Saarland University

Faculty of Mathematics and Computer Science

Bachelor'sThesis

Compiling Lola 2.0 to C

Author
Christoph Rosenhauer

Supervisor
Prof. Bernd Finkbeiner, Ph. D.

Advisor
B.Sc. Maximilian Schwenger

Reviewers
Prof. Bernd Finkbeiner, Ph. D.
Dr. Swen Jacobs

Submitted: 2nd February 2019

Erklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken,_____ _____

(Datum/Date) (Unterschrift/Signature)

Abstract

Runtime monitoring is a lightweight verification approach based on the extraction of information from a running system. This information is used to detect violations of a given specification to ensure the correctness of the system at runtime.
Lola 2.0 is a simple and expressive specification language and stream processing engine for monitoring temporal properties and computing complex statistical measurements. It supports dynamic stream generation as well as parameterization with data. This enables monitoring of new subtasks on their own timescale.
Lola 2.0 is currently applied to unmanned aircraft systems at DLR. For this purpose, specifications have so far only been evaluated by an interpreter. This thesis will change that, since a compiled C code monitor strongly improves the maintainability of a specification. Compared to the general interpreter code, the generated C code greatly eases the process of understanding and verifying the monitor. This enables the opportunity of easily modifying the C code monitor itself, whereby a higher flexibility in testing and also under actual deployment is granted. Furthermore, the compilation could speed up the monitoring process in general by eliminating the computational overhead that an interpreter brings along.
In this thesis we present an approach to translate *Lola 2.0* to C as well as the implementation of a compiler.

Contents

Chapter 1

Introduction

In recent years, reactive systems and especially autonomous systems obtain more and more acceptance in everyday life. Such systems have to make secure and reliable decisions in safety-critical environments, which leads to the necessity of verification.
Runtime Verification is a lightweight formal method to ensure the correctness of a system at runtime. Given a specification of desired properties, a monitor processes the execution trace of the system to detect possible violations. This technique is even applicable when other verification approaches like Model Checking fail due to state space explosion. Furthermore we are able to incrementally introduce Runtime Verification into the system during the development process.
Specifications may be expressed in formal languages, which enables formal reasoning and the automated generation of monitors. Their descriptive nature makes them easy to maintain compared to handwritten monitoring code.
Lola 2.0 is a stream-based specification language for runtime monitoring of synchronous systems [5]. Stream-based means that it continuously produces output streams from a given set of input streams. *Lola 2.0* can not only describe correctness assertions and statistical measures but also supports parameterization and dynamic stream generation. Despite its expressiveness, it is easy to understand and closes the gap between temporal logics and hand-written monitor code.
The applicability of *Lola 2.0* in practice was elaborated in several case studies at German Aerospace Center (DLR) [2][15]. *Lola 2.0* was capable to express required properties for the online and offline monitoring of Unmanned Aircraft Systems (UAS). Offline monitoring describes the analysis of log files, containing the output traces of a system, whereas online monitoring denotes the analysis of a system's output streams during runtime. The specifications were evaluated by an interpreter and the results show that *Lola 2.0* is efficient enough to be used in practice.
An interpreter always brings along some computational overhead which limits its computation speed. To make *Lola 2.0* even more interesting for industrial use, this

thesis will present a compiler that generates C code from a given *Lola 2.0* specification. Thereby we achieve an eased understanding and verifying process as main advantage compared to the general interpreter code. Due to the improved code readability the significantly increased maintainability of the whole monitoring process is attained, which promotes the industrial use of *Lola 2.0*.
Furthermore, the compilation eliminates the computational overhead that an interpreter brings along. The ability of speeding up the whole monitoring process will be examined by a computation speed comparison between interpreted and compiled versions of some example specifications.
Now we will take a more detailed look in how the compiler works.
The compilation starts with the abstract syntax tree (AST) of a specification. Out of that, we will generate a Dependency Graph (DG) as an intermediate representation. The analysis of the DG reveals information about what kind of specification is given. *Lola 2.0* enables accessing past and future values of streams by giving an offset. A negative offset takes past values, whereas a positive offset references a future value. Since most applications do not need positive offsets, we solely deal with the zero and negative ones, whose evaluation is much simpler. After constructing the DG, we describe our compilation algorithm and give examples of the generated C code. Finally, we conduct some experiments with example specifications to determine the performance of our generated code compared to an interpreted *Lola 2.0* specification.

1.1 Related Work

Lola 2.0 is a stream-based specification language that was mainly developed for Runtime Verification of synchronous systems [3, 5].
A very common approach to specify correctness properties are temporal logics like *LTL* [13]. They are well known and easy to maintain compared to hand-written monitor code. Furthermore it is possible to automatically generate automata out of a given *LTL* formula [6, 7, 4]. In a further step the automaton can be automatically translated to a C code monitor [12]. *LTL* specifications can be expressed by a boolean subset of *Lola 2.0*. But *Lola 2.0* is also able to take statistical measures of the system, which *LTL* itself can not do.
BeepBeep 3 [8] is an event stream processing tool with similar expressiveness to *Lola 2.0*. It can also be used for online runtime monitoring as well as the offline analysis of logs. One difference is the ability of *Lola 2.0* to reference not only past but also future values of a stream. Unfortunately *BeepBeep 3* is implemented as *Java* library and therefore not ideally applicative for runtime monitoring embedded systems, which are not necessarily able to run *Java*.
TeSSLa [11] was developed as specification language for runtime verification on multi-core systems. It can be seen as an extension, based on asynchronous streams,

of *LOLA*, the predecessor of *Lola 2.0*. So far there exists only a *TeSSLa* compiler for FPGA synthesis. Due to an intermediate representation, changing the specification requires no complete re-synthesizing of the FPGA, in contrast to other hardware-based techniques. A big difference to *Lola 2.0* despite the asynchrony, is that cyclic dependencies into the past are limited and future references are not allowed.
There exists also an approach to translate the predecessor of *Lola 2.0* to VHDL [9]. By using an FPGA we benefit from the processing speed on the one hand, but we are heavily restricted in expressiveness compared to *Lola 2.0* on the other hand. For the translation to VHDL, cyclic positive offsets are prohibited and the most important features of *Lola 2.0* namely parameterization and dynamic stream generation get lost.
Another idea is the transformation of a *Lola 2.0* specification to Quantitative Regular Expressions (QREs) [16]. Such a translation is feasible in several cases, but often very expensive. Furthermore *Lola 2.0* and QREs differ in their expressiveness.

