming language. An interface of the bitmap operation was abstracted in order to adapt with the bitmap compression algorithms all of which are implemented in open source libraries. After the programming job, a throughout benchmark was made to observe the process speed without compression and also the performance of both speed and space in condition with the compress algorithms.

1.4 THESIS ORGANIZATION

This thesis consists of five chapters. The content of each chapter is as follows:

Chapter one is an introduction to the background of the thesis, presents the tasks, describes

the methods applied in the research and in the end states the thesis' structure.

Chapter two briefs the relevant knowledge of runtime verification, linear temporal logics and introduces some runtime verification systems.

Chapter three is one of the contributions from this research. It presents the solution of one-way communication channel consisting of flickering QR Codes. It is actually a reformatted version of the publication "Real-Time Streaming Communication with Optical Codes" (Xie et al., 2016).

Chapter four is another contribution, the solution of calculating LTL formulæs with the assistance of bitmaps. The chapter details the definition of mapping, the algorithms and the experiment. It is based on the paper "Offline Evaluation of LTL Formulæ with Bitmap Manipulations".

Finally, chapter five concludes this research with the summary of our contributions and looks forward to our future work.

CHAPTER 2

REVIEW OF RUNTIME VERIFICATION AND RELATED WORK

In this chapter, we firstly review the common information of runtime verification, including the history, the definition and the comparison with other verification techniques. Secondly, linear temporal logic, as the common specification formalism of runtime verification, is presented with the syntax, the operators and the semantics. At last, a few well-known runtime verification frameworks are introduced, as well as a simple comparison of them.

2.1 RUNTIME VERIFICATION

2.1.1 CONCEPTION

Software verification and validation, as an important aspect of project management and software engineering, is the process of employing various necessary techniques to detect the violations or satisfactions and to evaluate the software quality and the performance of a system (iee, 2012).

There are normally two kinds of verification techniques: static and dynamic analysis. Some well-knowns traditional techniques of static analysis are model checking (Clarke et al., 1999)

and theorem proving (Heisel et al., 1990). Static analysis is usually applied to verify the behaviors of a system before its execution, but such techniques have natural shortcomings. For example, model checking cannot work on a system of which the size or the number of states might grow beyond the capacity of computational power owing to the "state-explosion problem".

Dynamic analysis is to monitor running systems. Although sometimes its result might be false negatives because of its incompleteness, this incompleteness enables the techniques of dynamic analysis to break the limitation of static methods and thus become their complementary verification methods. (Falcone et al., 2013)

Runtime verification (RV) is a kind of verification technique based on dynamic analysis. In 2001, the Runtime verification workshop [1] was founded, as the terminology "runtime verification" was officially introduced (Wikipedia, 2016). It is a relatively new technique which is lightweight and aims to complement other traditional verification techniques like model checking and testing.

Leucker et Schallhart (2009) defines runtime verification as follows:

> 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 correctness property.

Normally when a violation is observed, runtime verification does not fix the detected program itself, but its result is an important guide and basis for other component in the same system to deal with the problem.

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

Leucker et Schallhart (2009) also defines a *run* of a system as a sequence of infinite traces of the system, and an *execution* of a system as a finite trace and also a *finite prefix* of a *run*. The work of runtime verification mainly focuses on the analysis of *executions* which are performed by *monitors*.

A *monitor* is a decision procedure generated (or "synthesized") from one of the formal properties which is to be verified against the execution of the given system. During verification, a *monitor* enumerates the finite traces of an *execution*, checks whether they satisfy the correctness properties and produces a *verdict* as the result. A *verdict*, which is normally a truth value, indicates the satisfaction of the property against the gathered events.

A verdict in most simple cases normally can be expressed as true/value, yes/no or 1/0, depending on the context. But actually many runtime verification systems have to introduce other values to provide a more accurate result. For example, thanks to the incompleteness of runtime verification system, a verdict cannot be easily issued when the monitor needs more successive events, so a *inconclusive* (written as "?") value is introduced to indicate the current status of the monitored system. (Falcone et al., 2013)

2.1.2 PROCEDURE

Figure 2.1 describes the process of a typical runtime verification system which contains the following four steps (Falcone et al., 2013):

1. *Monitor synthesis*: A monitor is synthesized from a property.
2. *System instrumentation*: Extra instruments are integrated with the system under scrutiny in order to generate the events for the *monitor*.
3. *Execution*: The system is executed and starts to generate events and send them to the

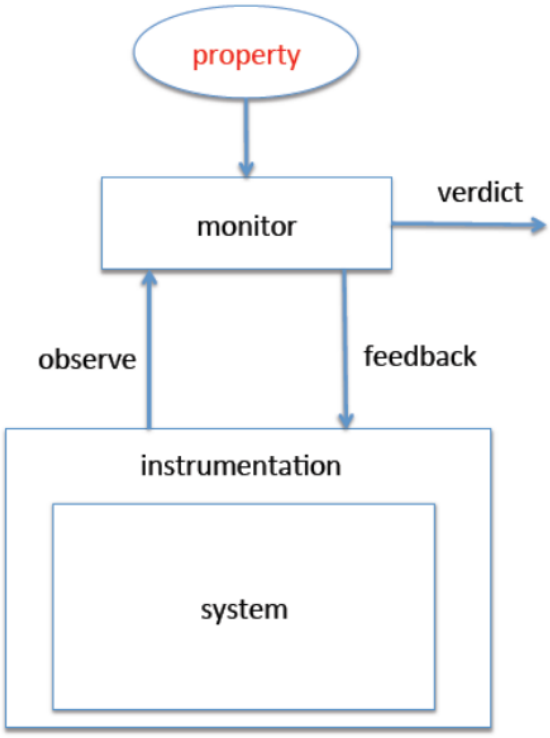

Figure 2.1: Runtime verification process (from Falcone et al. (2013))

monitor.

4. *Analysis and response*: The monitor analyzes the collected events, emits a *verdict* and sends additional information, i.e. *feedback* to the system if necessary.

Monitors can be classified into several modes from different aspects (Chen et Roşu, 2007):

- *online/offline* depends on when the monitors and the system work. *Online* if they work at the same time, and *offline* if the monitors start to work after the termination of the execution of the system.
- *inline/outline* depends on where the monitors are executed. *inline* if the monitors are embedded into the system and *outline* if the monitors run all alone while receiving the event traces from the system by certain methods, e.g. via file system or wireless signal.

- *violation/validation* is determined by the specification of the verdict.

From the definitions of the modes, we can see that a monitor working in offline mode is also working in outline mode and the inline mode implies the online mode.

2.1.3 COMPARISON WITH OTHER TECHNIQUES

Comparing with model checking (Clarke et al., 1999) which aims to verify finite state systems, the methods of generating monitors in runtime verification and generating automatas in model checking have similarities. However, whereas model checking deals mainly with infinite traces, runtime verification deals only with finite traces, i.e. the executions. For this reason, the monitors of runtime verification working in online monitoring mode have to be able to accept incremental traces.

Another important difference between model checking and runtime verification is that, unlike model checking which checks if all executions of a system satisfy a correctness property, runtime verification is interested only with whether existing an execution which belongs to a set of valid executions. Moreover, runtime verification only requires to analyze the observed events of a given system without having to watch its internal running information, but in model checking, the proper model of the system must be acknowledged in order to prepare every possible execution before running the system. (Leucker et Schallhart, 2009)

Software testing (Broy et al., 2005) is another verification technique. It is a process of running a program with a finite set of input-output sequences which is also namely test suite. Comparing with runtime verification, test suites are defined directly, unlike properties of runtime verification which are generated from formalism specification. Besides, “exhaustive testing” which is a common method in software testing, is normally not an option of runtime

verification.

2.2 LINEAR TEMPORAL LOGICS

In runtime verification, a monitor is translated from a correctness property, and correctness properties are specified in linear-time temporal logics, such as LTL.

Temporal logic is an extension of classical logic, and it provides a convenient language with the expressions of the properties to reason about the change of the states in terms of time. Although there are a lot of different temporal logics invented to meet various requirements, the temporal logics are normally classified by whether the time is linear or branching. The temporal logic with linear time is called *Linear Temporal Logic* (LTL) which was first proposed by Pnueli (1977). Time in LTL is turned into a sequence of states which extend to the infinite future. The sequence of states is a computation *path*. (Clarke et al., 1999) (Huth et Ryan, 2004)

Leucker et Schallhart (2009) summarizes LTL as a well-accepted linear-time temporal logic used to specify properties of infinite traces. However, in runtime verification, LTL is employed to check finite executions.

2.2.1 SYNTAX

A well-formed *LTL* formula consists of a finite set of atomic propositions, boolean operators $\neg, \wedge, \vee, \rightarrow$ and temporal logic operators **F**(future), **G**(global), **X**(next), **U**(until), **W**(weak-until) and **R**(release). It can be represented in the Backus Naur form as follows (Huth et Ryan,

2004):

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

2.2.2 SEMANTICS

For a sequence of states $s_0, s_1, s_2, ..., s_i, s_{i+1}, ...$ where s_{i+1} is a future state of s_i, we define a path with $\pi^i = s_i \rightarrow s_{i+1} \rightarrow ...$ where i is the first state in this path. Given that $\pi(i)$ is the set of atomic propositions which are true at the ith state, whether a path π^i satisfies an *LTL* formula is defined as follows (Rozier, 2011):

- $\pi^i \vDash \top$ (2.2)
- $\pi^i \nvDash \bot$ (2.3)
- $\pi^i \vDash p \iff p \in \pi(i)$ (2.4)
- $\pi^i \vDash \neg\psi \iff \pi^i \nvDash \psi$ (2.5)
- $\pi^i \vDash \psi \wedge \varphi \iff \pi^i \vDash \psi$ and $\pi^i \vDash \varphi$ (2.6)
- $\pi^i \vDash \psi \vee \varphi \iff \pi^i \vDash \psi$ or $\pi^i \vDash \varphi$ (2.7)
- $\pi^i \vDash \psi \rightarrow \varphi \iff \pi^i \vDash \varphi$ whenever $\pi^i \vDash \psi$ (2.8)
- $\pi^i \vDash \mathbf{X}\ \psi \iff \pi^{i+1} \vDash \psi$ (2.9)
- $\pi^i \vDash \mathbf{G}\ \psi \iff \forall j \geq i, \pi^j \vDash \psi$ (2.10)
- $\pi^i \vDash \mathbf{F}\ \psi \iff \exists j \geq i, \pi^j \vDash \psi$ (2.11)
- $\pi^i \vDash \psi\ \mathbf{U}\ \varphi \iff \exists j \geq i, \pi^j \vDash \varphi$ and $\forall k, i \leq k < j, \pi^k \vDash \psi$ (2.12)

- $\pi^i \vDash \psi \ \mathbf{W} \ \varphi \iff$ either $\exists j \geq i, \pi^j \vDash \varphi$ and $\forall k, i \leq k < j, \pi^k \vDash \psi$, or $\forall k \geq i, \pi^k \vDash \psi$ (2.13)
- $\pi^i \vDash \psi \ \mathbf{R} \ \varphi \iff$ either $\exists j \geq i, \pi^j \vDash \psi$ and $\forall k, i \leq k \leq j, \pi^k \vDash \varphi$, or $\forall k \geq i, \pi^k \vDash \varphi$ (2.14)

Formulæs 2.2 and 2.3 suggest that the states in the path π^i should be always true or false.

In formulæ 2.4, p is an atomic proposition belonging to the finite set of atomic propositions of LTL, and this formulæ demands to check only the ith state.

Formulæs 2.5—2.8 are boolean operators of propositional logic following the rules of Table 2.1

ψ	φ	$\neg\psi$	$\psi \wedge \varphi$	$\psi \vee \varphi$	$\psi \rightarrow \varphi$
True	True	False	True	True	True
True	False	False	False	True	False
False	True	True	False	True	True
False	False	True	False	True	True

Table 2.1: The truth table of boolean operators of propositional logic

Formulæs 2.9, 2.11 and 2.10 are unary temporal logic connectives. **X** means "next time" and it skips the ith state and evaluates the path π^{i+1}. **F** stands for "sometimes in the future" which requires that from the ith state, a property holds in a future state on the path. And **G** ("globally" or "always") denotes that a property should hold on the every state from the ith state until the end or the infinite future.

Formulæs 2.12, 2.13 and 2.14 are binary temporal logic operators. **U** is the abbreviation of "until". The formulæ $\psi \ \mathbf{U} \ \varphi$ holds if φ holds at a future state on the path, and before this state the property ψ holds at every state. **W**, "weak-until", is a weak version of **U**, except that for the formulæ $\psi \ \mathbf{W} \ \varphi$, φ does not have to hold eventually in some future state. **R**, which

stands for “release”, is actually a logic dual of $\mathbf{U}$, i.e. $\psi\ \mathbf{U}\ \varphi \equiv \neg(\neg\psi\ \mathbf{R}\ \neg\varphi)$. $\mathbf{R}$requires that for the formulæ $\psi\ \mathbf{R}\ \varphi$, the property φ should hold continuously until *psi* holds or *psi* does not hold eventually.

It is worth noting that in LTL, the two-valued logics might yield a premature result which is either true or false. As is mentioned above, LTL is defined to work with infinite traces whereas monitoring of runtime verification is only able to treat finite traces, which might lead to a conflict, especially in some running system. For instance, $\mathbf{F}\ \psi$ states that ψ should hold in a future state. In a running system, as long as ψ does not hold in the observed states, the results of the formulæ are always $false$, but if ψ holds in the next observation, the former results will become corrupted and obsolete. Therefore, Bauer et al. (2006) proposed the three-valued logic LTL_3 which introduces an new value *inconclusive* for the cases where the property cannot be evaluated immediately.

