Parallel Run-Time Verification

by

Shay Berkovich

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2012

© Shay Berkovich 2012

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Run-time verification is a technique to reason about a program correctness. Given a set of desirable properties and a program trace from the inspected program as an input, the monitor module verifies that properties hold on this trace. As this process is taking place at a run time, one of the major drawbacks of run-time verification is the execution overhead caused by a monitoring activity. In this thesis, we intend to minimize this overhead by presenting a collection of parallel verification algorithms. The algorithms verify properties correctness in a parallel fashion, decreasing the verification time by dispersion of computationally intensive calculations over multiple cores (first level of parallelism). We designed the algorithms with the intention to exploit a data-level parallelism, thus specifically suitable to run on Graphics Processing Units (GPUs), although can be utilized on multi-core platforms as well. Running the inspected program and the monitor module on separate platforms (second level of parallelism) results in several advantages: minimization of interference between the monitor and the program, faster processing for non-trivial computations, and even significant reduction in power consumption (when the monitor is running on GPU).

This work also aims to provide a solution to automated run-time verification of C programs by implementing the aforementioned set of algorithms in the monitoring tool called GPU-based online and offline Monitoring Framework (GooMF). The ultimate goal of GooMF is to supply developers with an easy-to-use and flexible verification API that requires minimal knowledge of formal languages and techniques.

Acknowledgements

I would like to thank University of Waterloo for giving me the chance.

Dedication

This is dedicated to Kate and Alyssa, two precious flowers, whom I am lucky to call my family.

This is also dedicated to my personal heroes Paolo Borsellino and Giovanni Falcone, who made the world a better place to live by sacrifising their own lives.

Table of Contents

List of Figures **ix**

1 Introduction **1**

1.1 Run-Time Verification and Execution Overhead Problem 1

1.2 Shortage in RV Frameworks for C and C++ 4

1.3 Contributions 5

1.4 Thesis Overview 6

2 Background **7**

2.1 Linear Temporal Logic (LTL) 7

2.2 3-Valued LTL 9

2.3 Graphics Processing Unit (GPU) Architecture 12

2.4 OpenCL Background 16

3 Algorithms Hierarchy **18**

3.1 Sequential Algorithm 18

3.2 Parallel Algorithms 20

3.2.1 History of LTL Property 20

3.2.2 Partial–Offload Algorithm 23

3.2.3 Finite-history Algorithm 24

3.2.4 Infinite-history Algorithm 27

3.3 Evaluation of Algorithms 30
3.3.1 Experiment Setup 30
3.3.2 Throughput Analysis 31
3.3.3 UAV Case Study 35

4 GooMF Framework **39**
4.1 Run-Time Verification Tools Overview 39
4.2 GooMF Architecture and Implementation 41
4.2.1 Online Monitoring 41
4.2.2 Offline Monitoring 43
4.2.3 Synchronous vs. Asynchronous Invocation 44
4.2.4 Other Features 45
4.2.5 Implementation Issues 46
4.3 GooMF and RiTHM 47
4.3.1 RiTHM Overview 47
4.3.2 GooMF as a Back End for RiTHM 51

5 Towards Parameterized Monitoring **53**
5.1 Introduction to Parameterized Monitoring 53
5.2 State of the Art in Parameterized Monitoring 58
5.3 Parallel Verification amid Parameterized Monitoring 60
5.3.1 Algorithms and Parameterized Monitoring 60
5.3.2 GooMF and Parameterized Monitoring 63
5.3.3 Future Extensions 64
5.4 Evaluation of Algorithms amid Parameterized Monitoring 65
5.4.1 Experiment Setup 65
5.4.2 Execution Time Analysis 68

6 Related Work **70**

7 Conclusions and Future Work **73**

7.1 Fitness for Properties 73

7.2 Further Research Directions 75

APPENDICES **77**

A GOOMF Online API Header **78**

B GOOMF Offline API Header **84**

C Configuration File Example **87**

References **97**

List of Figures

1.1 Run-time verification flow 2

2.1 The monitor for property $\varphi \equiv (\neg spawn \ \mathbf{U} \ init)$ 11
2.2 GPU architecture on the example of Nvidia GeForce GTX 480 13
2.3 GPU memory hierarchy 14
2.4 Example of a program in C and in OpenCL kernel. 17

3.1 Sequential algorithm flow 19
3.2 Monitors for properties, where (a) $\|\mathbb{H}^{\varphi}\| = 0$, and (b) $\|\mathbb{H}^{\varphi}\| = 1$. 20
3.3 Monitors for properties $\varphi_1 \equiv p \wedge (q \ \mathbf{U} \ r)$ and $\varphi_2 \equiv \Box(p \Rightarrow (q \ \mathbf{U} \ r))$. 22
3.4 Eliminating interdependencies for parallel execution 29
3.5 GPU-based architecture for experiments 30
3.6 Evaluation results 32
3.7 Throughput vs number of work items 34
3.8 Effect of the buffer and data sizes 35
3.9 Case study experiments 36

4.1 GooMF work–flow 42
4.2 Building blocks and data flow in RiTHM. 49
4.3 Example of a program and CFG. 50
4.4 Instrumented programs. 51

5.1 Monitors for a sample of parametric properties. 57
5.2 Algorithms execution times 67

Chapter 1

Introduction

In this chapter, we outline the motivation behind the current work. First part of the chapter regards the problem of verification overhead. As verification happens at a run time, the verification time adds up to the original execution time of the program under scrutiny. For obvious reasons, this might be a problem for real-time, embedded, mission-critical systems, and in general for systems with tight time constraints. Second part of the chapter describes the shortage of run-time verification tools for the software written in C and C++ . At the end of the chapter, we spell out the main contributions of this thesis and provide an overview of the thesis structure.

1.1 Run-Time Verification and Execution Overhead Problem

In computing systems, *correctness* refers to the assertion that a system satisfies its specification. *Formal verification* is a technique for checking such an assertion at design time. One instance of formal verification is *model checking*, where a state–space and a transition relation of a system are generated, and proof of correctness with respect to the system specification is automatically derived. The three major issues with model checking are (1) the infamous *state explosion problem*, (2) complexity of inferring the model from the inspected program, and (3) running the checking procedure on the model rather than on the real program. The first drawback is a combinatorial expansion of the state–space and usually observed for programs with a large set of variables, where model checking reaches its available memory limits rapidly. The second drawback might be problematic when the

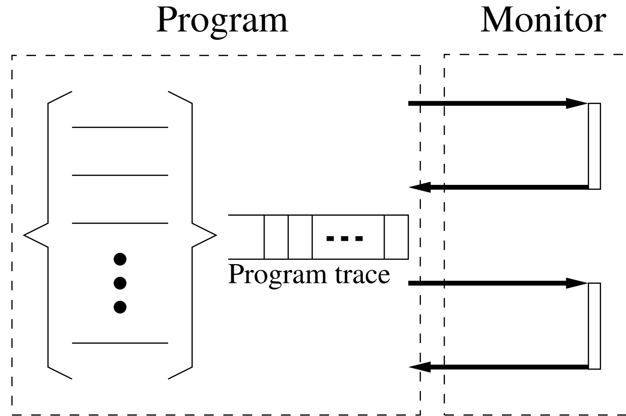


Figure 1.1: Run-time verification flow

system is large or complex, and the model derivation steps are not straightforward. The third issue is related to the second, when running the model checking on the wrong model can lead to erroneous conclusions.

As opposite to model checking, testing and debugging provide us with under-approximated confidence about the correctness of a system as these methods only check for the presence of defects for a limited set of scenarios. In general, most of the industrial projects tend to contain thousands of lines of code and complexity proportional to its size. For such software projects, it is virtually impossible to come up with an effective path coverage. Another aspect of software testing is the cost associated with it. The non-exhaustive nature of the testing makes it difficult to set a target testing level. To deal with the issue of software confidence, software reliability models were developed to estimate the amount of bugs in the system. However, this solution still bears an amount of uncertainty.

Run-time verification [15, 21, 36] refers to a technique where a *monitor* checks at a run time whether or not the execution of a system under inspection satisfies a given correctness property. Figure 1.1 shows generalized picture of run-time verification flow: while running, program under scrutiny generates a program trace, which, in turn, is inspected by the monitor under the set of previously specified properties. Whereas the general architecture is stable, different aspects may vary. For example, the length of the program trace aggregated before being verified can be one program state (yielding *event-triggered* monitoring) or more (*time-triggered* or *buffer-triggered* monitoring). The monitor can be implemented as a part of the program or as an external instance. The verification itself can be synchronous (program waits until it finishes) or asynchronous (program submits the trace and relies on the monitor to report back). There could be other variations, such as different formalisms

for property specifications, but the core remains the same: verification occurs at a run time on the current program trace. *Run-time monitoring* refers to more general monitoring process, where the program trace is monitored at a run time, but without specific goal of verifying the properties. In this thesis, we do not distinguish between two processes and use both names alternatively.

Run-time verification complements offline verification methods, such as model checking and theorem proving, as well as incomplete solutions, such as testing and debugging. As such, run-time verification takes the best of both worlds: exhaustive nature of the offline verification techniques, while applying them to the real program trace as in testing and debugging. This advantage obviously comes at a price. The main challenge in augmenting a system with run-time verification is dealing with its run-time *overhead*. It can come from several sources:

1. Invocation of a monitor
2. Calculation and evaluation of property predicates given a program state
3. Possible slowdown due to program instrumentation and program trace extraction
4. Possible interference between program and monitor, as monitor shares resources with a program

All of these sources may contribute to the overall overhead different amounts, depending on several factors, such as type of the inspected program, type of the properties being verified, and the way the monitor is implemented. This work aims at reducing the run-time overhead coming from sources 2 and 4. The advantage comes from the fact that we propose to use a separate platform for the monitor — thus the first meaning of *parallel run-time verification*. The interference between the monitor and the program (source 4) is minimized, since most of the computations are carried out on the GPU side (except some IO operations), so that the inspected program has an exclusive rights for the CPU as in the scenario without monitoring. As the program's demand to the CPU resources grows, the benefit of the proposed architecture becomes clearer. As an aside note, same architecture can be built with specialized hardware, such as FPGAs, but the cost of developing dedicated hardware for the monitoring is not comparable with the cost of the GPU, which is already a standard feature on the majority of machines. In addition, recent advances in the General-Purpose GPU (GPGPU) programming allow relatively easy reprogramming of the GPUs as monitored properties evolve. This is not the case with FPGAs and ASICs.

GPU-based monitoring also benefits form the engagement of multiple cores in the verification. This is particularly critical when the verification involves computationally intensive operations (overhead source 2), such as mathematical calculations, which are especially common in the properties designed for programs in safety-critical domain. In our approach, the burden of computations is distributed between the ample of processing elements offered by the GPUs — hence the second meaning of *parallel run-time verification*. Each processing element performs calculations on separate program state, thus reducing the verification time and, as shown later, decreasing the power consumption of the board the CPU and the GPU reside on.

Another significant problem introduced by run-time verification is the *overhead control*, as monitoring often introduces two types of defects to the system under scrutiny: (1) unpredictable execution and (2) possible bursts of monitoring intervention in program execution. Extensive research has been carried on these aspects [10, 11, 41], proposing time-aware instrumentation of the programs. These works suggest to buffer frequent bursts of events into the internal history to reduce the overhead coming from the excessive invocation. The common feature of these works is the element of buffering the events. Parallel run-time verification is particularly suitable for this scheme, because the power of GPU is only leveraged when program states are buffered and the trace processed in parallel on multiple cores. The synergy between time-triggered and parallel run-time verification resulted in a tool called Run-time Time-triggered Heterogeneous Monitoring (RiTHM), which employs time-aware instrumentation and smart buffering [10, 41] as a front end, and uses parallel verification algorithms from this work as a back-end verification engine.

1.2 Shortage in RV Frameworks for C and C++

C and C++ are de-facto primary languages for embedded, mission-critical, and real-time systems. Unfortunately, there is a lack of run-time verification tools for C programs. This is due to the fact that C and C++ source code is processed by compilers. Therefore, the ability to instrument an intermediate code representation is dependent on the compiler capabilities rather than on the original code. Java, on the other hand, offers an intermediate representation of its code called byte code, which enables instrumentation and manipulation with its structure even without having a source code.

Due to these limitations, the research community mainly focused on developing tools for Java, rather than for C and C++. The most prominent examples of the monitoring framework for Java are JavaMOP (monitoring-oriented variation of JavaPATH) [26], J-LO [7], JavaMaC [28], RuleR [4], Eagle [3], and Tracematches [1]. The majority of these

frameworks use the instrumentation engine AspectJ [27], which leverages the structure and the semantics of Java byte code to weave the monitoring code into specific points in the inspected program. The success of AspectJ pushes the research community farther towards using Java as a default target language for instrumentation-based tools and frameworks. Interestingly, some of the works in this area outline the monitoring architecture rather than specific language instance (Monitoring and Checking (MaC), Monitoring-Oriented Programming (MOP) [13]), but eventually choose Java for the implementation.

As a partial solution to this problem, we present GPU-based online and offline Monitoring Framework (GooMF). GooMF implements the verification algorithms presented in Chapter 3 and serves as a back-end verification engine for programs under examination. The developer of the program only has to specify the properties using one of the two formalisms (Ltl or FSM) and call the API functions for the actual verification. The user can switch between the verification algorithms merely by changing one parameter in the initialization function. Moreover, when the program specifications evolve, the only thing that needs to be changed is the properties file, whereas the source code of the program remains unmodified. GooMF is also capable of working in both blocking and non-blocking mode and offers some other features discussed in Section 4.2. As a result, the user has full control over the verification, while, at the same time, the implementation details remain hidden behind clean and minimalistic API. We designed the tool believing that simplicity-of-use is the main factor to attract users.

GooMF is only a partial solution to the deficit of tools for C, because it represents only the back end. Without automatic instrumentation of the inspected program, the responsibility of calling the API functions lays on the developer. The first steps towards solving this issue are described in Section 4.3, where we wrap the instrumentation and verification parts in one tool RiTHM. Also, several instrumentation tools for C and C++ appeared recently, mostly based on GCC and LLVM plug-ins. It might be appealing to try to interface these tools with GooMF in one monitoring framework.

1.3 Contributions

This thesis presents a research for practical run-time verification over many-core platforms, and an application of this research in the novel verification tool GooMF. The key contributions of this work are driven by motivation presented in the previous section, and can be summarized in two main parts:

- One sequential and three parallel algorithms for run-time verification. These algo-

rithms can run on CPU or GPU or other many-core platform that supports OpenCL standard. The algorithms form a hierarchy based on the degree of GPU involvement in the verification process and, consequently, on the suitability for the specific property.

- Implementation of these algorithms in GooMF— easy-to-use and flexible verification framework that allows C/C++ developers to state the desirable program properties and verify them at run time using one of four presented verification algorithms. Besides being a first GPU-based monitoring engine, GooMF fills the shortage in run-time verification tools for C and C++.

1.4 Thesis Overview

Chapter 2 provides background information on the formalisms used to specify the properties and on the generation of the sound monitors for these properties. It also contains high-level overview of the GPU architecture and the implementation language for the parallel algorithms.

Chapter 3 presents four verification algorithms that, given a property and a program trace, output verification result. The algorithms are introduced in the order of the GPU engagement. This chapter also contains experimental results for various performance measurements, as well as an evolved case study, which demonstrates the practicality of the algorithms.

Chapter 4 describes the author's work on the implementation of GooMF— the verification framework that has in its core four algorithms from Chapter 3. The chapter elaborates on the architecture and the most noteworthy features of the tool, as well as reports about interesting implementation issues encountered during the development.

Chapter 5 offers insight on the parameterized monitoring, its advantages over the regular, non-parameterized monitoring, and the difficulties that it introduces. After the general discussion, the chapter describes how parameterized monitoring modifies the algorithms and how GooMF handles parameterized monitoring.

Finally, Chapter 6 discusses related work, and Chapter 7 concludes the thesis with the discussion about the applicability of parallel run-time verification and opportunities for future research.

Chapter 2

Background

In this chapter, we present the body of knowledge the current work is based upon. Namely, LTL formalism (Section 2.1), which serves as a main specification language for the properties used in this work. It is also one of the two formalisms accepted by GooMF. LTL_3 (Section 2.2) is an extension of LTL and is particularly suitable for the run-time monitoring, because of additional logic value "?". The intuition behind this logic is as follows: while monitoring a program at a run time, only finite sequence of the program states is available at any given time moment. While some of the properties are satisfied ($\top$) / violated ($\bot$) given this program trace, other properties will need additional (possibly infinite) sequence of program states to converge. This intermediate state of the evaluation is represented by logical value "?".

If the reader is unfamiliar with the concept of GPGPU and heterogeneous programming, the last two sections will fill the gaps. Section 2.3 describes the memory architecture of Graphics Processing Units — our target device for the parallel algorithms. It also sheds light on some performance issues related to the memory hierarchy of the verification platform. Section 2.4 provides the reader with the required information about OpenCL— language for writing programs running on heterogeneous platforms. In GooMF, OpenCL code is used to express the core functionality of the verification algorithms.

2.1 Linear Temporal Logic (LTL)

Linear temporal logic (LTL) is a popular formalism for specifying properties of (concurrent) programs. The set of well-formed linear temporal logic formulas is constructed from a set

of atomic propositions, the standard Boolean operators, and temporal operators. Precisely, let AP be a finite set of *atomic propositions* and $\Sigma = 2^{AP}$ be a finite *alphabet*. A letter a in Σ is interpreted as assigning truth values to the elements of AP; i.e., elements in a are assigned true (denoted $\top$), and elements not in a are assigned false (denoted $\bot$). A *word* is a finite or infinite sequence of letters $w = a_0 a_1 a_2 \ldots$, where $a_i \in \Sigma$ for all $i \geq 0$. We denote a set of all finite words over Σ by Σ^* and a set of all infinite words by Σ^ω. For a finite word u and a word w, we write $u \cdot w$ to denote their *concatenation*. We write ω_i to denote the suffix of ω starting at a_i.

Definition 1 (LTL Syntax) LTL *formulas are defined inductively as follows:*

$$\varphi ::= \top \mid p \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \bigcirc\varphi \mid \varphi_1 \mathbf{U} \varphi_2$$

where $p \in \Sigma$*, and,* $\bigcirc$ *(next) and* $\mathbf{U}$ *(until) are temporal operators.* ■

An interpretation for an LTL formula is an infinite *word* $\omega = x_0, x_1, x_2, \ldots$ over the alphabet 2^{AP}; i.e. a mapping from the naturals to 2^{AP}. In this thesis ω will usually denote an infinite word as opposite to finite u.

Definition 2 (LTL Semantics) *Let* $w = a_0 a_1 \ldots$ *be an infinite word in* Σ^ω*,* i *be a non-negative integer, and* $\models$ *denote the* satisfaction *relation. Semantics of* LTL *is defined inductively as follows:*

$$
\begin{array}{lll}
w, i \models \top & & \\
w, i \models p & \textit{iff} & p \in a_i \\
w, i \models \neg\varphi & \textit{iff} & w, i \not\models \varphi \\
w, i \models \varphi_1 \vee \varphi_2 & \textit{iff} & w, i \models \varphi_1 \ \vee \ w, i \models \varphi_2 \\
w, i \models \bigcirc\varphi & \textit{iff} & w, i+1 \models \varphi \\
w, i \models \varphi_1 \, \mathbf{U} \, \varphi_2 & \textit{iff} & \\
\quad \exists k \geq i : w_k \models \varphi_2 \ \wedge \ \forall i \leq j \leq k : w, j \models \varphi_1. & &
\end{array}
$$

In addition, $w \models \varphi$ *holds* *iff* $w, 0 \models \varphi$ *holds.* ■

An LTL formula φ defines a set of words (i.e., a *language* or a *property*) that satisfies the semantics of that formula. We denote this language by $L(\varphi)$. We note that the results presented in this work remain valid if the logic LTL is built over atomic propositions as well as over letters. For simplicity, we introduce abbreviation temporal operators. $\Diamond\varphi$ (*eventually* φ) denotes $\top \, \mathbf{U} \, \varphi$, and $\Box\varphi$ (*always* φ) denotes $\neg\Diamond\neg\varphi$. For instance, formula $\Box(p \Rightarrow \Diamond q)$ means that 'it is always the case that if proposition p holds, then eventually proposition q holds'. One application of this formula is in reasoning about non-starvation in mutual exclusion algorithms: 'if a process requests entering a critical section, then it eventually is granted to do so'.

In order to reason about a correctness of programs with respect to an LTL property, we describe a program in terms of its state space and transitions. A *program* is a tuple $P = \langle S, T \rangle$, where S is the non-empty *state space* and T is a set of *transitions*. A transition is of the form (s_0, s_1), where $s_0, s_1 \in S$. A state of a program is normally obtained by valuation of its variables and transitions are program instructions. In this context, an atomic proposition in a program is a Boolean predicate over S (i.e., a subset of S).

We define a *trace* of a program $P = \langle S, T \rangle$ as a finite or infinite sequence of subsets of atomic propositions obtained by valuation of program variables (i.e., program states). Thus, a program trace can be defined as $\sigma = s_0 s_1 s_2 \ldots$, so that $s_i \in S$ and each $(s_i, s_{i+1}) \in T$, for all $i \geq 0$. A program trace σ *satisfies* an LTL property φ (denoted $\sigma \models \varphi$) iff $\sigma \in L(\varphi)$. If σ does not satisfy φ, we say that σ *violates* φ. A program P satisfies an LTL property φ (denoted $P \models \varphi$) iff for each trace σ of P, $\sigma \models \varphi$ holds. For example, consider the following piece of code:

```
x := 10;
while (x <= 100) {
x := x + 2;
}
```

It is straightforward to observe that this code satisfies the following properties: $\Box p$ and $\Diamond q$, where $p \equiv (x \geq 10)$ and $q \equiv (x = 100)$.

2.2 3-Valued LTL

Implementing run-time verification boils down to the following problem: given the current program *finite* trace $\sigma = s_0 s_1 s_2 \ldots s_n$, whether or not σ belongs to a set of words defined by some property φ. This problem is more complex than it looks, because LTL semantics

is defined over infinite traces and a running program can only deliver a finite trace at a verification point. For example, given a finite trace $\sigma = s_0 s_1 \ldots s_n$, it may be impossible for a *monitor* to decide weather the property $\Diamond p$ is satisfied. To clarify this issue, consider the case where for all i, $0 \leq i \leq n$, $p \notin s_i$, then the program still has the chance to satisfy $\Diamond p$ in the future. In other words, a monitor can decide whether a property is violated only if all the reachable states of the program under inspection cannot possibly satisfy the property. Respectively, the monitor reports satisfaction of the property if any sequence of program states that follows σ satisfies the property.

To formalize satisfaction of LTL properties at run time, in [6], the authors propose semantics for LTL, where the evaluation of a formula ranges over three values '$\top$', '$\bot$', and '?' (denoted LTL$_3$). The latter value expresses the fact that it is not possible to decide on the satisfaction of a property, given the current program finite trace.

Definition 3 (LTL$_3$ semantics) *Let $u \in \Sigma^*$ be a finite word. The truth value of an* LTL$_3$ *formula φ with respect to u, denoted by $[u \models \varphi]$, is defined as follows:*

$$[u \models \varphi] = \begin{cases} \top & \textit{if } \forall w \in \Sigma^\omega : u \cdot w \models \varphi, \\ \bot & \textit{if } \forall w \in \Sigma^\omega : u \cdot w \not\models \varphi, \\ ? & \textit{otherwise.} \blacksquare \end{cases}$$

Note that the syntax $[u \models \varphi]$ for LTL$_3$ semantics is defined over finite words as opposed to $u \models \varphi$ for LTL semantics, which is defined over infinite words. For example, given a finite program trace $\sigma = s_0 s_1 \cdots s_n$, property $\Diamond p$ holds iff $s_i \models p$, for some i, $0 \leq i \leq n$ (i.e., σ is a good prefix). Otherwise, the property evaluates to ?. Thus, one can reason about satisfaction of properties in LTL$_3$ through the notion of *good/bad* prefixes defined below. This is the reason why to the great extent monitor operation may be entirely based on LTL_3 semantics: return $\top$ once the property is satisfied independently of the further input; return $\bot$ once the property is violated independently of the further input; return ? if both outcomes are possible.

Definition 4 (Good and Bad Prefixes) *Given a language $L \subseteq \Sigma^\omega$ of infinite words over Σ, we call a finite word $u \in \Sigma^*$*

- *a* good prefix *for L, if $\forall w \in \Sigma^\omega : u \cdot w \in L$*
- *a* bad prefix *for L, if $\forall w \in \Sigma^\omega : u \cdot w \notin L$*

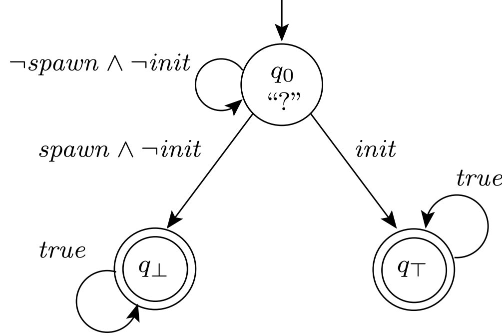


Figure 2.1: The monitor for property $\varphi \equiv (\neg \mathit{spawn}\ \mathbf{U}\ \mathit{init})$

- *an* ugly prefix *otherwise.* ■

In order to declare a verification verdict by a monitor more efficiently, it is advantageous to recognize good and bad prefixes as early as possible.