1.2 Outline

In Chapter 2, we propose the specification language *Lola 1.0* by formally defining its syntax and semantics. After that, we present the syntactical extension of *Lola 1.0*, which leads to *Lola 2.0*, and define corresponding semantics according to Faymonville et al. [5]. We additionally specify the dependency graph for *Lola 2.0* specifications and its usage to define well-formed specifications.
Chapter 3 deals with the compilation approach for *Lola 2.0*. We first define the abstract syntax tree (AST) for *Lola 2.0* specifications and specify admissible types. After that, we present the structure of the implemented dependency graph (DG) and how it is built from the AST. By means of the DG, we impose some restrictions on *Lola 2.0* specifications and name limitations of our implemented compiler. Subsequently, we define our algorithm for generating an evaluation order from a given DG. We complete Chapter 3 by giving code examples to explain the composition of the generated `main.c` code file, and the auxiliary data structures with associated functions.
In Chapter 4, we discuss the experimental results and compare the performance of the generated code to the interpreted version. Chapter 5 points out possible extensions of the compiler and identifies potential improvements of the generated code. We conclude our thesis in Chapter 6 by briefly summing up our findings.

Chapter 2

Lola

Lola is a stream-based specification language for online and offline monitoring of synchronous systems. It is not only suitable for monitoring temporal properties but also for computing complex statistical measures. As we will see in the following sections, *Lola* is a simple and expressive specification language which allows for past and future references.
First we introduce the original *Lola* which we call now *Lola 1.0*. Afterwards we define the extensions that lead us to *Lola 2.0*.

2.1 Lola 1.0

A *Lola 1.0* specification defines typed **input** and **output** streams. Input streams $t_1 \dots t_m$ with corresponding types $T_1 \dots T_m$ are called *independent variables*. Output streams $s_1 \dots s_n$ with corresponding types $T_{m+1} \dots T_{m+n}$ are called *dependent variables*.

2.1.1 Lola 1.0 Syntax

Definition 2.1. (*Lola 1.0* specification) [3]
A *Lola 1.0* specification is a set of equations over *dependent* and *independent stream variables*, of the form

$$
\begin{aligned}
&\textbf{input} && T_1\ t_1 \\
&&& \vdots \\
&\textbf{input} && T_m\ t_m \\
&\textbf{output} && T_{m+1}\ s_1 \ := \ e_1(t_1, \dots, t_m, s_1, \dots, s_n) \\
&&& \vdots \\
&\textbf{output} && T_{m+n}\ s_n \ := \ e_n(t_1, \dots, t_m, s_1, \dots, s_n)
\end{aligned}
$$

where $e_1, \ldots, e_n$ are *stream expressions* over $t_1, \ldots, t_m$ and $s_1, \ldots, s_n$.
A *Lola 1.0* specification can also declare one or more *triggers* over boolean expressions. Triggers are specified as

$$\textbf{trigger} \ \varphi$$

where φ is a boolean expression over stream variables.

At any instant of time, a trigger expression evaluates to $true$, the trigger generates a notification. That is the way how we are able to take notice of violations of given system properties.

Definition 2.2. (*Lola 1.0* stream expressions) [3]
A *Lola 1.0 stream expression* is constructed by the following rules:

- if c is a constant value of type T,
  then c is a stream expression of type T
- if s is a (*dependent* or *independent*) stream variable of type T,
  then s is a stream expression of type T
- let $f : T_1 \times T_2 \times \cdots \times T_k \longmapsto T$ be a k -ary operator.
  If for $1 \leq i \leq k$, e_i is an expression of type T_i, then $f(e_1, \ldots, e_k)$ is a stream expression of type T
- if b is a boolean stream expression and e_1, e_2 are stream expressions of type T,
  then $\textbf{ite}(b, e_1, e_2)$ is a stream expression of type T
- if e is a stream expression of type T, d is a constant of type T,
  and i is an integer,
  then $e[i, d]$ is a stream expression of type T.

In the above definition, the first two kinds of expressions are called *atomic* expressions. The $\textbf{ite}(b, e_1, e_2)$ expression denotes the common *if-expression*. The expression $e[i, d]$ of the last construction rule is called *offset expression* and represents the value of expression e at the position with offset i dependent on the current time stamp. If expression e is not defined at that position, since it is past the end or before the beginning of the stream, the constant d is taken as *default* value.

2.1.2 Lola 1.0 Semantics

To characterize the semantics of *Lola 1.0* we formally define the relation between input streams and output streams. To accomplish this, we inductively define an evaluation function val that leads us to the concept of *evaluation models*.

Definition 2.3. (*Lola 1.0* Evaluation Models) [3]
Let P be a *Lola 1.0* specification over independent stream variables $t_1, \ldots, t_m$ and dependent stream variables $s_1, \ldots, s_n$. Further, let $\tau_1, \ldots, \tau_m$ be the input streams of length N with corresponding types.
The tuple $\langle \sigma_1, \ldots, \sigma_n \rangle$ of output streams with length N is called *evaluation model* for specification P, if for every equation

$$\textbf{output } T_i\ s_i\ :=\ e_i(t_1, \ldots, t_m, s_1, \ldots, s_n)$$

of P, the associated *evaluation equation*

$$\sigma_i(j) = val(e_i)(j) \qquad \text{for } 0 \leq j < N$$

is satisfied and the type of σ_i complies T_i.

Given a stream expression e and a position j, the evaluation function val returns the value of e at that position. val is inductively defined as follows:
Base cases:

- $val(c)(j) \ = \ c$
- $val(t_i)(j) \ = \ \tau_i(j)$
- $val(s_i)(j) \ = \ \sigma_i(j)$

Inductive cases:

- $val(f(e_1, \ldots, e_k))(j) \ = \ f(val(e_1)(j), \ldots, val(e_k)(j))$
- $val(\textbf{ite}(b, e_1, e_2))(j) \ = \ \text{if } val(b)(j) \text{ then } val(e_1)(j) \text{ else } val(e_2)(j)$
- $val(e[k, d])(j) \ = \begin{cases} val(e)(j+k) & \text{for } 0 \leq j+k < N \\ d & \text{otherwise} \end{cases}$

Note that contradictions and non-terminating recursive stream definitions can lead to the existence of none or multiple evaluation models. A *Lola 1.0* specification that has *exactly one* evaluation model for any kind of appropriate input, is called *well-defined* [3].

2.2 Lola 2.0

Lola 2.0 was originally proposed in combination with network intrusion detection. For this purpose *Lola 1.0* was extended by *stream equation templates* that introduced two new key features. They effect that *Lola 2.0* obtains the abilities to carry data along the streams as well as it enables the dynamic generation of new output streams.

2.2.1 Lola 2.0 Syntax

We will now take a look at the syntactical extension that leads us to *Lola 2.0*.