2.2.3 VARIOUS LTLS

Metric Temporal Logic

Metric Temporal Logic (MTL) (Chang et al., 1994) was invented to reason about real-time properties. To precise time accurately, MTL cuts the time into numbered pieces which is also called timed transition modules, and employs boundary operators to constrain the temporal logic operators. Its formulæ is defined as follows:

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

$$\text{where } \sim \in \{<, =, >, \equiv_d\} \text{ and } c \geq 0, d \geq 2$$

$\bigcirc_{\sim c}\phi$ means "Next", $\ominus_{\sim c}\phi$ means "Previous", $\phi\ U_{\sim c}\ \phi$ means "Until" and $\phi\ S_{\sim c}\ \phi$ means "Since" (Chang et al., 1994). Given that T_i indicates the time of the ith state of the path π^i, whether a formulæ holds at the position j of the path π is defined as follows (we ignore the propositional operators here):

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

Past time LTL

Whereas LTL mentioned in the last part is defined to check the future states, Past time LTL (ptLTL) aims to verify the states in the past. A ptLTL formulæ is defined as follows (Havelund et Roşu, 2004):

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

As can be seen in the definition of the formulæ, ptLTL keeps several basic operators as LTL and introduces five standard part time operators and four monitoring operators.

The five standard part time operators are $\odot$ which means "previous", $\diamond$ "eventually in the

past", $\boxdot$ "always in the past", S_s "strong since" and S_w "weak since".

The monitoring operators $\uparrow\downarrow [,)_s[,)_w$ mean respectively "start", "end", "strong interval" and "weak interval".

The temporal operators' semantics are described as follows, in the same form of the last section:

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

EAGLE

EAGLE (Barringer et al., 2004) is a temporal finite trace monitoring logic supporting parameterized equations by combining minimal/maximal fix-point semantics with temporal operators.

Online runtime verification requires to accept incremental traces which means there are

possible boundaries between traces. Minimal/maximal fix-point rules were designed to treat this problem. Before evaluating the next trace, the equations with maximal rules are required to be always right and the ones with minimal rules are only needed to be eventually right.

2.3 RUNTIME VERIFICATION FRAMEWORKS

In runtime verification, monitors are generated from formal specifications by runtime verification frameworks. There are four monitoring modes: offline, online, inline and outline as we discussed in the early part of this section. Many frameworks and systems using some variant or extension of LTL have been proposed, as Table 2.2 shows.

Name	Logic	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)	Various	inline/outline/offline

Table 2.2: Runtime Verification Frameworks

Java PathExplorer (JPaX) (Havelund et Rosu, 2001) is an online runtime verification system aiming to monitor the execution traces of Java program. It has three modules (shown in Figure 2.2): an instrumentation module, an observe module and an interconnection module. The program working with the instrumentation module sends necessary event traces to the interconnection module which then transmits the traces to the observe module possibly running on another computer. The instrumentation module is driven by a user-specified script written in Java or Maude which is applicable for the specification of runtime monitoring.

JavaMaC (Kim et al., 2004) is a "run-time assurance system" for Java programs while Mac means Monitoring and Checking. Its architecture is shown in Figure 2.3. Two definition languages are proposed: one for high-level specification which specifies required properties,

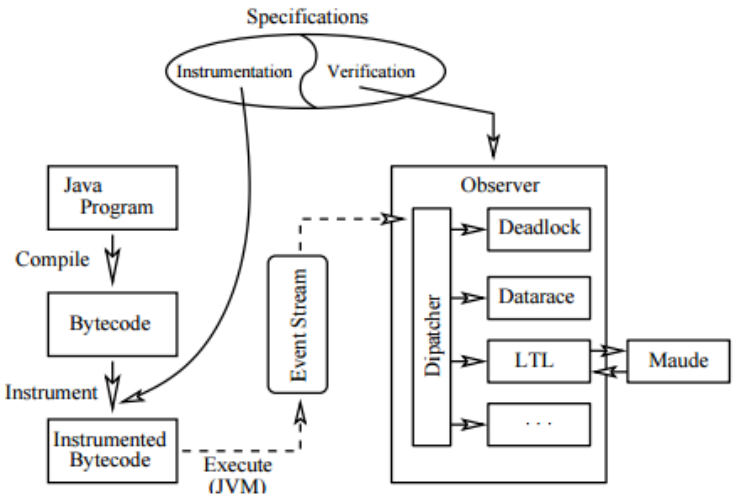

Figure 2.2: JPaX Architecture (from Havelund et Rosu (2001))

another for low-level specification which defines the events and conditions. During the preparation, a "filter" and an "event recognizer" which are used to collect the necessary event traces are generated from the low-level specification, and a "run-time checker" is generated from the high-level spec. When running with the target program, events collected by the "filter" and "event recognizer" are sent to the "run-time" checker which is then responsible for the runtime verification work.

d'Amorim et Havelund (2005) presents a logic named HAWK and its tools for RV of Java programs. HAWK is in fact built on EAGLE, another temporal logic which is considered more expressive (Barringer et al., 2004). Although HAWK is event-based in contrast to state-based EAGLE, HAWK specifications are translated to EAGLE monitors. As Figure 2.4 describes, during program execution, the EAGLE state is updated by instrumented program which then notifies the EAGLE observer. After that, the observer assesses the formulæ in the current state to produce a result.

Temporal Rover (Drusinsky, 2000) is a commercial runtime verification tool based on LTL

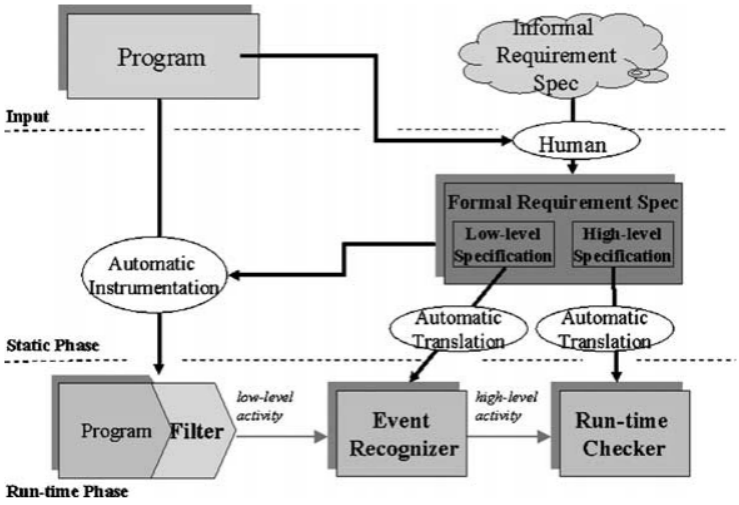

Figure 2.3: MaC Architecture (from Kim et al. (2004))

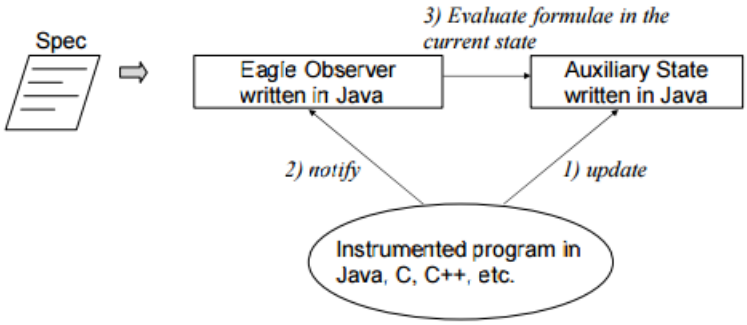

Figure 2.4: Eagle Architecture (from d'Amorim et Havelund (2005))

and MTL (Metric Temporal Logic). The specification code of Temporal Rover is inserted into the source code of Java, C, C++ or HDL and then converted into a compilable source file of corresponding language. A Temporal-Rover runtime verification system normally has two parts: host and target. The host is responsible for the verification while the target performs the computation of propositional formulæ and sends back the results to the host via serial port, RPC or other customizable protocol.

Each of the frameworks discussed above hardwires a different specification formalism, which

suggests that a general specification formalism serving all purposes does not exist. To be more expressive and generic, Chen et Roşu (2007) introduced customizable and extensible "logic-plugins" in their runtime framework MOP and designed its architecture shown in Figure 2.5 with two layers: one is called "language clients" which support different programming languages, while another is "logic repository" which includes and manages various "logic-plugins" to support different specification formalisms, such as : Linear Temporal Logic (ltl), Finite State Machines (fsm), Extended Regular Expressions (ere), Context Free Grammars (cfg) and String Rewriting Systems (srs).

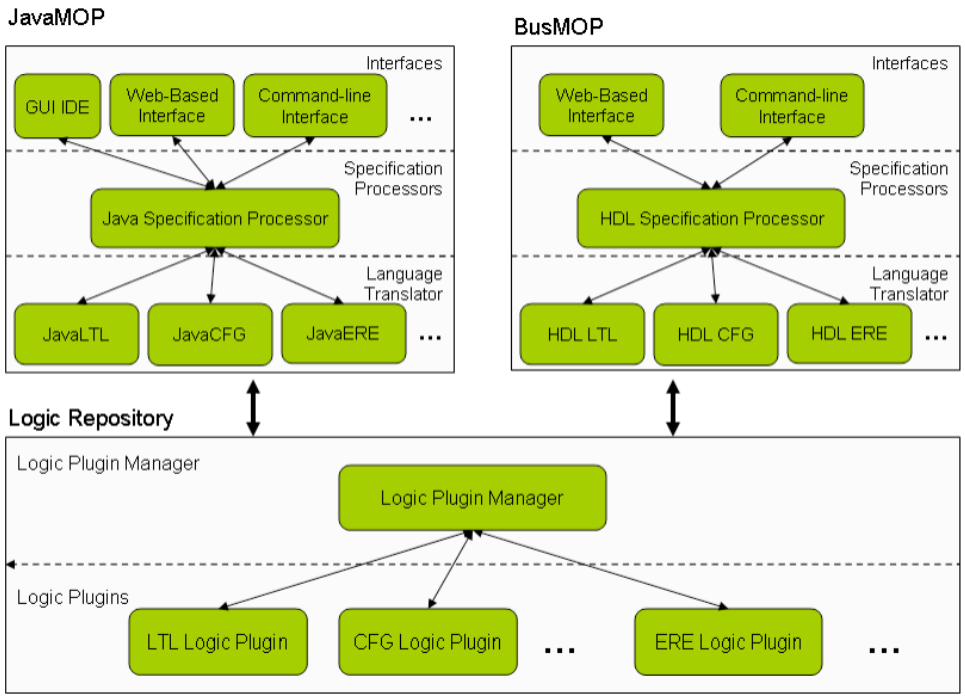

Figure 2.5: MOP Architecture (from Chen et Roşu (2007))

Besides the frameworks presented above, there are also lots of other frameworks invented for their corresponding requirements or specific temporal logics. Comparing these frameworks, we can find that they have their specialties as well as they share some common characters. For example, nearly all frameworks support online monitoring mode, Java programming

language and network communication perhaps because these features are the most popular requirement in industry. Temporal Rover is a commercial RV framework, so it has to support more programming languages and supply more data collection options in order to expand its marketing. MOP was designed to be extremely general, resulting that most components can be changed or separately optimized.

CHAPTER 3

REAL-TIME STREAMING COMMUNICATION WITH OPTICAL CODES

This chapter represents a reformatted version of a paper published in 2016 in IEEE Access, v.4, p. 284-298 by K. Xie, S. Gaboury and S. Hallé.

ABSTRACT

Optical codes have long been used to carry small amounts of static data, such as URLs, IDs or other short binary sequences. In this chapter, we experiment on the use of sequences of optical codes to form a one-way communication channel. In this context, a sender is made of a surface displaying rapidly changing codes, which are picked up by a receiver's camera and converted back into a binary data stream. After presenting experimental results seeking the combination of frame rate, code size and error correction level maximizing effective bandwidth, we describe the implementation of a robust communication protocol designed specifically for lossy, simplex, and low-bandwidth data links. Our findings indicates that such a protocol is sufficient for carrying at least voice-quality audio in realtime.

3.1 INTRODUCTION

Wireless communication is a technology that allows two or more peers to communicate without electric cables or conductors (Tse et Viswanath, 2005). While the majority of wireless communication technologies uses radio waves as their transmission medium, a few others use light, especially in situations where radio technology is difficult to apply. Optical communication in itself reaches back to the use of flags, smoke signals and signal lamps to communicate information between two points with the use of a specific code (Burns, 2004).

Recently, a simple form of optical communication, called *Quick Response Code* (QR code) (Denso Wave Inc., 2015), emerged as a refinement over the existing technology of one-dimensional barcodes. Owing to their accuracy and their considerable capacity, QR codes have been used in many areas; applications for handling such codes have also been ported to a variety of devices, including desktop computers, smartphones, and even television sets.

However, so far QR codes have been used for the transmission of *static* data. Generally a code is printed on a physical medium, such as a sheet of paper, and is read by an optical device (typically a camera) for decoding at a later time. In this chapter, we explore the idea of extending QR codes and turn them into a dynamic, one-way communication channel. In such a channel, a stream of data is transmitted through a *sequence* of codes; typically, these codes are continuously generated and displayed on one device, and simultaneously captured and decoded by another, hence performing the same task as other kinds of communication technologies.

After briefly describing in Section 3.2 the basics of QR codes and other wireless communication technologies, we shift our focus in Section 3.3 to the principle of raw QR code communication. In particular, we attempt to find the intrinsic limits of such a communication

channel, by analyzing the influence of a variety of factors, such as code density, number of codes displayed per second, etc. The results of an experimental benchmark considering more than a hundred different combinations of parameters allow us to extract optimal conditions that minimize the rate of error in decoding the images, while maximizing the amount of data that can be transmitted per unit of time.

These first results indicate that QR code streams can indeed be used as a simple, one-way channel, but that error-free communication and high-bandwidth are more or less impossible. Therefore, as a second step, we design a protocol suitable for the specific nature of QR code communication. This protocol, called BufferTannen, is described in Section 3.5. It is able to encapsulate raw data, provides various signaling capabilities, can represent semi-structured data (such as JSON) in a compact binary form, and supports data framing/deframing and streaming.