Definition 5 (Minimal Good/Bad Prefixes) *A bad (good) prefix u for language $L \subseteq \Sigma^\omega$ is called* minimal *if each strict prefix of u is not a bad (good) prefix.* ■

Implementing run-time verification for an LTL$_3$ property involves synthesizing a monitor that realizes the property. In [6], the authors introduce a stepwise method that takes an LTL$_3$ property φ as input and generates a deterministic finite state machine (FSM) $\mathcal{M}^\varphi$ as output. Intuitively, simulating a finite word u on this FSM reaches a state that illustrates the valuation of $[u \models \varphi]$.

Definition 6 (Monitor) *Let φ be an* LTL$_3$ *formula over alphabet Σ. The* monitor $\mathcal{M}^\varphi$ *of φ is the unique FSM $(\Sigma, Q, q_0, \delta, \lambda)$, where Q is a set of states, q_0 is the initial state, δ is the transition relation, and λ is a function that maps each state in Q to a value in $\{\top, \bot, ?\}$, such that:*

$$[u \models \varphi] = \lambda(\delta(q_0, u)). \blacksquare$$

For example, consider the property $\varphi \equiv (\neg \mathit{spawn}\ \mathbf{U}\ \mathit{init})$ (i.e., a thread is not spawned until the program is initialized). The corresponding monitor is shown in Figure 2.1. The proposition *true* denotes the set AP of all propositions. Examples of monitors appear also in Figures 3.2 and 3.3. We use the term a *conclusive state* to refer to monitor states $q_\top$

and $q_\bot$; i.e., states where $\lambda(q) = \top$ and $\lambda(q) = \bot$, respectively. Other states are called an *inconclusive state*. A monitor $\mathcal{M}^\varphi$ is constructed in a way that it recognizes minimal good and bad prefixes of $L(\varphi)$. Hence, if $\mathcal{M}^\varphi$ reaches a conclusive state, it stays in this *trap* state. If a monitor has only ugly inconclusive state, all monitoring activity becomes obsolete, which leads us to the notion of *monitorable* properties.

Definition 7 (Monitorable Property (Bauer et al. 2011)) *An* LTL$_3$ *property* φ *is* monitorable *if* $L(\varphi)$ *has no ugly prefixes. We denote the set of all monitorable* LTL$_3$ *properties by* LTL_3^{mon}. ■

Definition 8 (Monitorable Property) *An* LTL$_3$ *property* φ *is* monitorable *if there exists some good/bad prefix for* $L(\varphi)$*. We denote the set of all monitorable* LTL *properties by* LTL_3^{mon}.

In other words, a property is *monitorable* if for every finite word, there still exists a (possibly) infinite continuation that will determine whether the property is violated or satisfied.

Our notion of monitorable properties in Definition 8 is more relaxed than the one in [6]; i.e., our definition allows existence of ugly prefixes as long as useful information can be extracted from the input sequence. To demonstrate this difference, consider LTL property $\varphi \equiv \Box\Diamond q$. The monitor for this property does not include a trap state $q_\top$ or $q_\bot$. Hence, φ is non-monitorable according to Definition 8. On the contrary, property $\psi \equiv p \vee \Box\Diamond q$ is monitorable according to our definition, as given a prefix $u \in \Sigma^*$ and depending upon the value of proposition p in the current state, the monitor can reach a conclusive state if proposition p is observed.

2.3 Graphics Processing Unit (GPU) Architecture

In recent years, the area of heterogeneous computing has taken a giant leap forward. As processing power of CPUs is restrained by Moore's law, energy consumption and thermal considerations has become a design focus for future processors. The scaling in the computing power is achieved mainly through the multiplication of cores, and, as a result, the current CPU architecture reminds more and more of that of the GPU. As we discuss the GPU architecture, we denote the CPU as the *host*, and GPU as the *device*, although device could actually be any computing device, including the same or other CPU. In the modern

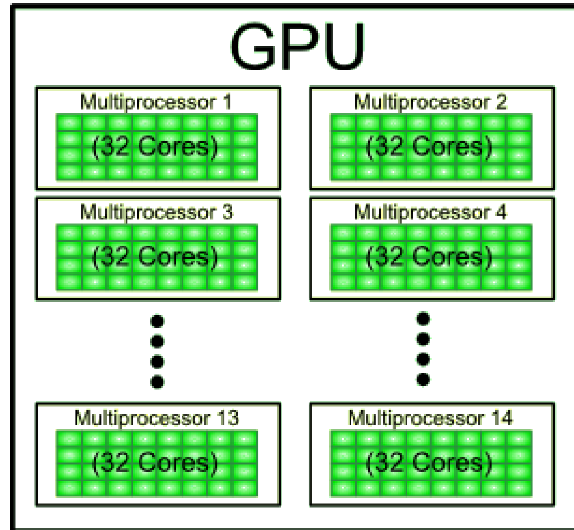


Figure 2.2: GPU architecture on the example of Nvidia GeForce GTX 480

architectures, a CPU commonly has 4 to 8 fast, flexible cores clocked at 2-3Ghz, whereas a GPU has hundreds of relatively simple cores clocked at about 1Ghz.

In Chapter 3, our proposed verification algorithms leverage the power of the GPUs by evaluating the program trace in parallel on multiple stream cores the GPU offers. Among the selection of many-core architectures, the GPU platform is particularly appealing, because it conforms to the Single Instruction Multiple Data (SIMD) pattern, which is ideal for our algorithms. Thus, for the informative and efficient estimation of the algorithms performance, it is vital to understand the GPU memory hierarchy.

Figure 2.2 illustrates the architecture and Figure 2.3 illustrates the memory layout of the most existing GPUs. The host communicates with the device via the *global memory*. Every buffer sent from the host to the device automatically loaded to the global memory and from there can be dispersed over other memories. The same applies to the opposite direction: to transfer the computational results from the device, the results must first reside in the global memory. *Constant memory* is a subset of global memory that is not a subject for change and thus is perfect for caching. Therefore, if the data is read-only, it is advantageous to load it to the constant memory to get faster access. Global memory is accessible by all the threads across multiple compute units, and thus, the slowest memory type in the hierarchy. Sometimes, when the number of memory accesses is significant, it is better to copy the data first to the *local memory* and work with it.

Local memory (shared memory in Nvidia architectures) is faster than global memory,

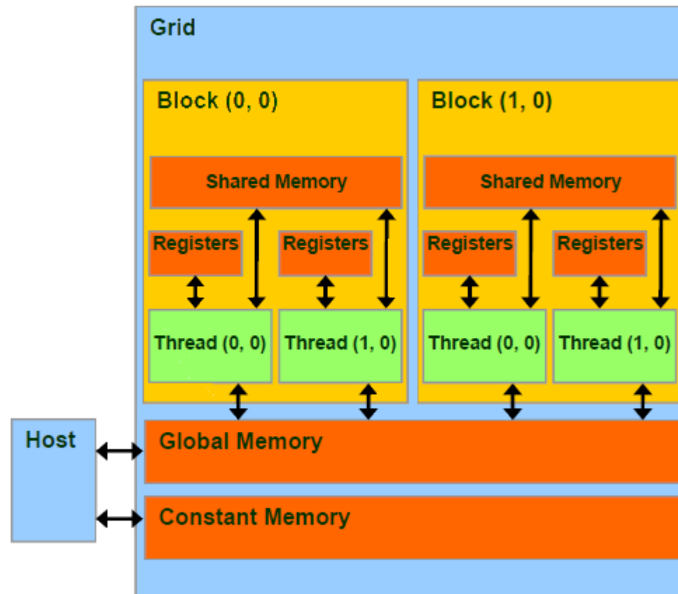


Figure 2.3: GPU memory hierarchy

and in general, local to the *compute unite*. As Figure 2.2 suggests, every device consists of one or more compute units, which can operate autonomously. In CPUs, compute unit corresponds to one CPU core. In GPUs, compute unit (called also *streaming multiprocessor*), comprises *workgroup* — multiple worker threads (called also *streaming processor*, *processing element*, *GPU core*), which can share the same local memory. This memory is particularly useful, since the memory model does not encourage global synchronization. Instead, the synchronization should be carried out on the workgroup level. In the modern architectures, the amount of local memory available reaches 64 KB per streaming multiprocessor (AMD Radeon HD7970).

The last type is the *private memory*, which is, naturally, private to the worker thread. As this is the fastest memory in the hierarchy, it is usually restricted by relatively small number of registers available per thread. For example, in Nvidia Fermi architecture, there are 32K of 32-bit registers per streaming multiprocessor, which means up to 63 registers per thread. This memory is used for private and temporary calculations or for data that requires especially frequent access. If an application requires more private memory than available, it "spills" to the local, and from there to the global memory.

There are several key steps in the algorithms that might be affected by the memory

parameters:

1. Transfer of the program states from the host to the device. The amount of memory copied from the host to the device for every algorithm iteration is affected by two factors: number of program states in the buffer and the size of the program state. The transfer volume, thus, can reach high numbers and high transfer speed is crucial for algorithms throughput.

2. Transfer of the bound property parameters from the host to the device. When the parametric monitoring is in use (Chapter 5), verifier has to know the values of the bound variables. The amount of memory copied from the host to the device for every algorithm iteration is affected by two factors: number of the parametric properties and the size of the parameter values.

3. Transfer of the monitor state from the device to the host. After the algorithm iteration is over, the result is to be transferred back for the final monitoring state update. The amount of the memory is negligible comparing to the previous steps.

4. During the verification phase on the device, multiple worker threads access the memory simultaneously to read the program states, current monitor states, and parameter values. The number of memory accesses is equal or greater of the number of program states in the buffer. Therefore, the memory access speed is a key factor of the overall algorithm performance.

To summarize the effect of memory layout on the algorithms performance: for the first step, the speed of the memory transfer from the host to the device is crucial, as we want the memory volume (buffer of the program states) to be scalable. The considerations about global memory size are insignificant, since modern GPUs offer up to 5 GB of the global memory and the verification buffer is unlikely to reach this size. For the fourth step, the memory access speed is critical. It may be beneficial to copy the frequently accessed data to the local or even private memory before the computation stage. In essence, there is a tradeoff between the cost of copying the data between the memories and the speedup in the access speed.

During the last several years, there is a tendency in the industry towards unifying CPU and GPU memory spaces. The prominent example of this trend is AMD Fusion architecture (recently renamed to Heterogeneous Systems Architecture (HSA)), which combines a regular processor execution and the GPU functions into a single die. There are already some papers being published that report on performance comparison between different

architectures [39]. It will be intriguing to see how these architecture changes affect the memory transfer rates.

2.4 OpenCL Background

With the recent developments in heterogeneous computing, it became clear that a new, cross-platform language is required, that will ideally run in different environments (single-core, multi-core, many-core) with minimal portability effort. This language should facilitate the use of the GPU and general-purpose GPU programming, but also should work on other many-core platforms after recompiling the source code. As a result, several years ago emerged a need in the cross-platform language that will unify the vendors architectures and will offer the unified programming model for the parallel-intensive tasks as well as for sequential ones.

In 2008, Khronos group[1] introduced OpenCL— language based on C99 (ISO/IEC 9899:1999) standard with some restrictions and several specialized additions. Particularly, OpenCL prohibits function pointers, custom header files (so that all the code resides in one file), recursion, and bit-arrays. On the other hand, it introduces new vector data types (to facilitate the use of the stream processors), built-in synchronization mechanisms and rich library of mathematical functions. It also provides designated memory region qualifiers that correspond to the memory hierarchy from the previous section: `__global`, `__constant`, `__local` and `__private`.

OpenCL code can be of two types: host code (running on the host) and device code, also called *kernel* (running on the compute device). The essence of the OpenCL is in task-oriented programming, where parallel-intensive computations are wrapped into tasks and sent to the device. The responsibility of the host code is to manage compute devices, command queues, compute tasks, and potentially synchronize between different kernels (global synchronization). Luckily, in GooMF the majority of these operations is performed during the initialization phase. The compiler (usually vendor-specific) compiles the kernel to the device code. Examples of SDKs that provide such compiler and include OpenCL API implementation are: AMD Accelerated Parallel Processing (APP) SDK and NVIDIA OpenCL SDK. Figure 2.4 shows an example of simple vector summation program written in C (Figure 2.4a) and kernel in OpenCL (Figure 2.4b).

OpenCL adheres a *relaxed memory consistency model* [22], which (1) allows work items to access data within private memory; (2) permits sharing of local memory by work

[1] http://www.khronos.org/opencl/

```
void vector_summation_C
            (const float* src_a,
            const float* src_b,
            float*  res,
            const int num)
{
   for (int i = 0; i < num; i++)
      res[i] = src_a[i] + src_b[i];
}
```

(a) Adding two vectors in C

```
__kernel void vector_summation_opencl
            (__global const float* src_a,
            __global const float* src_b,
            __global float* res,
            const int num)
{
  const int idx = get_global_id(0);

  if (idx < num)
    res[idx] = src_a[idx] + src_b[idx];
}
```

(b) Adding two vectors as an OpenCL kernel

Figure 2.4: Example of a program in C and in OpenCL kernel.

items during the execution of a work–group; (3) only guarantees memory consistency after various synchronization points, such as barriers within kernels and queue events; (4) does not offer communication and synchronization between different work–groups as consistency of variables across a collection of workgroup items is not guaranteed; and (5) defines and satisfies data dependencies via work queues and atomic operations.

There are multiple mechanisms in OpenCL that facilitate the use of the memory model. For internal synchronization, there are multiple atomic functions, such as atomic_min(), atomic_max(), etc. As the memory model stresses the in-group synchronization, applying those functions to the parameters residing in the global memory results in extremely high latency (see Subsection 4.2.5 for our experience). Additional synchronization mechanisms are local memory barriers, placed to ensure that data is not overwritten in the shared memory before all of the work–items in a work–group have accessed it. The barrier can be set either on load access, store access, or both [40]. This coordinated loading and sharing of data reduces the number of slow global memory accesses. In addition to in-group synchronization, OpenCL offers global synchronization between the kernels, by employing an event-based approach. Compute kernel can register to the event fired upon completion of other kernel. Moreover, the number of events required (or fired) by kernel can vary.

Chapter 3

Algorithms Hierarchy

In this chapter, we introduce a set of algorithms for verifying LTL_3^{mon} properties at run time. Three of the algorithms employ Single Instruction Multiple Data (SIMD) parallelism and, as a result, are particularly suitable to run on Graphics Processing Unit or other many-core platform. The presentation of the algorithms is sequenced in the order of GPU involvement in the verification process. We start from Section 3.1, which describes CPU-based **sequential** counterpart of the parallel algorithms. Then in Subsection 3.2.1 we introduce the notion of *property history* as a performance metric of **finite-history** parallel algorithm. Finally, in Section 3.2, we present three parallel algorithms: **partial–offload** (Subsection 3.2.2), finite-history (Subsection 3.2.3) and **infinite-history** (Subsection 3.2.4). Comparing to the finite-history algorithm, the infinite-history algorithm is more general but less efficient, and suitable for properties with both infinite and finite history lengths. Section 3.3 concludes the chapter with the evaluation results.

3.1 Sequential Algorithm

Before discussing the parallel algorithms, we first describe their **sequential** counterpart. It will serve us two purposes: (1) better understanding of other algorithms, and (2) representing a reference point in algorithms performance comparison in Section 3.3. Being sequential by nature, this algorithm performs the verification totally on the CPU side. After the verification buffer is flushed, for every program state the algorithm performs the following steps:

1. New program state s_i is received.

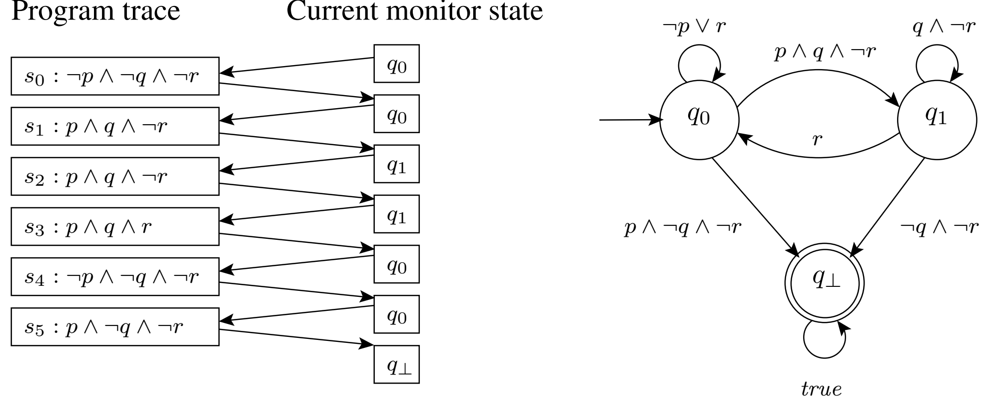


Figure 3.1: Sequential algorithm flow

2. Based on s_i, perform calculations to map predicates (the LTL formula consists of) to either $true$ or $false$.

3. Based on the values of the predicates, make the appropriate transition if needed.

More formally, let $P = \langle S, T \rangle$ be a program and $\sigma = s_0 s_1 s_2 \ldots$ be a trace of P. Also, let φ be a property in LTL_3^{mon} and $\mathcal{M}^\varphi = (\Sigma, Q, q_0, \delta, \lambda)$ be its monitor, which is intended to inspect program P. One can build a sequential implementation of $\mathcal{M}^\varphi$ by employing a sequence of conditional statements as follows. By every change of program state s_i, $i \geq 0$, monitor $\mathcal{M}^\varphi$ calculates $q_{i+1} = \delta(q_i, u_i)$, where u_i is a mapping of program state s_i onto an alphabet Σ. The output of each algorithm step is $\lambda(q_{n+1})$.

Figure 3.1 illustrates the algorithm run on the monitor FSM received from property $\phi = \Box(p \rightarrow (q \mathbf{U} r))$. $s_0, s_1, s_2, s_3, s_4, s_5$ are the program states in the buffer with corresponding predicate values in the first column. q_0, q_1, $q\bot$ are the states of the monitor with middle column showing the current monitor state. As can be seen, the monitoring run follows the following path on the FSM: $q_0 \rightarrow q_0 \rightarrow q_1 \rightarrow q_1 \rightarrow q_0 \rightarrow q_0 \rightarrow q_\perp$.

The time required to evaluate n program states can be described by the following equation:

$$T_{seq} = n.(t_{calc} + t_{branch}) \tag{3.1}$$

where t_{branch} is the time spent in a conditional statement to compute the next state of $\mathcal{M}^\varphi$ and t_{calc} is the (proposition to alphabet) translation time. For instance, consider the

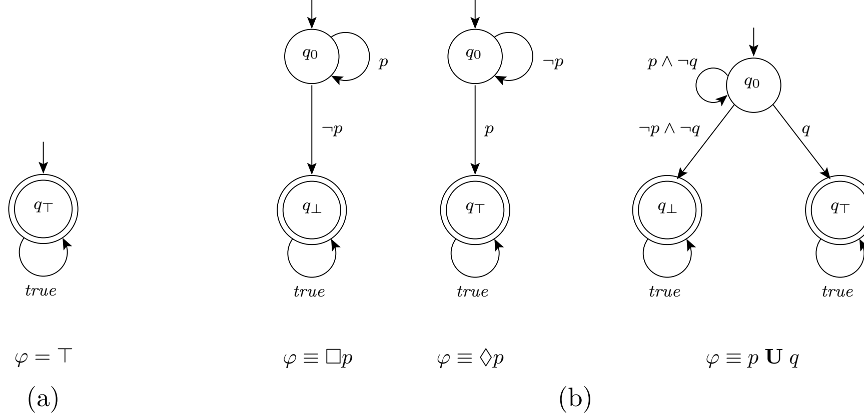


Figure 3.2: Monitors for properties, where (a) $\|\mathbb{H}^\varphi\| = 0$, and (b) $\|\mathbb{H}^\varphi\| = 1$.

LTL_3^{mon} property $\varphi \equiv p \wedge (q \, \mathbf{U} \, r)$ in Figure 3.3a, where $p \equiv (\log(x) \leq 1.0)$, $q \equiv (\sin(y) \leq 2)$, and $r \equiv (\tan(z) \geq 3.0)$. Here, x, y, and z are program variables and p, q, and r are atomic propositions over the program states. Hence, t_{calc} involves calculation of $\log(x)$, $\sin(y)$, and $\tan(z)$.

t_{calc} and, consequently, T_{sec}, increase linearly with computational complexity of (proposition to alphabet) translation. Now, if we are to evaluate multiple LTL_3^{mon} properties (say m), the algorithm complexity becomes $\mathcal{O}(m.n)$ and this linear increase appears in all evaluations:

$$T_{seq} = n.\Big(\sum_{j=1}^{m}(t^j_{calc} + t^j_{branch})\Big) \tag{3.2}$$

Our parallel algorithms described in this section tackle this linear increase by distributing the calculation load across multiple processing units.

3.2 Parallel Algorithms

3.2.1 History of LTL Property

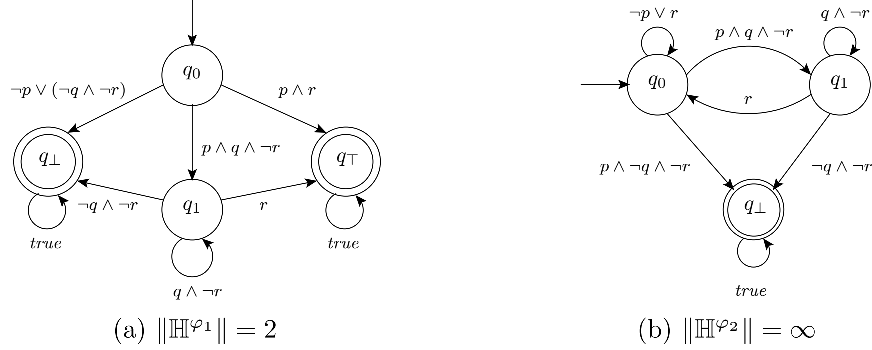
The construction of a monitor for an LTL_3 property as described in Section 2.2, allows evaluation of a prefix with respect to the property by observing state-by-state changes in

the program under scrutiny. In other words, the monitor processes each change of state in the program individually. Such state-by-state monitoring is inherently sequential. The core of our idea to leverage parallel processing in run-time verification of a property is to buffer a finite program trace and then somehow assign one or more sub-trace to a different monitoring processing units. We assume that the length of the program trace is given as an input parameter by the system designer. This length may depend on factors such as: (1) hardware constraints (e.g., memory limitations), (2) tolerable detection latency (i.e., the time elapsed since a change of state until a property violation is detected), or (3) the sampling period in time-triggered run-time verification [10] (e.g., for scheduling purposes in real time systems).

Although our idea seems simple and intuitive, its implementation may become quite complex. This is due to the fact that truthfulness of some LTL_3 properties is sensitive to the causal order of state changes. For example, monitoring property $\varphi \equiv \Box(p \Rightarrow \Diamond q)$ has to take the order of occurrence of atomic propositions p and q into account. In other words, occurrence of q is of interest only if p has occurred before. This leads us to our notion of *history* carried by an LTL_3 formula. This history can be seen as a measure of "parallelizability" of the property; it can also serve as a performance metric for the algorithm introduced in 3.2.3.

One can observe that each state of $\mathcal{M}^\varphi$ for a property φ in LTL_3^{mon} represents a different logical step in evaluation of φ. Thus, the structure of $\mathcal{M}^\varphi$ characterizes the temporal complexity of φ. Running a finite program trace on $\mathcal{M}^\varphi$ results in obtaining a sequence of states of $\mathcal{M}^\varphi$. This sequence encodes a history (denoted $\mathbb{H}^\varphi$) of state changes in the program under inspection and consequently in the monitor. In the context of monitoring, we are only concerned with the longest minimal good or bad prefixes in $L(\varphi)$. Thus, the length of the history for the property φ (denote $\|\mathbb{H}^\varphi\|$) is the number of steps that the monitor needs at most to evaluate φ. For example, for the trivial property $\varphi \equiv \top$, we have $\|\mathbb{H}^\varphi\| = 0$, since φ is evaluated in 0 steps (see Figure 3.2a). Figure 3.2b, shows three properties and their corresponding monitors, where $\|\mathbb{H}^\varphi\| = 1$. Figure 3.3a demonstrates the monitor of property $\varphi_1 \equiv p \wedge (q\ \mathbf{U}\ r)$, where $\|\mathbb{H}^{\varphi_1}\| = 2$. This is because the length of the longest path from the initial state q_0 to a conclusive state is 2 and the monitor has no cycles. transition if, for example, $p \wedge r$ is observed in the first letter of the input word.