Definition 2.4. (Stream Equation Templates) [5]
Lola 2.0 generalizes *Lola 1.0 stream equations* to **stream equation templates** of the following form:

$$\begin{aligned}&\textbf{output } T\ s\langle p_1 : T_1, \ldots, p_l : T_l\rangle\\ &\quad\textbf{inv}: s_{inv}\\ &\quad\textbf{ext}: s_{ext}\\ &\quad\textbf{ter}: s_{ter}\\ &:= e(t_1, \ldots, t_m, s_1, \ldots, s_n, p_1, \ldots, p_l)\end{aligned}$$

A stream equation template defines a *template variable* s of type T. This template variable depends on *parameters* $p_1, \ldots, p_l$ with corresponding types $T_1, \ldots, T_l$.
For given values $v_1, \ldots, v_l$ of matching types $T_{p_1}, \ldots, T_{p_l}$,
an *instance* of s is defined as:

$$s\langle v_1, \ldots, v_l\rangle = e(t_1, \ldots, t_m, s_1, \ldots, s_n, p_1, \ldots, p_l)[p_1/v_1, \ldots, p_l/v_l]$$

The template stream expression $e(t_1, \ldots, t_m, s_1, \ldots, s_n, p_1, \ldots, p_l)$ extends the definition of *Lola 1.0* stream expressions as follows:

- let p_i for $i \in \{1, \ldots, l\}$ be a parameter of type T_i ,
  then p_i is a template stream expression of type T_i
- let s be a template variable and Op be an *aggregation operator* of type T ,
  then $Op(s)$ is a template stream expression of type T
  - $\textbf{any}(s)$ is an aggregation operator of type $bool$
  - $\textbf{count}(s)$ is an aggregation operator of type int

Stream equation templates depend on template variables s_{inv} , s_{ext} and s_{ter} which are called *auxiliary streams*:

- s_{inv} denotes the *invocation stream* variable of s and has type $T_{p_1} \times \cdots \times T_{p_l}$.
  Whenever an instance of s_{inv} has value $(v_1, \ldots, v_l)$, a new instance $s\langle v_1, \ldots, v_l\rangle$ of s is invoked (if it does not exist yet)
- s_{ext} denotes the *extension stream* variable of s and has type $bool$.
  If s gets invoked by parameter value $\alpha = (v_1, \ldots, v_l)$, then an instance s^{α}_{ext} of the template variable s_{ext} is invoked with the same parameter value.
  If at some position s^{α}_{ext} evaluates to $true$, then the output stream $s\langle v_1, \ldots, v_l\rangle$ is computed at that position

- s_{ter} denotes the *termination stream* variable of s and has type $bool$.
  If s gets invoked by parameter value $\alpha = (v_1, \ldots, v_l)$, then an instance s^{α}_{ter} of the template variable s_{ter} is invoked with the same parameter value.
  If at some position s^{α}_{ter} evaluates to $true$, then the output stream $s\langle v_1, \ldots, v_l \rangle$ is terminated and does not exist until it is invoked again

Now we take a look at some simplifying arrangements of template streams and show how we obtain *Lola 1.0* stream equations as special cases of template streams.

If for some template stream s

- the invocation stream s_{inv} is omitted, it is set to the default stream σ_0 that produces the empty tuple $()$ at every position.
- the extension stream s_{ext} is omitted, it is set to the default stream σ_{true} that produces $true$ at every position
- the termination stream s_{ter} is omitted, it is set to the default stream σ_{false} that produces $false$ at every position

We establish that *Lola 1.0* stream equations can be constructed by taking a *Lola 2.0* template stream equation with no parameters, and omitting all auxiliary streams.

2.2.2 Lola 2.0 Semantics

To formally define the semantics of *Lola 2.0*, we extend the previously proposed concept of *evaluation models*.

Definition 2.5. (*Lola 2.0* Evaluation Models) [5]
Let P be a *Lola 2.0* specification over independent stream variables $t_1, \ldots, t_m$ and dependent stream variables $s_1, \ldots, s_n$ with corresponding types $T_1, \ldots, T_{m+n}$. We fix a natural number $N \geq 0$ and define $\tau_1, \ldots, \tau_m$ as input streams of length N.
A set Φ of output streams of length N is called an *evaluation model* of specification P, if the below described conditions are satisfied. To indicate that an output stream does not exist, since it has not been invoked before, we add the symbol # to the types. Thus the type for an output stream is $T_{m+i} \cup \{\#\}$ for $1 \leq i \leq n$. To denote an instance of a template variable s_i with parameter value α, we use s^{α}_i. To refer to the corresponding output stream in Φ, we use σ^{α}_i. Now we take a look at the two conditions for an evaluation model:

1. *Condition 1:*
   Since new streams can be invoked during runtime, we can not predetermine of *how many*, respectively of *which* streams the evaluation model consists. To guarantee the condition that the evaluation model consist of a sufficiently large set of streams, we build the set Φ inductively as follows:

- $\sigma_0 \in \Phi$, $\sigma_{true} \in \Phi$ and $\sigma_{false} \in \Phi$,
  where σ_0 , σ_{true} and σ_{false} denote the above described *default streams*
- for every template stream variable s_i we do the following:
  Let s_{inv} denote the invocation stream variable of s_i. If for some parameter value α, the set Φ contains a stream σ^{α}_{inv} , then Φ must also contain the streams σ^{β}_{i} corresponding to the instances of s_i , that are invoked by s^{α}_{inv}. The parameter values β are given by the equation $\beta = \sigma^{\alpha}_{inv}(j)$, where $j < N$ and $\sigma^{\alpha}_{inv}(j) \neq \#$

2. *Condition 2:*
   Similar to the evaluation model of *Lola 1.0*, the second condition claims that the streams $\sigma^{\alpha}_{i} \in \Phi$ satisfy their corresponding stream expressions e_i. For *Lola 2.0* we have to distinguish whether a stream exists at some position j or not:

$$\sigma^{\alpha}_{i}(j) = \begin{cases} val(e_i[p_1/v_1, \ldots, p_l/v_l])(j) & \text{if } alive(s_i, \alpha, j) \\ \# & \text{otherwise} \end{cases}$$

   where $\alpha = (v_1, \ldots, v_l)$

   We define $alive(s_i, \alpha, j)$ as a function that evaluates to $true$, if the corresponding stream σ^{α}_{i} exists at position j. This means that at some position $j' < j$ an instance of s_i has been invoked with parameter value α, and it has not been terminated at any intermediate position j'' with $j' < j'' \leq j$.
   As we did for *Lola 1.0*, we now define the evaluation function val inductively:
   Let σ^{α}_{ext} be the extension stream of σ^{α}_{i}:

   – If $\sigma^{\alpha}_{ext}(j) = true$ then
     Base cases:
     - $val(c)(j) \;=\; c$
     - $val(t_h)(j) \;=\; \tau_h(j) \qquad$ for $1 \leq h \leq m$

     Inductive cases:
     - $val(f(e_1, \ldots, e_k))(j) \;=\; f(val(e_1)(j), \ldots, val(e_k)(j))$
     - $val(\mathbf{ite}(b, e_1, e_2))(j) \;=$ if $val(b)(j)$ then $val(e_1)(j)$ else $val(e_2)(j)$
     - $val(\mathbf{count}(s_h))(j) = \big|\{\beta \mid alive(s_h, \beta, j) = true\}\big|$
     - $val(\mathbf{any}(s_h))(j) \;= \begin{cases} true & \text{if for any instance } \sigma^{\beta}_{h}(j) = true \\ false & \text{otherwise} \end{cases}$
     - $val(s^{\beta}_{h}[0, d])(j) = \begin{cases} \sigma^{\beta}_{h}(j) & \text{if } alive(s_h, \beta, j) \\ d & \text{otherwise} \end{cases}$