A second experiment reveals the robustness of this transmission scheme: using our specially designed protocol, the communication channel created by a person pointing a smartphone at arm's length towards a flickering QR display produces sufficient bandwidth to transmit voice-quality audio in real time. Section 3.4 presents the environment and the results of the experiments of this second step. We also show how a piece of data, cut into multiple codes with the use of BufferTannen, can be reconstructed automatically by a user hovering his camera over a sheet of such printed codes in no particular order.

This work was originally motivated by a practical application in the field of runtime verification. In past research, we suggested and informally experimented the use of optical codes as a form of loosely coupled communication between a software system and an external monitor receiving events produced by that system (Lavoie et al., 2014). In this context, communication through optical means ensured a complete isolation between the system and its monitor.

Although the use of sequences of QR codes has been informally suggested in the past, to the best of our knowledge, our study is the first systematical investigation of the potential of QR codes to send streams of data in real time.

The solution we propose provides a method for distributing streaming data without dedicated communication devices. The only required devices are a common camera (such as a webcam or the camera on any cellphone) and a small flat surface to display the sequential QR codes —for example, a computer, television or smartphone screen, or even a sheet of paper.

3.2 WIRELESS COMMUNICATIONS

This section recalls a few common wireless communication technologies. Despite their popularity and performance, each has its own limitations and application scenarios.

3.2.1 RADIO WAVES

The first obvious means of wireless communication is through the use of radio waves, and is best exemplified through WiFi (Comer, 2008), used for local area wireless networking. Its variants are based on the IEEE 802.11 family of standards, and support both centralized (routing) and decentralized (*ad hoc*) networking. Table 3.1 shows popular 802.11 standards and part of their specifications.

A second contender in this family is Bluetooth (Comer, 2008), which is used for short-range and generally point-to-point communication between devices. Its range varies from about 1 to 100 meters depending on the class of power. Table 3.2 shows different versions of Bluetooth and their specific data rates.

Protocol	Frequency	Max. physical data rate	Indoor range
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

Table 3.1: A summary of WiFi protocols(Theng, 2008; Perahia et Stacey, 2013)

Version	Data rate	Range
Version 1.2	1 Mbps	Class 1: 100 m; Class 2: 10 m; Class 3: 1 m
Version 2.0 + EDR	3 Mbps	
Version 3.0 + HS	24 Mbps	
Version 4.0	24 Mbps	

Table 3.2: Bluetooth specifications (Gupta, 2013)

Finally, ZigBee (Farahani, 2011), based on the IEEE 802.15.4 standard, aims to implement short-range wireless communication with low power and long battery life. Table 3.3 shows its performance in different frequency bands.

Frequency Band	Data rate	Range
868–870 MHz	20 kbps	10–100 m, depending on power output and environment
902–928 MHz	40 kbps	
2.4–2.4835 GHz	250 kbps	

Table 3.3: ZigBee specifications (Lee et al., 2007)

All these protocols have on point in common: before allowing any form of communication between two endpoints, some form of *discovery* or *setup* of devices is necessary. This process is generally finalized through the establishment of a long-running and stateful *connection* between the endpoints.

3.2.2 IRDA

In a different family, we find technologies using infrared waves instead of radio (Sarkar et al., 2007). In addition to the different wavelength, these technologies generally relax the requirements on the establishment of a connection, and allow for a more "on-the-fly" communication between two devices. Typically, one end of an infrared link waits for incoming data, while periodically, another device "points" at the receiver and beams short bursts of data without requiring prior notice.

Of particular importance is the IrDA standard (Infrared Data Association); its transceivers send infrared pulses with a cone angle and a moderate irradiance while its receivers can be less than one meter or several meters away depending on the power of the transceivers and the position in the cone. The IrDA communication is half-duplex and provides basic CRC. Table 3.4 shows several IrDA schemas and their data rates in the specific range.

Schema	Data rate	Range
SIR	2.4–115.2 kbps	Up to one meter
MIR	0.576–1.152 Mbps	
FIR	4 Mbps	
GigaIR	512 Mbps–1 Gbps	

Table 3.4: Data rates for IrDA Physical Layer schemas(Millar et al., 1998)

3.2.3 VISIBLE LIGHT COMMUNICATION

As its name implies, Visible Light Communication (VLC) (Komine et Nakagawa, 2004) uses wavelengths in the visible range (from 400 to 700 nm) to communicate data between peers —this is typically achieved by rapidly switching a light source on and off, enabling a form of Morse-like encoding of data. A receiver (such as a photo-electrical cell) pointed at the light

source detects this flickering and converts it back into digital data. With fluorescent lamps as the light source, the data rate can reach 10 kbps, while with LED technology, the data rate can be as high as 500 Mbps. The range depends mostly on different specifications, but because it cannot penetrate walls and can also be impacted by dense weather or other light sources, its range and reliability are limited (Arnon, 2015).

This mode of communication is intrinsically one-way, as one cannot reply back to the light source, acknowledge reception of information, or request for a retransmission in case of lost data. Therefore, this technology is also the one that requires the least coupling between a transmitter and a receiver; by all practical means, the transmitter is unaware of the presence of a receiver, which, on its side, may elect to start listening at any point in time. We shall see later on that this characteristic is also shared with the QR communication channel we attempt to design.

3.3 READING STREAMS OF QR CODES

Not mentioned in the previous short survey of wireless communication technologies are optical codes, which are also a means of carrying data without the need for a physical media. In this section, we review the concept of QR codes, and discuss the idea of producing streams of data through sequences of such codes.

3.3.1 OVERVIEW OF QR CODES

A QR code (officially called “Quick Response Code”) (Denso Wave Inc., 2015) is a two-dimensional barcode that stores data, as shown in Figure 3.1. Compared with the well-known UPC barcode, which is a linear (i.e. one-dimensional) barcode, a QR code can store more

Figure 3.1: A QR code with the text "Hello world!"

Error correction level	Data bits	Numeric	Alphanumeric	Binary	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

Table 3.5: Maximum storage capacities for Version-40 codes

information in a smaller printout size.[1] The QR code standard stipulates that these codes can have a capacity as high as 7,089 numeric characters or 2,953 8-bit characters, represented in a square array of a maximum of 177×177 "pixels", called *modules* (International Organization for Standardization, 2006).

The capacity of a QR code mainly depends on its data type, version and error correction level. The data type can be *Numeric only*, *Alphanumeric*, *Binary*, or *Kanji*. The version, from 1 to 40, determines a code's dimensions, which range from 21×21 to 177×177 modules. QR codes use a form of error-correction coding which can take four levels: *Low* (L), *Medium* (M), *Quartile* (Q), and *High* (H), as is shown in Table 3.5. Obviously, as the level increases, more redundancy is introduced in the code's content, which decreases its storage capacity; however, more data can be restored if the code is dirty or damaged. With position detection patterns included in the symbol, a QR code can be decoded in 360 degrees.

Before generating a QR code from a piece of data, a code generator needs to analyze that

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

input data to decide the most efficient mode and version. In the data encoding, the characters are converted into a bit stream, and in this progress, some *mode indicators* and *terminators* are inserted for the mode changes. The bit stream is then split into 8-bit codewords, and pad characters are required to fill the number of codewords for the chosen version. The generated codeword sequence is divided into blocks according to the specific error correction level and an error correction codeword is generated for each block. Then the codewords from each block are interleaved and some remainder bits are added as necessary.

In the next step, the generator places the codeword modules in a black-and-white matrix with the finder pattern, separators, timing pattern and alignment Patterns; it applies the masking patterns, evaluates and then selects the appropriate pattern. Finally, it generates the *Format* and *Version Information* and completes the QR code (International Organization for Standardization, 2006).

The decoding steps are simply the reverse of the encoding procedure. At first, the QR code needs to be located and the black and white modules are recognized as 0s and 1s that form a binary array. From the binary array, the decoder gets the format information and version information. With that information, it can begin to read the characters and the error correction codewords, and then tries to detect and correct the errors with the error correction codewords according to the appropriate error correction level. In the next step, the data codewords are divided according to the *Mode Indicators* and *Character Count Indicators*, and the data characters are finally decoded and output.

3.3.2 THE CASE FOR QR CODE COMMUNICATION

The aforementioned process applies for the encoding and decoding of a single code containing static data. We now investigate the idea of using QR codes as a communication channel, where

realtime data would be transformed on-the-fly as a *sequence* of QR codes, which could then be optically picked up by some device, and converted back into the original data stream at the receiving end.

The use of QR code communication presents several advantages in a handful of scenarios. For example, the US Navy has investigated the use of QR codes as a "digital semaphore". The proposed technology focuses on the detection of low-resolution codes from very long ranges, and highlights the interest and possible use cases for this technology in a military context:

> 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)

However, in the cited work, the codes are considered to be *static*, i.e. they do not change over time to form a stream of data, and more or less act as a substitute for flags or signs. Nevertheless, the absence of any emission of radio waves in QR code communication proves to be an appealing advantage in some scenarios.

We have also seen in the previous section how all other technologies, such as Bluetooth or IrDA, require dedicated hardware. In contrast, using QR codes to communicate can be done through codes printed on a hard surface, or through any device capable of displaying images in sufficient resolution: TV screens, computer monitors, tablets and cell phones. Similarly, reception can be done through any device equipped with a standard, off-the-shelf camera. This can turn devices equipped with this common hardware into communicating devices, even though they were not designed for that purpose in the first place. One can even imagine

emergency situations in which all digital means of communications between two points are not working. If line of sight can be established and a display and a camera are available, the use of QR codes nevertheless makes it possible to transmit digital data —arguably much faster than the manual handwriting/transcribing that would otherwise need to be done.

Finally, we have mentioned in the beginning how the use of an optical and strictly one-way communication channel can also be desirable, even in situations where radio or cable communication is available. For example, in the context of runtime verification, the execution of a system is being observed by an external process called a *monitor*. To prevent the monitor from interfering with the execution of the system, it is often placed on a separate machine, with some communication channel carrying events from the former to the latter. However, in traditional protocols such as TCP, the bidirectional nature of a connection presents too high a risk of attacks on the program to monitor. Moreover, some software setup is required to hook up the monitor to the program: defining IP addresses, pipe names, ports, etc., which represents too much coupling in many scenarios. We have argued in past work (Lavoie et al., 2014) how the use of an optical communication channel can alleviate these problems by providing greater isolation between the system and its monitor.

3.3.3 ESTIMATING BANDWIDTH AND ERROR RATE

However, one-way transmission introduces the possibility of losing frames during the process, due to the limitation of the physical devices or the vulnerability of the software. Moreover, it is impossible for the transmitter to be aware of any missing frames on the receiving end and resend them. Therefore, we need to analyze thoroughly this approach to estimate the *recognition rate* and the *transmission bandwidth* of such a communication channel.

The transmission of codes takes multiple parameters: each frame's data size, the number

of generated frames per second (fps) and the error correction level. All three can have an important effect on the generation of QR codes and the resulting bandwidth. Larger data size leads to a higher symbol version of the QR code and more symbol modules, and with the same frame size, a higher correction level requires more symbol modules than a lower one.

Because the sender cannot be aware of any codes missed by the receiver, this one-way communication channel is actually a lossy channel, of which the *effective* bandwidth can be calculated with the measured frame recognition rate. This represents the number of bits that are correctly received.

$$bandwidth = fps \times frame\_bits \times recognition\_rate$$

If the receiver finds that not all the frames are received, the only way is to make sure the sender sends all the frames again and again until the receiver gets all frames and stops; the actual bandwidth is:

$$bandwidth = fps \times frame\_bits \div actual\_sent\_times$$

The recognition rate is normally determined by the ability of the camera and the screen, the accuracy of the recognition algorithm and the code's complexity (i.e. the number of the displayed modules). However, within the ability of the camera and the screen, if we can send the same frame for more than once, meanwhile the value of *fps* doesn't need to change, the practical recognition rate can be improved.

$$practical\_recognition\_rate = 1 - (1 - recognition\_rate)^{times}$$

3.4 EXPERIMENTS

In this section, we describe experiments in which we measure the accuracy of reading sequences of QR codes in various conditions. The purpose of these experiments is threefold:

1. assess whether data can be successfully transmitted through the reading of sequences of optical codes;
2. determine the parameters that maximize the decoding rate and bandwidth of the transmitted data;
3. from these results, determine the characteristics of a typical QR stream communication channel.

3.4.1 EXPERIMENTAL SETUP

Our set of experiments involves producing and displaying sequences of QR codes on one end, and capturing and decoding these sequences on the other. In our experimental environment, we used a Samsung 19-inch LED monitor as the transmitter and a high-definition Logitech webcam as the receiver. The camera was placed at a fixed distance of 50 cm from the screen. The resolution of the monitor was 1280×1024 pixels. The camera was placed on a stable surface, with the optical code zone correctly in focus and covering the whole field of view. The computer used for the experiments is a laptop with the Intel Core i7-3632QM processor and 16 GB of memory. Figure 3.2 shows the setup used for the experiments.