Definition 9 (History) *Let φ be a property in LTL_3^{mon} and $w \in \Sigma^\omega$ be an infinite word. The* history *of φ with respect to w is the sequence of states $\mathbb{H}^\varphi_w = q_0 q_1 \ldots$ of $\mathcal{M}^\varphi = (\Sigma, Q, q_0, \delta, \lambda)$, such that $q_i \in Q$ and $q_{i+1} = \delta(q_i, w_i)$, for all $i \geq 0$.*



(a) $\|\mathbb{H}^{\varphi_1}\| = 2$ (b) $\|\mathbb{H}^{\varphi_2}\| = \infty$

Figure 3.3: Monitors for properties $\varphi_1 \equiv p \wedge (q\ \mathbf{U}\ r)$ and $\varphi_2 \equiv \Box(p \Rightarrow (q\ \mathbf{U}\ r))$.

Definition 10 (History Length) *Let φ be a property in LTL_3^{mon} and $w \in \Sigma^\omega$ be an infinite word. The* history length *of φ with respect to w (denoted $\|\mathbb{H}^\varphi_w\|$) is the number of state transitions in history $\mathbb{H}^\varphi_w = q_0q_1q_2\ldots$, such that $q_i \neq q_{i+1}$, for all $i \geq 0$. The history length of a property is then:*
$\|\mathbb{H}^\varphi\| = \max\{\|\mathbb{H}^\varphi_w\| \mid w \in \Sigma^\omega \ \wedge\ w \text{ has a minimal good/bad prefix}\}$

We also say that the good/bad minimal prefix of input word $w' \in \Sigma^\omega$ that leads to the maximum length history is an identifying minimal prefix*.*

We clarify a special case, where a monitor contains a cycle reachable from its initial state and a conclusive state is reachable from the cycle. In this case, according to Definition 10, history length of the associated property is infinity. For example, Figure 3.3b illustrates such a monitor. Obtaining length of infinity is due to the existence of cycle $q_0 - q_1 - q_0$. In other words, the monitor may encounter infinite number of state changes when running an infinite word before reaching a conclusive state.

Theorem 1 *Let φ be a property in LTL_3^{mon}. $\mathcal{M}^\varphi$ is cyclic iff $\|\mathbb{H}^\varphi\| = \infty$.*

Proof 1 *We distinguish two cases:*

- ($\Rightarrow$) *If a cycle exists in $\mathcal{M}^\varphi$, then it does not involve conclusive state. This is because any conclusive state is a trap. Thus, given a cycle $\bar{q} = q_0 - q_1 - q_2 \ldots q_k - q_0$ of inconclusive states, one can obtain an infinite word $w \in \Sigma^\omega$, such that the corresponding history $\mathbb{H}^\varphi_w$ will run infinitely on $\bar{q}$ and has infinite number of state changes. Therefore, $\|\mathbb{H}^\varphi_w\| = \|\mathbb{H}^\varphi\| = \infty$.*

- ($\Leftarrow$) *The opposite direction also holds. If the length of a property history is ∞, then it has to be the case that some states in the history are revisited. This implication is trivial, because the number of states of a monitor is finite. Hence, the monitor must contain a cycle.*

In general, the structure of a monitor depends on the structure of its LTL_3 formula (i.e., the number of temporal and Boolean operators as well as nesting depth of operators). For instance, the temporal operator $\bigcirc$ increments the history length of an LTL_3 formula by 1. If a formula does not include the **U** temporal operator, then the structure of its monitor will be a linear sequence of states and will not contain any loops. However, the reverse direction does not hold. For instance, properties $\varphi \equiv p \vee (q\,\mathbf{U}\,r)$ and $\psi \equiv p\,\mathbf{U}\,(q\,\mathbf{U}\,r)$ have both history of length 2.

Note that a finite history length for a property φ (i.e., $\|\mathbb{H}^\varphi\| = n$, where $n < \infty$) does not necessarily imply that any history will have a finite number of states before reaching a conclusive state. This is because self-loops in $\mathcal{M}^\varphi$ may exist. Finiteness of $\|\mathbb{H}^\varphi\|$ simply means that monitoring φ takes at most n state changes in $\mathcal{M}^\varphi$ to reach a verdict.

Using these definitions, we introduce a natural hierarchy of LTL formulas. Given $\text{LTL}^{mon}_{\mathbb{H}=i}$ a set of monitorable properties with history of length i,

$$\text{LTL}^{mon}_3 = \bigcup_{i=0}^{\infty} \text{LTL}^{mon}_{\mathbb{H}=i}$$

It is easy to see that this hierarchy is semantically strict. One can always find $\varphi \in \text{LTL}_{\mathbb{H}=i+1}$, such that $\varphi \notin \text{LTL}_{\mathbb{H}=i}$. For instance, property $\varphi = \underbrace{\bigcirc(\ldots(\bigcirc}_{\text{n+1 operators}} p)\ldots) \in \text{LTL}_{\mathbb{H}=n+1}$, but $\notin \text{LTL}_{\mathbb{H}=n}$.

3.2.2 Partial–Offload Algorithm

As stated previously, the algorithms in this chapter are presented in the order of the GPU involvement in the verification process. The **partial–offload algorithm** is very similar to the sequential algorithm in its logic. The only difference is that some of the predicates calculations are offloaded to the GPU. It is up to the developer to specify which properties should be offloaded. For example, if for the set of desirable properties only one of the properties is computationally intensive, it makes sense to perform the calculation of those special predicates on the GPU side. Thus, the only part that is effectively parallelized is the evaluation of predicates. Input and output are the same as in sequential algorithm. The algorithm flow is as follows:

1. New buffer of program states $s_0, \ldots, s_n$ is received.
2. Based on $s_0, \ldots, s_n$, perform calculations to map the predicates (the LTL formula consists of) to either $true$ or $false$:
   (a) Send the program states to the GPU.
   (b) The GPU evaluates the offloaded predicates to either $true$ or $false$.
   (c) Send the offloaded predicate values back to the CPU.
   (d) The CPU evaluates the rest of the predicates.
3. Based on the values of the predicates, make the appropriate transition if needed.

Note that even if the call for the GPU evaluation is synchronous, the CPU can perform evaluations of the non-offloaded predicates simultaneously, thus cutting the total execution time of the algorithm. Another possible optimization is to send only the partial program states to the GPU, as offloaded predicates might use only some of the variables from the program state. This optimization will reduce the memory transfer time.

3.2.3 Finite-history Algorithm

Finite-history algorithm takes a program trace and processes it in a parallel fashion by assigning different program states to the different GPU cores. To handle the causal order between the state transitions, the algorithm starts a new iteration every time a state transition occurs. Consequently, in each new iteration, a different current monitor state will be given. Intuitively, the number of iterations required to process the whole program trace, is bounded by the history length of the monitored property.

We now describe the algorithm in detail. The algorithm takes a finite program trace $\sigma = s_0 s_1 \ldots s_n$, a monitor $\mathcal{M}^\varphi = (\Sigma, Q, q_0, \delta, \lambda)$, and a *current* state $q_{current} \in Q$ (possibly q_0) of the monitor as input and returns a resulting state q_{result} and the monitoring verdict $\lambda(q_{result})$ as output (see Algorithm 1). The algorithm works as follows:

1. *Initialization:* The algorithm initializes tuple $\mathcal{I}$ to $\langle n+1, q_{current} \rangle$ (line 2). This tuple is used to keep the index of the left-most (in trace σ) program state that causes a state transition in the monitor and the resulting state of this transition. The tuple may return either the key (program state index) as in line 7 or the value (resulting monitor state) as in line 15. *StartIndex* points to the first program state to be processed in the current iteration (0 in the beginning).

2. *Parallel computation of next monitor state:* In lines 4–11, the algorithm computes the contents of tuple $\mathcal{I}$ in parallel. Tuple $\mathcal{I}$ is set to $\langle i, q_{result} \rangle$ only if program state s_i results in enabling a transition from current monitor state $q_{current}$ and i is strictly less than the previous value of $key(\mathcal{I})$. This way, at the end of this phase tuple $\mathcal{I}$ will contain the left-most transition occurred in the monitor. Observe that lines 7–9 are protected as a critical section and, hence, different processing units are synchronized. This synchronization is implemented at the GPU work–group level within the same computation unit. Thus, there is no global synchronization and parallelism among work groups is ensured. Moreover, observe that the most costly task in the algorithm (i.e., predicate evaluation on Line 5) is not in the critical section and executes in parallel with other such evaluations.

3. *Obtaining the result:* The third phase of the algorithm computes the final state of the monitor. If key of $\mathcal{I}$ is set to the initial value $n+1$, then no transition of the monitor gets enabled by program trace σ. In this case, the algorithm terminates and returns $(q_{result}, ?)$ as output (line 13). Otherwise, if a change of state in the monitor occurs and results in a conclusive state, then the algorithm returns this state and the monitoring verdict (line 17). Transition to an inconclusive state yields update of both $q_{current}$ and *StartIndex* and a new iteration (lines 19–21).

For illustration, consider property $\varphi_1 \equiv p \wedge (q \ \mathbf{U} \ r)$ and its monitor in Figure 3.3a. Let the current state of the monitor be $q_{current} = q_0$ and the input program trace be $\sigma = (p \wedge q \wedge \neg r) \cdot (\neg p \wedge q \wedge \neg r) \cdot (p \wedge q \wedge \neg r) \cdot (p \wedge \neg q \wedge \neg r)$. By applying this trace, first, the monitor makes a transition to state q_1 after meeting $p \wedge q \wedge \neg r$. Although the next two program states (i.e., $(\neg p \wedge q \wedge \neg r) \cdot (p \wedge q \wedge \neg r)$) of the input trace also cause program state transitions, the algorithm considers only the left-most transition, thus the first iteration records transition to q_1 and the second iteration starts from s_1. Finally, during the second iteration only the last program state $p \wedge \neg q \wedge \neg r$ enables the monitor transition from q_1 to $q_\perp$.

It is straightforward to observe that algorithm performs 2 iterations, which is precisely $\|\mathbb{H}^{p \wedge (q \ \mathbf{U} \ r)}\|$. In general, the number of algorithm iterations is equal to the number of state transitions in the underlying automaton. Thus, the complexity of the algorithm on one property is $\mathcal{O}(\|\mathbb{H}^{\varphi}\|)$ and on m properties $\mathcal{O}(\max\{\|\mathbb{H}^{\varphi_i}\| \ \mid 1 \leq i \leq m \ \})$. So the increase is less than linear. Following Theorem 1, the monitor constructed from a finite-history property does not contain a cycle. In other words, the number of state transitions during monitoring (and, hence, before reaching a conclusive state) is bounded by $\|\mathbb{H}^{\varphi}\|$. Since there can be a maximum of $\|\mathbb{H}^{\varphi}\|$ algorithm iterations, this number may be less if an input trace imposes a shorter path from $q_{current}$ to a conclusive state of the monitor. Either way,

Algorithm 1 For finite-history LTL_3^{mon} properties

Input: A monitor $\mathcal{M}^\varphi = (\Sigma, Q, q_0, \delta, \lambda)$, a state $q_{current} \in Q$, and a finite program trace $\sigma = s_0 s_1 s_2 \dots s_n$.
Output: A state $q_{result} \in Q$ and $\lambda(q_{result})$.

```
    /* Initialization */
 1: StartIndex ← 0
 2: I ← ⟨n + 1, q_current⟩
 3: Let m be a mutex

    /* Parallel computation of next monitor state given the current state */
 4: for all (s_i, StartIndex ≤ i ≤ n) in parallel do
 5:     q_result ← δ(q_current, s_i)
 6:     lock(m)
 7:     if (q_current ≠ q_result ∧ i < key(I)) then
 8:         I ← ⟨i, q_result⟩
 9:     end if
10:     unlock(m)
11: end for

    /* Obtaining the result */
12: if key(I) = n + 1 then
13:     return q_result, ?
14: else
15:     q_result ← value(I)
16:     if λ(q_result) ≠ ? then
17:         return q_result, λ(q_result)
18:     end if
19:     q_current ← q_result
20:     StartIndex ← StartIndex + key(I) + 1
21:     goto line 2
22: end if
```

the overhead of interrupting the iteration in the middle is bounded by a constant. This overhead becomes negligible as input size and monitoring time increase.

In case of the infinite history length, there are two conditions that combined together may cause excessive iterations and lead to the performance degradation of the algorithm: (1) existence of a loop in $\mathcal{M}^\varphi$, and (2) sequence of the program states that results in constant loop traversing. The Algorithm presented in Subsection 3.2.4 addresses these two conditions.

More precise, the execution time of the algorithm on one chunk of data now can be described by the following formula:

$$T_1 = t_{mt_1} + E(n).\max\{\sum_{j=1}^{m}(t^i_{calc} + t^{i,j}_{branch}) \mid 1 \leq i \leq n\} \quad (3.3)$$

where t_{mt_1} is the memory transfer time to the processing unit, which, if a parallel evaluation takes place on a CPU or the buffer resides in the shared memory, is negligible (consider AMD Fusion architecture as mentioned in Section 1). $E(n)$ is the expected number of the monitor state transitions (and algorithm iterations consequently) per n data items. It encodes conditions (1) and (2) from the previous paragraph.

3.2.4 Infinite-history Algorithm

The second algorithm is an adaption of the algorithm in [24] for parallel execution of deterministic finite state machines. This algorithm does not tackle the issues with causal order of state changes directly. Instead, it calculates possible state transitions for every monitor state (except the conclusive states, since they are traps), regardless of the actual current state of the monitor. Then, in a sequential phase the algorithm aggregates all the results to compute one and only one current state. This sequential run reflects the transitions in the underlying monitor that are identical to those caused by sequential processing. Consequently, this algorithm does not depend on the history length of a property.

Now, we describe the algorithm in more detail. Similar to Algorithm 1, it takes a finite program trace $\sigma = s_0 s_1 \ldots s_n$, a monitor $\mathcal{M}^{\varphi} = (\Sigma, Q, q_0, \delta, \lambda)$, and a *current* state $q_{current} \in Q$ (possibly q_0) of the monitor as input and returns a resulting state q_{result} and the monitoring verdict $\lambda(q_{result})$ as output. For parallelization, we leverage a well-known exhaustive execution method, where we eliminate the input interdependencies by considering all possible outputs. The algorithm works as follows:

1. *Initialization:* First, the algorithm computes all inconclusive states of the monitor. We do not consider conclusive states, since they always act as a trap. In order to eliminate input interdependencies, we maintain a lookup matrix $\mathcal{A}$ for storing intermediate results, for each state of the monitor and a program state from trace σ (see Figure 3.4).

2. *Parallel exhaustive state computation:* In lines 3–7, the algorithm computes the columns of matrix $\mathcal{A}$ in parallel. Each element of $\mathcal{A}$ is calculated using a monitor state and a program state. A program state identifies the set of atomic propositions and, hence, the letter in Σ that holds in that state. Although calculation of step $i+1$

Algorithm 2 For infinite-history LTL_3^{mon} properties

Input: A monitor $\mathcal{M}^{\varphi} = (\Sigma, Q, q_0, \delta, \lambda)$, a state $q_{current} \in Q$, and a finite program trace $\sigma = s_0 s_1 s_2 \ldots s_n$.
Output: A state $q_{result} \in Q$ and $\lambda(q_{result})$.

```
    /* Initialization */
 1: Q_? = {q ∈ Q | λ(q) = ?}
 2: Let A be a |Q_?| × n matrix

    /* Parallel exhaustive computation of monitor states */
 3: for all (s_i, 0 ≤ i ≤ n) in parallel do
 4:    for all q_j ∈ Q_? do
 5:       A_{q_j,s_i} ← δ(q_j, s_i)
 6:    end for
 7: end for

    /* Sequential computation of actual monitor state */
 8: q_result ← q_current
 9: for all (0 ≤ i ≤ n) sequentially do
10:    q_result = A_{q_result,s_i}
11:    if λ(q_result) ≠ ? then
12:       return q_result, λ(q_result)
13:    end if
14: end for

15: return q_result, ?
```

depends on the output of step i, by calculating and storing q_{i+1} for all possible q_i, this interdependency can be eliminated (i.e., for every s_i and q_j, we calculate element $\mathcal{A}_{q_j,s_i} = \delta(s_i, q_j)$). For example, for property $\varphi \equiv \Box(p \Rightarrow (q \; \mathbf{U} \; r))$ from Figure 3.3b, the matrix is of size $2 \times n$ because of two inconclusive states in $\mathcal{M}^{\varphi}$: q_0 and q_1. Note that this part of the algorithm can be done independently for each s_i — in parallel in other words.

3. *Actual state computation:* The third phase of the algorithm consists of a sequential pass of length at most n over the columns of matrix $\mathcal{A}$. Initially, the resulting state is $q_{current}$. The output of step i is the content of $\mathcal{A}_{q_{result},s_i}$. Note that in this step, the value of q_{result} changes only if a transition to a different state of the monitor is enabled. This way, the algorithm starts from state $q_{current}$ and terminates in at most n steps. The value of q_{result} reflects every change of the state of the monitor when the algorithm executes lines 8–14. In Figure 3.4, for example, this step-by-step sequential evaluation jumps from $q_{current} = q_0$ to $q_{result} = q_{|Q_?|}$ after the second step and finally

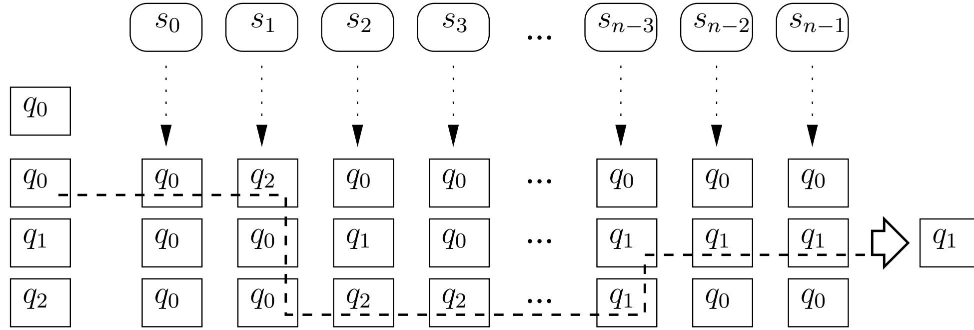


Figure 3.4: Eliminating interdependencies for parallel execution

concludes in $q_{result} = q_1$. At any point, if q_{result} happens to be a conclusive state, then the algorithm terminates and returns the verification verdict of the monitor (lines 11–13).

Finally, if the monitor does not reach a conclusive state, then the algorithm returns the reached state by trace σ and value ?.

Unlike the first algorithm, the performance of this algorithm does not depend on the structure of the monitor, as all possible state transitions are absorbed by the final sequential pass. On the other hand, the extra calculations undertaken for every inconclusive state, as well as memory transfer of the results of those calculations back to the host process on the CPU, add to the execution time. The complexity of the algorithm now depends on the size of the input and the number of inconclusive states in the underlying monitor: $\mathcal{O}(n.\sum_{i=1}^{m}|Q_?|_i)$. More precise, the total execution time can be described by the following equation:

$$T_1 = t_{mt_1} + t_{mt_2} + t_{seq} + \max\{\sum_{j=1}^{m}(t_{calc}^{i,j} + \sum_{k=1}^{|Q_?|_j} t_{branch}^{i,j,k}) \mid 1 \leq i \leq n\} \tag{3.4}$$

where t_{seq} is the time spent on the sequential phase (lines 8 to 14), and t_{mt_1} and t_{mt_2} are memory transfer times to and from processing unit respectively. Again, if a parallel evaluation takes place on a CPU or the buffer resides in the shared memory, then t_{mt_1} and t_{mt_2} are negligible.

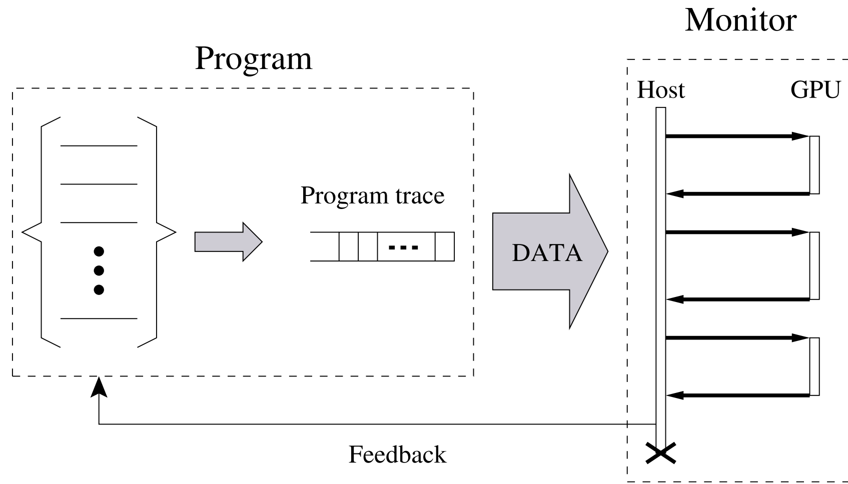


Figure 3.5: GPU-based architecture for experiments

3.3 Evaluation of Algorithms

3.3.1 Experiment Setup

Figure 3.5 shows the architecture of the experiment setup. Following is the detailed description of the main components:

- *Program instrumentation:* We instrument the original program under inspection, so that it stores the program states in a buffer as a program trace. From time to time (depending upon the design parameters), the program writes this trace to the shared memory.
- *Host process:* The host process is a C program that reads program traces from the shared memory, redirects them to the monitoring processing unit, and manages the verification results. The only overhead incurred by the host is for memory transfers. In fact, the host is blocked while waiting for the verification tasks to complete. Obviously, this is not the case in traditional sequential monitoring (i.e., the CPU carries all the computations, which increases potential interference with the program under scrutiny).
- *Monitor generation:* Generating code for a monitor is done in two steps:

1. Given an $\text{L}{\small\text{TL}}_3$ property, we generate a monitor as a deterministic finite state machine as defined in Definition 6 using the tool from [6].
2. Next, we automatically generate two modules of code to implement the monitor constructed in the previous step. The first module is a single-threaded C program that instantiates a simple sequential monitor. The second module is a concurrent program that implements either Algorithm 1 or 2 to parallelize execution of the monitor. The code of this module is generated in the OpenCL language for GPU programming. The generation tool called `GooMFGenerator` and is described in detail in Section 4.2.

When using the sequential algorithm, the need in memory transfer and the host process itself is eliminated. Instead, the inspected program directly calls for the verification functions. Also, the finite state machines are implemented in the simplest way. All this to ensure maximal efficiency of the sequential monitoring, which, in turn, will provide additional validity to the comparison between parallel and sequential monitoring carried out in the following sections.

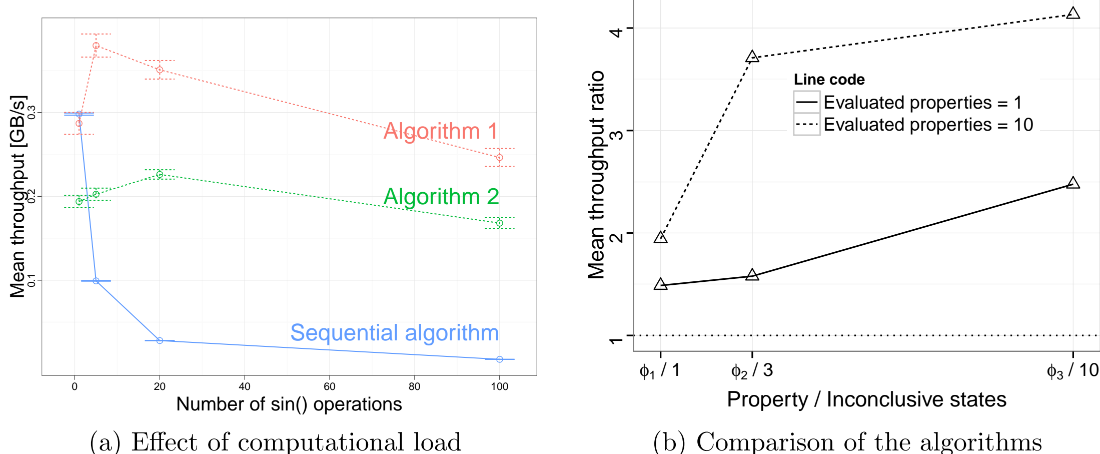
All the experiments are conducted on 32-bit Ubuntu Linux 11.10 using a 8x3.07GHz Intel Core i7 and 6GB main memory and an AMD Radeon HD5870 graphics card as GPU. At the time the experiments were conducted the partial–offload algorithm was not yet conceived, thus only three algorithms are inspected. Section 5.4 delivers the results on the partial–offload algorithm instead.

3.3.2 Throughput Analysis

In the context of monitoring, throughput is the amount of data monitored per second. In the following experiments, the program trace is fed to the monitor directly from the main memory, thus maximizing throughput of the algorithms.

Effect of the Computational Load

From now on, we refer to *computational load* as the number of evaluations of all mathematical and arithmetic operations in order to evaluate atomic propositions of a property. Computational load is a necessary step before verification of a property and has direct effect on the sequential monitoring time, as it is performed by the CPU. However, in our GPU-based approach, the computational load is spread over GPU cores. We hypothesize that our approach will outperform a sequential monitoring technique.