- $val(s_h^{\beta}[k,d])(j) = \begin{cases} d & \text{if } j \geq N \text{ or } j < 0 \\ val(e_h[k-1,d])(j+1) & \text{if } k > 0, \sigma_{ext}^{\beta}(j) = true \\ val(e_h[k+1,d])(j-1) & \text{if } k < 0, \sigma_{ext}^{\beta}(j) = true \\ val(e_h[k,d])(j+1) & \text{if } k > 0 \\ val(e_h[k,d])(j-1) & \text{otherwise} \end{cases}$

– otherwise for $\sigma_{ext}^{\alpha}(j) = false$

- $val(e_i[p_1/v_1, \ldots, p_l/v_l])(j) = \#$

After formally defining the semantics of *Lola 2.0*, we now briefly sum it up in words:
Whenever the invoke stream of some template evaluates to a value that is not parameter of an instance of the template yet, a new instance is invoked with this parameter value. Consequently new instances of the corresponding extend stream and terminate stream are invoked too. Whenever the extend stream of some template instance evaluates to $true$, a new value of that instance is evaluated. Similarly, if the terminate stream of a template instance evaluates to $true$, this instance is terminated at that position. Nevertheless, the corresponding value at this position exists, provided that the instance has been extended. Due to the extend stream of a template, the stream instances follow their own local clocks instead of having the same one clock for all streams as in *Lola 1.0*.

For *Lola 2.0*, the definition of *well-defined specifications* is the same as for *Lola 1.0*:

Definition 2.6. (Well-defined Specifications) [5]
A specification P is called *well-defined*, if it has a unique evaluation model Φ for any set of appropriately typed input streams of length N.

Checking a *Lola 2.0* specification for well-definedness is not easy, since it describes a semantic condition. A solution for this problem is given by a syntactic criterion, which can be checked by means of the *dependency graph*.

Definition 2.7. (Dependency Graph) [5]
The *dependency graph* of a *Lola 2.0* specification is a weighted and directed multi-graph $G = \langle V, E \rangle$ with $V = \{t_1, \ldots, t_m, s_1, \ldots, s_n\}$. If stream expression e_i of template variable s_i contains some subexpression of type $s_k[w,d]$, we add an edge $e = \langle s_i, s_k, w \rangle$ from s_i to s_k with weight w to the set E. Analogously, we add edges leading from s_i to independent stream variables t_k if necessary. An edge $\langle s_i, s_k, w \rangle$ represents the dependency of s_i from template variable s_k by an offset w. Since stream expressions can have multiple references to the same stream variable but with different offsets, we are talking about a multi-graph.

Cyclic dependencies:
A sequence $v_1 \xrightarrow{e_1,w_1} v_2 \dots v_k \xrightarrow{e_k,w_k} v_{k+1}$ where all $e_i = \langle v_i, v_{i+1}, w_i \rangle \in E$ and $v_1 = v_{k+1}$ is called a *cycle* of the graph. The sum of all weights $\sum_{i=1}^{k} w_i$ is called *total weight* of the cycle.

After formally defining the dependency graph of a *Lola 2.0* specification, we now specify the syntactical criterion of *well-formed* specifications, which leads to well-defined specifications.

Definition 2.8. (Well-formed Specifications) [5]
A specification P is called *well-formed*, iff its dependency graph G does not contain a cycle of total weight zero.
A well-formed specification is guaranteed to be *well-defined*.

Chapter 3

Compiling Lola 2.0 to C

In this chapter, we propose our compilation approach for *Lola 2.0*. We describe the abstract syntax tree of a specification and define the implemented dependency graph. After illustrating its build-up, we describe the analysis of the dependency graph and determine some restrictions of *Lola 2.0*. Finally, we present our algorithm to generate an evaluation order and explain the main file's structure as well as additionally generated data structures.

3.1 Abstract Syntax Tree (AST)

The starting point of our compilation is the AST of a *Lola 2.0* specification, whose structure is described only briefly. Furthermore we specify the types allowed for the usage in a *Lola 2.0* specification.

Structure

The root node of our AST, which represents the given specification, contains information about all input streams, as well as stream template and trigger definitions. Additionally, we allow for the definition of typed *constant stream variables* s_c of the form

$$\textbf{constant } T_c \; s_c = d$$

where d is a *constant value* of type T_c. The corresponding stream σ_c produces the constant value d at every position. This small syntactical extension fits problem-free into the above described semantics of *Lola 2.0* and is not discussed in further detail.
As expected the tree structure of input and constant streams is simply composed of the type, the variable name and, in the latter case, the fixed value. Output streams additionally consist of a list of typed parameters, stream variables of the corresponding invoke, extend and terminate stream, and a stream expression. Different

from the specification, where triggers are solely defined through their expression, we additionally mark them by a numbered name. This allows for a differentiation between several trigger notifications. The structure of stream and trigger expressions follows general expectations, except that aggregation operators **any** and **count** are not permitted in stream expressions.

Types

Since every constant, input and output stream is defined with a type, we have to determine which types are valid. For our AST, we consider the basic types **bool**, **int** and **string**, which we call *atom types*. In types of the form $T_1 \times \cdots \times T_k$, which we call *tuple types*, we restrict T_i to be of atom type for $1 \leq i \leq k$. Thus we achieve that tuples can not consist of further tuples.

3.2 Dependency Graph (DG)

The DG serves as an intermediate representation of a given specification. In this chapter, we present the implemented design, as well as how the DG is build from the AST. Afterwards we talk about the information and resultant restrictions we obtain by analyzing the DG.

3.2.1 Design

The implemented DG is an extended version of the one presented in Definition 2.7. Although, we have extended sets of nodes and edges, we are still talking about a directed multi-graph $G = \langle V, E \rangle$.

Nodes

So far the set of nodes has been composed of the input and output stream variables. Now we extend V by adding a new node r_i for each trigger definition. Additionally we add new nodes for the default empty, default true and default false stream called q_0, q_{true} and q_{false} respectively. Furthermore, we add a node q_{none} , which is similar to q_0, but has no effect on the evaluation. We will discuss the reason in Section 3.2.2. Thus, the new set of nodes is

$$V = \{t_1, \ldots, t_m, s_1, \ldots, s_n, r_1, \ldots, r_u, q_0, q_{true}, q_{false}, q_{none}\}$$

In Section 3.2.2, we also justify why the constant streams, introduced in Section 3.1, are not represented in the DG.