In the development, we chose OpenCV[2] to capture the images from the camera and ZXing[3] to generate and decode the QR codes. To reduce the CPU and memory's overhead of capturing

[2] `http://opencv.org/`
[3] `https://github.com/zxing/zxing`

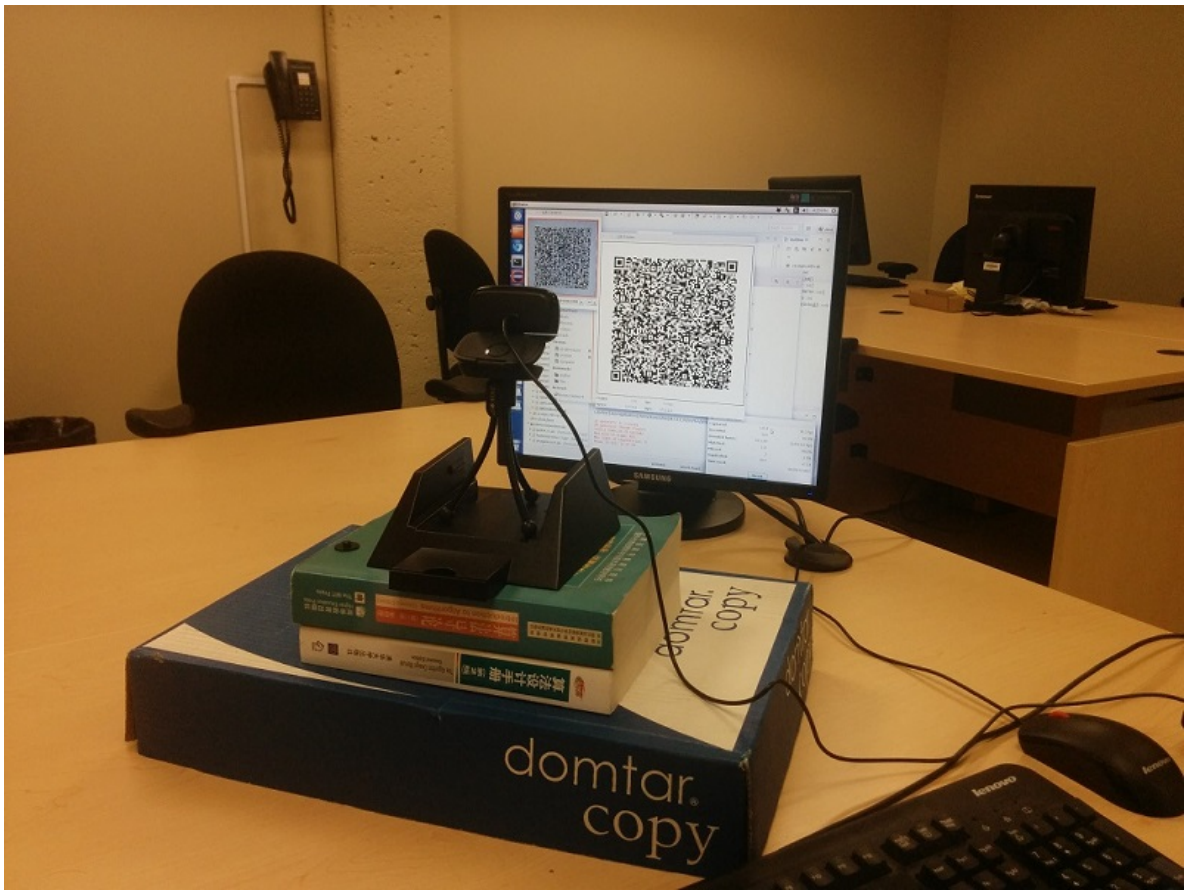

Figure 3.2: Experimental setup for reading codes.

and decoding, the captured images were transformed to 16-level grayscale. The data used to generate QR codes were randomly generated *alphanumerics*. All benchmarking code is implemented in Java and is freely available.[4]

The code decoding depends on the quality and the complexity of the captured image. If the image is broken or blurred, it will be difficult to decode. And the algorithms of image recognition may have the probability of failure(Adelmann et al., 2006). Therefore, our first step is to measure the ability of optical recognition libraries to properly read sequences of codes, irrespective of the actual data contained in these codes. Sequences of codes were generated by producing a character string of the form `dddd#rrrr...`, where `dddd` is a sequential number

[4] `http://github.com/sylvainhalle/GyroGearloose`

starting from zero and incrementing by one on each successive code, and `rrrr...` is a random string of characters (different on each code) long enough to fill the code up to its maximum size. Each test consisted in filming the sequence of such codes and storing the sequential number of each correctly decoded image into a file. This allows us to determine the fraction of all codes that were correctly read; given the size of each code and the number of codes sent, this makes it possible to compute the bandwidth and decoding error rate.

Our experiments quickly tripped over what appears to be a bug in the ZXing image decoding library. When analyzing sequences of images captured by the camera to look for decoding errors, we discovered that a number of times, the decoding failed while the corresponding image seemed to have no apparent problem (no blurring, correct framing, etc.). Putting the offending codes back on screen and trying to properly decode them with the camera yielded no success, even after changing the code's size, the camera's position, lighting conditions, etc. This is all the more puzzling that codes immediately before and after the problematic one were correctly decoded in multiple frames, while being captured in the same conditions. Even sending the code's "pure" image directly back into the decoding algorithm, without going through a camera, produces a decoding error.

It therefore seems the library cannot recognize some of the codes it itself produces (Figure 3.3 shows such an example). This most probably indicates a bug in the library, which has persisted up to the latest version available at the time this chapter was written. Therefore, in the following, the reader should keep in mind that an unknown proportion of reading errors may be due to this purported bug, and not to the particular experimental conditions. This is the case, for example, for the gaps in the correction rate we shall observe in Figures 3.4 and 3.5.

Figure 3.3: A QR code generated by ZXing that ZXing itself cannot decode in the experiment.

3.4.2 EXPERIMENTAL PARAMETERS

The experiment seeks the combination of parameters that could maximize the bandwidth and minimize the error rate for the transmission of codes. The parameters that were considered are the following.

Code Resolution

The first parameter is code data size (i.e. the number of data bits contained in each code) and physical size (number of pixels used to display the code on screen). We varied the data size in increments of 500 bits, from 500 up to 4,500 bits. As shown in Table 3.6, the largest QR code, which contains 4,500 bits of data using the highest error correction level, is 101×101 modules large. We also fixed the code's physical size to 700×700 pixels, which makes each module a square of at least 6×6 pixels.

Input data bits	Error correction level	Symbol version	Symbol size
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

Table 3.6: Sample QR code sizes, according to their data size and error correction level (International Organization for Standardization, 2006)

Code Rate

The second experimental parameter we considered is the code rate, i.e. the number of codes displayed per unit of time. We initially selected 2, 4, 6, 8, and 10 codes per second (cps), and also considered up to 16 cps in a later phase of the experiment.

Error Correction Level

As we have seen, QR codes include additional data intended for error correction. We hence also varied the level of error correction used in each experiment, using either its highest setting (H) or its lowest (L).

Camera Resolution and Rate

The resolution of the camera was not considered as an experimental parameter. It was fixed to its maximal setting, 1920×1080 pixels. Similarly, its frame rate was kept fixed at 30 frames per second. This corresponds to 1080p high-definition video, a setting expected to be found in the majority of recent and future video capture devices. We performed some informal tests with lower resolutions (down to 640×480), which were globally conclusive, but did not deem relevant of including them in our detailed analysis.

3.4.3 EXPERIMENTAL RESULTS

The product of all combinations of code size, error correction level and code rate produces a total of 90 different experiments. These experiments were repeated in three sets, differing in the way in which codes were displayed.

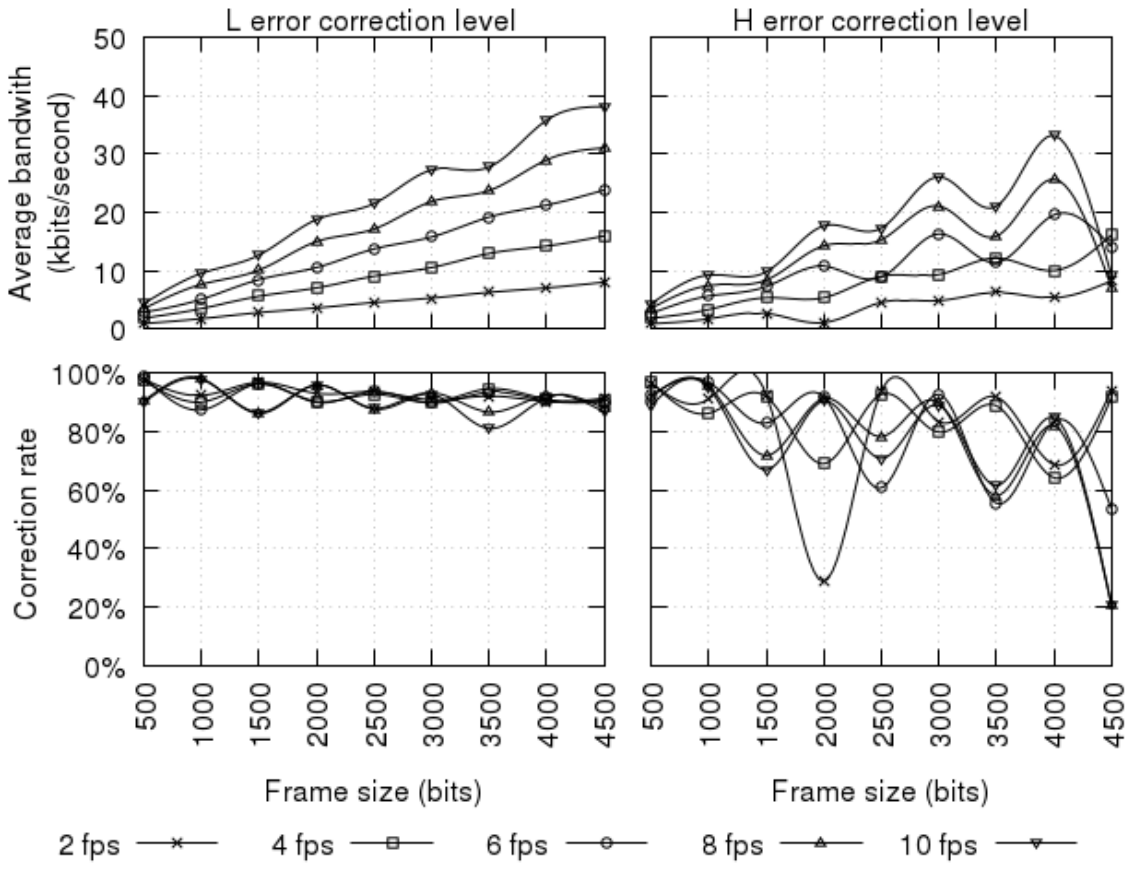

Figure 3.4: Bandwidth and decoding rate in the first experiment

Single Display

In a first experiment, each code was displayed in sequence for a duration of $1/f$ second, where f is the code rate. The bandwidth and decoding rate are shown in Figure 3.4 for combinations of all parameters.

As one can see, the recognition rates of higher correction level were lower than the ones of lower correction level, with all other parameters being equal. This can be explained by the fact that the same amount of data, carried inside a code with a higher correction level, has to display more modules. For example, according to Table 3.6, the modules of a 2,000-bit,

H-level code are as small as those of a 4,500-bit, L-level code. Smaller modules, in turn, yields increased difficulty in recognition by the camera. Therefore, a first conclusion one can draw is that, surprisingly, effective bandwidth seems to be improved by using a *lower* level of error correction.

With the same data sizes and correction levels, the figure shows that the recognition rate decreases as the code rate increases. This can be explained by the fact that, in a higher code rate, the same code occupies fewer camera frames, and hence has fewer chances of being correctly decoded in one of the frames. Moreover, the probability that a code change occurs at the moment a frame is taken (resulting in a blurry image showing part of two different codes) is also increased. In the L level, the decrease is slight, but in the H level, the decrease is dramatic when the code size reaches 3,000 bits. As the data size increases, the recognition rate drops constantly and considerably.

These figures seem to indicate that the ideal configuration for level L is 4,500 bits and 10 fps, which yields an effective bandwidth of 39.0 kbps; for level H, 4,000 bits and 10 fps result in a bandwidth of 24.6 kbps.

Double Display

Considering that the camera might have missed several frames, we performed a second experiment in which every QR code is displayed twice within a small time window. Hence, instead of displaying each code once for $1/f$ second, each code was interleaved with neighbouring codes and displayed twice for $1/2f$ second each time. This results in the same total exposure time for each code, but increases the diversity in the images captured by the camera.

The results are plotted in Figure 3.5. They show an increase in all recognition rates, which

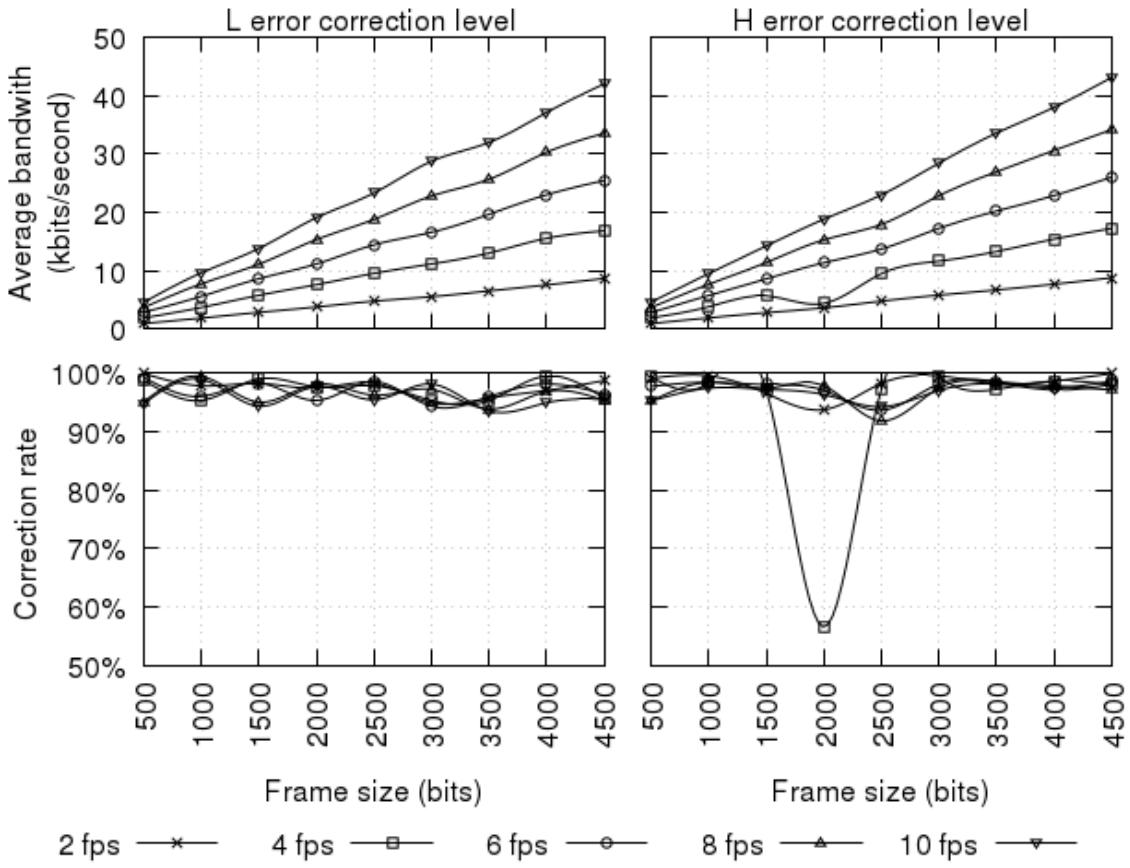

Figure 3.5: Bandwidth and decoding rate in the second experiment, where each code is displayed twice

are now all higher than 90%. This, in turn, increases the effective bandwidth; using the same settings as above, one can get a bandwidth of 43.0 kbps using level L, and 44.1 kbps using level H.