(a) Effect of computational load

(b) Comparison of the algorithms

Figure 3.6: Evaluation results

In order to validate our hypothesis, we emulate different computational load on property $\varphi = \Box(\neg a \vee \neg b \vee \neg c \vee \neg d \vee \neg e)$, where $\|\mathbb{H}^{\varphi}\| = 1$. The number and the type of the properties are not essential, as the goal is to isolate the load factor. The loads are 1x, 5x, 20x, and 100x, where 'x' is one sin() operation on a floating point operand. We run the experiment for the parallel algorithms on 1600 cores available on the GPU and measure the throughput of three algorithms (i.e., a sequential algorithm as well as Algorithms 1 and 2 from this paper). Figure 3.6a shows the results of this experiment (the error bars represent the 95% statistical confidence interval).

As can be seen in Figure 3.6a, both Algorithms 1 and 2 outperform a sequential algorithm as the computational load increases. For instance, for 100 sin() operations Algorithms 1 outperforms the sequential algorithm by a factor of 35. Thus, as Equations 3.3 and 3.4 in Subsection 3.2.3 and 3.2.4 suggest, there is a loose dependency between our parallel algorithms' throughput and the computational load. This observation along with the graph in Figure 3.6a conjectures that the computational load is not a bottleneck for our algorithms. On the contrary, the sequential algorithm performance is poorer than the linear dependency on the computational load as Equation 3.2 in Section 3.1.

Under the conditions of this experiment, the superiority of the GPU-based monitoring will only grow in the future: while the desktop configuration is one of the most powerful available, the GPU series used in the experiments is more than two years old. Newer GPU series will offer an increase in both the number of processing units and memory transfer throughput. The performance of the sequential algorithm, on the other hand, is

constrained by the staggering growth in the processing speed of the single core.

Performance Analysis of Algorithms 1 and 2

We hypothesize that Algorithm 1 would outperform Algorithm 2, as Algorithm 2 evaluates the next state of *all* inconclusive states in the monitor. In addition, Algorithm 1 does not transfer the calculated states back to the CPU (recall that lines 8–14 of Algorithm 2 run on the CPU). The only slowdown factor that appears in Equation 3.3 and does not appear in Equation 3.4, is $E(n)$. Given the finite number of monitor state transitions, this factor becomes negligible as the monitoring time increases.

The experimental setting consists of the following input factors: algorithm type, number of properties for monitoring, and the number of inconclusive states in the monitor of a property. This is due to the fact that the number of inconclusive states affects the amount of calculations the second algorithm should carry (see Equation 3.4). The size of the program trace is 16,384 states. We consider three different properties. Two of the properties are taken from the set of properties in [18] for specification patterns: $\varphi_1 \equiv \Box\neg(a \vee b \vee c \vee d \vee e)$ and $\varphi_2 \equiv \Box((a \wedge \Diamond b) \Rightarrow ((\neg c)\,\mathbf{U}\,b))$, where the number of inconclusive state are 1 and 3, respectively. The monitors of properties in [18] are relatively small (the largest monitor has five states). Thus, our third property is the following (rather unrealistic) formula whose monitor has 10 inconclusive states:

$$\varphi_3 \equiv \underbrace{\bigcirc \bigcirc \ldots \bigcirc}_{\text{10 next operators}} (a\,\mathbf{U}\,b)$$

Figure 3.6b confirms the hypothesis. We emphasize that the graphs in the figure represent the *ratio* of the throughputs of Algorithm 1 over Algorithm 2. X-axis represents the number of inconclusive states in the monitored property (1 for φ_1, 3 for φ_2, and 10 for φ_3). The solid line shows the experiments for one property and the dashed line plots the experiments for a conjunction of ten identical properties. Algorithm 1 consistently performs better on all of the properties. The ratio only grows when the the number of inconclusive states in the property increases.

Scalability

As illustrated in Figure 3.5, the host submits *chunks* of the program trace to the monitoring tasks running on the GPU. A chunk is the array containing a sub-trace to the GPU internal

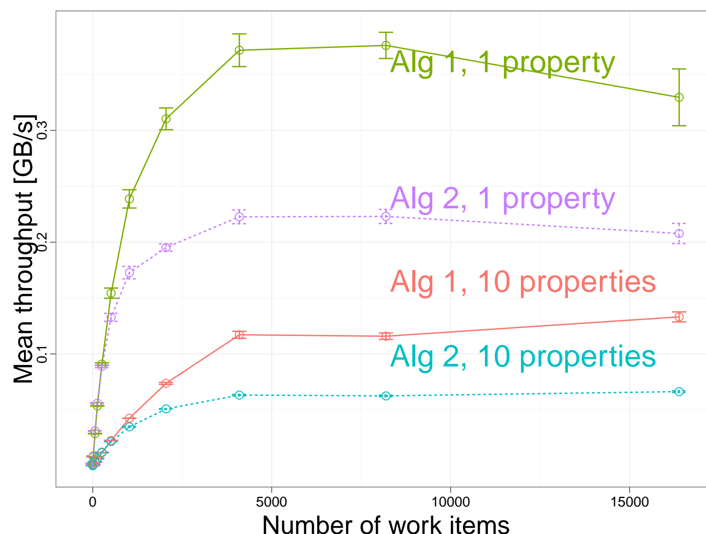


Figure 3.7: Throughput vs number of work items

buffers. Once the GPU is available, it will pull the next task from the queue and will run the task on the input data following the Single Instruction Multiple Data (SIMD) pattern. The GPU also implements the internal scheduling mechanism that controls the process of assigning the data to the GPU cores. However, input array can be represented as a number of *work items*. Thus, by setting this number to one, we ensure that only one core participates in the evaluation. Consequently, by increasing the number of work items to the number of items in the input data, we control the number of cores engaged in monitoring. If the chunk size is greater than the number of cores, then at some point the GPU scheduler will have to assign several work items to the same core.

In this experiment, we fix all contributing factors as constants (including the chunk size of $16,384$) and only change the monitoring algorithm, the number of work items, and the number of properties for monitoring. The results (shown in Figure 3.7) imply that the algorithms are clearly scalable with respect to the number of cores. The error bars represent the 95% confidence interval. The result shows that the mean throughput increases with the number of cores engaged in monitoring. At some point, both algorithms reach the optimum, where all the core are utilized. From that point and on, the throughput will be mostly affected by the GPU thread scheduler.

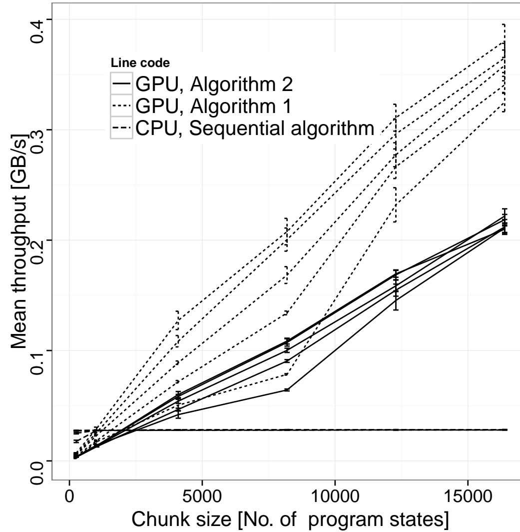


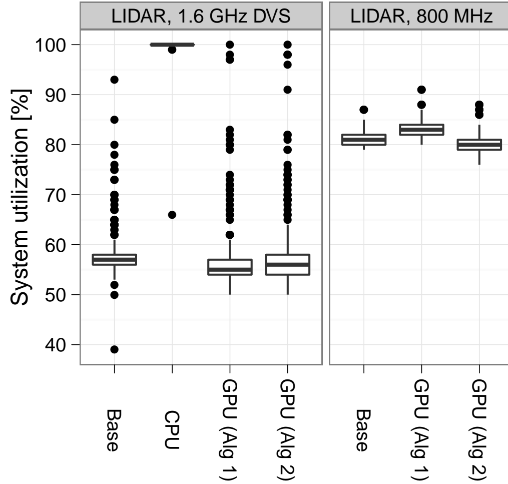
Figure 3.8: Effect of the buffer and data sizes

Effect of Data and Chunk Sizes on Throughput

The factors of this experiment comprise the algorithm type, the number of simultaneously evaluated properties, and the amount of data in each run (see Figure 3.8). The error bars indicate the 95% confidence interval. The line type indicates the type of algorithm. The five lines for each algorithm show results for different amounts of data (i.e., 10, 20, 30, 40, and 50 data chunks). The results show that sequential monitoring is neither influenced by the chunk size nor the amount of data. The mean throughput of the parallel monitoring shows a strong correlation with the chunk size. This is expected, since an increase of the chunk size results in a higher number of the program states processed in parallel. This, in turn, leads to the more efficient distribution of the work items among the cores. However, this throughput gain is limited by the amount of memory available on the GPU and the tolerable delay of the program. The minor differences between the results with different amounts of data show the independence of the mean throughput from the amount of data. This is expected behavior and increases our confidence in the results.

3.3.3 UAV Case Study

This section offers additional benefits of using the GPU-based monitoring. When engaging the parallel algorithms instead of sequential for the UAV flight monitoring, the numbers



(a) CPU utilization

	Type	Name	Avg Watt
1	Nothing	Idle	15.56
2	No LIDAR	CPU	20.20
3		GPU (Alg 1)	15.82
4		GPU (Alg 2)	15.72
5	LIDAR, 1.6 GHz DVS	Base	27.81
6		CPU	28.80
7		GPU (Alg 1)	28.73
8		GPU (Alg 2)	28.04
9	LIDAR, 800MHz	Base	22.29
10		GPU (Alg 1)	23.00
11		GPU (Alg 2)	22.42

(b) Power consumption

Figure 3.9: Case study experiments

showed 25% of reduction in power consumption as well as significant decrease in CPU utilization during the flight.

The monitor checks at run time an unmanned aerial vehicle (UAV) autopilot software running on Beagle Board with QNX 6.5 OS. The monitor runs on an ASUS E35-M1 board containing dual-core AMD E350 APU and Ubuntu 11.10 OS. Another process running alongside with monitor is LIDAR — a grid reconstruction algorithm — that emulates the load of the autopilot. The scale of the software being examined:

- Size of Monitor: 52K
- Size of Autopilot: 78K
- Lines of code in Monitor: 3000
- Lines of code in Autopilot: 4000

Every 10ms, the autopilot sends the program state consisting 14 float numbers to the monitor over the crossover Ethernet cable using the UDP protocol. The host thread in the

monitor interpolates the data and fills up the buffer. The GPU then processes the buffer using one of the three evaluation algorithms. We verify the following five properties:

1. $\varphi_1 \equiv \Box(a \wedge b \wedge c)$
   This property verifies the sanity of the IMU sensors during the flight. IMU (Inertial Measurement Unit) measures values of pitch, yaw, roll and their velocities. The property compares the numerical values of the angular acceleration from the sensors with calculated analytical values from the two consecutive program states. Predicates a, b and c stand for the calculations on the three dimensions.

2. $\varphi_2 \equiv \Box(d \Rightarrow (\bigcirc\neg d) \vee (\bigcirc\bigcirc\neg d) \vee (\bigcirc\bigcirc\bigcirc\neg d))$
   Predicate d stands for "number of satellites less than three". The property verifies that this undesirable state will not last in more than four consecutive program states.

3. $\varphi_3 \equiv \Box(e)$
   This property verifies whether the autopilot translates the coordinates of latitude, longitude, and altitude correctly. Predicate e stands for the transformation function.

4. $\varphi_4 \equiv \Box(f \Rightarrow \Diamond g)$
   This property verifies that the UAV eventually lands. Predicate f stands for "altitude higher than 500", predicate g stands for "altitude less than 358" — the altitude above the sea level on the university campus.

5. $\varphi_5 \equiv \Box(h \wedge i \wedge j \wedge k \wedge l)$
   This property verifies the requirements of Transport Canada Agency stated in Safety Flight Operation Certificate Requirements document: the flight is permitted only within specific coordinates on the campus of the University of Waterloo.

If one of the properties is violated/satisfied, the monitor will send a UDP packet indicating the property that has converged, back to the autopilot.

The experiment consists of two parts: first part reports the CPU utilization measurements (Figure 3.9a) and the second part reports the power consumption during the flight (Table 3.9b). As the monitoring takes place during the flight, we do not measure execution time. Rather, we measure CPU utilization. As Figure 3.9a shows, LIDAR alone consumes

about 57% of CPU time when running the system with default “ondemand” policy. When this CPU scheduling policy is on, the OS switches the CPU to highest frequency upon demand and back to the lowest frequency after the task is done. While switching the frequencies, Dynamic Voltage Scaling (DVS) takes place. This explains the outliers in the GPU-based evaluation algorithms, which otherwise barely increase the regular consumption. On the contrary, running the CPU-based monitoring, results in utilizing practically all the CPU resources. In safety critical systems with real-time constraints, this can lead to the missing cycles, which is obviously problematic. When GPU-based monitoring is on, the amount of idle CPU time indicates that the frequency can be lowered without risk of missing cycles. Second part of the graph shows the system running on the “userspace” scheduling policy with constant frequency of 800 MHz. The utilization rises to 81%, but never reaches 100%.

Table 3.9b reports the average of the power consumption measurements over the time and closely correlates with the Figure 3.9a. E350 APU board is given constant power supply of 15 Volts and by recording the current level we calculate the consumed power. Rows 2–4 report the consumption of the system without background load of LIDAR, while rows 5–8 show close consumption levels with LIDAR running. Although without background load the GPU-base monitoring consumes 25% less power than CPU-based, this approach is not practical, as we aim at running the monitor and the inspected program on the same board. To deal with this issue, we set the constant CPU frequency to 800 MHz (rows 9–11 in the table correspond to the second part of the Figure 3.9a). In addition to solid 25% reduction in power consumption, this gives us predictable CPU utilization with minimum outliers.

Chapter 4

GooMF Framework

This chapter is dedicated to GPU-based online and offline Monitoring Framework (GooMF) — a verification framework that enables any C developer to monitor and verify her program given a set of desirable properties. The ultimate goal of GooMF is to supply the developers with an easy-to-use and flexible tool that requires minimal knowledge of formal languages and techniques. The core of GooMF is the hierarchy of algorithms presented in Chapter 3. Therefore, the monitoring action can be performed on GPU or on CPU almost seamlessly for the user. With GPU support enabled, the inspected system benefits from the range of advantages discussed in the previous chapter, and in particular, minimization of the interference between the monitor and the program, faster processing for non-trivial computations, and even significant reduction in power consumption comparing to the CPU-based monitoring.

4.1 Run-Time Verification Tools Overview

We already discussed the shortage of verification tools for C and C++ languages in Section 1.2. As the area of run-time verification already offers a selection of frameworks and tools, in this section, we are trying to identify the characteristics of successful tools. The majority of the tools in the area can be easily sorted by the language of the target program: JavaMOP, JavaMaC, J-Lo, Eagle, Hawk, RuleR, Tracematches are for Java; Spec# [2] is for C#; P2V [31] and Temporal Rover [17] are for C and C++. P2V, although intended for C programs, is essentially a translation tool for the PSL assertions and requires dedicated hardware (eMIPS and FPGA) to run on.

One interesting exception of this division is the commercial tool Temporal Rover, which, according to authors, is capable of interfacing with C, C++, Java, Verilog, and VHDL altogether. Temporal Rover is a run-time verification tool based on future time metric temporal logic. It allows programmers to insert formal specifications in programs via annotations, from which monitors are generated. The obvious drawbacks are manual instrumentation of the source code, commercial nature of the framework, and the overhead, which remains unexplored.

The majority of these tools implementation is tightly coupled with specification formalism, but few are able to incorporate plug-ins for **different formalisms**. The most prominent example is JavaMOP, which offers seven formalisms for properties specifications: LTL, Past-Time LTL (PTLTL), Finite State Machine (FSM), Extended Regular Expressions (ERE), Context Free Grammars (CFG), String Rewriting Systems (SRS), and Past-Time LTL with Calls and Returns (PTCaReT). More important, this impressive set of specification languages contains the most common LTL formalism. This feature is particularly valuable, because LTL is a de-facto standard in model checking. Thus, combining offline and online verification is made easy by not requiring modifications to the set of properties.

Parameterized monitoring is a salient feature that enables users of tools to specify a reacher set of properties than that available for non-parameterized systems. The classical example is a property that checks whether open file is eventually closed. Specifying this requirement with non-parametric properties is tricky, as the file descriptor may be known only at run time. Verification tool that implements parameterized monitoring, on the other hand, will spawn new monitoring instance every time new file is opened. Thus, in addition to adding the expressibility to the properties, parameterized monitoring also contributes to the clarity and simplicity of the specification. Consider, for example, a program that potentially operates with thousands of files. Even if all the file descriptors are known beforehand, specifying property per file would be frustrating. Obviously, parametrized monitoring is a must-have attribute in monitoring frameworks. We discuss parameterized monitoring and how it is implemented in GOOMF in Chapter 5 in detail.

Both JavaMOP and J-Lo implement parameterized monitoring, but it comes at a price of **space and time efficiency**, as well as **overhead**. In monitoring real systems, problem size can scale extremely fast both in terms of a number of monitoring instances and a number of monitored events in the system. Space and time efficiency and especially reasonable overhead are crucial for real-time and safety-critical systems, as unpredictable overhead can easily be a cause for missing real-time cycles. This is obviously an impediment for the process of incorporating run-time verification tools into real-time systems. Unfortunately, these aspects of monitoring usually do not draw attention of the tool developers. This is

partly due to the tradeoff between expressibility of the tool and its efficiency.

4.2 GooMF Architecture and Implementation

GooMF is a verification framework that enables C developers to monitor and verify their program given a set of desirable properties. The monitoring action can be performed on GPU and / or CPU. The ultimate goal of GooMF is to supply the developers with an easy-to-use and flexible tool that requires minimal knowledge of formal languages and techniques. In addition, by incorporating the parallel algorithms into the tool, we achieved all the advantages of GPU-based verification discussed in the previous section.

GooMF borrows some of the functionality from the setup described in Subsection 3.3.1. The detailed work–flow is shown in Figure 4.1: the user specifies the properties in a special configuration file using one of the two formalisms: Ltl or FSM. The properties are specified through the predicates, which, in turn, consist of program variables listed in the same file. GooMFGenerator — the generation module of GooMF— automatically generates the necessary structures and headers for storing the snapshot of the program state. This program state includes all the program variables comprising the predicates and thus involved in the monitoring. Along with the header comprising the program state structure, GooMFGenerator produces three files that contain OpenCL code: `GooMF_GPU_monitor_alg_partial.cl`, `GooMF_GPU_monitor_alg_finite.cl`, and `GooMF_GPU_monitor_alg_infinite.cl`. Each of these files corresponds to a different parallel algorithm. The code in the files embodies the functionality of the algorithms and the structure of the finite state machines of the properties. As mentioned in Subsection 3.3.1, given Ltl_3 properties, GooMFGenerator generates monitors as deterministic finite state machines as defined in Definition 6 using the tool from [6]. This architecture ensures that the configuration file is the only place that needs to be changed after the properties evolve.

These steps are executed by the script GooMFMake, which should run offline before running the actual program under inspection. The final result of this offline processing is `libGooMF.so` — the shared library that contains all the verification functionality and can be linked from the program.

4.2.1 Online Monitoring

The first method of GooMF usage is calling the Online API directly from the verified program. The second method is calling the Offline API functions to parse / analyze the

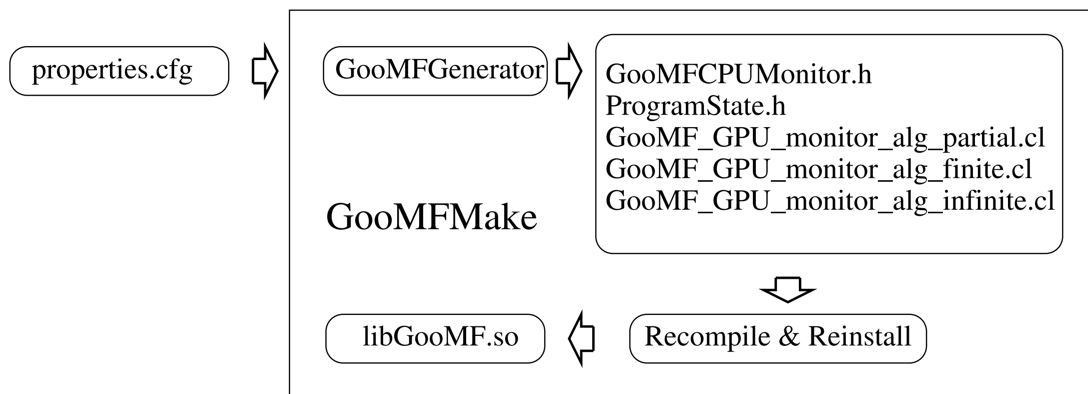


Figure 4.1: GooMF work–flow

input stream. The former approach gives the developer a full control over the monitoring process, while the latter allows to implement the stream parser with minimal effort. In its instance, GooMF is a shared library, and as such must be dynamically linked to the program. Behind the scenes, GooMF Offline API uses the functions of Online API, which is the reason why `GooMFOnlineAPI.h` header is included from `GooMFOnlineAPI.h`. During the execution of the inspected program, the program states and events are buffered into the verification buffer managed by GooMF. Given the type of the verification trigger, this buffer is flushed either when it reached its maximal limit, or when specific function is called.

Following is the list of the key functions in Online API:

- `_GOOMF_initContext` — initializes the monitoring context. This function accepts parameters that define the type of the verification algorithm, the type of the verification trigger (buffer- or user-triggered), and the type of the call (blocking or non-blocking — see Subsection 4.2.3).
- `_GOOMF_destroyContext` — destroys the monitoring context; usually called at the end of the program execution.
- `_GOOMF_nextState` — loads the next program state to the verification buffer.
- `_GOOMF_nextEvent` — loads the next event to the verification buffer. This function is useful when only one program variable has changed, and there is no need to copy the entire program state.

- `_GOOMF_flush` — flushes the verification buffer; usually called if user-triggered verification is selected.

For the full Online API see Appendix A.

4.2.2 Offline Monitoring

Run-time monitoring is useful when the decision needs to be taken while the program is running and particularly appealing for fault prevention and recovery. Sometimes, however, the requirement is to monitor the system offline. The prominent example is a postmortem log analysis, when the goal is to examine the log by checking if it conforms to the specified property. With this in mind, we designed GooMF so that it can be used for both purposes.

Callbacks. The Offline API is useful when the input stream of data has to be Ltl- or FSM- parsed/analyzed with minimal effort and within minimal time. It is similar to the way one would use libpcap / Winpcap packets capture modules. The developer is required to provide four callback functions and then call a blocking function that will parse the input stream until its done. The callbacks will be called upon the following events:

- Opening the input stream: `typedef int (*open_handler)()`.
- Parsing the next program state: `typedef int (*get_next_state_handler)(void* next_ps_ptr)` Here, `next_ps_ptr` is the pointer to the newly parsed program struct. The content of the struct is copied to the verification buffer, so that the developer does not have to allocate the memory dynamically.
- Property has been satisfied / violated: `typedef int (*report_handler_type)(int prop_num, _GOOMF_enum_verdict_type verdict_type, const void* program_state)` Here, `prop_num` is the number of the converged property, `verdict_type` is the verdict and `program_state` is the final state in the sequence that caused the convergence.
- Closing the input stream: `typedef int (*close_handler)()`.

Each callback function should return a negative number in case of failure or 0 in case of success.

Analyze. To start the parsing, the developer should call `_GOOMF_analyze` function, with the callback functions provided as arguments:

```
int _GOOMF_analyze(open_handler oh_callback,
.......get_next_state_handler gnsh_callback,
.......close_handler ch_callback,
.......report_handler_type rh_callback,
......._GOOMF_enum_trigger_type trigger_type,
......._GOOMF_enum_alg_type alg_type,
......._GOOMF_enum_invocation_type invocation_type,
.......FILE* logger,
.......unsigned int buffer_size);
```

where first four parameters are the callback function pointers; `trigger_type` is type of the trigger that causes the buffer of the program states to flush (user-invoked or buffer-size–invoked); `alg_type` is type of the verification algorithm: sequential, partial, `finite_history` or `infinite_history`; `invocation_type` controls if the verification is performed in the separate thread (asynchronous) or the same thread with the blocking call (synchronous); `logger` is the optional file descriptor for the additional output; `buffer_size` is the size of the verification buffer. Once the maximum capacity is reached — the buffer needs to be flushed. `_GOOMF_analyze` returns `_GOOMF_SUCCESS` if successful or predefined error. For the full Offline API see Appendix B.

4.2.3 Synchronous vs. Asynchronous Invocation

The current version of GooMF optionally implements a multi-threaded monitoring architecture. The verification process might be either blocking (main thread invokes the flush function and waits until its done) or non-blocking (main thread designates a worker thread from the pool to perform the verification). In the non-blocking mode, the program under scrutiny does not need to wait until the buffer is verified, rather the flush function is done once the task is wrapped and sent to the verification device (GPU or CPU). The non-blocking mode is more appealing, as the program can continue to run with the verification happening at the background, which will obviously shorten the execution time of the program. However, some considerations are to be thought of before using the asynchronous mode. Namely, whether the program can afford additional delay in the report of property violation/satisfaction. Also, the developer should provide a reliable report function that GooMF will call in the case of convergence. Therefore, additional synchronization mechanisms may be required for the safe implementation, because multiple threads are operating simultaneously and possibly share same data.