Edges

Before we talk about how the set of edges E is composed, we give the extended definition of an edge by

$$e = \langle v_b, v_g, w, \gamma \rangle$$

with *base* node v_b, *goal* node v_g , weight w and *edge type* $\gamma \in \Gamma$. The set of edge types is given by

$$\Gamma = \{invBy, invOf, extBy, extOf, terBy, terOf, evalBy, evalOf\}$$

We now build the set of edges as follows: For each stream template definition with template variable s_i, and corresponding invocation, extension and termination stream variables s_{inv}, s_{ext} and s_{ter} respectively

- we add edges $e = \langle s_i, s_{inv}, 0, invBy \rangle$ and $e' = \langle s_{inv}, s_i, 0, invOf \rangle$ to E
- we add edges $e = \langle s_i, s_{ext}, 0, extBy \rangle$ and $e' = \langle s_{ext}, s_i, 0, extOf \rangle$ to E
- we add edges $e = \langle s_i, s_{ter}, 0, terBy \rangle$ and $e' = \langle s_{ter}, s_i, 0, terOf \rangle$ to E

Furthermore, if stream (or trigger) expression e_i of template variable s_i, (or trigger r_i respectively,) contains some subexpression of type $s_k[w, d]$

- we add edges $e = \langle s_i, s_k, w, evalBy \rangle$ and $e' = \langle s_k, s_i, w, evalOf \rangle$
  (or $e = \langle r_i, s_k, w, evalBy \rangle$ and $e' = \langle s_k, r_i, w, evalOf \rangle$ respectively)
  to E

where s_k denotes a dependent or independent stream variable.
Thus, we obtain a multi-graph, which always contains an edge *and* its reverse. This eases the gathering of information during code generation.

3.2.2 Construction

After giving a formal definition of our DG, we now describe how it is stepwise build from the AST. Before starting the construction, we have to get rid of the constant stream definitions, which are not represented in the DG. Since constant streams produce the same value at every position, we replace their occurrences in the AST by the constant value itself. After that, we build up the set of nodes V and the set of edges E separately and combine them in a further step to the implemented DG, which is exclusively existing of nodes, containing all information of the edges.

Nodes

We gradually build up the set of nodes by iterating over the AST and generating a new node for every independent and dependent stream variable, and for every trigger definition.

Edges

The set of edges is build by an iteration over all output stream and trigger definitions of the AST. Considering the output streams, we create the edges to their invocation, extension and termination stream together with the corresponding antagonist edges in reversed direction. For the stream expressions as well as the trigger expressions, we recursively search for subexpressions of the form $s_k[w, d]$ and add appropriate edges in either direction.

The implemented DG

As mentioned above, the implemented DG is solely composed of nodes, which are stocked with references according to their incoming and outgoing edges. Thus, every node has following references to dependent nodes, analogically to the edge types.
Node v has references

- $invBy$ to its invocation node
- $extBy$ to its extension node
- $terBy$ to its termination node
- $evalBy$ to the set of nodes, combined with associated offsets, which are required for the evaluation of its stream expression
- $invOf$ to the set of nodes, which v is invocation node of
- $extOf$ to the set of nodes, which v is extension node of
- $terOf$ to the set of nodes, which v is termination node of
- $evalOf$ to the set of nodes, combined with associated offset, that require v for the evaluation of their stream expressions

To write those information into the nodes, we iterate over the set of edges. For each edge $e = \langle v_b, v_g, w, \gamma \rangle$, a new reference to the goal node v_g (combined with offset w) is added to the base node v_b according to the given edge type γ. For every node v_b, we additionally calculate the maximum absolute offset value w_b of its $evalOf$ dependencies. w_b denotes the number of past values of stream v_b that have to be stored, since other streams depend on them. Thus, the buffer size of to v_b corresponding streams is fixed to at minimum $w_b + 1$ for all necessary past, and the current value.

At the beginning of this chapter, in Section 3.2.1, we added a new default node, called q_{none}, to the set of nodes. To justify that, we assume a stream template definition s, whose invocation stream is set to the default stream s_0. Nevertheless s is defined with some parameters $\langle T_1 : p_1, \ldots, T_l : p_l \rangle$. Since the type of s_0 does not fit the parameter type $T_1 \times \cdots \times T_l$, s can never be invoked by its invocation stream. To make such cases obvious, we redirect the original invocation dependency q_0 to q_{none} .
Although s can not be invoked by its invocation stream, it however is not negligible, since it could be invoked as extension or termination stream of some other stream definition s'.

3.2.3 Circular Dependencies and Restrictions

In this chapter, we talk about dependencies inside *Lola 2.0* specifications and how they lead to several restrictions. Since we only consider well-defined specifications, we have to check if a given specification is well-formed, as is exposed in Definition 2.8. After that we justify further restrictions for *Lola 2.0* in general and for the implementation of the compiler.

Finding Circular Dependencies

For our DG $G = \langle V, E \rangle$, the general definition of a *cycle* and the corresponding *total weight*, given in Definition 2.7, remains the same. However we have to remind, that we always added edges in combination with their reverse edges to E, for having easy access to all important information during code generation. Hence we have to restrict the set of regarded edges depending on their edge types.

Relevant Edges
For the detection of circular dependencies, an edge $e = \langle v_b, v_g, w, \gamma \rangle$, leading from node v_b to node v_g, is regarded, if its edge type $\gamma \in \{invBy, extBy, terBy, evalBy\}$. In words, edge e is only interesting, if v_b depends on v_g, i.e. the stream variable name of v_g occurs in the stream template definition of v_b. Moreover we are able to discard some more edges from the analysis of cycles. If we take a closer look at the terminate dependencies, it is recognizable that they have no effect on the evaluation process itself. They only serve as an indicator whether a stream is terminated at the end of the evaluation step. Thus, we can safely discard edges of type $terBy$ from the analysis of cycles too. The remaining set of edges, which are relevant for cycle detection, is composed of edges $e_i = \langle v_b, v_g, w, \gamma \rangle$ with type $\gamma \in \{invBy, extBy, evalBy\}$.

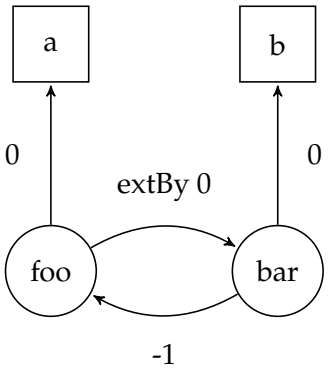
Finding cycles
In order to find cycles, we first divide the DG into strongly connected components (SCCs) by using Tarjan's depth first search algorithm [14]. In doing

```
input int a
input int b

output int foo <>
  ext: bar[0,false]
:= a

output bool bar <>
:= (foo[-1,0]=b)
```

(a) Specification



(b) Dependency Graph

Figure 3.1: $extBy$ in circular dependency

so, we additionally extract possible self-loops.
In the next step, the remaining SCCs are searched for contained cycles by using Johnson's algorithm for finding all elementary circuits of a directed graph [10].

Restrictions

Now we define some further restrictions of *Lola 2.0* specifications to guarantee the existence of exactly one evaluation model. After that we specify some limitations of our implemented compiler.