Random Padding

However, as we discussed earlier, not all QR codes are created equal; for the same resolution and error correction level, experimental results indicate that some codes seem to be more difficult to read than some others. Therefore, merely repeating the same image multiple

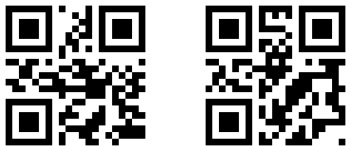

Figure 3.6: Examples of two codes with slightly different data, but widely different dot patterns. The code on the left contains the string "abcdefg", while the one on the right contains "abcdeff".

times has no impact on that intrinsic "hardness". Our third experiment introduces yet another mechanism for boosting recognition rate.

This time, we tried to make the codes from the same input data different by appending, at the end of the data to be encoded, a small random string intended to change every time the code is to be displayed. Hence the same original data, if displayed twice, is prepended to a different random padding each time, yielding a slightly different array of bits. However, by virtue of the QR encoding schema, even a small change at the end of an array produces a completely different pattern of dots in the resulting QR code. Figure 3.6 shows an example of this phenomenon. Therefore, if a code is harder to read, the same data is also displayed in a largely different pattern of dots, increasing the odds of being properly picked up at least once.

Although the objective reason for some codes being harder to read is unknown and out of the focus of this chapter, experimental results seem to confirm this hypothesis. We performed a third experiment where every input data was displayed three times with different generated QR codes. The recognition rate is better than before when the code rate is lower than 10 fps, as shown in Figure 3.7.

These results led us to experiment with higher code rates; we added 12 cps, 14 cps and 16 cps. The codes were displayed twice. As the Figure 3.8 shows, the maximum effective bandwidth in the result is 65.5 kbps using level L, and 68.3 kbps in level H level, using 16 cps and 4,500-bit codes.

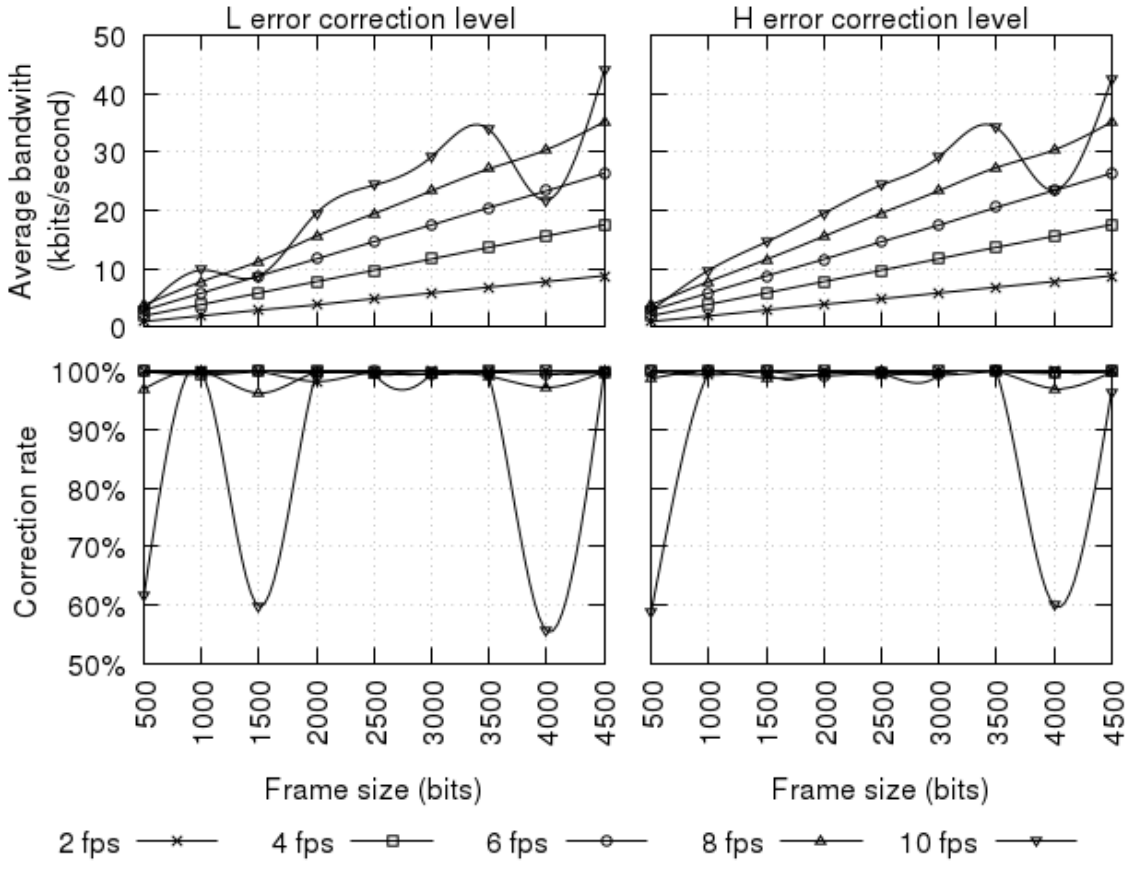

Figure 3.7: Third experiment: triple display and random padding

3.4.4 PARTIAL CONCLUSIONS

These initial experiments allow us to draw a few conclusions on the nature of a QR-based communication channel. First, although higher code rate and code size have a negative impact on the recognition ratio, the increased data that can be carried globally compensates for the higher error rate in terms of *effective* bandwidth. Second, introducing repetition and varying the dot pattern for the same data increases the effective bandwidth; that is, showing two different codes for half the time is more effective than a single code for the same interval. Third, even for the smallest code sizes, the error rate of the channel is never zero, indicating

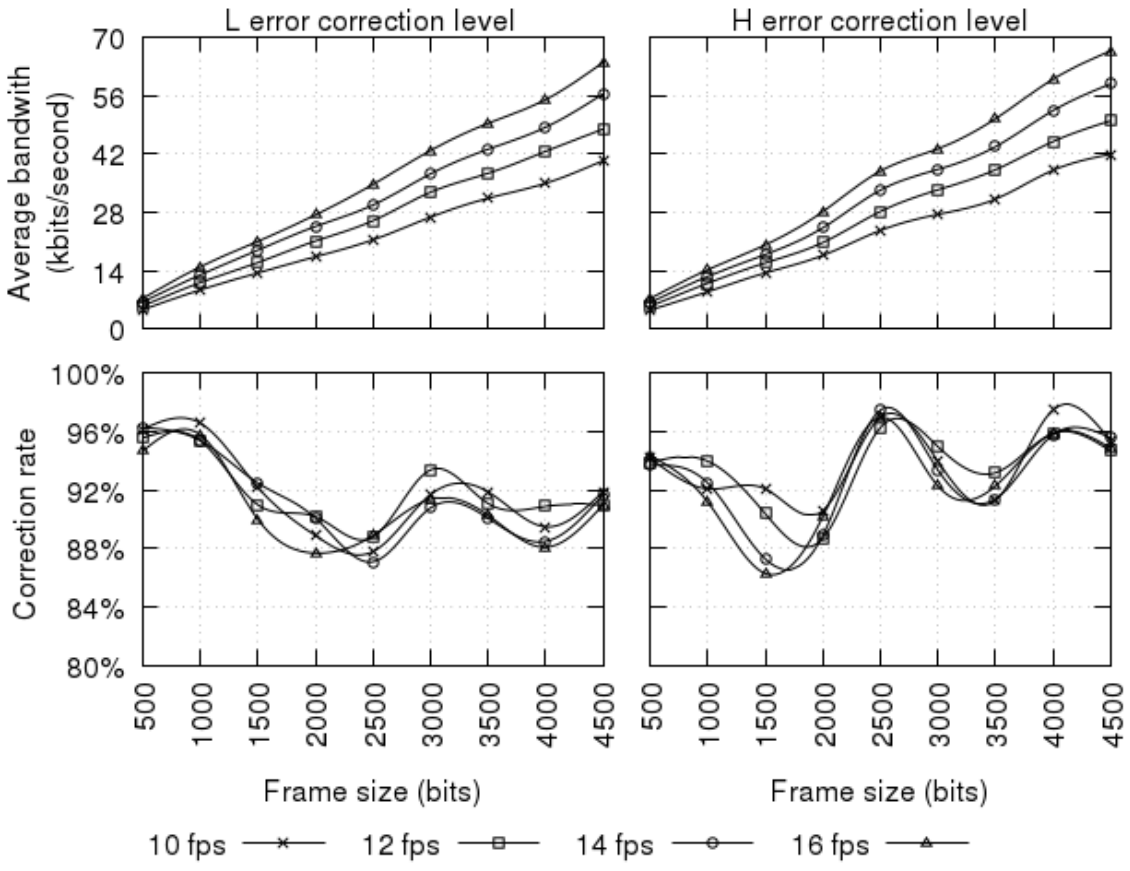

Figure 3.8: Fourth experiment: double display and higher code rates

that the channel is intrinsically lossy.

From these findings, one can reasonably expect a QR code stream to provide a channel with an effective bandwidth of about 40 kbps, when displaying 10 4,000-bit codes per second using the random padding technique and L-level error correction. The decoding rate of the channel using these parameters should be of at least 95%. Obviously, these findings apply to a fixed-camera setting. They do not take into account potential jitter, blurring or other effects that may occur in other contexts —although an informal experiment described in Section 3.5.8 tends to indicate the technology is relatively robust.

3.5 A PROTOCOL FOR ONE-WAY, LOSSY COMMUNICATION CHANNELS

In this section, we propose an approach which uses continuous QR codes as a medium to achieve one-way data transmission.

3.5.1 DESIGN GOALS

In order to implement a communication channel, a specific protocol is essential; it should be well designed so that the data can be serialized and transferred without significant overhead. Besides, the protocol has to have the ability of splitting the to-be-transferred data into frames to generate the QR images. The result is BufferTannen, a Java software package dedicated to the serialization and transmission of structured data over limited communication channels.[5] It provides a set of classes allowing the representation of structured data in a compact binary form. Contrarily to other systems, like Google's Protocol Buffers [6], defining new message types can be done at runtime and does not require compiling new classes to be used. Moreover, messages in BufferTannen cannot be encoded and decoded without prior knowledge of their structure. However, since messages do not contain information about their structure, they use much less space.

BufferTannen also defines a protocol allowing the transmission of messages. Although any channel (TCP connection, etc.) can be used, BufferTannen was designed to operate on a channel with the following specifications, which are based on our initial experimental results:

- The channel is *point-to-point*. The goal is to send information directly from A to B; no addressing, routing, etc. is provided.

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

- The channel is *low-bandwidth* (that is, able to transmit a few hundred bytes at a time, possibly less than 10 times per second).

- The channel is *one-way*: typically, one side of the communication sends data that is to be picked up by some receiver. This entails that the receiver cannot acknowledge reception of data or ask the sender to transmit again, as in protocols like TCP.

- The channel is *lossy*. However, we assume that the channel provides a mechanism (such as some form of checksum) to detect when a piece of data is corrupted and discard it.

- A receiver can start listening on the channel at any time, and be able to correctly receive messages from that point on. As such, the communication does not have a formal "start" that could be used, for example, to advertise parameters used for the exchange.

Therefore, the communication channel envisioned as the transmission medium for BufferTannen's messages can be likened in many ways to a slow broadcast signal, such as Hellschreiber (Evers, 1979), slow-scan television (Bretz, 1984), Teletext (tel, 1976) or RBDS (rbd, 2011).

BufferTannen's protocol aims at transmitting messages as reliably as possible under these conditions, while preserving the integrity of data and the ordering of messages. The low-bandwidth nature of the channel explains the emphasis on serializing messages in a compact binary form. Since the receiver cannot ask for any form of re-transmission, the protocol must provide for automatic re-transmissions of each message to maximize their chances of being picked up, while at the same time not confusing a re-transmission with a new message with identical content. Moreover, as the receiver can start listening at any moment, and that the schema of messages must be known in order to decode them, the schemas used in the communication must also be transmitted at periodic intervals.

3.5.2 SCHEMAS

The declaration of a data structure is called a *schema*. Information can be represented in three different forms:

- Smallscii: A variable-length string of characters. Since BufferTannen is aimed towards limiting as much as possible the number of bits required to represent information, these strings are restricted to a subset of 63 ASCII characters (letters, digits and punctuation). Each character in a Smallscii string takes 6 bits, and each string ends with the 6-bit string `000000`.
- Integer: The only numerical type available in BufferTannen. When declared, integers are given a "width", i.e. the number of bits used to encode them. The width can be anything between 1 and 16 bits.
- Enumeration: A list of predefined Smallscii constants. Enumerations can be used to further reduce the amount of space taken by a data element when its set of possible values is known in advance.

These basic building blocks can be used to write schemas by combining them using compound data structures:

- List: a variable-length sequence of elements, all of which must be of the same type (or schema). List elements are accessed by their index, starting with index 0.
- FixedMap: a table that associates strings to values. The structure is fixed and the exact strings that can be used as keys must be declared. However, each key can be associated to a value of a different type.