The type of the invocation is controlled by the fourth parameter in `_GOOMF_initContext()` function. The following code initializes GPU-based monitoring context that uses infinite-history algorithm with a synchronous invocation and a size-triggered buffer flush:

```
_GOOMF_initContext(&context,
....._GOOMF_enum_buffer_trigger,
....._GOOMF_enum_alg_infinite,
....._GOOMF_enum_sync_invocation,
.....1024);
```

This code initializes CPU-based monitoring context that uses the sequential verification algorithm with an asynchronous invocation and a user-triggered buffer flush:

```
_GOOMF_initContext(&context,
....._GOOMF_enum_no_trigger,
....._GOOMF_enum_alg_sequential,
....._GOOMF_enum_async_invocation,
.....4096);
```

4.2.4 Other Features

The predicates for the properties are automatically embodied into the verification function code, thus stated in C language. This allows several interesting features, namely using C functions to evaluate the predicate. The property file can contain separate section for user functions, thus enabling evaluation of complex and evolved predicates as well as using the power of C language for this evaluation. Consider, for example, property from Subsection 3.3.3. φ_3 verifies whether the conversion of the coordinates from coordinates from World Geodetic System (WGS) to Earth-Centered, Earth-Fixed (ECEF) to East, North, Up (ENU) Cartesian coordinate system performed right. Without the ability to encompass the logic of the predicate into the function, it would be virtually impossible to express this predicate. Instead, it would take a set of other, smaller predicates to form the original one. To verify e, and, consequently, φ, GooMF calls the function specified in the beginning of the property file in the "functions" section. Obviously, it should return boolean variable at the end. Same section allows the user to specify constant and other global variables used for the predicate evaluation.

The properties can be specified using one of the two formalisms: Ltl or FSM. For FSM, GooMF requires AT&T FSM format[1]. Behind the scenes, GooMF calls the script from [6] to generate the monitor FSM in the same format. This action takes place in function `buildFSMs()` in GooMFGenerator module. To introduce additional formalism, a future developer should add functionality only to this function. For example, an Extended Regular Expression (ERE) formalism can be added by calling the new script from

[1] http://www2.research.att.com/~fsmtools/fsm/tech.html

`buildFSMs()`. This script should generate FSM from ERE and convert this FSM into AT&T FSM format.

There are several formats for expressing a Finite State Machine. The most logical way to express FSM is a sequence of `if` operators. Every `if` clause checks the conditions for a transition given a current FSM state and an input event. Although this is not the most efficient way in terms of decision–making time and code size, it is definitely the most intuitive. Another, more efficient approach is to store the FSM as a matrix. All possible input combinations are encoded in first dimension, and all possible states are encoded in second dimension. This way, the decision about next state is reduced to an action of assignment. The disadvantage of this method is in its succinctness: if a transition is accompanied with an action, it is impossible to incorporate the latter into the code. Hence, the decision about FSM representation is left to the GooMF user. It can be specified on a per-property basis in the configuration file.

GooMF allows enabling and disabling monitoring on a per-property basis. More importantly, this can be done at run time. If for some reason a property known to be not active, it can be disabled using `_GOOMF_disableProperty()` API call. This feature saves evaluation time of unnecessary predicates.

4.2.5 Implementation Issues

While working on GooMF, we encountered several implementation problems. Most of the issues were GPU-related, since the performance of GPU-based execution is highly sensitive to changes in the OpenCL kernel code.

As mentioned in Section 2.3, heterogeneous memory hierarchy promotes local synchronization. This became especially obvious when we tried to use function `atomic_min()` on the variable residing in the global memory. Function `atomic_min()` is used for evaluating the left-most state transition in Algorithm 1. The first version of the kernel included `atomic_min()` applied to the global state transition index, which resulted in approximately 30% of the slowdown of the complete algorithm run. To solve this problem, we reduced the problem domain to the local memory. After every work-group of 256 threads evaluated its left-most state transition, one thread passes over the results and picks the first-in-order transition.

Another compelling problem is related to the work scheduling between the worker threads. Sometimes, when the worker is blocked or is waiting for the IO operation to complete, it can automatically switch to the next work–item to continue its work. We leverage this feature by assigning several work–items to the same thread — depending on

the number of items in the work buffer. Obviously, this will only work if the number of program states in the verification buffer is less than the number of worker threads available in the compute device.

At the end of Section 2.3, we pointed out the trend in the industry to couple the CPU and the GPU on the same die, which results in faster memory transfer between the two. The specific location of the memory allocation is controlled by the memory flags when calling the allocation function `clCreateBuffer()`. The flags are: `CL_MEM_USE_HOST_PTR`, `CL_MEM_ALLOC_HOST_PTR`, and `CL_MEM_COPY_HOST_PTR`. `CL_MEM_USE_HOST_PTR` indicates that the program commands the OpenCL to use the memory referenced by the host pointer as the storage space for the memory object. OpenCL implementation is allowed to cache the buffer contents pointed to by host pointer in device memory. This cached copy can be used when kernels are executed on a device. `CL_MEM_ALLOC_HOST_PTR` specifies that the application commands the OpenCL to allocate the memory from the host-accessible memory. `CL_MEM_COPY_HOST_PTR` indicates that the program commands the OpenCL to allocate the memory for the memory object and copy the data from memory referenced by the host pointer. We discovered that, giving the OpenCL ability to allocate memory by specifying `CL_MEM_ALLOC_HOST_PTR` flag, results in performance gain on the run of both Algorithm 1 and Algorithm 2.

4.3 GooMF and RiTHM

Run-time Time-triggered Heterogeneous Monitoring (RiTHM) takes a C program and a set of Ltl properties as input and generates an instrumented C program that is verified at run time by a *time-triggered* monitor. It supports two monitoring modes: the program either *self-monitors* itself (i.e., the monitoring code is weaved with the input program), or it incorporates an *external monitor* thread. In both cases, the monitor is invoked in a time-triggered fashion, ensuring that the states of the program can be reconstructed at each invocation by using efficient instrumentation. The verification decision procedure is sound and complete and takes advantage of the GPU many-core technology to speedup monitoring and reduce the run-time monitoring overhead.

4.3.1 RiTHM Overview

Figure 4.2 shows the different modules and detailed data flow of RiTHM. The tool takes a C program and a set of Ltl properties as input and generates an *instrumented* C program as output. We, in particular, use the 3-valued linear temporal logic (LTL_3) designed

particularly for RV [6]. Each $\text{L}\textsc{tl}_3$ property is specified in terms of variables of the given C program. For instance, `G(x >= 10 and foo.y = z)` is one such property, where `x` and `z` are two global variables and `y` is a local variable of function `foo`.

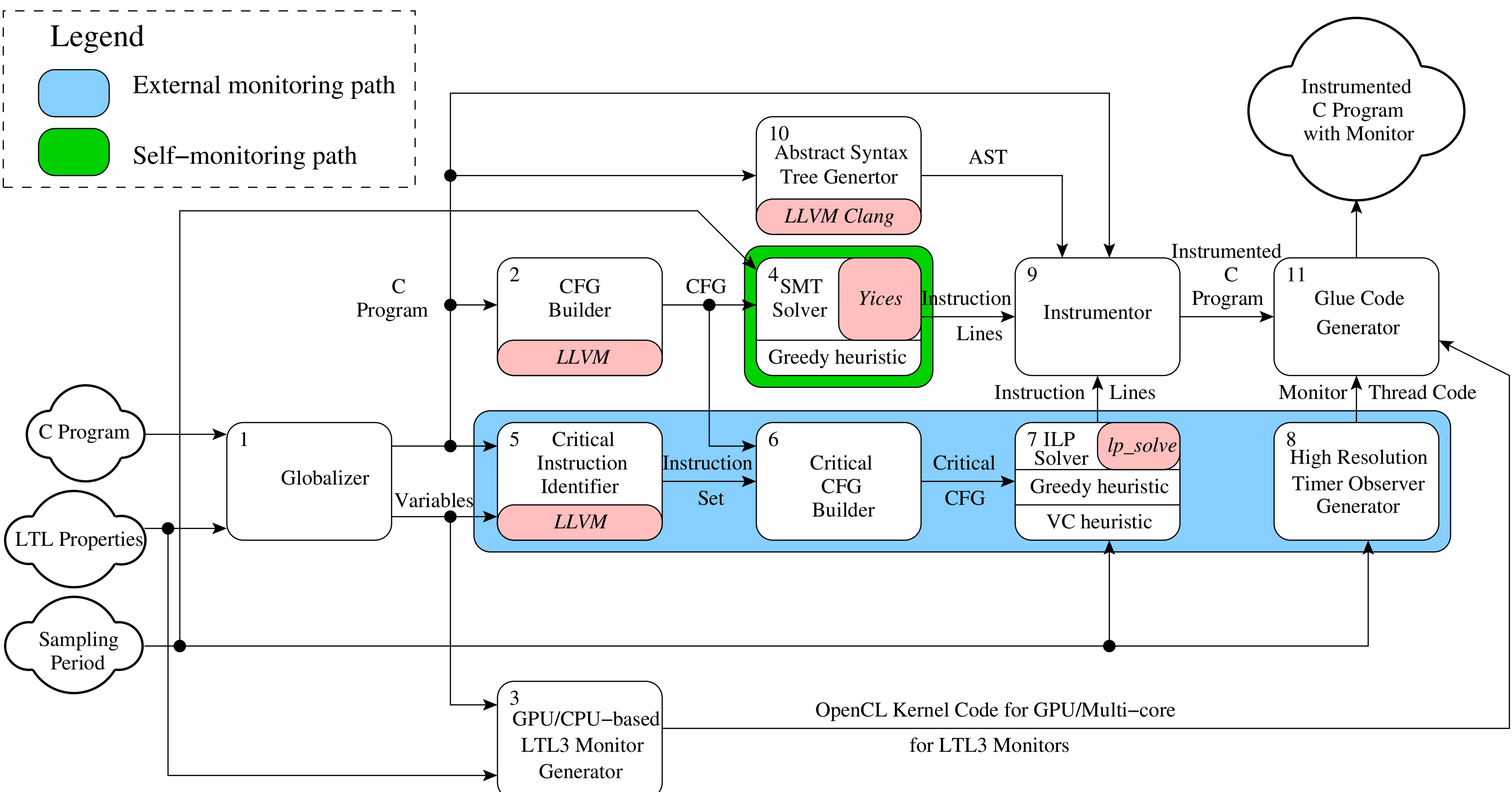
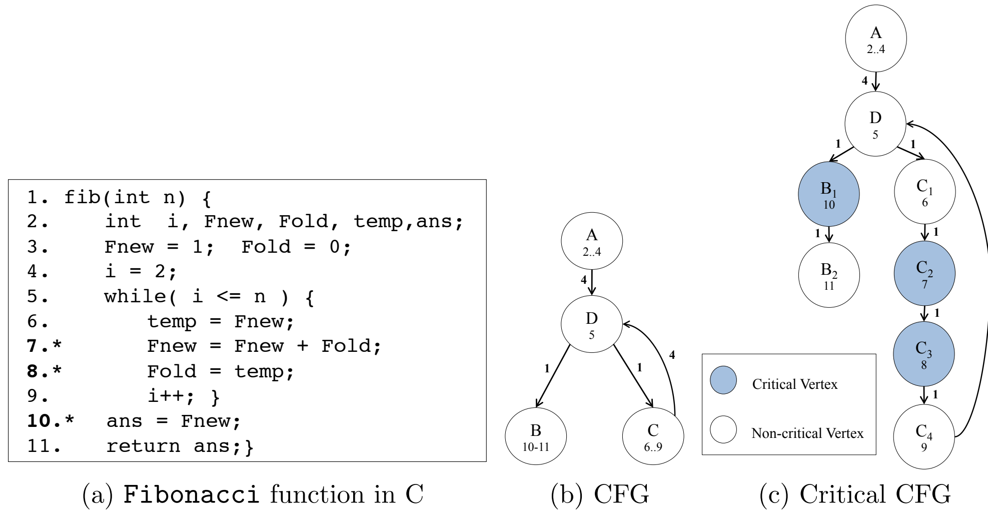


Figure 4.2: Building blocks and data flow in RiTHM.



(a) `Fibonacci` function in C (b) CFG (c) Critical CFG

Figure 4.3: Example of a program and CFG.

Module 1 (in Figure 4.2) is *Globalizer*, which takes the C program and LTL specification as input and generates a C program, where all the variables that participate in the LTL properties are changed into global variables in the C program. Globalizer also generates the list of the globalized variables and passes it to the LTL_3 *Monitor Generator* (module 3) and *CFG Builder* (module 2) that generates the C program's *control–flow graph*. This module is implemented over `LLVM` [30]. For example, the CFG of the program in Figure 4.3a (Fibonacci function) is shown in Figure 4.3b, where the weight of each arc is the best-case execution time of the instructions in the originating vertex. For simplicity, we assume that each instruction takes one time unit.

Using the generated CFG, *Critical CFG Builder* identifies the vertices (called *critical vertices*) that may change the valuation of the LTL properties (module 6). The resulting CFG is called *critical* CFG (see Figure 4.3c).

Given a CFG, RiTHM can generate two types of instrumented programs with respect to the techniques described in detail in [10, 11, 33]. Based on those instrumentation techniques, the program undergoes the following transformations:

- A program, where monitoring instructions are weaved into the program (module 4); i.e., the program self-monitors itself (the green path in Figure 4.2).
- A program augmented with a TTM thread (modules 5-8); i.e., an external thread that monitors the program (the blue path in Figure 4.2).

```
1. fib(int n) {
2.      int  i, Fnew, Fold, temp,ans;
3.      Fnew = 1;  Fold = 0;
4.      i = 2;
5.      while( i <= n ) {
6.         temp = Fnew;
7.*     Fnew = Fnew + Fold;
8.         current_program_state.Fnew = Fnew;
9.         next_state(context,
                (void*)(&current_program_state));
10.*    Fold = temp;
11.        i++; }
12.*  ans = Fnew;
13.   current_program_state.ans = ans;
14.   next_state(context,
                (void*)(&current_program_state));
15.   flush(context);
16.   return ans;}
```

(a) Instrumented `Fibonacci` function for external monitoring.

```
1. fib(int n) {
2.    int  i, Fnew, Fold, temp,ans;
3.    current_program_state.Fnew = Fnew;
4.    current_program_state.ans = ans;
5.    next_state(context,(void*)(&current_program_state));
6.    Fnew = 1;  Fold = 0;
7.    i = 2;
8.    while( i <= n ) {
9.       current_program_state.Fnew = Fnew;
10.      current_program_state.ans = ans;
11.      next_state(context,
         (void*)(&current_program_state));
12.      temp = Fnew;
13.      Fnew = Fnew + Fold;
14.      Fold = temp;
15.      i++; }
16.   ans = Fnew;
17.   current_program_state.Fnew = Fnew;
18.   current_program_state.ans = ans;
19.   next_state(context,(void*)(&current_program_state));
20.   flush(context);
21.   return ans;}
```

(b) Instrumented `Fibonacci` function for self-monitoring.

Figure 4.4: Instrumented programs.

The output of either technique is a set of instructions in the input C program that needs to be instrumented to enable sound external/self-monitoring. The lines of code corresponding to the instructions are located using the abstract syntax tree generator (module 10) of `LLVM Clang`, and instrumented by *Instrumentor* (module 9). Finally, *Glue Code Generator* (module 11) augments the instrumented code with the synthesized LTL_3 monitors and proper function calls for verifying properties at run time.

4.3.2 GooMF as a Back End for RiTHM

GooMF serves as a back-end infrastructure for the RiTHM. Since GooMF is a shared library, the integration of the front- and middle- end modules with the verification engine is straightforward: after the instrumentation stage is over, GooMF API calls are inserted at the end of the instrumented vertices in the Critical CFG.

Figure 4.4 demonstrates the results of such instrumentation on the example of previously shown `Fibonacci` function. The result of the external TTM instrumentation path is displayed in Figure 4.4a, whereas Figure 4.4b illustrates the outcome of the self-monitoring path. As can be seen, the number and the instance of instrumented vertices differ for two different methods.

Depending on the instrumentation, `_GOOMF_nextState()` or `_GOOMF_nextEvent()` may be called. The latter is particularly useful for the event-based monitoring, when every change in the variable is monitored. In this case, call to `_GOOMF_nextEvent()` saves the time of copying the whole program state into the verification buffer. Instead, only one (modified) variable is copied, and GooMF takes the rest from previous program state.

For the sake of broader usability of the tool, we allow the user to choose the verification platform by choosing appropriate verification algorithm when specifying the instrumentation parameters. Therefore, RiTHM can work on machines with or without GPU.

Chapter 5

Towards Parameterized Monitoring

This chapter describes the first steps towards integration of parameterized monitoring and the proposed parallel algorithms. Section 5.1 offers a high-level overview of parameterized monitoring and presents basic definitions. Section 5.2 surveys the state-of-the-art in parameterized monitoring and the problems raised by it. Finally, Section 5.3 discusses the steps towards practical incorporation of parameterized monitoring into the finite- and infinite-history algorithms. Section 5.4 evaluates the algorithms' performance in light of the aforementioned changes.

5.1 Introduction to Parameterized Monitoring

Parameterized run-time monitoring is an extension of a more traditional run-time monitoring, where verified properties may have one or more unbound universal quantifiers, which, when instantiated, produce a new *monitoring instance*. In addition, the verified properties may depend on variables other than those in program state. These special variables are recorded at specific points of time and virtually add state to the monitoring instance.

To illustrate the concept, consider commonly used property "if file is open, it must be eventually closed": $\Box(open \rightarrow \Diamond close)$. With conventional future-time LTL formalism, this property requires separate definition for every possible file:

$$\Box((action == open \wedge file == f_1) \rightarrow \Diamond(action == close \wedge file == f_1));$$
$$\Box((action == open \wedge file == f_2) \rightarrow \Diamond(action == close \wedge file == f_2));$$
$$\ldots$$
$$\Box((action == open \wedge file == f_n) \rightarrow \Diamond(action == close \wedge file == f_n));$$

This cumbersome definition imposes several problems. First, and the most obvious, need of maintaining multiple properties definitions: when the property or predicate evolve, the developer is required to update all the definition instances. Second, the developer has to know in front about all possible files open / closed in the system. Third, every property definition by default represents separate monitoring instance. Therefore, there is an unnecessary overhead of running the monitoring instances for yet-to-be-created files.

On the contrary, one can use the parameterized syntax for the same property: regular LTL syntax augmented with first order quantification:

$$\forall f \in F \ : \ \Box((action == open \wedge file == f) \rightarrow \Diamond(action == close \wedge file == f))$$

where F is the set of all possible file descriptors and f is a *parameter*. Properties that use parameters are called *parametric properties*. Note that set F can be unknown before the actual program run. During the program execution, new monitor instance is spawned every time the *open* event is met with new parameter f, thus saving the overhead caused by not-yet-created monitor instances. Even shorter version of the property $\Box(open\,\langle f\rangle \rightarrow \Diamond close\,\langle f\rangle)$ assumes implicit universal parameter f. We use pointy brackets to distinguish between regular parentheses and parameterized events.

Defining Variables. More formally, parameter f can be seen as a special program state variable that defines the monitoring instance of the property. Given the parameterized property φ, we call the set of such variables *defining variables of the property* and denote it as $DV(\varphi)$. The notion of *defining variables* was also introduced in [7] and used excessively in run-time verification framework J-Lo. Naturally, $DV(\varphi) \in V$, where V is a set of variables used in the program at any given time. Normally, a program state represents a mapping of a subset of V to values. The event of associating specific value with parameter and, consequently, with monitoring instance, is called *bind* or *binding event*. The opposite event of dissociating a parameter value is called *unbind* or *unbinding event*. Note that $DV(\varphi)$ can contain several variables, thus delaying the actual instantiation of the monitor up to the point when all the variables are bounded. Only from that moment, the monitoring

will be intelligible. Therefore, the verification tool that would implement parameterized monitoring should handle the management of those partially instantiated monitors. Later in Section 5.2 we will see how verification frameworks J-Lo and JavaMOP deal with this issue. Classical example of multiple defining variables is the safety property that describes the map iteration in Java from [26]:

$$\begin{aligned}\forall m, c, i \;:\; & \Box(getset\,\langle m, c\rangle \wedge \Diamond(getiter\,\langle c, i\rangle \\ & \wedge \Diamond((modifyMap\,\langle m\rangle \vee modifyCol\,\langle c\rangle) \wedge \Diamond useiter\,\langle i\rangle)))\end{aligned}$$

Event *getset* represents the moment when the collection c is retrieved from the map m. This event effectively binds two of the parameters for this property. The monitoring instance of the property is only instantiated when *getiter* is met, which finally binds the actual iterator i. From that point and on, events $modifyMap$, $modifyCol$, and *useiter* relate to the specific monitoring instance.

Used Variables. As opposite to parametric properties that require instantiation and result in multiple instances, there is a class of parametric properties that will only have one instance. Consider, for example, third property from Subsection 3.3.3:

$$\varphi_1 = \Box\,(d \rightarrow (\bigcirc\neg d) \vee (\bigcirc\bigcirc\neg d) \vee (\bigcirc\bigcirc\bigcirc\neg d))$$

Predicate d stands for "number of satellites less than three". φ_1 verifies that this undesirable state will not last more than four consecutive program states. One can notice that the property syntax is not "scalable": if the undesirable state of insufficient number of satellites is extended to ten consecutive program states, the length of the property will explode linearly. To eliminate this dependency, parameter is required:

$$\varphi_1 = \Box\,(d\,\langle p\rangle \rightarrow ((ts < p + x)\ \mathbf{U}\ \neg d)))$$

Here, p is a parameter, ts is a variable in a program state representing the timestamp, and x is a constant representing maximal allowed duration of the unhealthy system state. Although this property is parameterized, it only has one instance active from the very beginning of the program run, since there can be only one unhealthy system state. As a result, the property will always have only one instance ($DV(\varphi) = \emptyset$), but will use ts variable as the parameter. We call such variables *used variables of the property* and denote the set of these variables as $UV(\varphi)$. Note that $UV(\varphi)$ does not have to be a subset of V, but a function on variables from V. For example, the property above may need to

store ts in a different format than that in the program. In this case, $UV(\varphi) = \{p = toNanoseconds(ts)\}$, where $toNanoseconds()$ is a conversion function. Similar to defining variables, the notion of *used variables* was introduced in [7].

In the previous example, the parameter p has to be bound upon receiving the first program state with $d = true$. This brings us to the point where LTL syntax is no longer enough to express the semantics of parameterized property, because the binding event depends not only on the program state, but also on the current state of the monitor. More precisely, binding the p to the value of ts is only necessary when first unhealthy state is encountered. FSM, on the other hand, is the perfect formalism for expressing the parametric specifications as it allows to capture the event of binding during specific state transition.

Figure 5.1a shows FSM structure for property φ_1. Figures 5.1b and 5.1c show the FSMs for properties φ_2 and φ_3 respectively. Binding actions are displayed on top of the transition edges with symbol "→" denoting the action of association of a parameter with a value. The defining and used variables sets are shown below. The properties displayed in the figure represent different types of the parametric properties:

- φ_1, as mentioned previously, maintains only one instance and binds the parameter p once per unhealthy state. Once p is bound, the monitor either unbound when the healthy state is over, or eventually ends up in the violation state. As the property has only one monitoring instance, the unbinding action does not result in a monitor destruction. $DV(\varphi_1) = \emptyset, UV(\varphi_1) = \{ts\}$.

- $\varphi_2 = \Box(open\,\langle file\rangle \rightarrow \Diamond close\,\langle file\rangle)$ is the file open–close property already discussed at the beginning of this section. The binding action occurs once the file is open, and the unbinding action occurs once one of the open files is closed. The binding action triggers the creation of the new monitoring instance; the unbinding action triggers the destruction of the instance. $DV(\varphi_2) = \{file\}, UV(\varphi_2) = \emptyset$.

- $\varphi_3 = \Box(a\,\langle item, bid_amount\rangle \rightarrow (b\ U\ c))$ describes a bidding process: once a bid on specific item is posted, the bid amount can only go higher. Predicate a stands for first bid on the item. It effectively binds variables *item* and *bid_amount* to parameters p_1 and p_2. Predicate b stands for $bid_amount > p2 \vee item \neq p1$: the legal (higher) bid on the bidden item or other activity on the unbidden items. Predicate c stands for the bid closure. This property has an instance for every bidden item. As expected, binding of *item* to p_1 triggers the creation of the new monitoring instance, while unbinding p_1 triggers the destruction. Binding *bid_amount* to p_2, however, does not trigger creation. $DV(\varphi_3) = \{item\}, UV(\varphi_3) = \{bid_amount\}$.