Restrictions of *Lola 2.0*

1. The DG of a *Lola 2.0* specification must not contain some cycle of total weight zero, as already stated in Definition 2.8.
2. A cycle of the DG must not contain an edge of type $extBy$, since this may lead to none, or multiple evaluation models.
   Example: We consider the *Lola 2.0* specification given by Fig. 3.1a and its corresponding dependency graph in Fig. 3.1b.
   Stream `foo` and stream `bar` are obviously in a circular dependency containing an $extBy$ edge. Now we try to evaluate the streams regarding following input values:
   First we have to evaluate `bar`. Since `foo` does not exist yet, we have to take the default value 0 and `bar` evaluates to $true$. That means `foo` is extended and produces the value 2.
   In the next evaluation step, the value of `foo` with offset -1 still does not

	0	1	2
a	2	2	$\cdots$
b	0	0	$\cdots$

exist. Therefore `bar` evaluates to $true$ again. Thus, `foo` is extended and produces the value 2 again.
Now that `foo` has been extended, its value with offset -1 exists and is 2. If we would take this value now for the evaluation of `bar`, it would evaluate to $false$, and thus, `foo` should not have been extended.

	0	1	2
foo	2	?	$\cdots$
bar	$true$	?	$\cdots$

This contradiction induces, that we do not permit $extBy$ dependencies in cycles.

Restrictions of our Implementation

1. As main restriction of our implemented compiler, we forbid the use of positive offsets in stream expressions. Thus, we obtain an *efficiently monitorable* fragment of *Lola 2.0*, with the assumption that the number of instances invoked during the monitoring process is bounded, which is often true in practice. *Efficiently monitorable* means that, for some specification, the memory consumption is constant in the length of the input streams [5].
2. Considering we have some stream s with its extension stream s_{ext} which is already evaluated. Now if a new instance of s, and therefore also a new instance of s_{ext} is invoked, this new instance of s_{ext} will not be evaluated during the current evaluation step. In that case, the compiler will show an appropriate warning.

3.3 Code Generation

In this chapter, we discuss how the implemented compiler works and what the generated code looks like. We first show the generation of the evaluation order and how it is embedded in the monitors `main.c` file. Then we explain the generated auxiliary data structures used to represent streams and stream templates.

3.3.1 Evaluation Order and `main.c`

The evaluation order is the core part of the generated monitors *main* function. For each position of the input streams, the output streams and triggers are evaluated according to the predetermined evaluation order.

Generating the Evaluation Order (EO)

The general idea of our algorithm is to determine, which stream template is next evaluable, since all its dependencies are already evaluated. Algorithm 1 gives a slightly simplified version of how the evaluation order is identified. We separate a set DONE of ready evaluated streams, and a set EVAL of streams, that are invoked but not evaluated so far. Furthermore we have a set of pairs called INVKD. If some pair $\langle v_1, v_2 \rangle \in$ INVKD, then v_1 has already been invoked by an invocation value from v_2. We start our algorithm with all sets being empty. Since the values of default and input streams are available at the beginning of an evaluation step, their nodes are directly added to DONE. Now we have to invoke some new streams to start the evaluation process. For every node in DONE, we are able to invoke all its $invOf$ dependencies. Thus, these invokes are the starting point of our evaluation order. We have to keep in mind, that due to the invocation of some stream s, its extension ($s.extBy$) and termination ($s.terBy$) streams are invoked recursively. All the nodes of corresponding newly invoked streams are added to EVAL. Furthermore we add appropriate pairs $\langle v_1, v_2 \rangle$, with v_2 being the node where v_1 gets its invoke value from, to INVKD.
In the main part of our algorithm, we iterate over the set EVAL, searching for nodes that are ready for being evaluated next. Looking at some node $v \in$ EVAL, we are able to evaluate v, if its extension stream and all its stream expression dependencies have already been evaluated. That means, $v.extBy$ and all dependencies $v.evalBy$ have to be in DONE. If some node of $v.evalBy$ is not yet in DONE, we can look at its corresponding offset. Considering the offset is of negative value, we are allowed to evaluate v anyway, since the past values always exist. Now if we found a node v that can be evaluated next, it is added to our EO. Aside from that, it is removed from EVAL and added to DONE. The evaluation of some stream v may cause some side effects. If v is invocation stream of some other template, i.e. $v.invOf \neq \emptyset$, we have to add that invocation to our EO. Additionally we again have to consider potential recursive invocations and add all new invokes to EVAL and to INVKD respectively. Furthermore we have to check, if v is termination stream of some other stream, i.e. $v.terOf \neq \emptyset$, and add this potential termination information to the EO. Since a node, that has been evaluated, is removed from EVAL, we are finished with our EO as soon as the set is empty.
At the end of an evaluation step, all collected terminations are executed, before continuing with the next step.

```
1  DONE : set of stream variables that are ready evaluated
2  EVAL : set of stream variables that have to be evaluated
3  INVKD : set of stream variable pairs ⟨v1, v2⟩, where v1 has been invoked by v2
4  EO : evaluation order
5  function EvaluationOrder()
6      add default and input streams to DONE;
7      foreach node ∈ DONE do
8          foreach n ∈ node.invOf do
9              FnInvoke(n, node);
10         end
11     end
12     while not EVAL.empty do
13         node = EVAL.get();
14         if node.extBy ∈ DONE then
15             bool allDone := true;
16             bool negOffset := true;
17             foreach n_dep ∈ node.evalBy do
18                 if n_dep ∉ DONE then
19                     allDone := false;
20                     if n_dep.offset = 0 then
21                         negOffset := false;
22             end
23             if allDone||negOffset then
24                 FnEvaluate(node);
25     end
26     EO.add(execute all terminations);
27 end
28 function FnInvoke(n, inv)
29     if not ((n, inv) ∈ INVKD) then
30         EO.add(invoke(n, inv));
31         INVKD.add(n, node);
32         EVAL.add(n);
33         FnInvoke(n.extBy, inv);
34         FnInvoke(n.terBy, inv);
35     end
36 end
37 function FnEvaluate(node)
38     EO.add(evaluate(node));
39     EVAL.remove(node);
40     DONE.add(node);
41     foreach n ∈ node.invOf do
42         FnInvoke(n, node);
43     end
44     foreach n ∈ node.terOf do
45         EO.add(collect terminate(n, node));
46     end
47 end
```

Algorithm 1: Generating the Evaluation Order (EO)

Structure of main.c

Now we look at the general structure of generated *main.c* files.
First we do some includes of C *standard libraries*.

```
#include <stdio.h>
#include <stdlib.h>
...
```

We have some further includes of the generated auxiliary data structures, that will be explained in detail in Section 3.3.2.

```
#include "HashMap.h"
#include "BUFFER/Buffer_b.h"
#include "TEMPLATE/Template_i_Bb.h"
...
```

Before starting the `main` function, we declare the function `getInput` that manages reading the input. Furthermore, we declare the global variable `input_count` to count the current input position.

```
bool getInput(FILE*);
unsigned long input_count = 0;
```