These constructs can be mixed freely. The following represents the declaration of a complex message schema:

```
FixedMap {
  "title" : Smallscii,
  "price" : Integer(5),
  "chapters" : List [
     FixedMap {
       "name" : Smallscii,
       "length" : Integer(8),
       "type" : Enum {"normal", "appendix"}
     }
  ]
}
```

The top-level structure for this message is a map (delimited by `{...}`). This map has three keys: `title`, whose associated value is a Smallscii string, `price`, whose associated value is a integer in the range 0-32 (i.e. 5 bits), and `chapters`, whose value is not a primitive type, but is itself a list (delimited by `[...]`). Each element of this list is itself a map with three keys: a string `name`, an integer `length`, and `type` whose possible values are `normal` or `appendix`.

Schemas can be represented in a compact and unequivocal binary representation as follows.

Integer The declaration of an integer is encoded as the following sequence of bits:

```
ttt wwwww ddddd s
```

The sequence `ttt` represents the element type, encoded on 3 bits. An integer contains the decimal value 6. The sequence of `w` indicates the integer's width in bits. The width itself is encoded over 5 bits. The sequence of `d` indicates the integer's width, in bits, when expressed as a delta value, i.e. as the difference with respect to an integer from a previous message. The width itself is encoded over 5 bits. The single bit `s` is the sign flag. If set to 0, the integer is unsigned; if set to 1, the integer is signed. Note that integers expressed as delta values are always encoded as signed integers; hence this flag only applies to integers occurring as full values.

Smallscii string The declaration of a Smallscii string is simply coded as three bits representing the element type; a string contains the decimal value 2.

Enumeration An enumeration must provide the list of all possible values it can take. It is formally represented as:

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

The element type is the decimal value 1, and the sequence `llll` is the number of elements in the enumeration, encoded on 4 bits. What follows is a concatenation of Smallscii strings defining the possible values for the enumeration. Each character is encoded on 6 bits, and the end of a string is signalled by the 6-bit sequence `000000`.

List The declaration of a list is as follows:

```
ttt llllllll ...
```

The element type is the decimal value 3; the 8-bit sequence `llllllll` defines the maximum number of elements in the list. What follows is the declaration of the element type for elements of that list.

Fixed Map The last element type is the fixed map, declared as follows:

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

The element type is the decimal value 4; what follows is a Smallscii string defining the name of a key, followed by the declaration of the element type for that key; this is repeated for as many keys the map declares.

3.5.3 MESSAGES

A *message* is an instance of a schema. For example, the following is a possible message abiding by the previous schema:

```
{
  "title" : "hello world",
  "price" : 21,
  "chapters" : [
    {
      "name" : "chapter 1",
      "length" : 3,
      "type" : "normal"
    },
```

```
    {
      "name" : "chapter 2",
      "length" : 7,
      "type" : "normal"
    },
    {
      "name" : "conclusion",
      "length" : 2,
      "type" : "appendix"
    }
  ]
}
```

The reader familiar with JSON or similar notations will notice strong similarities between BufferTannen and these languages. As a matter of fact, elements of a message can be queried using a syntax similar to JavaScript. For example, assuming that `m` is an object representing the above message, fetching the length of the second chapter would be written as the expression:

```
m[chapters][1][length]
```

This fetches the `chapters` value in the top-level structure (a list), then the second element of that list (index 1), and then the `length` value of the corresponding map element.

As with schemas, messages can be represented in a compact binary form.

Smallscii string Strings are represented as a sequence of 6-bit characters, terminated by the end of string delimiter `000000`.

Integer Numbers are represented by the sequence of bits that encodes their value, without any terminating sequence: the number of bits to read is dictated by the size of the integer, as specified by the corresponding schema element. If the integer is signed, the first bit represents the sign (0 = positive, 1 = negative) and the remainder of the sequence represents the absolute value.

Enumeration An enumeration is simply made of the sequence bits corresponding to the appropriate value. Again, the number of bits to read is dictated by the size of the enumeration, as specified in the schema of the message to read. For example, if the enumeration defines 4 values, then 2 bits will be read. The numerical value i corresponds to the i-th string declared in the enumeration.

List A list begins by 8 bits recording the number of elements in the list. The remainder of the list is the concatenation of the binary representation of each list element. Since the type of each element and the number of such elements to read are both known, no delimiter is required between each element or at the end of the list.

Fixed Map The contents of a fixed map is simply the concatenation of the binary representation of each map value. The key to which each value is associated, and the value type to read, are specified in the schema of the message to read, and are expected to appear exactly in the order they were declared. This spares us from repeating the map's keys in each message.

3.5.4 READING AND WRITING MESSAGES

In BufferTannen, both schemas and instances of schemas are represented by the same object, called `SchemaElement`. An empty SchemaElement must first be instantiated using some

schema; this can be done by either:

- Reading a character string formatted as above; or
- Reading a binary string containing an encoding of the schema. As a matter of fact, in BufferTannen both messages *and* schemas can be transmitted in binary form over a communication channel, and a method is provided to export the schema of some message into a sequence of bits.

Once an empty SchemaElement is obtained, it can be filled with data, again in two ways:

- By reading a character string formatted as above; or
- By reading a binary string containing an encoding of the data.

Similar methods exist to operate in the opposite way, and to *write* a message's schema or data contents either as a character string or as a binary string. This way, messages and schemas can be freely encoded/decoded using human-readable text strings or compact binary strings.

As one can see, for a message to be read or written, it is necessary first to instantiate an object with a schema. As a matter of fact, trying to decode a stream of data without first advertising the underlying schema will cause an error, even if the stream contains properly formatted data. Similarly, trying to read data that uses some schema with an object instantiated with another schema will also cause an error. In other words, no data can be read or written without knowledge of the proper schema to use.

This might seem restrictive, but it allows BufferTannen to heavily optimize the binary representation of messages. In the absence of a known schema, each message would require to carry, in addition to its actual data, information about its own structure. Practically speaking, this

amounts to repeating within each message the description of its schema, interspersed through the message data. On the contrary, if the schema is known, all this signaling information can be discarded: when receiving a sequence of bits, a reader that possesses the schema knows exactly how many bits to read, what data this represents and where to place it in the message structure being populated. This entails, however, that a receiver that does not know the schema to apply has no clue whatsoever on how to process a binary string.

To illustrate the interest of BufferTannen as a message encoding scheme, we consider the example of transmitting events from a video game to an external monitor.

3.5.5 SEGMENTS

Messages and schemas are encapsulated into a structure called a *segment*. A segment can be of four types:

Message segments contain the binary representation of a message, along with a sequential number (used to preserve the ordering of messages received), as well as the number referring to the schema that must be used to decode the message. A message segment consists of a header structured as follows:

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

The header starts with two bits describing the type of the segment; a message segment contains the decimal value 1. The `n` and `w` sections describe the segment's sequential number and total length, both encoded on 12 bits. The four `s` bits provide the schema number in the schema bank that should be used to read this segment. The remainder of the segment is comprised of a map, list, Smallscii string or number, whose binary representation was described above.

Schema segments contain the binary representation of a schema, which is associated to a number. Multiple schemas can be used in the same communication, hence creating a bank of schemas identified by their number. A schema segment consists of a header structured as follows:

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

The header starts with two bits describing the type of the segment; a schema segment contains the decimal value 2. The `n` section describes the segment's sequential number, and the `s` section gives the the schema number in the schema bank this segment should be assigned to. The remainder of the segment is comprised of a binary string describing the schema, whose representation was described above.

Blob segments are intended to carry raw binary data over the BufferTannen protocol.

Delta segments contain the binary representation of a message, expressed as the difference ("delta") between that message and a previous one used as a reference. Delta segments are used to further compress the representation of a message, in the case where messages don't change much over an interval of time.

```
tt nnnnnnnnnnnn wwwwwwwwwwww rrrrrrrrrrrr...
```

The header starts with two bits describing the type of the segment; a delta segment contains the decimal value 1. The `n` and `w` sections describe the segment's sequential number and total length, both encoded on 12 bits. The `r` section gives the sequential number of another segment, relative to which the delta of the current segment are expressed. What follows is a binary

string that describes the “difference” one must compute with respect to that segment to obtain the contents of the current one.

The computation of the delta is performed recursively on each element of the two messages to compare in the order they occur. It is defined for each element type as follows.

- Smallscii strings: if the corresponding strings are identical, emit the single bit `0`. Otherwise, emit the bit `1` followed by the Smallscii string of the target message.
- Integers: if the corresponding numbers are identical, emit the single bit `0`. Otherwise, emit the bit `1` followed by the difference between the source and the target integer.
- Enumerations: if the corresponding value of the enumerated type is the same, emit the single bit `0`. Otherwise, emit the bit `1` followed by the integer value corresponding to the index of the value in the target message.
- Lists: if both lists have the same elements in the same order, emit the single bit `0`. Otherwise, emit the bit `1` followed by the binary representation of the target list.
- Maps: recursively apply the previous rules for each key of the map.

One can see that delta segments apply only a coarse form of comparison. For example, no attempt is made to detect whether two lists differ by the addition or deletion of an element; the contents of the list is retransmitted in full whenever it is not identical to the original. Nevertheless, this technique allows substantial savings whenever a part of a data structure remains identical from one message to the next.

3.5.6 FRAMES

The communication channel sends binary data in units called *frames*. A frame is simply a set of concatenated segments in binary form, preceded by a header containing the version number of the protocol (currently "1") and the length (in bits) of the frame's content. Formally, the binary structure of a frame is as follows:

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

The `v` section consists of the 4-bit protocol version number, followed by 14 bits indicating the total length (in bits) of the frame. Each segment is appended directly to this 18-bit header. As each segment's header contains its own length, no further marshaling is required to correctly decode segment data.

When many segments are awaiting to be transmitted, the protocol tries to fit as many segments as possible (in sequential order) within the maximum size of a frame before sending it. This maximum size can be modified to fit the specifics of the communication channel that is being used. In the current incarnation of the protocol, segments cannot be fragmented across multiple frames. Hence a segment cannot exceed the maximum size of a frame.

Each frame is then converted into a QR code, with its binary content Base64-encoded as the code's text. This QR code can then be read at the receiving site, converted into a binary sequence, and parsed back into frames, segments and messages by applying the reverse transformations.

3.5.7 STREAMING MODES

BufferTannen is designed with two sending modes, respectively called “Lake” mode and “Stream” mode.

Lake mode is intended for the sending of a finite piece of data, such as a file, or a sequence of BufferTannen messages whose complete contents are known in advance. The data to be sent is divided into a finite set of segments, and the whole sequence of segments is repeatedly emitted through QR codes. If any frames are missed or incorrectly decoded, the infinite repetition of all segments makes possible to catch the missing data at the next loop. Ultimately, decoding errors may entail that the data needs to be read for more than one loop before it is completely received.

The use of Lake mode can be detected by frames carrying a non-zero value to their “total segments” header field. Hence a receiver that starts reading at any point through the sequence of frames knows how many segments in total are to be received, and the relative position of each segment in the data to be reconstructed. This makes Lake mode a relatively slow, but very robust optical data transmission scheme.

In Stream mode, the data is continuously read into segments which then form a stream of frames, and the frames are immediately sent. The reading process stops only when there is no more data to read. The frames already sent are removed from the memory, so there is no way to resend the data for several times. However, for the sake of the data consistency, we made a buffer for the clones of the sent frames, and after having sent a specific amount frames fetched from the original data, the frames in the buffer are resent again and then removed from the buffer. Therefore, Stream mode is intended to send realtime data, typically where where the data loss is acceptable and the consistency can be slightly sacrificed (e.g. audio or video).

```
------------------------------------------------------------------------
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: Part of the text interface of the QR code receiver operating in Lake mode

Figure 3.9 shows a portion of the text interface of our QR code receiver implementation. The interface shows that the frames being received are in Lake mode. The buffer state field indicates the progress of the reception. In the example, it shows that 130 out of 219 segments have been correctly received; the text bar at the left indicates to what portions of the total sequence these segments correspond. A section of the sequence that has not been received at all is indicated by a blank space; increasingly full portions of the sequence are represented respectively by the symbols `.`, `:` and `|`. The `>` symbol indicates the relative position of the last segment that was correctly read.

The "Link quality" field gives a realtime indication of the decoding rate. It shows that 22 of the last 30 images captured by the camera were correctly decoded, and that globally, 339 images were decoded out of 454 captured. The resource identifier and data stream index, carried by each frame, is also displayed.

3.5.8 EXPERIMENTAL RESULTS

The experiments of Section 3.4 confirmed our intuition that optical code streams are an inherently unreliable and low-bandwidth communication channel. The compactness of the BufferTannen protocol can be motivated by an example from runtime verification. A particular

```
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: The schema of events produced by an instrumented video game.

video game, called Pingus, was instrumented to produce events containing the state of every character in the game. The schema for these events is shown in Figure 3.10.

File Transfer

An event typically contains data for 50 characters, hence the map structure is repeated that many times. Sending such an event in clear-text format, without any whitespace, takes roughly 3,750 bytes. At a rate of 30 events per second, it takes 879 kbps of bandwidth to transmit the event stream. The same event in BufferTannen takes 1,856 *bits*, or 232 bytes. This divides by more than 16 the bandwidth requirements for sending a stream of such events, yielding a bandwidth of 54 kbps.[7] From that point on, delta segments can be used to further reduce the stream's bandwidth, and transmit the remaining events using slightly more than 100 bytes each, consuming a bandwidth of approximately 24 kbps. Our previous experiments show that this is within the range of what one can reasonably expect to transmit using QR codes.

[7] Sending the same character string into Gzip shrinks it down to 716 bytes, which makes standard compression a less appealing alternative in that context.

We then tested the ability of the BufferTannen protocol to mitigate these defects, through its use of repetition and its compact binary representation.[8]

We chose to encode data into 4,000-bit codes, which, after the encapsulation of BufferTannen, amounts to a QR code of about 5,800 bits. With the combination of 5800 bits and L correction level, according to Table 3.6, the symbol size is about 93 $\times$ 93 which is between the combinations of 3,500-bit, H-level and 4,000-bit, H-level. From the result of the last experiment, the combinations of 4,000-bit, H level and any sample fps has more than 95% correction rate which is reliable. Secondly, the code rates were 4, 6, 8, 10, 12 fps. According to the last experiment, this configuration is reliable and supposed to be able to supply 23.2–69.6 kbps, and 16.0–48.0 kbps of bandwidth, respectively. In the consideration of the practical application, we chose to transfer a sample file of which the size is 37,656 bytes, and we performed each experiment 20 times.