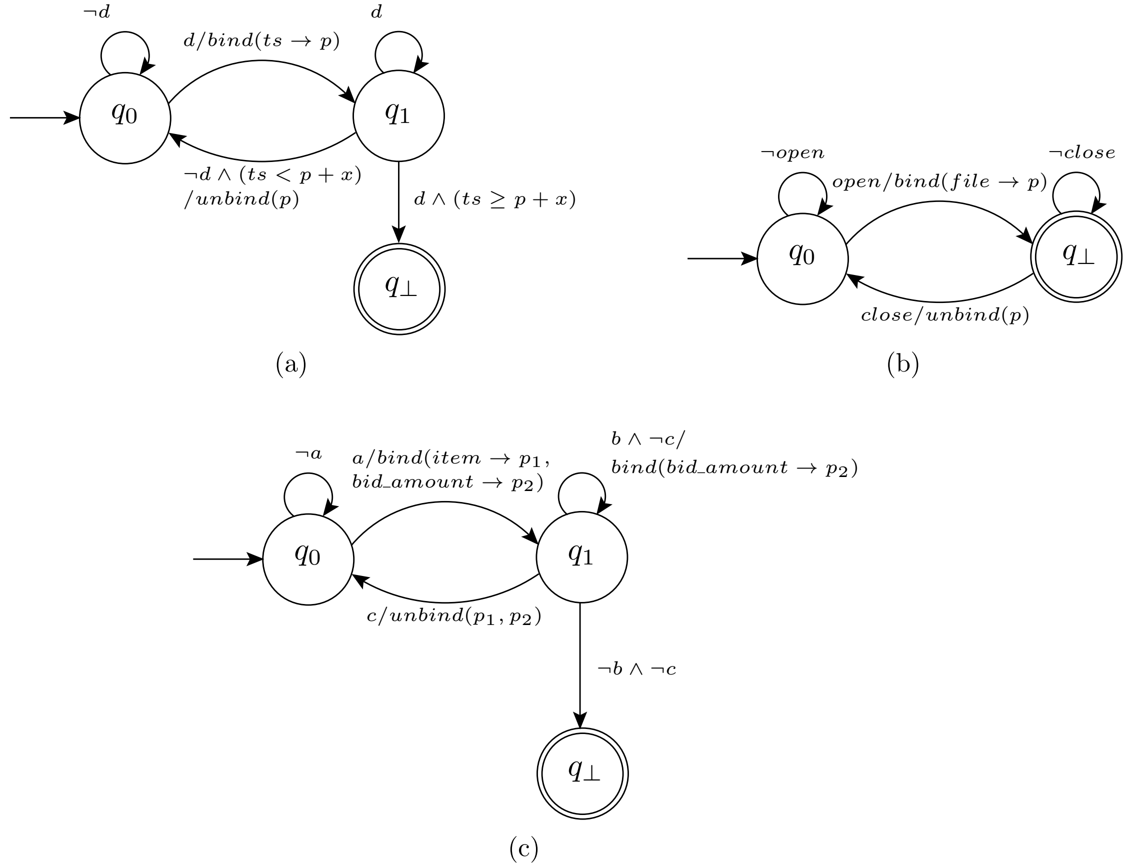


Figure 5.1: Monitors for a sample of parametric properties.

These examples demonstrate the value of the parameters: both used and defining variables empower monitor to save additional history inside the monitor, but there is a basic difference between them. Defining variables are only used to relate the events in the program to the correct monitoring instance, whereas used variables simply add another dimension to the state of the monitor.

After introducing the notions of used and defined variables and augmenting monitoring FSMs with bind–unbind actions, we revise the definition of the monitor:

Definition 11 (Monitor for Parametric Property) *Let φ be a parametric* LTL$_3$ *formula over alphabet $\Sigma \cup P$, where P is a set of parameters. The* monitor $\mathcal{M}^{\varphi}$ *of φ is the unique FSM $(P, \Sigma \cup P, Q, q_0, \delta, \Phi, \rho, \lambda)$, where Q is a set of states, q_0 is the initial state, δ is*

a transition relation, and λ *is a function that maps each state in* Q *to a value in* $\{\top, \bot, ?\}$*, such that:*

$$[u \models \varphi] = \lambda(\delta(q_0, u)).$$

In addition, Φ *is a set of bind–unbind actions on* P *plus empty action and* ρ *is a function that maps each transition to a bind–unbind action:* $\rho : \{Q \times Q\} \rightarrow \Phi$. ■

This definition abstracts away the notion of defined and used variables. Also, it does not limit Φ to be an assignment to a parameter. Rather, action in Φ can be seen as a function in C (i.e. $toNanoseconds()$ for φ_1).

5.2 State of the Art in Parameterized Monitoring

The mechanism for parameterized monitoring was first implemented in verification tool Eagle and in Hawk [16], built on top of Eagle. Although lacking the semantic division between used and defining variables, Eagle and Hawk support instantiation of multiple monitoring instances per property. The verification ability of these tools, however, is constrained by incomplete property specification semantics (only $\Box$ and $\Diamond$ temporal operators are allowed) and by the restrictions on monitoring points (the events are only reported upon function execution and return).

Like Eagle and Hawk, J-Lo binds variables to free parameters, but is not restricted to specific monitoring points. This is due to the use of AspectJ as an instrumentation engine. The authors specifically distinguish between the defining and used variables and introduce new formalism *Dynamic LTL* to define parametric properties over the program space. One of the interesting features of J-Lo is the ability to automatically differentiate between defining and used variables and treat them differently.

Most of the tools that support parameterized monitoring, tightly couple the handling of parameter bindings with the property checking, yielding monolithic but supposedly efficient monitors. Such combinations of parameter binding and property checking result in extremely complex and formalism-specific algorithms, hard or impossible to adapt to other formalisms. More recent attempt of implementing parameterized monitoring is JavaMOP — a variation of JavaPaX tool, but specifically dedicated to run-time monitoring. JavaMOP was built with the intention to decouple specification formalism and implementation of parameterized binding, which resulted in a variety of possible formalisms the framework can work with (see Section 4.1). Specifically, JavaMOP uses the technique called "trace slicing": the execution trace is sliced for each parameter instance, so that a monitor for each

parameter instance receives only relevant program events. As a result, every monitoring instance is independent of parameters, resulting in a formalism-independent architecture. The major problem with this approach is the fact that the system needs to dispatch each event to the corresponding monitor given the parameter instance of the event.

Another big challenge in fully implemented parameterized monitoring is the difficulty of managing partly and fully instantiated monitoring instances. Consider the following property from Section 5.1:

$$\begin{aligned}\forall m, c, i \ : \ & \Box(getset\,\langle m, c\rangle \wedge \Diamond(getiter\,\langle c, i\rangle \\ & \wedge \Diamond((modifyMap\,\langle m\rangle \vee modifyCol\,\langle c\rangle) \wedge \Diamond useiter\,\langle i\rangle)))\end{aligned}$$

The monitoring of this property should begin from the point where all the defining parameters are bound. Naturally, this will not happen at once. First, the framework will bind map variable, then, given the bound parameter, the collection variable. Then, given both bound parameters, iterator variable will be bound. In the meanwhile, the monitoring framework should keep these partially initialized structures in the memory, where they can be fetched from and connected to the upcoming events.

The aforementioned two challenges were the reason why the first version of JavaMOP could not handle properties with more than one defining variable [14]. Jin in his doctoral thesis [26] presents the next version of JavaMOP and describes the proposed solution in detail. The new data structure for the indexing tree is proposed, which contains all of the mappings from parameter objects to monitors. The indexing tree is an efficient means to locate the monitors for a given parameter instance. The authors implement it as a multi-level map that, at each level, indexes each parameter object of the parameter instance. As expected, the performance check revealed that this data structure is the key bottleneck in terms of both run time and memory performance. This is because the size of the indexing tree grows as the specification creates more monitors. The authors tackle this issue to some extent by combining the indexing trees if their defined parameter types share the same prefix.

Some other impressive JavaMOP features are: (1) supporting specification inheritance for reusing specifications (similarly to object inheritance); (2) special optimization that avoids monitoring partially instantiated parametric instances; (3) garbage collection of the monitor instances that reached a final state. All these qualities make JavaMOP positively one of the most successful and mature monitoring frameworks. However, similarly to J-Lo, JavaMOP relies on some of the features of Java (objects structure, objects meta data, garbage collection, hash functions incorporated in every object, existence of weak references etc). This dependency makes it difficult to port the tools to other, possibly

non-object-oriented, languages. For example, JavaMOP keeps the initialized monitoring instances in a special hash–tree data structure, which is accessible by hash–code of the monitored objects. Obviously, this solution is not suitable for implementation in C and C++.

5.3 Parallel Verification amid Parameterized Monitoring

In the previous section, we discussed the difficulties associated with proper employment of parameterized monitoring on the example of JavaMOP. In our case, complete and sound implementation of parameterized monitoring is also constrained by application of parallel algorithms. This work is a first step towards incorporation of parameterized monitoring into the finite- and infinite-history algorithms.

5.3.1 Algorithms and Parameterized Monitoring

Intuitively, parameters introduce an additional level of uncertainty for parallel algorithms. The decision about monitor state transition now depends not only on current monitor state and a program state, but also on the values of bound parameters. When processing several program states in parallel, the $(i+1)$-th transition does not now the real values of the bound parameters, as those may alter during the processing of i-th transition. Following paragraphs give a detailed description of how parameterized monitoring influences the set of proposed algorithms.

Parameterized monitoring with sequential and partial–offload algorithms. Both sequential and partial–offload algorithms can handle parameterized monitoring without serious changes. Let φ be a parametric property in LTL_3^{mon} and $\mathcal{M}^{\varphi} = (P, \Sigma \cup P, Q, q_0, \delta, \Phi, \rho, \lambda)$ be its monitor; for every program state both algorithms perform the following steps:

1. New program state s_i is received.

2. Based on s_i, perform calculations to map predicates (the LTL formula consists of) to either $true$ or $false$ (possible on GPU for partial–offload algorithm).

3. Based on the values of the predicates, make the appropriate transition from q_{i-1} to q_i. If transition carries an action ($\rho(q_{i-1}, q_i) \neq \emptyset$), perform it.

This flow is similar to the one in conventional sequential and partial–offload algorithms from Chapter 3 except for the last step, where bind–unbind actions are taking place.

Algorithm 3 For parametric finite-history LTL_3^{mon} properties

---

Input: A monitor $\mathcal{M}^{\varphi} = (P, \Sigma \cup P, Q, q_0, \delta, \Phi, \rho, \lambda)$, a state $q_{current} \in Q$, and a finite program trace $\sigma = s_0 s_1 s_2 \ldots s_n$.
Output: A state $q_{result} \in Q$ and $\lambda(q_{result})$.

    /* *Initialization* */
1: $StartIndex \leftarrow 0$
2: $\mathcal{I} \leftarrow \langle n+1, q_{current} \rangle$
3: $\mathcal{P} \leftarrow \emptyset$
4: Let m be a mutex

    /* *Parallel computation of next monitor state given the current state and state of the parameters* */
5: **for all** $(s_i, StartIndex \leq i \leq n)$ in parallel **do**
6:     $q_{result} \leftarrow \delta(q_{current}, s_i)$
7:     lock(m)
8:     **if** $((q_{current} \neq q_{result} \vee \rho(q_{current}, q_{result}) \neq \emptyset) \ \wedge \ i < key(\mathcal{I}))$ **then**
9:         $\mathcal{I} \leftarrow \langle i, q_{result} \rangle$
10:         Perform action $\Phi(\rho(q_{current}, q_{result}))$ and update $\mathcal{P}$
11:     **end if**
12:     unlock(m)
13: **end for**

    /* *Obtaining the result* */
14: **if** $key(\mathcal{I}) = n+1$ **then**
15:     **return** $q_{result}, ?$
16: **else**
17:     $q_{result} \leftarrow value(\mathcal{I})$
18:     **if** $\lambda(q_{result}) \neq ?$ **then**
19:         **return** $q_{result}, \lambda(q_{result})$
20:     **end if**
21:     $q_{current} \leftarrow q_{result}$
22:     $StartIndex \leftarrow StartIndex + key(\mathcal{I}) + 1$
23:     **goto** line 2
24: **end if**

Parameterized monitoring with finite-history parallel algorithm. Amid parameterized monitoring, the decision about monitor state transition depends not only on

current monitor state and a program state, but also on the values of bound parameters. Upon every bind–unbind action, the algorithm should update the state of the parameters, and thus, will need additional iterations for every such action. The changes are shown in parametric version of finite-history algorithm in Algorithm 3. The algorithm takes a finite program trace $\sigma = s_0 s_1 \ldots s_n$, a monitor $\mathcal{M}^{\varphi} = (P, \Sigma \cup P, Q, q_0, \delta, \Phi, \rho, \lambda)$, and a *current* state $q_{current} \in Q$ (possibly q_0) of the monitor as input and returns a resulting state q_{result} and the monitoring verdict $\lambda(q_{result})$ as output.

We only describe changes to the original algorithm. Line 3 initializes a structure that holds the state of the parameters to the default value. In lines 8–10, tuple $\mathcal{I}$ is set to $\langle i, q_{result} \rangle$ and the state of the parameters is updated. It happens not only if program state s_i results in enabling a transition from current monitor state $q_{current}$, but also if the transition is accompanied with the action from Φ. Note that the transition can be a self–loop. This way, at the end of this phase tuple $\mathcal{I}$ will contain either the left-most transition, or the left-most bind–unbind action and P will be updated with right values. The rest of the algorithm remains unchanged.

Parameterized monitoring with infinite-history parallel algorithm. Infinite-history algorithm relies on exhaustive computations to eliminate interdependencies among the data items. Not knowing current monitor state, the algorithm calculates next state for every possible current state given a program state as an input. With the addition of parameterized monitoring, there is an additional input to the algorithm — the state of the parameters. Although it possible to pre-calculate predicate value for every inconclusive state in the monitor FSM, it is more difficult to do this for every parameter value. In most cases, the domain of the defining and used variables in programs is unbound (think about all possible file names for the file open–close property), which makes it virtually impossible to predict. This is the reason why we adapted the finite-history algorithm for parameterized properties, but not the infinite-history algorithm.

The above explanation illustrates the difficulties associated with incorporating parameters binding into the algorithms core. It is clear that every bind–unbind action changes the state of the monitoring and, consequently, the performance of the parallel algorithms. We revise the definition of the history and history length accordingly:

Definition 12 (History amid Parameterized Monitoring) *Let φ be a parametric property in* LTL_3^{mon} *and $w \in \Sigma^{\omega}$ be an infinite word. The* history *of φ with respect to w is the sequence of states $\mathbb{H}_w^{\varphi} = q_0 q_1 \ldots$ of $\mathcal{M}^{\varphi} = (P, \Sigma \cup P, Q, q_0, \delta, \Phi, \rho, \lambda)$, such that $q_i \in Q$ and $q_{i+1} = \delta(q_i, w_i)$, for all $i \geq 0$.*

Definition 13 (History Length amid Parameterized Monitoring) *Let φ be a parametric property in* LTL_3^{mon} *and $w \in \Sigma^\omega$ be an infinite word. The* history length *of φ with respect to w (denoted $\|\mathbb{H}_w^\varphi\|$) is the number of state transitions in history $\mathbb{H}_w^\varphi = q_0 q_1 q_2 \ldots$, such that $q_i \neq q_{i+1}$ or $\rho(q_i, q_{i+1}) \neq \emptyset$, for all $i \geq 0$. The history length of a property is then:* $\|\mathbb{H}^\varphi\| = \max\{\|\mathbb{H}_w^\varphi\| \mid w \in \Sigma^\omega \ \wedge \ w \text{ has a minimal good/bad prefix}\}$

The definitions of good and bad prefixes remain unchanged. The history length of parametric property is at least the history length of same-structured non-parametric property. As a history length of a property is a measure of performance of the finite-history algorithm, the execution time of this algorithm on parametric properties is expected to rise due to increased history length. The execution time of sequential and partial–offload algorithms is expected to rise due to the bind–unbind actions, but not as significantly as the execution time of the finite history algorithm.

5.3.2 GooMF and Parameterized Monitoring

GooMF implements parameterized monitoring only partially. Specifically, it allows monitoring parametric properties with an empty set of defining variables. In other words, GooMF can manage additional state introduced by used variables, but does not handle online creation and destruction of monitoring instances introduced by defining variables. In general, if defining variables set of the property is empty, it is enough to keep only the current state for each monitoring instance and, consequently, for each property. Therefore, the evolution from regular to parameterized monitoring is reduced to the following changes: (1) for every property, the system should keep track not only of current monitor state, but also of the current value of the parameters; (2) for every binding event, the parameter value is updated with the value of used variable; (3) for every unbinding event, the parameter value should be set the the “undefined” value.

As mentioned in Section 5.1, proper specification of parametric properties requires specification formalism other then LTL. To accommodate this requirement, GooMF allows two formalisms for properties specification: LTL and FSM. FSM should be used when parametric properties are monitored. In this case, binding actions are specified on the edges. This brings us to the question of FSM encoding in the code. If the FSM is encoded as an array, the event of the monitor state update is merely an assignment to the variable. On the contrary, if the FSM is encoded as a sequence of `if` operators, it will allow arbitrary code to be inserted as a result of following the specific edge. Thus, the choice of `if` and FSM representation in the configuration file is obligatory if parametric

properties are to be monitored. Following is the proper example of parametric property definition from GooMF configuration file:

```
properties = (
{name = "prop1"; formalism="LTL"; encoding = "array"; syntax = "[] (a && b)"},
{name = "prop2"; formalism="FSM"; encoding = "ifs"; syntax =
"digraph G {
(0, 0) -> (1, 1) [label = (e&&f), action = "used_params->xtimestamp = gps_second;
used_params->x_d = GetX(llh_lat_dest, llh_lon_dest, llh_alt_dest); used_params->xtsat =
XTSAT;"];
(0, 0) -> (1, 1) [label = (e), action = "used_params->xtimestamp = gps_second;
used_params->x_d = GetX(llh_lat_dest, llh_lon_dest, llh_alt_dest); used_params->xtsat =
XTSAT;"];
(0, 0) -> (0, 0) [label = (f)];
(0, 0) -> (0, 0) [label = (<empty>)];
(1, 1) -> (1, 1) [label = (e&&f)];
(1, 1) -> (-1, 2) [label = (e)];
(1, 1) -> (0, 0) [label = (f)];
(1, 1) -> (0, 0) [label = (<empty>)];
(-1, 2) -> (-1, 2) [label = (e&&f)];
(-1, 2) -> (-1, 2) [label = (e)];
(-1, 2) -> (-1, 2) [label = (f)];
(-1, 2) -> (-1, 2) [label = (<empty>)];
(-1, 2) [label=(-1, 2), style=filled, color=red]
(1, 1) [label=(1, 1), style=filled, color=pink]
(0, 0) [label=(0, 0), style=filled, color=darkseagreen1]
}"
} );
```

Here, property `prop1` is regular and property `prop2` is parametric (φ_1 from Section 5.1) in AT&T FSM format. Two first transition lines (from state $(0, 0)$ to $(1, 1)$) are instrumented with the binding action.

Another aspect of GooMF that needed to be changed in order to accommodate parameterized monitoring is the input to the algorithm kernel. When using parallel algorithms, the predicates are evaluated on the GPU side; hence, the parameters' values should be readily available to the GPU threads. This is the reason for keeping the special structure of current parameters for all the parametric properties in the system. This structure passed to the GPU upon every iteration of Algorithm 3.

5.3.3 Future Extensions

The infinite-history algorithm uses exhaustive computations to eliminate future dependencies on the data. Particularly, it calculates future monitor state transitions given every possible current monitor state. As such, it sacrifices the calculation time for the precise

future prediction of the next monitor state. Parameterized monitoring introduces additional uncertainty: to calculate the next monitor state, the algorithm has to know not only current monitor state, but also values of the bound parameters if such exist. This uncertainty spans over the domain of the variables. For example, for the classic open–close property, the infinite-history algorithm has to figure out what file descriptors are to be opened / closed. For some programs and parameters, this domain can be limited to several values and known beforehand; for others the domain can be infinite. The algorithm can get help from static analysis to learn about the domain of the parameters.

Another option is to employ a learning phase to learn about possible reduction of the parameter domain. Off course, as the size of the domain grows, the performance of the algorithm slows down. In its behavior, the size of the parameters domain is similar to the number of the inconclusive states in the monitor FSM generated from the property.

5.4 Evaluation of Algorithms amid Parameterized Monitoring

5.4.1 Experiment Setup

The goal of this experiment section is two-fold: (1) to assess the effect of the parameters on the performance of sequential, partial–offload, and finite-history algorithms; (2) to evaluate the performance of the partial–offload algorithm, missing from the evaluation in Section 3.3. As mentioned in the previous section, parameterized monitoring leads to an additional functional complexity and an additional amount of unpredictability. As a result, the infinite-history algorithm cannot handle parametric properties and therefore, is not included in current experiments.

As an input to the algorithms, we use the log file from the same UAV from the case study in Subsection 3.3.3. The size of the log file is about 30,000 program states. It was recorded during the flight conducted in September, 2012 near Greater Napanee, Ontario. The flight's purpose was to conduct the aerial inspection of the solar station and to pinpoint possible problems with the structure of the solar plant. By using the same input for all the experiments, we eliminate the need in throughput measurements. Instead, we measure the execution time of the log analysis as a performance metric of the algorithms.

We use GooMF for the experiments. During the initialization phase, the program reads the log file into the memory and then repeatedly calls for blocking `_GOOMF_nextState()` until the end of the input log. This way the total execution time is not constrained by IO, but

only by verification time. The time measurement starts only after the first buffer of the program states is processed. This allows GooMF to warm up and eliminates possible time skew due to the initialization of the first buffer. All the experiments are conducted on 32-bit Ubuntu Linux 11.10. The sequential algorithm measurements were collected on AMD A6 3500 APU with 32GB of 1500MHz main memory. The parallel algorithms run on AMD Radeon HD5870 discrete GPU and AMD Radeon HD6530 integrated GPU. Every configuration of factors was tested 100 times, and confidence intervals were updated accordingly.

We monitor the following three properties:

1. Convergence to the destination point for X axes: $\varphi_1 = \Box\,(a \rightarrow \bigcirc(b\ \mathbf{U}\ c)))$
2. Convergence to the destination point for Y axes: $\varphi_2 = \Box\,(d \rightarrow \bigcirc(e\ \mathbf{U}\ f)))$
3. Convergence to the destination point for altitude: $\varphi_3 = \Box\,(g \rightarrow \bigcirc(h\ \mathbf{U}\ k)))$

These properties verify the convergence towards new destination point over three axes. Each property has three parameters: x_d — new destination coordinate; *timestamp* — timestamp of setting the new destination; and t_sat — settling time. Once new destination coordinate x_d is set, the UAV must be in the vicinity of this coordinate within t_sat seconds. Every time new destination coordinate is set, the monitor binds the parameters and jumps to the next state. If the UAV is close enough, the monitor unbinds the parameters and jumps back. The property is violated if after t_sat seconds the UAV is still too far from the new destination. Predicates a, d, g stand for the condition of setting the new destinations. Predicates b, e, h stand for time checking. Predicates c, f, and k include checking of the UAV proximity to the destination. They also contain the conversion of the coordinates from World Geodetic System (WGS) to Earth-Centered, Earth-Fixed (ECEF) to East, North, Up (ENU) Cartesian coordinate system, and thus are especially rich in mathematical operations. The computational load of each property is roughly equal to 14 trigonometric operations + 8 division operations + 2 square root operations + a set of less computationally demanding multiplication, addition and subtraction operations. For the full configuration file defining the properties, predicates, parameters, and program variables for this section, refer to Appendix C.