Input streams and templates are declared as global variables too. Since input streams can not have multiple instances, we treat them as buffers. in1 denotes an auxiliary variable of `inputStream` used in function `getInput` as explained later on.

```
bool in1;
Buffer_b* inputStream;
Template_i_Bb* someTemplate;
```

At the beginning of our `main` function, we get the path of the input file from the argument list and open the file to read data.

```
int main(int argc, char** argv){
  char* filename  = argv[1];
  FILE* inputfile = fopen(filename, "r");
```

Next, we initialize the above declared input stream and template data structures by calling their appropriate functions.

```
  inputStream = buffer_b_new(1);
  userAction  = template_i_Bb_new(1);
```

Now we are ready to start the evaluation. For every new input, read from the `inputfile` by the function `getInput`, the generated code of the EO is executed. If it exists no new input, since the inputfile is at its end, the evaluation is finished.

```
  while (getInput(inputfile)){
    // generated code of the EO
  }
  // end of main function
```

Finally we take a look at the `getInput` function, that reads the values from the input file into the appropriate data structures. Since C has no format specifier for `bool`ean values, they are read to an auxiliary variable `b1` of type `int` and converted to boolean type later on. After that, the input values are written to the corresponding data structures by calling their appropriate function.

```
bool getInput(FILE* tracefile) {
  int b1;
  if (fscanf(tracefile, "%d\n", &b1) == EOF){
    return false;
  }
  input_count++;
  in1 = b1 ? true : false;
  buffer_b_writeValue(inputStream, 0, in1);
  return true;
}
```

3.3.2 Generated Data Structures

Now we look at the compiler-generated data structures, namely `Tuple`, `Buffer` and `Template`, for managing stream instances and corresponding values. In general, the instances of a template are accumulated in a generic hash map. Since there is nothing special about the hash map, we do not discuss its code here. During an evaluation cycle, all streams that need to be terminated at the end are collected in a data structure called `TerminateSet`, which is explained later on.

Tuple

As defined in Section 3.1, a tuple is a collection of values of atom type. To store these values, the tuple is represented by a `struct` as follows:

```
struct Tuple_bis {
  bool  exists;
  char* key;
  bool  value1;
  int   value2;
  char* value3;
};
```

In this case, we deal with a tuple called `Tuple_bis`, where the abbreviation `bis` denotes the types of the collected values. That is, our tuple consists of a **b**oolean (`value1`), an **i**nteger (`value2`) and a **s**tring (`value3`). The string `key` is built as concatenation of all values and serves as key for hashing. Only if the key has been built, the tuple is marked as existent (`exists = true`).
With the function

```
Tuple_bis* tuple_bis_new();
```

a new tuple is allocated.
For every position of the tuple, a corresponding function is generated for writing and reading the values.

```
  void tuple_bis_writePos1(Tuple_bis*, bool value);
  void tuple_bis_writePos2(Tuple_bis*, int value);
  void tuple_bis_writePos3(Tuple_bis*, char* value);

  bool  tuple_bis_readPos1(Tuple_bis*);
  int   tuple_bis_readPos2(Tuple_bis*);
  char* tuple_bis_readPos3(Tuple_bis*);
```

Since every value of a tuple is written individually, we do not want to rebuilt the `key` every time a value changes. After writing all new values into the tuple, the `key` is built by a call of the function `tuple_bis_writeKey`. Furthermore there are functions for reading and deleting the `key`.
If a new `key` is written, the tuple is marked as existent, which can be checked by the function `tuple_bis_exists`.

```
  void  tuple_bis_writeKey(Tuple_bis*);
  char* tuple_bis_readKey(Tuple_bis*);
  void  tuple_bis_deleteKey(Tuple_bis*);
  bool  tuple_bis_exists(Tuple_bis*);
```

For copying a tuple's values to an other tuple, or comparing the values of two tuple, there are appropriate functions given.

```
  void tuple_bis_copy(Tuple_bis*, Tuple_bis*);
  bool tuple_bis_cmp(Tuple_bis*, Tuple_bis*);
```

Finally we of course have a function to delete a tuple and free its allocated memory.

```
  void tuple_bis_delete(Tuple_bis*);
```

With all these functions, tuple are easy to handle for other data structures.

Buffer

Every stream, stores its values in a buffer. Since a stream needs to store only a certain amount of past values, we implemented a ring-buffer, which always overwrites the oldest value. In the following, we look at the generated code of a buffer named `Buffer_Tbis`. The `Tbis` indicates that the stored values are **T**uple containing a **b**oolean, **i**nteger and **s**tring value, as described above. If the buffer stores values of atom type, the name only contains the appropriate abbreviation. An element of the buffer is represented by a `struct` composed of the stored `value` and a flag to indicate its existence.

```
struct Buffer_TbisElement {
  bool exists;
  Tuple_bis* value;
};
```

The `struct` of the buffer itself contains an array of size `size` of buffer elements. `current_index` denotes the position which is written next.

```
struct Buffer_Tbis {
  Buffer_TbisElement* element;
  unsigned int size;
  unsigned int current_index;
};
```

The function

```
Buffer_Tbis* buffer_Tbis_new(unsigned int size);
```

creates a new buffer with `size` as the number of elements. With `offset` zero, new values are written to the buffer. With offsets other than zero, we are able to write values at arbitrary positions, but this is not used in our generated code.

```
void buffer_Tbis_writeValue(Buffer_Tbis*, int offset,
   Tuple_bis* value);
```

Offset zero is also used to read the most recent value. Preceding positions are accessed by negative offsets. Since reading a non-existing value leads to an error, the existence of a value should always be checked first.

```
Tuple_bis* buffer_Tbis_readValue(Buffer_Tbis*, int offset);
bool buffer_Tbis_existsValue(Buffer_Tbis*, int offset);
```

To delete a buffer and all its elements we call the function:

```
void buffer_Tbis_delete(Buffer_Tbis*);
```

Template

The Template data structure is generated to manage all instances of a template. We look at an example template, namely `Template_i_BTbis`, that is invoked by an **i**nteger value. The instances of this template produce values of the above type of **T**uple (`Tbis`), which are stored in a corresponding **B**uffer.
The `struct` of an instance is composed of a string `key`, its invocation value `inv_value` and a buffer of appropriate type, to store the stream's values.

```
struct Template_Tsib_BTbisStream {
  char* key;
  Tuple_sib* inv_value;
  Buffer_Tbis* buffer;
};
```

The instances of a template are managed in a hash map with corresponding iterator to be able to iterate over all instances of the template. Apart from that, the `buffer_size` of the instances is stored in the template, since this information is needed whenever a new instance is generated.

```
struct Template_Tsib_BTbis {
  unsigned int buffer_size;
  HashMap* instances;
  HashMapIterator* iterator;
};
```

To obtain a new template data structure, we call the function

```
Template_Tsib_BTbis* template_Tsib_BTbis_new(unsigned int
   buffer_size);
```

with the information of an instance's buffer size as argument. Thereby, a hash map

of fixed initial size and its iterator are created.
The following function is used to invoke a new instance for a given invocation value `inv_value`.

```
void template_Tsib_BTbis_invoke(Template_Tsib_BTbis*,
   Tuple_sib* inv_value);
```