The result of the Lake mode experiment in Figure 3.11 shows that the best fps value is 10, and in this case the sample file needed to be transferred on average 2.4 times to make sure that the receiver could get all the frames. The average spent time is 17.27 seconds, from which the bandwidth is about 17.0 kbps.

In Stream mode, the percentage of received codes is important. In the experiment, according to Figure 3.12, the average completion ratios of all configurations are over 99%, and the configuration of 12 fps needs approximately 13.11 seconds to send all frames with a data streaming channel of 22.4 kbps.

[8] A video of BufferTannen in action is available online: `https://www.youtube.com/watch?v=GSL0md0TlY8`

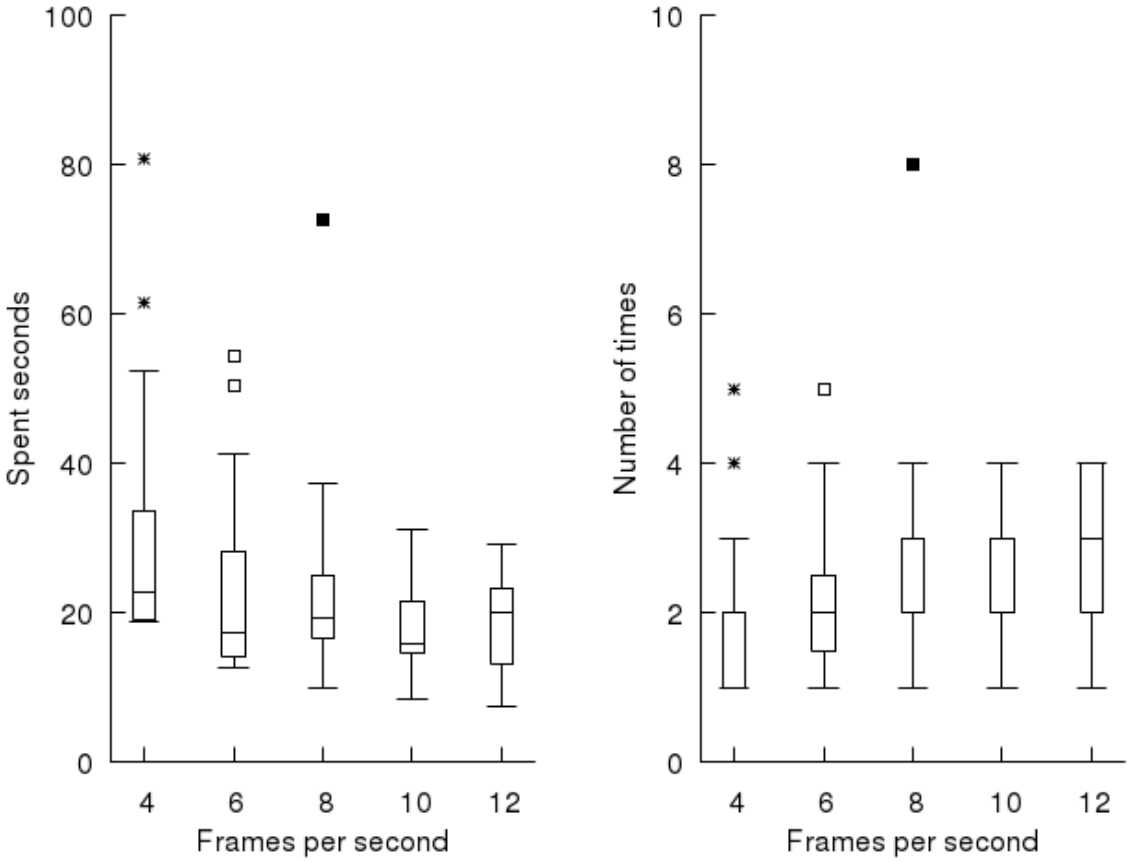

Figure 3.11: Time to send data in Lake mode

Paper Swiping

The ability to send streams of data in Lake mode can also be used to supplement QR codes' inherently limited capacity. When displaying a printed code on a piece of paper, large amounts of data can be carried only through increasing the code's resolution; however, resolution can only be increased up to a certain, predefined limit.[9] Moreover, for higher resolutions, the code may become hard to read using entry-level, low-resolution cameras. Therefore, it is safe to assume that, using existing QR code technology, no more than 3,000 bytes of data can be transferred using a QR code.

[9] 4,296 alpha-numeric characters, or 3,222 bytes assuming Base-64 encoding.

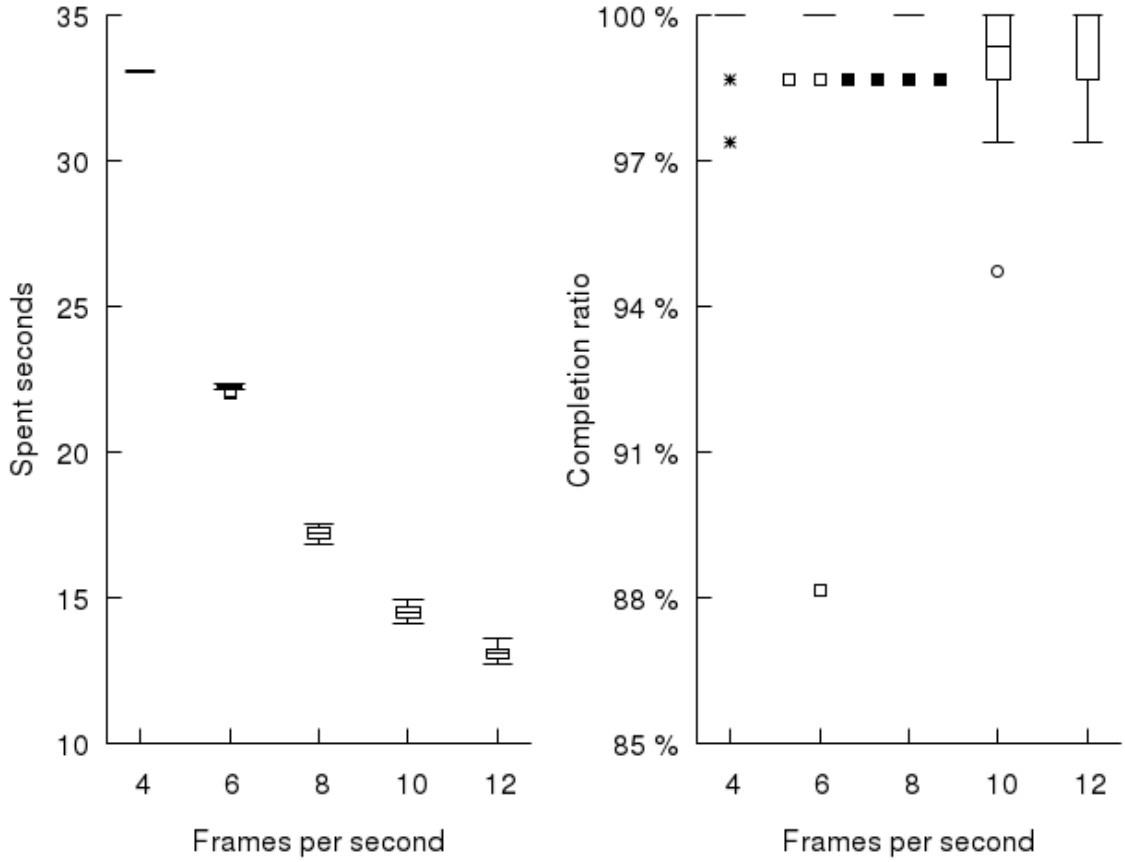

Figure 3.12: Time to send data in Stream mode

This limitation can be overcome by the use of the BufferTannen protocol. Although no code larger than roughly 4,000 bytes can be created, multiple such codes can be lined on a piece of paper. Each such code can be formatted to contain a single frame of data sent by the BufferTannen protocol in Lake Mode. It suffices for a user to swipe the camera over these multiple codes; by virtue of Lake Mode, the order in which the codes are scanned is irrelevant, and the complete piece of data can be correctly reconstructed from the individual frames. It is therefore possible to transmit theoretically unlimited amounts of data, while using codes of a lower resolution (this lower resolution being compensated by the presence of more than one code).

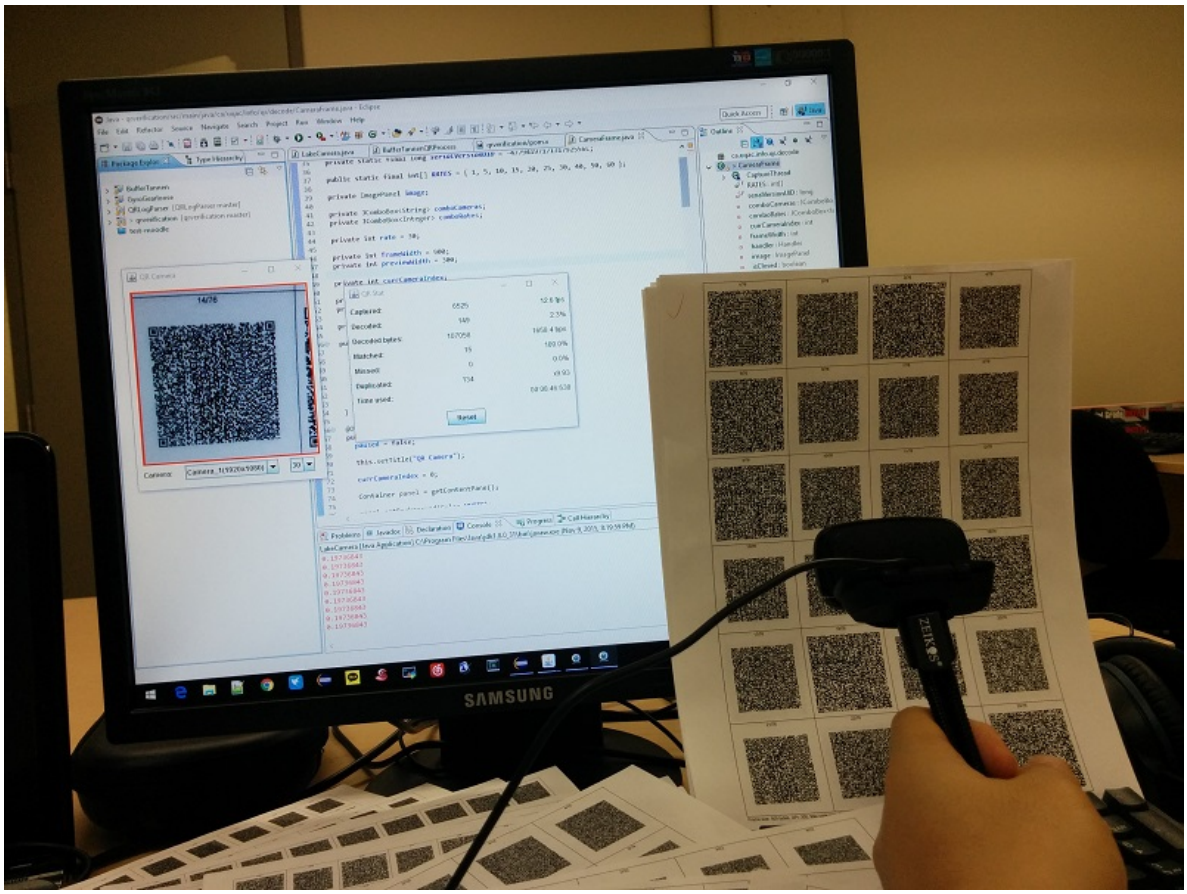

Figure 3.13: Swiping the camera over a set of QR codes to reconstruct the contents of a larger file.

To verify this claim, we printed on a piece of paper the contents of a 37 kb file as a sequence of QR codes, processed as frames through BufferTannen in Lake Mode. We then hovered the camera over that sheet of paper at arm's length (see Figure 3.13). The software's user interface displayed in real time the number of frames remaining to be decoded and their location in the complete stream, giving indications to the user as to which codes to swipe over. It shall be noted that the camera operated in "film" mode, and not in "snapshot" mode. In other words, images were continuously captured by the camera as it was being moved over the sheet; the user did not need to point and click at each individual QR code (which would be fastidious).

The error correction level chosen was L and the sizes of raw data per code that we tested

were 500, 750, 1,000, 1,250, 1,500, 1,750, and 2,000 bytes. With 500 bytes there were 76 codes while with 2,000 bytes there were only 19. After being encapsulated by BufferTannen protocol, the final sizes of data used to generate the QR codes became correspondingly 723, 1,055, 1,391, 1,724, 2,054, 2,389, and 2,722 bytes. The codes were printed at 300 dpi and 600 dpi on standard office paper, and we programmed to give each edge of every code 500 dots in the paper.

At 300 dpi, the codes of size ranging from 500 to 1,250 bytes were all decoded successfully and smoothly. However, when the size reaches 1,500 bytes, the decoding becomes more problematic. For some codes, we had to retry for several times by sticking the camera near the paper for a few seconds, yet among the total 31 codes, three remained impossible to decode at all. In the case of codes of 1,750 and 2,000 bytes, none of the codes could be decoded. This can be explained by the fact that higher density codes entail that each module of the code is smaller and hence harder to capture. For example, the edge of a 500-bytes and L-level code is about 77 modules, so each module can have approximately 6 dots in the paper, while the edge of a 2000-bytes, L-level code is 149 modules, so each module amounts to only 3 printed dots.

At 600 dpi, none of the codes could be decoded, no matter how many bytes they carried. The reason is that the codes printed at 600 dpi are too small and the camera has to approach very closely to the paper, but the captured images were all blurred and out of focus. It therefore seems that codes printed at 600 dpi are beyond the ability of a standard web camera.