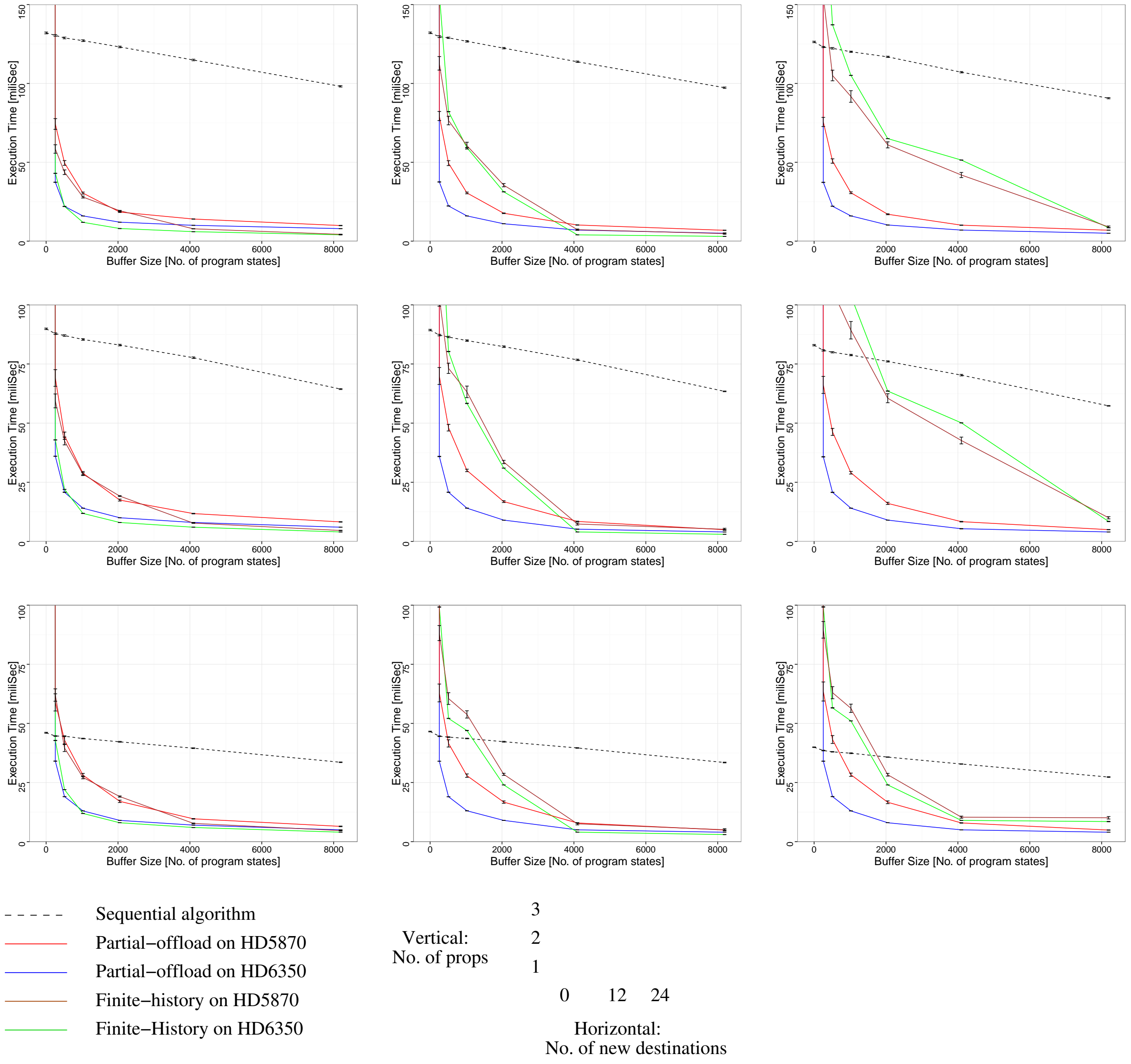


Figure 5.2: Algorithms execution times

5.4.2 Execution Time Analysis

Effect of the Parametric Load

We hypothesize that the execution time of the finite-history algorithm on the parametric properties is expected to rise due to increased history length of the parametric properties. We also believe that the execution times of sequential and partial–offload algorithms are expected to rise due to the bind–unbind actions, but not as significantly as the execution time of the finite-history algorithm. We regard this increase in execution time as *parametric load*. To test the hypothesis, we conducted the experiments with increasing parametric load. Originally, the number of destination points in the log file was 12. We modified the log file to create two additional versions of the input with 0 and 24 destination points. Every destination point set results in two jumps between the monitor states and, consequently, in two finite-history algorithm iterations for each property.

Figure 5.2 shows the results of this experiment (the error bars on each one of the graphs display the 95% statistical confidence interval). The external X-axis represent the parametric load (0, 12, and 24 destination points). The first left-most column of figures shows the execution times of the algorithms with 0 destination points. The middle column of figures shows the execution times of the algorithms with 12 destination points. The right-most column of figures shows the execution times of the algorithms with 24 destination points. As expected, performance of the sequential algorithm (dashed line) and the partial–offload algorithm (blue and red lines) does not change significantly. The execution time of the finite-history algorithm (green and brown lines), on the other hand, grows with the increase in destination points. The execution times of the finite-history and sequential algorithms come closer as the effect from parameterized monitoring grows, although the finite-history algorithm outperforms the sequential on all the scenarios when buffer size reaches 2000 or more.

Effect of the Device

We hypothesize that the parallel algorithms run faster on HD6350 than on HD5870, because shared memory hierarchy on HD6350 results in faster memory transfer between the CPU and the GPU. This is particularly true on the first column of experiments with 0 destination points. Parallel algorithms on integrated HD6350 exhibit similar behavior and outperform the algorithms running on discrete HD5870. For all three computational loads, the algorithms on HD6350 run twice as fast as those on HD5870. However, the parametric

load is a more dominant factor than the device type. On 12 and 24 destination points, the finite-history algorithm is slower than the partial–offload algorithm for both devices.

Effect of the Computational Load

We hypothesize that computational load has the same effect on the algotihms employing parameterized monitoring as on the algorithms that perform regular monitoring. Namely, an advantage of the parallel algorithms over the sequential will grow with an increase in computational load. To test the hypothesis, we conducted the experiments with three different computational loads.

In Figure 5.2, the external Y-axis represent the computational load in number of monitored properties. By utilizing the ability of GooMF to enable and disable properties, we enable only one property in the bottom row of figures, two properties in the middle row, and all three properties in the upper row. As predicted, performance of the sequential algorithm doubles with every enabled property. This is due to the linear dependency of the sequential algorithm on the computational load. This result conforms to the conclusions from Subsection 3.3.2. The execution time of the parallel algorithms rises in much slower pace than that of the sequential algorithm.

In general, for buffer size of 256 and more, the parallel algorithms are always preferable over the sequential. The biggest gap in performance is reached with maximal computational load and minimal parametric load. Due to its immunity from parametric load fluctuations, partial–offload algorithm showed good performance, especially on the third column scenarios. Another intriguing result is the similarity in execution times between the partial–offload and the finite-history algorithms. This result is yet to be examined to find an explanation.

Chapter 6

Related Work

Several techniques have been introduced in the literature for reducing and controlling run-time overhead, and each of the proposed approaches remedies the overhead issue to some extent and in specific contexts. To the best of our knowledge, this work is the first that combines the following: (1) using non-dedicated parallel architecture (GPU) and (2) systematic and formal verification of properties at a run time in a parallel fashion. The suggested architecture allows efficient reduction of the monitoring time originated from properties evaluation and from monitor–program resource-sharing conflicts. In general, we identified four main sources of the monitoring overhead:

1. Invocation of a monitor
2. Calculation and evaluation of property predicates given a program state
3. Possible slowdown due to program instrumentation and program trace extraction
4. Possible interference between a program and a monitor, as monitor shares resources with a program

The overhead caused by each of the above items can be minimized up to a certain degree and in specific contexts. Following is the list of proposed approaches that target overhead originated from the different sources:

- Improved instrumentation (e.g., using aspect-oriented programming [12, 38]) — targets sources 1,3

- Combining static and dynamic analysis techniques (e.g., using typeset analysis [8] and PID controllers [25]) — targets sources 1,3
- Efficient monitor generation and management (e.g., in the case of parametric monitors [32]) — targets source 2
- Schedulable monitoring (e.g., using time-triggered monitors [10]) — targets sources 1,3,4

The main focus in the literature of run-time verification is on *event-triggered* monitors [29], where every change in the state of the system triggers the monitor for analysis. Alternatively, in *time-triggered* monitoring [10], the monitor samples the state of the program under inspection in regular time intervals. The latter approach involves an optimization problem that aims at minimizing the size of auxiliary memory needed for state reconstruction at sampling time. Several heuristics have been introduced to tackle the high complexity of the optimization problem [33]. Similar concept of *predictable* run-time monitoring was proposed in [43], where authors suggest to bind the latency of the error detection by presenting the concepts of monitoring budgets and violation detection latency specifications. All these works, however, utilize sequential monitoring techniques.

There is a body of works that aim at reducing the run-time overhead through an additional pre-processing phase. In [8], the authors use static analysis to reduce the number of residual run-time monitors. The framework in [9] statically optimizes monitoring aspects. Although the objective of this thesis is also reducing the run-time overhead, GPU-based run-time monitoring targets shipping the monitoring costs to a different processing unit. Thus, our approach is orthogonal to the existing approaches in run-time monitoring.

This work proposes to offload the monitoring process to the GPU or other many-core platform parallel to the CPU (which already carries the execution of the inspected program). There are already several works that aim to reduce the overhead by isolating the monitor in a separate processing unit. For example, parallel monitoring has been addressed in [23] by focusing on low-level memory architecture to facilitate communication between application and analysis threads. This approach is not quite relevant to the proposed method in this thesis. The concept of separation of a monitor from the monitored program is also considered in [44, 43, 34]. However, to the best of our knowledge, our work is the first that combines both using non-dedicated parallel architecture (GPU) and systematic and formal verification of properties at a run time in a parallel fashion. Specifically, in [44] the authors suggest using a dedicated co-processor as a parallel profiling architecture. This co-processor implements some common profiling operations as hardware primitives. Similar to our method, instead of interrupting the processor after every sample, multiple

program samples are pre-processed and buffered into a $1K$ buffer before processing. The work in [44, 34] also concentrate on hardware-based monitoring and, hence, the need in a dedicated hardware. On the contrary, our approach utilizes the available many-core platform (GPU or multi-core CPUs) in a computing system.

Finally, in [19] the authors propose using GPU for run-time monitoring, but their approach is limited to data race detection scenarios and not a general formal specification language such as LTL. Consequently, no systematic approach is proposed for generating the code that runs on GPU. For instance, to run experiments, the authors have parallelized ERASER and GOLDILOC algorithms and adjusted them to the syntax of CUDA. In addition, the authors do not resolve interdependencies in data frames. More importantly, their approach is not sound and may result in obtaining false positives.

Sections 1.2, 4.1, 5.2 present an extensive discussion about existing run-time verification tools and frameworks. We shortly mentioned JavaMaC [28], RuleR [4], Eagle [3], Tracematches [1], Spec# [2], P2V [31] and Temporal Rover [17]. We also discussed in detail J-Lo [7] and JavaMOP [26]. We outlined the common characteristics of the existing tools and identified theirs most successful features.

Chapter 7

Conclusions and Future Work

We have presented a concept of parallel run-time verification. The proposed architecture exploits two levels of parallelisms: (1) parallelism between the inspected program and the monitor, and (2) data-level parallelism that allows to evaluate monitor state over multiple program states in parallel. This design reduces the run-time overhead caused by run-time verification and isolates the monitoring module in a separate platform.

The major contributions of this work are the set of novel verification algorithms (Chapter 3) and the monitoring framework for run-time verification of C and C++ programs (Chapter 4). We explored the influence of different factors on the performance of the algorithms, and also presented their practical application on the example of the UAV (Section 3.3). In addition to minimized execution overhead, utilization of the parallel algorithms on practice resulted in significant reduction in power consumption of the system.

Further, we presented a partial integration of parameterized monitoring and our algorithms. Although more powerful in terms of expressiveness, parameterized monitoring introduces additional slowdown in the performance of the algorithms. This tradeoff is explored in experiment Section 5.4.

GooMF source code and documentation are available on https://bitbucket.org/sberkovich/goomf.

7.1 Fitness for Properties

The proposed set of algorithms consists of four algorithms and forms an hierarchy based on the involvement of the GPU in the verification process. In the experimental sections,

we explored the influence of different factors on the performance of the algorithms. In addition, we used history length of the property to predict the performance of the finite-history algorithm. We summarize the most influencing factor effects on the performance of the algorithms:

1. Execution time of the sequential algorithm rises at least linearly with increase in computational volume and number of properties.
2. As history length of the property rises, so does the execution time of the finite-history algorithm. History length of the monitored properties does not affect the execution time of the infinite-history algorithm.
3. As a number of inconclusive states in monitor rises, so does the execution time of the infinite-history algorithm. Number of inconclusive states does not affect the execution time of the finite-history algorithm.
4. GPU-based monitoring is more power-efficient than CPU-based monitoring.

In general, we have shown that given a considerable amount of computations in the properties, the parallel algorithms are advantageous. The same may apply to the large set of computationally trivial properties, just because the sequential execution time grows linearly with the number of monitored properties. Partial–offload algorithm is beneficial for monitoring process with a majority of trivial properties and some computationally intensive properties. In this case, the most "heavy" predicates may be offloaded to GPU. This partial–offload ability makes this algorithm perfect compromise between fully parallel and fully sequential algorithms. When a monitoring is mostly event-based, and predicates evaluations are trivial, sequential algorithm will be the algorithm of choice. Finally, if energy efficiency is a priority, GPU-based verification is preferable over CPU-based verification.

Off course, the actual execution time will depend also on the actual program trace. This closely correlates with the area of applicability of the inspected program. In some cases, only a developer or an end-user can tell how likely the certain condition to happen and how many times. And consequently, how many monitor state transitions are expected to occur. Consider, for example, the properties from Section 5.4. The performance of the finite-history algorithm depends directly on the number of times the new destination point will be set during the UAV flight.

This brings us to an additional aspect that affects the algorithms' performance — number of parametric bindings. As shown in Section 5.1, a history of the property is

affected by a number and a location of a parameters bind–unbind actions. In Section 5.4, this aspect is represented by the factor of the number of new UAV destinations.

Another factor to consider when choosing among the algorithms, is the tolerable delay of the program under inspection. The parallel algorithms are only suitable for systems that can tolerate delay resulted from buffering the program states and verifying all the batch in parallel. If the question of delay is critical, the maximal reaction time can be calculated given the length of the buffer, the size of the program state snapshot, and the speed of the bus.

To summarize, a set of considerations needs to be taken into account when deciding about the instance of the optimal verification algorithm. However, the action of choosing among the algorithms is simplified by GooMF ability to switch among the algorithms by merely modifying one parameter in `_GOOMF_initContext()` function.

7.2 Further Research Directions

Several possible directions for further research exist. In [42], the authors propose a novel verification approach — *hybrid monitoring*. At a run time, the monitor switches between the event-triggered and the buffer-triggered modes to minimize the overhead resulted from the invocation of the monitor. The logic of switching between the monitoring modes could be incorporated into GooMF and be based not only on monitor invocation times, but also on reliable verification times. In this case, it will run in buffer-triggered mode, but will alter the maximal buffer size at a run time. Another step in this direction is allowing GooMF to choose automatically among the verification algorithms. This will probably require additional AI module and some kind of learning phase.

The last version of OpenCL (1.2) introduced new capability: “fission” (i.e. split) a single device into multiple sub-devices at run time. This gives a program greater freedom in assigning work to specific cores, allowing for task-level parallelism and/or better exploitation of temporal and spatial cache locality [35]. This feature can be used in GooMF to assign different properties to different sub-devices, thus achieving not only data-level parallelism, but also functional-level parallelism. Intuitively, number of processing elements in a sub-device assigned to a specific property, can vary according to the computational load that this property introduces.

One of the possible directions to further minimize monitoring overhead is to prioritize the monitored properties. This method is particularly applicable when strict time constraints are given that cannot be violated. In [5], the authors propose to shut down

monitors based on estimation of their proximity to satisfaction / violation. Instead, the monitoring module can disable less important properties, and leave critical properties alive.

Due to its ability to work in Offline mode, GooMF can be used for offline–online log monitoring. There are several interesting works in this area. For example, in [20], the authors propose to use Ltl-based specification formalisms to enforce privacy legislation rules from Health Insurance Portability and Accountability Act (HIPAA). As our verification algorithms are capable of analyzing any deterministic FSM, GooMF can be easily converted into the offline log checker. The same can be done with online sniffing of the network traffic. In fact, the rules of the network intrusion detection system SNORT [37] are all stateless properties and can be easily expressed with Ltl.

Finally, complementing GooMF with code–instrumentation front end will be a logical next step. By implementing RiTHM, we made a first step towards providing the solution that combines automatic instrumentation and verification. The front-end instrumentation provided by RiTHM is very specific and bounded to specific time-triggered monitoring solutions. Using AspectC as a front end for GooMF will offer flexible instrumentation front end without restrictions on instrumentation points and type of inspected programs.

APPENDICES

Appendix A

GooMF Online API Header

```
#ifndef GOOMFONLINEAPI_H_
#define GOOMFONLINEAPI_H_

#include <stdbool.h>
#include <stdio.h>

#define MAIN_FUNCTION_NAME "verification"


/*********************** errors declarations ***************************/
#define _GOOMF_SUCCESS                          (0)
#define _GOOMF_NO_CONTEXT_ERROR                 (-1)
#define _GOOMF_MALLOC_ERROR                     (-2)
#define _GOOMF_ARGUMENT_ERROR                   (-3)
#define _GOOMF_CL_CONTEXT_ERROR                  (-4)
#define _GOOMF_CL_DEVICE_ERROR                  (-5)
#define _GOOMF_CL_PROGRAM_ERROR  (-6)
#define _GOOMF_CL_QUEUE_ERROR    (-7)
#define _GOOMF_CL_BUILD_ERROR    (-8)
#define _GOOMF_CL_KERNEL_ERROR   (-9)
#define _GOOMF_CL_ALLOCATE_ERROR  (-10)
#define _GOOMF_CL_KERNEL_ARGUMENT_ERROR  (-11)
#define _GOOMF_CL_MEMORY_WRITE_ERROR  (-12)
#define _GOOMF_CL_KERNEL_RUN_ERROR (-13)
```

```
#define _GOOMF_CL_MEMORY_READ_ERROR (-14)
#define _GOOMF_CALLBACK_ERROR (-15)
#define _GOOMF_BUFFER_SIZE_ERROR (-16)
#define _GOOMF_THREAD_ERROR (-17)
#define _GOOMF_PARTIAL_OFFLOAD_ERROR (-18)


/************************** types declarations ************************/
//type for the fsm state
typedef enum{_GOOMF_enum_sat = 0, _GOOMF_enum_psat, _GOOMF_enum_pviol,
_GOOMF_enum_viol} _GOOMF_enum_verdict_type;

//type for the trigger that triggers the flush of the program trace
typedef enum{_GOOMF_enum_no_trigger = 0, _GOOMF_enum_buffer_trigger,
_GOOMF_enum_time_trigger} _GOOMF_enum_trigger_type;

//type for the GooMF_flush() invocation - synchronous (blocking) or
//asynchronous (non-blocking)
typedef enum{_GOOMF_enum_sync_invocation = 0, _GOOMF_enum_async_invocation}
_GOOMF_enum_invocation_type;

//enum for algorithm type
typedef enum{_GOOMF_enum_alg_seq = 0,
     _GOOMF_enum_alg_partial_offload,
     _GOOMF_enum_alg_finite,
     _GOOMF_enum_alg_infinite} _GOOMF_enum_alg_type;

//type for GOOMF monitoring context structure that keeps the current
//state of the monitor
typedef struct _GOOMF_context_struct* _GOOMF_context;

//type for the callback function to call upon the property convergence
typedef int (*report_handler_type)
(int prop_num, _GOOMF_enum_verdict_type verdict_type,
const void* program_state);


/*********************** functions declarations ***********************/
```

```
/*
 * Allocates and initializes the monitoring context structure along
 * with all other initialization stuff
 * - context - the structure to be allocated
 * - trigger_type - triggers the program trace flush
 * - device type - device the monitoring is performed on
 * - buffer_size - if trigger is _GOOMF_enum_buffer_trigger,
 * specify the maximal size of the buffer
 */
extern int _GOOMF_initContext(_GOOMF_context* context,
                             _GOOMF_enum_trigger_type trigger_type,
                             //_GOOMF_enum_device_type device_type,
                             _GOOMF_enum_alg_type alg_type,
                             _GOOMF_enum_invocation_type invocation_type,
                             int buffer_size);

/*
 * Destroys context structure allocated previously
 * - context - the structure to be destroyed
 */
extern int _GOOMF_destroyContext(_GOOMF_context* context);

/*
 * Copy next program state to the buffer and flush if needed
 * - context - GooMF monitoring context
 */
extern int _GOOMF_nextState(_GOOMF_context context,
void* program_state_ptr);

/*
 * Process event, copy next program state to the buffer and flush if needed
 * - context - GooMF monitoring context
 * - var_order - order of the variable in program state that changed
 * - var_value - new value of the variable
 */
extern int _GOOMF_nextEvent(_GOOMF_context context, unsigned int var_order,
                            void* var_value);
```

```
/*
 * Register report callback for specific property
 * - prop_num - report callback is called if this property has converged
 * - report_handler - callback function to call; should be concise as it
 *   called during the processing
 */
extern int _GOOMF_registerReport(_GOOMF_context context, int prop_num,
                                 report_handler_type report_handler);

/*
 * Unregister report callback for specific property
 * - prop_num - report callback is called if this property has converged
 * - report_handler - callback function to call
 */
extern int _GOOMF_unregisterReport(_GOOMF_context context, int prop_num);

/*
 * Enables monitoring of the specific property
 * - context - GooMF monitoring context
 * - prop_num - property to enable
 */
extern int _GOOMF_enableProperty(_GOOMF_context context, int prop_num);

/*
 * Disables monitoring of the specific property
 * - context - GooMF monitoring context
 * - prop_num - property to disable
 */
extern int _GOOMF_disableProperty(_GOOMF_context context, int prop_num);

/*
 * Resets the property to the initial monitoring state
 * - context - GooMF monitoring context
 * - prop_num - property to reset
 */
extern int _GOOMF_resetProperty(_GOOMF_context context, int prop_num);

/*
```

```
 * Force flush of the content of the buffer
 * - context - GooMF monitoring context
 */
extern int _GOOMF_flush(_GOOMF_context context);

/*
 * Get current status of all properties
 * - context - GooMF monitoring context
 * - verdict - output array of the statuses; it is the responsibility
 * of the caller to deallocate it
 */
extern int _GOOMF_getCurrentStatus(_GOOMF_context context,
                                   _GOOMF_enum_verdict_type* verdict);

/*
 * Translate verdict to string
 * - verdict - verdict
 * - str - output string containing the verdict phrase
 */
extern int _GOOMF_typeToString(_GOOMF_enum_verdict_type* verdict,
char* str);

/*
 * Check if all properties has converged (either satisfied or violated)
 * - context - GooMF monitoring context
 */
extern bool _GOOMF_allPropertiesConverged(_GOOMF_context context);

/*
 * Set the logger to log the basic events
 * - context - GooMF monitoring context
 * - logger - file descriptor to write into. Assuming the descriptior is
 *       open. It is the responsibility of the caller to open and
 *            close the stream.
 */
extern int _GOOMF_setLogger(_GOOMF_context context, FILE* logger);

/*
```

```
 * Translate status to the meaningful sentence. It is the responsibility
 * of the caller to allocate a string for the message
 * - result - status
 */
extern int _GOOMF_getErrorDescription(int result, char* message);

#endif /* GOOMFONLINEAPI_H_ */
```

Appendix B

GooMF Offline API Header

```
#ifndef GOOMFOFFLINEAPI_H_
#define GOOMFOFFLINEAPI_H_
#include "GooMFOnlineAPI.h"


/********************** function callbacks types ***********************/
/*
 * This function is called to open input data stream.
 * - Should return negative number in case of failure or non-negative in
 *   case of success
 */
typedef int (*open_handler)();

/*
 * This function is called to parse next program state from input data
 * stream. The next program state should be placed into next_program_state.
 * - Should return negative number in case of failure or non-negative
 * in case of success.
 */
typedef int (*get_next_state_handler)(void* next_program_state_ptr);

/*
 * This function is called to close the input data stream.
 * - Should return negative number in case of failure or non-negative
```

```
 * in case of success
 */
typedef int (*close_handler)();

/*
 * This function is called upon any property convergence; should be
 * concise as it called during the processing
 * - prop_num - number of converged property
 * - verdict_type - violation/satisfaction
 * - program_state - program state that caused the convergence
 * - Should return negative number in case of failure or non-negative
 * in case of success.
 */
typedef int (*report_handler_type)
   (int prop_num, _GOOMF_enum_verdict_type verdict_type,
   const void* program_state);



/*********************** functions ***********************************/
/*
 * Analyzer function - uses provided callback functions to analyze the
 * input data stream
 * - trigger_type - triggers the program trace flush
 * - device type - device the monitoring is performed on
 * - buffer_size - if trigger is _GOOMF_enum_buffer_trigger, specify the
 *   maximal size of the buffer
 */
extern int _GOOMF_analyze(open_handler oh_callback,
                          get_next_state_handler gnsh_callback,
                          close_handler ch_callback,
                          report_handler_type rh,
                          _GOOMF_enum_trigger_type trigger_type,
                          //_GOOMF_enum_device_type device_type,
                          _GOOMF_enum_alg_type alg_type,
                          _GOOMF_enum_invocation_type invocation_type,
                          FILE* logger,
                          unsigned int buffer_size);
```

```
#endif /* GOOMFOFFLINEAPI_H_ */
```