Nevertheless, this experiment demonstrates the viability of the concept of hovering a camera across an array of QR codes. Our empirical findings indicate that a stream of data can be split into a set of QR codes of about 1,000 bytes each, the contents of which correspond to individual frames of the BufferTannen protocol containing the data to be transmitted. Swiping the camera over this set of codes, in no particular order, is sufficient to reconstruct on the

device side the complete data contents.

3.6 CONCLUSION

In this chapter, we presented a solution for an one-way communication channel based on QR codes, and performed experiments to measure its performance. We first experimentally tested the characteristics of a QR data stream under various conditions, and extracted the parameters maximizing the effective bandwidth of the channel. However, since that channel is inherently error-prone and low-bandwidth, we then introduced BufferTannen, a protocol designed especially for this kind of channel. BufferTannen takes care of splitting, marshaling, and to some extend compressing the data to transmit in order to maximize the efficiency of the QR code stream. The feasibility of this approach was then empirically observed through a new set of experiments.

Within the limits of the protocol and of the communication channel, the presented results can be put to good use in a variety of situations. In limited environments where the use of radio signal or cables is forbidden or difficult, our approach can provide an easy way to communicate between peers, either as an emergency backup or as a primary means. Furthermore, the evolution of the quality of both display and image capture devices makes it possible to foresee increased transmission rates in the future. Finally, the aforementioned techniques could be turned into a bidirectional communication link in the case of endpoints equipped with both a camera and a display. In such a case, acknowledgments of correctly decoded images could be exchanged, which in turn would allow resending data on demand and increase the effective bandwidth.

CHAPTER 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, sucha 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 $(11\ldots1)$ or just $w-1$ zeros $(00\ldots0)$, 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 $2^{32}-1$

Table 4.1: Parameters of RLE-model algorithms

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

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

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

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

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

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

Table 4.2: The semantics of LTL. Here $\overline{s}^i$ denotes the subtrace of $\overline{s}$ 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 **F** (eventually), **G** (globally), **X** (next), and **U** (until).

Let $\overline{s} = s_0, s_1, s_2, ..., s_n$ be a finite sequence of *events*, and let $\pi(i)$ be the set of atomic propositions that are true in s_i. The trace $\overline{s}$ 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 $\overline{s}$ is the empty trace, $\overline{s} \not\models \mathbf{F}\ \varphi$, $\overline{s} \not\models \mathbf{X}\ \varphi$, $\overline{s} \not\models \varphi\ \mathbf{U}\ \psi$, but $\overline{s} \models \mathbf{F}\ \varphi$.

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

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

Table 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 $(s_0, s_1, ..., s_{n-1})$ and an LTL formula φ, the principle is to compute a bitmap $(b_0 b_1 ... b_i b_{i+1} ... b_{n-1})$ of length n, noted B_φ, whose content is defined follows:

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

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

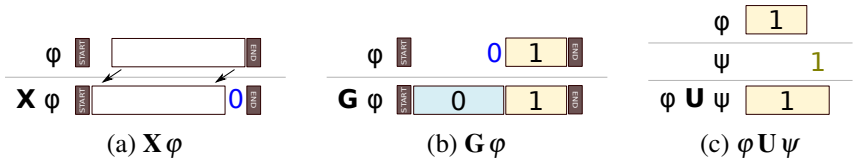

Figure 4.1: A graphical representation of the computation of three temporal operators on bitmaps

Figure 4.1a, and formalized in Algorithm 1.

Algorithm 1 Computing $\mathbf{X}\,a$

Require: Bitmap a
1: $out \leftarrow$ removeFirstBit(a)
2: addMany(out, 0, 1)
3: **return** out

To compute the vector for $\mathbf{G}\ \psi$, it suffices to find the smallest position i such that all subsequent bits are 1. In $B_{\mathbf{G}\ \psi}$, 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 $\mathbf{G}\,a$

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

$\overline{s} \models \varphi \ \mathbf{U} \ \psi$. 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.

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

As the dual of the operator **U**, the operator **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 $\pi^i, \pi^{i+1}, ... \pi^j$ satisfy φ when π^j satisfies ψ; and the second is simply $G\varphi$. 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 **U** b

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

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 **W** b

Require: Bitmaps a and b

$out \leftarrow \langle\,\rangle$
$p, a_0, a_1, b_0, b_1 \leftarrow 0$
while p < |a| **do**
  if $a_1 \leq p$ **then**
    $a_1 \leftarrow$ next(1, a, p)
  end if
  if $b_1 \leq p$ **then**
    $b_1 \leftarrow$ next(1, b, p)
  end if
  if $a_1 = -1$ or $b_1 = -1$ **then**
    break
  end if
  $nearest1 \leftarrow \min(a_1, b_1)$
  if $nearest1 > p$ **then**
    addMany(out, 0, $nearest1 - p$)
    $p \leftarrow nearest1$
    continue
  end if
  if $p = b_1$ **then**
    if $b_0 \leq b_1$ **then**
      $b_0 \leftarrow$ next(0, b, b_1)
      if $b_0 = -1$ **then**
        $b_0 \leftarrow |a|$
      end if
    end if
    addMany(out, 1, $b_0 - p$)
    $p \leftarrow b_0$
    continue
  end if
  if $a_0 \leq a_1$ **then**
    $a_0 \leftarrow$ next(0, a, a_1)
    if $a_0 = -1$ **then**
      $a_0 \leftarrow |a|$
    end if
  end if
  if $a_0 \geq b_1$ **then**
    addMany(out, 1, $b_1 - p + 1$)
    $p \leftarrow b_1 + 1$
  else
    addMany(out, 0, $a_0 - p + 1$)
    $p \leftarrow a_0 + 1$
  end if
end while
if $b_1 = -1$ **then**
  if $a_1 = -1$ **then**
    addMany(out, 0, $|a| - |out|$)
  else
    $last0 \leftarrow$ last(0, a)
    if $last0 = -1$ or $last0 < p$ **then**
      addMany(out, 1, $|a| - |out|$)
    else
      addMany(out, 0, $last0 - p + 1$)
      addMany(out, 1, $|a| - |out|$)
    end if
  end if
else if $a_1 = -1$ **then**
  copyTo(out, b, $|b| - b_1 - 1$, $|b| - b_1$)
end if
return out

Algorithm 6 Computing a **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: 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: 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: $a_0 \leftarrow \text{next}(0, b, a_1)$
34: **if** $a_0 = -1$ **then**
35: $a_0 \leftarrow |a|$
36: **end if**
37: **end if**
38: $nearest0 \leftarrow \min(a_0, b_0)$
39: addMany(out, 1, $nearest0 - p$)
40: $p \leftarrow nearest0$
41: **end while**
42: **if** $a_1 = -1$ and $b_1 \neq -1$ **then**
43: $last0 \leftarrow \text{last}(0, b)$
44: **if** $last0 = -1$ and $last0 < p$ **then**
45: addMany(out, 1, $|a| - |out|$)
46: **else**
47: addMany(out, 0, $last0 - p + 1$)
48: addMany(out, 1, $|a| - |out$)
49: **end if**
50: **else**
51: addMany(out, 0, $|a| - |out|$)
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 fill words in one operation. Computing the bitmap for a **F** or **G** operator requires a single such lookup for the entire trace.

Another interesting point is the fact that operators **F** and **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 **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`

Table 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

Table 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 0[2], 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 the 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 cycles 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`

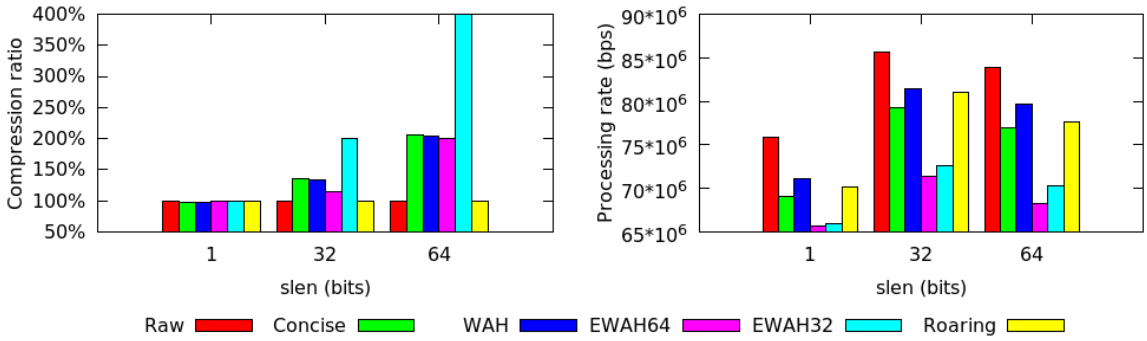

Figure 4.2: Bitmap generation with compression algorithms

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 formulæs in the previous experiments. Detailed results covering all the operators and formulæs 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

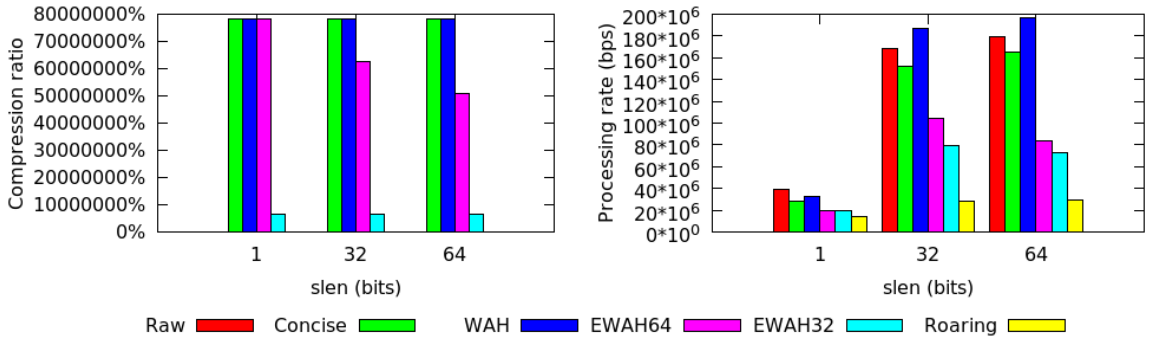

Figure 4.3: Comparison of compression ratio and processing rate for LTL formula 1, for various bitmap compression libraries and various values of *slen*

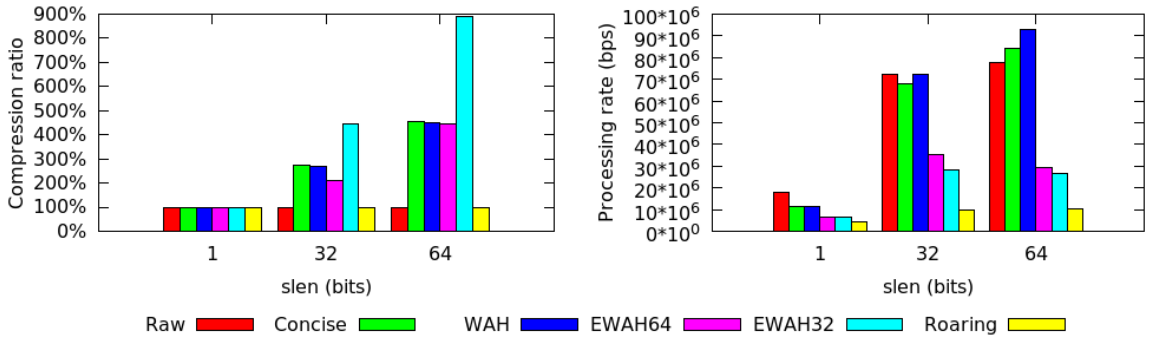

Figure 4.4: Comparison of compression ratio and processing rate for LTL formula 14, for various bitmap compression libraries and various values of *slen*

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. (Pel-

lizzoni 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 mutilcore 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 subformulæ 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 it 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 formulæ 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 formulæ 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 thair 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 parallellize the evaluation of large segments of btimaps on multiple machines.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Table 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

Table 4.7: Running time for the evaluation of LTL formulæ of Table 4.6, without the use of a compression library.

CHAPTER 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 formulæs' 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

CHAPTER 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.

	slen	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

Table A.1: Bitmap generation with compression algorithms

	slen	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

Table A.2: Calculation of $\neg s_0$ with compression algorithms

	slen	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

Table A.3: Calculation of $s_0 \wedge s_1$ with compression algorithms

	slen	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

Table A.4: Calculation of $s_0 \vee s_1$ with compression algorithms

	slen	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

Table A.5: Calculation of $s_0 \vee s_1$ with compression algorithms

	slen	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

Table A.6: Calculation of X s_0 with compression algorithms

	slen	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

Table A.7: Calculation of G s_0 with compression algorithms

	slen	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

Table A.8: Calculation of F s_0 with compression algorithms

	slen	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

Table A.9: Calculation of s_0Us_1 with compression algorithms

	slen	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

Table A.10: Calculation of $s_0\mathbf{W}s_1$ with compression algorithms

	slen	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

Table A.11: Calculation of $s_0\mathbf{R}s_1$ with compression algorithms

	slen	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

Table A.12: Formulæ 1 calculation with compression algorithms

	slen	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

Table A.13: Formulæ 2 calculation with compression algorithms

	slen	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

Table A.14: Formulæ 3 calculation with compression algorithms

	slen	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

Table A.15: Formulæ 4 calculation with compression algorithms

	slen	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

Table A.16: Formulæ 5 calculation with compression algorithms

	slen	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

Table A.17: Formulæ 6 calculation with compression algorithms

	slen	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

Table A.18: Formulæ 7 calculation with compression algorithms

	slen	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

Table A.19: Formulæ 8 calculation with compression algorithms

	slen	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

Table A.20: Formulæ 9 calculation with compression algorithms

	slen	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

Table A.21: Formulæ 10 calculation with compression algorithms

	slen	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

Table A.22: Formulæ 11 calculation with compression algorithms

	slen	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

Table A.23: Formulæ 12 calculation with compression algorithms

	slen	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

Table A.24: Formulæ 13 calculation with compression algorithms

	slen	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

Table A.25: Formulæ 14 calculation with compression algorithms

BIBLIOGRAPHY

1976. Broadcast teletext specification. Rapport.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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, éditeurs, *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.