Appendix C

Configuration File Example

```
//A configuration file that describes monitoring objects

//name of the process under scrutiny
program_name = "ParametricUAV";

//user functions
functions = (
"const float XDELTA = 1.0;",
"const float YDELTA = 1.0;",
"const float ZDELTA = 1.0;",
"const long XTSAT = 100; //settling time for x position",
"const long YTSAT = 100; //settling time for y position",
"const long ZTSAT = 100; //settling time for z position",
"bool IMUSanityCheckPhi(float phi1, float teta1, float phi2, float p1,
float q1, float r1, float delta)
{
  float temp = ((phi2 - phi1) / 0.01) -
  (p1 + q1*sin(phi1)*tan(teta1) + r1*cos(phi1)*tan(teta1));
  return (temp < delta && temp > -delta);
}",
"bool IMUSanityCheckTeta(float phi1, float teta1, float teta2,
float q1, float r1, float delta)
{
  float temp = ((teta2 - teta1) / 0.01) -
```

```
  (q1*cos(phi1) - r1*sin(phi1));
  return (temp < delta && temp > -delta);
}",
"bool IMUSanityCheckKsi(float phi1, float teta1, float ksi1,
float ksi2, float q1, float r1, float delta)
{
  float temp = ((ksi1 - ksi1) / 0.01) -
  (q1*sin(phi1)/cos(teta1) + r1*cos(phi1)/cos(teta1));
  return (temp < delta && temp > -delta);
}",
"/*
 * Converts LLH to XYZ in the ECEF frame using WGS84 convention
 * Parameters: float lat, float lon, float alt is the LLH coordinates,
 * float* X, float* Y, float* Z is the memory where the
 * solution will be stored
 */
void Convert_WGS84_To_ECEF(float lat, float lon, float alt, float* X,
float* Y, float* Z)
{
  //Need to convert lat and lon to radians first
  lat = (lat/180)*3.14159265;
  lon = (lon/180)*3.14159265;

  //WGS84 parameters
  float a = 6378137; //semi-major axis of earth
  //float b = 6356752.314245; //semi-minor axis of earth
  //float e2 = ((a*a)-(b*b))/(a*a); //first eccentricity squared
  float e2 = 0.00669437999014;

  //Geodetic to ECEF coordinates
  //normal: distance from surface to the Z axis along the ellipsoid normal
  float N = a/(sqrt(1-(e2*pow(sin(lat),2))));

  *X = (N+alt)*cos(lat)*cos(lon);
  *Y = (N+alt)*cos(lat)*sin(lon);
  *Z = (N*(1-e2)+alt)*sin(lat);
}",
"/*
```

```
 * Converts ECEF to ENU
 * Parameters: float dx, float dy, float dz are ECEF displacements between
 * desired position and ENU origin
 * float* x, float* y, float* z is the memory where the ENU solution will
 * be stored
 */
void Convert_ECEF_To_ENU(float lat_origin, float lon_origin, float dx,
float dy, float dz, float* x, float* y, float* z)
{
  //first convert lat and lon to radians
  float lat = (lat_origin/180)*3.14159265;
  float lon = (lon_origin/180)*3.14159265;

  *x = (-1*sin(lon)*dx)+(cos(lon)*dy);
  *y = (-1*sin(lat)*cos(lon)*dx)+(-1*sin(lat)*sin(lon)*dy)+(cos(lat)*dz);
  *z = (cos(lat)*cos(lon)*dx)+(cos(lat)*sin(lon)*dy)+(sin(lat)*dz);
}",
"/*
 * Converts LLH in WGS84 standard to ENU
 * Parameters: float lat, float lon, float alt is LLH of the desired
 * position, float* x, float* y, float* z is the memory where the ENU
 * solution will be stored
 */
bool CompareToReferences(float lat, float lon, float alt,
float lat_origin, float lon_origin, float alt_origin,
float x_ref, float y_ref, float z_ref,
float delta)
{
  //convert desired position and ENU origin from LLH to ECEF
  float X, Y, Z;
  float X_origin, Y_origin, Z_origin;
  Convert_WGS84_To_ECEF(lat, lon, alt, &X, &Y, &Z);
  Convert_WGS84_To_ECEF(lat_origin, lon_origin, alt_origin, &X_origin,
&Y_origin, &Z_origin);

  //convert from ECEF to ENU
  float x, y, z;
  Convert_ECEF_To_ENU(lat_origin, lon_origin, X-X_origin, Y-Y_origin,
```

```
Z-Z_origin, &x, &y, &z);

  return ((x - x_ref) < delta && (x - x_ref) > -delta &&
  (y - y_ref) < delta && (y - y_ref) > -delta &&
  (z - z_ref) < delta && (z - z_ref) > -delta);
}",
"float GetX(float lat, float lon, float alt)
{
  //convert desired position and ENU origin from LLH to ECEF
  float X, Y, Z;
  float X_origin, Y_origin, Z_origin;
  Convert_WGS84_To_ECEF(lat, lon, alt, &X, &Y, &Z);
  Convert_WGS84_To_ECEF(44.295193, -76.941707, 116.1366, &X_origin,
&Y_origin, &Z_origin);

  //convert from ECEF to ENU
  float x, y, z;
  Convert_ECEF_To_ENU(44.295193, -76.941707, X-X_origin, Y-Y_origin,
Z-Z_origin, &x, &y, &z);
  return x;
}",
"float GetY(float lat, float lon, float alt)
{
  //convert desired position and ENU origin from LLH to ECEF
  float X, Y, Z;
  float X_origin, Y_origin, Z_origin;
  Convert_WGS84_To_ECEF(lat, lon, alt, &X, &Y, &Z);
  Convert_WGS84_To_ECEF(44.295193, -76.941707, 116.1366, &X_origin,
&Y_origin, &Z_origin);

  //convert from ECEF to ENU
  float x, y, z;
  Convert_ECEF_To_ENU(44.295193, -76.941707, X-X_origin, Y-Y_origin,
Z-Z_origin, &x, &y, &z);
  return y;
}",
"float GetZ(float lat, float lon, float alt)
{
```

```
  //convert desired position and ENU origin from LLH to ECEF
  float X, Y, Z;
  float X_origin, Y_origin, Z_origin;
  Convert_WGS84_To_ECEF(lat, lon, alt, &X, &Y, &Z);
  Convert_WGS84_To_ECEF(44.295193, -76.941707, 116.1366, &X_origin,
&Y_origin, &Z_origin);

  //convert from ECEF to ENU
  float x, y, z;
  Convert_ECEF_To_ENU(44.295193, -76.941707, X-X_origin, Y-Y_origin,
      Z-Z_origin, &x, &y, &z);
  return z;
}");


//LTL properties specified in Future-LTL syntax
//this property is really [] (a -> X (b U c)) in LTL
properties = ( {name = "xposition"; formalism = "FSM"; syntax =
"digraph G {
\"(0, 0)\" -> \"(1, 1)\" [label = \"(a&&b&&c)\", action =
\"used_params->xtimestamp = gps_second; used_params->x_d =
GetX(llh_lat_dest, llh_lon_dest, llh_alt_dest);
used_params->xtsat = XTSAT\"];
\"(0, 0)\" -> \"(1, 1)\" [label = \"(a&&b)\", action =
\"used_params->xtimestamp = gps_second; used_params->x_d =
GetX(llh_lat_dest, llh_lon_dest, llh_alt_dest);
used_params->xtsat = XTSAT\"];
\"(0, 0)\" -> \"(1, 1)\" [label = \"(a&&c)\", action =
\"used_params->xtimestamp = gps_second; used_params->x_d =
GetX(llh_lat_dest, llh_lon_dest, llh_alt_dest);
used_params->xtsat = XTSAT\"];
\"(0, 0)\" -> \"(1, 1)\" [label = \"(a)\", action =
\"used_params->xtimestamp = gps_second; used_params->x_d =
GetX(llh_lat_dest, llh_lon_dest, llh_alt_dest);
used_params->xtsat = XTSAT\"];
\"(0, 0)\" -> \"(0, 0)\" [label = \"(b&&c)\"];
\"(0, 0)\" -> \"(0, 0)\" [label = \"(b)\"];
\"(0, 0)\" -> \"(0, 0)\" [label = \"(c)\"];
```

```
\"(0, 0)\" -> \"(0, 0)\" [label = \"(<empty>)\"];
\"(1, 1)\" -> \"(1, 1)\" [label = \"(a&&b&&c)\"];
\"(1, 1)\" -> \"(1, 1)\" [label = \"(a&&b)\"];
\"(1, 1)\" -> \"(1, 1)\" [label = \"(a&&c)\"];
\"(1, 1)\" -> \"(-1, 2)\" [label = \"(a)\"];
\"(1, 1)\" -> \"(0, 0)\" [label = \"(b&&c)\"];
\"(1, 1)\" -> \"(1, 1)\" [label = \"(b)\"];
\"(1, 1)\" -> \"(0, 0)\" [label = \"(c)\"];
\"(1, 1)\" -> \"(-1, 2)\" [label = \"(<empty>)\"];
\"(-1, 2)\" -> \"(-1, 2)\" [label = \"(a&&b&&c)\"];
\"(-1, 2)\" -> \"(-1, 2)\" [label = \"(a&&b)\"];
\"(-1, 2)\" -> \"(-1, 2)\" [label = \"(a&&c)\"];
\"(-1, 2)\" -> \"(-1, 2)\" [label = \"(a)\"];
\"(-1, 2)\" -> \"(-1, 2)\" [label = \"(b&&c)\"];
\"(-1, 2)\" -> \"(-1, 2)\" [label = \"(b)\"];
\"(-1, 2)\" -> \"(-1, 2)\" [label = \"(c)\"];
\"(-1, 2)\" -> \"(-1, 2)\" [label = \"(<empty>)\"];
\"(-1, 2)\" [label=\"(-1, 2)\", style=filled, color=red]
\"(1, 1)\" [label=\"(1, 1)\", style=filled, color=darkseagreen1]
\"(0, 0)\" [label=\"(0, 0)\", style=filled, color=darkseagreen1]
}"
},
{name = "yposition"; formalism = "FSM"; syntax =
"digraph G {
\"(0, 0)\" -> \"(1, 1)\" [label = \"(d&&e&&f)\",
action = \"used_params->ytimestamp = gps_second; used_params->y_d =
GetY(llh_lat_dest, llh_lon_dest, llh_alt_dest);
used_params->ytsat = YTSAT\"];
\"(0, 0)\" -> \"(1, 1)\" [label = \"(d&&e)\",
action = \"used_params->ytimestamp = gps_second; used_params->y_d =
GetY(llh_lat_dest, llh_lon_dest, llh_alt_dest);
used_params->ytsat = YTSAT\"];
\"(0, 0)\" -> \"(1, 1)\" [label = \"(d&&f)\",
action = \"used_params->ytimestamp = gps_second; used_params->y_d =
GetY(llh_lat_dest, llh_lon_dest, llh_alt_dest);
used_params->ytsat = YTSAT\"];
\"(0, 0)\" -> \"(1, 1)\" [label = \"(d)\",
action = \"used_params->ytimestamp = gps_second; used_params->y_d =
```

```
GetY(llh_lat_dest, llh_lon_dest, llh_alt_dest);
used_params->ytsat = YTSAT\"];
\"(0, 0)\" -> \"(0, 0)\" [label = \"(e&&f)\"];
\"(0, 0)\" -> \"(0, 0)\" [label = \"(e)\"];
\"(0, 0)\" -> \"(0, 0)\" [label = \"(f)\"];
\"(0, 0)\" -> \"(0, 0)\" [label = \"(<empty>)\"];
\"(1, 1)\" -> \"(1, 1)\" [label = \"(d&&e&&f)\"];
\"(1, 1)\" -> \"(1, 1)\" [label = \"(d&&e)\"];
\"(1, 1)\" -> \"(1, 1)\" [label = \"(d&&f)\"];
\"(1, 1)\" -> \"(-1, 2)\" [label = \"(d)\"];
\"(1, 1)\" -> \"(0, 0)\" [label = \"(e&&f)\"];
\"(1, 1)\" -> \"(1, 1)\" [label = \"(e)\"];
\"(1, 1)\" -> \"(0, 0)\" [label = \"(f)\"];
\"(1, 1)\" -> \"(-1, 2)\" [label = \"(<empty>)\"];
\"(-1, 2)\" -> \"(-1, 2)\" [label = \"(d&&e&&f)\"];
\"(-1, 2)\" -> \"(-1, 2)\" [label = \"(d&&e)\"];
\"(-1, 2)\" -> \"(-1, 2)\" [label = \"(d&&f)\"];
\"(-1, 2)\" -> \"(-1, 2)\" [label = \"(d)\"];
\"(-1, 2)\" -> \"(-1, 2)\" [label = \"(e&&f)\"];
\"(-1, 2)\" -> \"(-1, 2)\" [label = \"(e)\"];
\"(-1, 2)\" -> \"(-1, 2)\" [label = \"(f)\"];
\"(-1, 2)\" -> \"(-1, 2)\" [label = \"(<empty>)\"];
\"(-1, 2)\" [label=\"(-1, 2)\", style=filled, color=red]
\"(1, 1)\" [label=\"(1, 1)\", style=filled, color=darkseagreen1]
\"(0, 0)\" [label=\"(0, 0)\", style=filled, color=darkseagreen1]
}"
},
{name = "zposition"; formalism = "FSM"; syntax =
"digraph G {
\"(0, 0)\" -> \"(1, 1)\" [label = \"(g&&h&&k)\",
action = \"used_params->ztimestamp = gps_second; used_params->z_d =
GetZ(llh_lat_dest, llh_lon_dest, llh_alt_dest);
used_params->ztsat = ZTSAT\"];
\"(0, 0)\" -> \"(1, 1)\" [label = \"(g&&h)\",
action = \"used_params->ztimestamp = gps_second; used_params->z_d =
GetZ(llh_lat_dest, llh_lon_dest, llh_alt_dest);
used_params->ztsat = ZTSAT\"];
\"(0, 0)\" -> \"(1, 1)\" [label = \"(g&&k)\",
```

```
action = \"used_params->ztimestamp = gps_second; used_params->z_d =
GetZ(llh_lat_dest, llh_lon_dest, llh_alt_dest);
used_params->ztsat = ZTSAT\"];
\"(0, 0)\" -> \"(1, 1)\" [label = \"(g)\",
action = \"used_params->ztimestamp = gps_second; used_params->z_d =
GetZ(llh_lat_dest, llh_lon_dest, llh_alt_dest);
used_params->ztsat = ZTSAT\"];
\"(0, 0)\" -> \"(0, 0)\" [label = \"(h&&k)\"];
\"(0, 0)\" -> \"(0, 0)\" [label = \"(h)\"];
\"(0, 0)\" -> \"(0, 0)\" [label = \"(k)\"];
\"(0, 0)\" -> \"(0, 0)\" [label = \"(<empty>)\"];
\"(1, 1)\" -> \"(1, 1)\" [label = \"(g&&h&&k)\"];
\"(1, 1)\" -> \"(1, 1)\" [label = \"(g&&h)\"];
\"(1, 1)\" -> \"(1, 1)\" [label = \"(g&&k)\"];
\"(1, 1)\" -> \"(-1, 2)\" [label = \"(g)\"];
\"(1, 1)\" -> \"(0, 0)\" [label = \"(h&&k)\"];
\"(1, 1)\" -> \"(1, 1)\" [label = \"(h)\"];
\"(1, 1)\" -> \"(0, 0)\" [label = \"(k)\"];
\"(1, 1)\" -> \"(-1, 2)\" [label = \"(<empty>)\"];
\"(-1, 2)\" -> \"(-1, 2)\" [label = \"(g&&h&&k)\"];
\"(-1, 2)\" -> \"(-1, 2)\" [label = \"(g&&h)\"];
\"(-1, 2)\" -> \"(-1, 2)\" [label = \"(g&&k)\"];
\"(-1, 2)\" -> \"(-1, 2)\" [label = \"(g)\"];
\"(-1, 2)\" -> \"(-1, 2)\" [label = \"(h&&k)\"];
\"(-1, 2)\" -> \"(-1, 2)\" [label = \"(h)\"];
\"(-1, 2)\" -> \"(-1, 2)\" [label = \"(k)\"];
\"(-1, 2)\" -> \"(-1, 2)\" [label = \"(<empty>)\"];
\"(-1, 2)\" [label=\"(-1, 2)\", style=filled, color=red]
\"(1, 1)\" [label=\"(1, 1)\", style=filled, color=darkseagreen1]
\"(0, 0)\" [label=\"(0, 0)\", style=filled, color=darkseagreen1]
}"
});

//all the parameters being involved in the monitoring and their types
parameters = ( {name = "xtimestamp"; type = "float"},
  {name = "x_d"; type = "float"},
  {name = "xtsat"; type = "float"},
  {name = "ytimestamp"; type = "float"},
```

```
  {name = "y_d"; type = "float"},
  {name = "ytsat"; type = "float"},
  {name = "ztimestamp"; type = "float"},
  {name = "z_d"; type = "float"},
  {name = "ztsat"; type = "float"});

//mapping of the predicates on the program variables
predicates = ( {name = "a"; syntax = "llh_lat_dest_prev != llh_lat_dest"},
  {name = "b"; syntax =
  "gps_second < used_params->xtimestamp + used_params->xtsat"},
  {name = "c"; syntax =
  "(GetX(llh_lat, llh_lon, llh_alt) - used_params->x_d) < XDELTA";
  device = "GPU"},
  {name = "d"; syntax =
  "llh_lon_dest_prev != llh_lon_dest"},
  {name = "e"; syntax =
  "gps_second < used_params->ytimestamp + used_params->ytsat"},
  {name = "f"; syntax =
  "(GetY(llh_lat, llh_lon, llh_alt) - used_params->y_d) < YDELTA";
  device = "GPU"},
  {name = "g"; syntax =
  "llh_alt_dest_prev != llh_alt_dest"},
  {name = "h"; syntax =
  "gps_second < used_params->ztimestamp + used_params->ztsat"},
  {name = "k"; syntax =
  "(GetZ(llh_lat, llh_lon, llh_alt) - used_params->z_d) < ZDELTA";
  device = "GPU"});

//all program variables being involved in the monitoring and their types
program_variables = ( {name = "x"; type = "float"},
  {name = "y"; type = "float"},
  {name = "llh_lat"; type = "float"},
  {name = "llh_lon"; type = "float"},
  {name = "llh_alt"; type = "float"},
  {name = "gps_second"; type = "float"},
  {name = "llh_lat_dest"; type = "float"},
  {name = "llh_lon_dest"; type = "float"},
  {name = "llh_alt_dest"; type = "float"},
```

```
{name = "llh_lat_dest_prev"; type = "float"},
{name = "llh_lon_dest_prev"; type = "float"},
{name = "llh_alt_dest_prev"; type = "float"});
```

References

[1] P. Avgustinov, J. Tibble, and O. de Moor. Making trace monitors feasible. In *ACM SIGPLAN Notices*, volume 42, pages 589–608. ACM, 2007.

[2] M. Barnett, K. Leino, and W. Schulte. The spec# programming system: An overview. *Construction and analysis of safe, secure, and interoperable smart devices*, pages 49–69, 2005.

[3] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Program monitoring with ltl in eagle. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, page 264. IEEE, 2004.

[4] H. Barringer, D. Rydeheard, and K. Havelund. Rule systems for run-time monitoring: from eagle to ruler. *Journal of Logic and Computation*, 20(3):675–706, 2010.

[5] E. Bartocci, R. Grosu, A. Karmarkar, S.A. Smolka, S.D. Stoller, E. Zadok, and J. Seyster. Adaptive runtime verification. 2012.

[6] A. Bauer, M. Leucker, and C. Schallhart. Runtime Verification for LTL and TLTL. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 20(4):14:1–14:64, 2011.

[7] E. Bodden. J-lo-a tool for runtime-checking temporal assertions. *Master's thesis, RWTH Aachen university*, 2005.

[8] E. Bodden. Efficient hybrid typestate analysis by determining continuation-equivalent states. In *International Conference on Software Engineering (ICSE)*, pages 5–14, 2010.

[9] E. Bodden, P. Lam, and L. Laurie. Clara: A framework for partially evaluating finite-state runtime monitors ahead of time. In *Runtime Verification (RV)*, pages 183–197. 2010.

[10] B. Bonakdarpour, S. Navabpour, and S. Fischmeister. Sampling-based runtime verification. In *Formal Methods (FM)*, pages 88–102, 2011.

[11] B. Bonakdarpour, Johnson J. Thomas, and S. Fischmeister. Time-triggered program self-monitoring. In *IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 260–269, 2012.

[12] F. Chen and G. Roşu. Java-MOP: A monitoring oriented programming environment for java. In *Tools and Algorithms for the construction and analysis of systems (TACAS)*, pages 546–550, 2005.

[13] F. Chen and G. Roşu. Towards monitoring-oriented programming: A paradigm combining specification and implementation. *Electronic Notes in Theoretical Computer Science*, 89(2):108–127, 2003.

[14] F. Chen and G. Roşu. Parametric trace slicing and monitoring. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 246–261, 2009.

[15] S. Colin and L. Mariani. *Run-Time Verification*, chapter 18. Springer-Verlag LNCS 3472, 2005.

[16] M. d'Amorim and K. Havelund. Event-based runtime verification of java programs. *ACM SIGSOFT Software Engineering Notes*, 30(4):1–7, 2005.

[17] D. Drusinsky. The temporal rover and the atg rover. *SPIN Model Checking and Software Verification*, pages 323–330, 2000.

[18] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *International Conference on Software Engineering (ICSE)*, pages 411 –420, 1999.

[19] T. Elmas, S. Okur, and S. Tasiran. Rethinking runtime verification on hundreds of cores: Challenges and opportunities. Technical Report UCB/EECS-2011-74, EECS Department, University of California, Berkeley, June 2011.

[20] Deepak Garg, Limin Jia, and Anupam Datta. Policy auditing over incomplete logs: theory, implementation and applications. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, pages 151–162, New York, NY, USA, 2011. ACM.

[21] D. Giannakopoulou and K. Havelund. Automata-Based Verification of Temporal Properties on Running Programs. In *Automated Software Engineering (ASE)*, pages 412–416, 2001.

[22] Khronos OpenCL Working Group. The opencl specification, 2011. Version 1.1.

[23] J. Ha, M. Arnold, S. M. Blackburn, and K. S. McKinley. A concurrent dynamic analysis framework for multicore hardware. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 155–174, 2009.

[24] J. Holub and S. Stekr. On parallel implementations of deterministic finite automata. In *Implementation and Application of Automata (CIAA)*, pages 54–64, 2009.

[25] X. Huang, J. Seyster, S. Callanan, K. Dixit, R. Grosu, S. A. Smolka, S. D. Stoller, and E. Zadok. Software monitoring with controllable overhead. *Software tools for technology transfer (STTT)*, 14(3):327–347, 2012.

[26] D. Jin. *Making runtime monitoring of parametric properties practical*. PhD thesis, University of Illinois, 2012.

[27] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of aspectj. *ECOOP 2001???Object-Oriented Programming*, pages 327–354, 2001.

[28] M. Kim, M. Viswanathan, H. Ben-Abdallah, S. Kannan, I. Lee, and O. Sokolsky. Formally specified monitoring of temporal properties. In *Real-Time Systems, 1999. Proceedings of the 11th Euromicro Conference on*, pages 114–122. IEEE, 1999.

[29] O. Kupferman and M. Y. Vardi. Model Checking of Safety Properties. In *Computer Aided Verification (CAV)*, pages 172–183, 1999.

[30] C Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *International Symposium on Code Generation and Optimization: Feedback Directed and Runtime Optimization*, page 75, 2004.

[31] H. Lu and A. Forin. The design and implementation of p2v, an architecture for zero-overhead online verification of software programs. Technical report, Technical Report MSR-TR-2007–99, Microsoft Research, 2007.

[32] P.k Meredith, D. Jin, F. Chen, and G. Roşu. Efficient monitoring of parametric context-free patterns. *Journal of Automated Software Engineering*, 17(2):149–180, June 2010.

[33] S. Navabpour, C. W. Wu, B. Bonakdarpour, and S. Fischmeister. Efficient techniques for near-optimal instrumentation in time-triggered runtime verification. In *International Conference on Runtime Verification (RV)*, 2011. To appear.

[34] R. Pellizzoni, P. Meredith, M. Caccamo, and G. Rosu. Hardware runtime monitoring for dependable cots-based real-time embedded systems. In *Real-Time Systems Symposium, 2008*, pages 481 –491, 30 2008-dec. 3 2008.

[35] SJ Pennycook, SD Hammond, SA Wright, JA Herdman, I. Miller, and SA Jarvis. An investigation of the performance portability of opencl. *Journal of Parallel and Distributed Computing*, 2012.

[36] A. Pnueli and A. Zaks. PSL Model Checking and Run-Time Verification via Testers. In *Symposium on Formal Methods (FM)*, pages 573–586, 2006.

[37] M. Roesch et al. Snort-lightweight intrusion detection for networks. In *Proceedings of the 13th USENIX conference on System administration*, pages 229–238. Seattle, Washington, 1999.

[38] J. Seyster, K. Dixit, X. Huang, R. Grosu, K. Havelund, S. A. Smolka, S. D. Stoller, and E. Zadok. Aspect-oriented instrumentation with GCC. In *Runtime Verification (RV)*, pages 405–420, 2010.

[39] K.L. Spafford, J.S. Meredith, S. Lee, D. Li, P.C. Roth, and J.S. Vetter. The tradeoffs of fused memory hierarchies in heterogeneous computing architectures. In *Proceedings of the 9th conference on Computing Frontiers*, pages 103–112. ACM, 2012.

[40] J.E. Stone, D. Gohara, and G. Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(3):66, 2010.

[41] J. J. Thomas, S. Fischmeister, and D. Kumar. Lowering overhead in sampling-based execution monitoring and tracing. In *Languages, compilers, and tools for embedded systems (LCTES)*, pages 101–110, 2011.

[42] Chun Wah Wallace Wu. Methods for reducing monitoring overhead in runtime verification. M.a.sc. thesis, University of Waterloo, 2012.

[43] H. Zhu, M. B. Dwyer, and S. Goddard. Predictable runtime monitoring. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 173–183, 2009.

[44] Craig B. Zilles and Gurindar S. Sohi. A programmable co-processor for profiling. In *International Symposium on High Performance Computer Architecture (HPCA)*, pages 241–253, 2001.