To invoke a new template instance, the function checks whether an appropriate stream already exists. If that is not the case, a new stream is created and inserted into the hash map.

```
bool template_Tsib_BTbis_exists(Template_Tsib_BTbis*,
   Tuple_sib* inv_value);
Template_Tsib_BTbisStream* template_Tsib_BTbisStream_new(
   Tuple_sib* inv_value, unsigned int buffer_size);
```

To obtain more flexibility, there are two functions given to terminate an instance. The first one receives the appropriate invocation value and the second one receives the stream to be terminated itself.

```
void template_Tsib_BTbis_terminate(void* template, void*
   inv_value);
void template_Tsib_BTbisStream_terminate(Template_Tsib_BTbis*
   template, Template_Tsib_BTbisStream* stream);
```

Both call the following function to delete the instance and remove it from the hash map.

```
void template_Tsib_BTbisStream_delete(
   Template_Tsib_BTbisStream* stream);
```

For reading, writing and checking the existence of values, there are functions with the invocation value as argument to specify the stream

```
void template_Tsib_BTbis_writeValue(Template_Tsib_BTbis*,
   Tuple_sib* inv_value, int offset, Tuple_bis* value);
Tuple_bis* template_Tsib_BTbis_readValue(Template_Tsib_BTbis*,
    Tuple_sib* inv_value, int offset);
bool template_Tsib_BTbis_existsValue(Template_Tsib_BTbis*,
   Tuple_sib* inv_value, int offset);
```

as well as functions with the appropriate stream itself as argument.

```
void template_Tsib_BTbisStream_writeValue(
   Template_Tsib_BTbisStream* stream, int offset, Tuple_bis*
   value);
Tuple_bis* template_Tsib_BTbisStream_readValue(
   Template_Tsib_BTbisStream* stream, int offset);
bool template_Tsib_BTbisStream_existsValue(
   Template_Tsib_BTbisStream* stream, int offset);
```

To read the invocation value of a stream, we can use the function:

```
Tuple_sib* template_Tsib_BTbisStream_getInv(
   Template_Tsib_BTbisStream* stream);
```

For tuple invocation types, we additionally need to have the possibility to directly access the parameters of the tuple.

```
char* template_Tsib_BTbisStream_getParam1(
   Template_Tsib_BTbisStream* stream);
int template_Tsib_BTbisStream_getParam2(
   Template_Tsib_BTbisStream* stream);
bool template_Tsib_BTbisStream_getParam3(
   Template_Tsib_BTbisStream* stream);
```

After resetting the iterator, we can repeatedly call its function `template_Tsib_BTbisIterator_next` until it returns `NULL`, to access all streams of the template.

```
void template_Tsib_BTbisIterator_reset(Template_Tsib_BTbis*);
Template_Tsib_BTbisStream* template_Tsib_BTbisIterator_next(
   Template_Tsib_BTbis*);
```

There is one last function to determine the number of instances existing in the template, which implements the aggregation operator **count**.

```
unsigned int template_Tsib_BTbis_getSize(Template_Tsib_BTbis*);
```

Given above functions to manage stream templates, it is quite simple to transform the, with Algorithm 1, generated EO into C code. However our compiler produces the code directly, without generating the EO in terms of some intermediate representation.

Special Cases Apart from the general template data structure , we differentiate two special cases of template definitions:

1. *Invoked by default empty stream*
   A template that is invoked by the default empty stream and has no parameters, can only have a single instance invoked by the empty tuple. Such templates are represented by a single buffer and a flag to mark if the stream is currently invoked.

2. *Invoked by bool stream*
   A template that is invoked by a stream of type **bool**, can at most have two instances invoked by *true* and *false*. Such a template is represented by two streams with corresponding invocation value and flags to mark if the stream is currently invoked.

TerminateSet

The `TerminateSet` data structure is used to collect all stream instances, that have to be terminated at the end of an evaluation step. After finishing an evaluation step all streams that have been collected will be terminated.
Since we are talking about C code, it is not enough to collect the streams, but we also need to know, which function has to be called to terminate some stream. The termination function of a stream generally receives a pointer to the template and the appropriate invocation value of the stream. Therefore, a pointer to a termination function is defined as follows:

```
typedef void (*TerminateFun)(void*, void*);
```

The `TerminateSet` is implemented as a `struct` composed of a linked list of `elements` and a pointer to the `next` free spot in the list.

```
struct TerminateSet{
  TerminateSetElem* elements;
  TerminateSetElem* next;
};
```

An element of the set contains a pointer to the template (`temp`) and to the invocation value (`inv`), as well as the function pointer to the corresponding termination function (`terminate`). Furthermore it stores a pointer to the next element in the list. Initially, new memory is allocated for every new element added to the list. Instead of freeing the memory after terminating the elements, we set the flag `empty`, to avoid unnecessary reallocations.

```
struct TerminateSetElem{
  bool empty;
  TerminateSetElem* next;
  TerminateFun terminate;
  void* temp;
  void* inv;
};
```

Before starting the evaluation process of the monitor, the `TerminateSet`, if needed, is created and initialized.

```
TerminateSet* terminateSet_new();
```

The following function is used during the evaluation for adding new elements to the set.

```
void terminateSet_add(TerminateSet* ts, TerminateFun fun,
   void* temp, void* inv);
```

At the end of every evaluation step the function

```
void terminateSet_term(TerminateSet* ts);
```

is called to terminate all collected streams and to reset the list.

Chapter 4

Experiments

In this chapter we compare the performance of our compiled C code monitors to an interpreter implemented in *Swift* [1]. We first describe the considered example specifications and then present our findings.

4.1 Specifications

4.1.1 Web Application Fingerprinting

The first example is on the application of network intrusion detection and was primarily presented in [5]. A slightly modified version of the specification, describing a pattern for detecting a web application fingerprinting attack, is given in Fig. 4.1. Such an attack is performed by sending arbitrary HTTP requests to a server and waiting for its response. Since the hostile client mostly performs random requests, the server replies in many cases a *Bad Request* or a *Page Not Found* message. To detect such an attack, we monitor IP addresses initiating bad replies and check whether they keep on sending further requests.
Our specification in Fig. 4.1 slightly differs from the one given in Faymonville et al. [5]. To avoid expensive string comparisons we define input streams of integer type for `Source` and `Destination` addresses. The `Protocol` is also specified by an integer number, where 1 represents the HTTP protocol. We only separate positive and negative server responses to be able to use a boolean value as `ResponsePhrase`. In the specification, the stream `badRequest` evaluates to *true*, whenever a bad HTTP request is detected. If that is the case, the stream `badHttpRequestInvoke` is extended by the corresponding pair of `Source` and `Destination`. This starts the monitoring process by invoking corresponding template instances for `badHttpRequestExtend`, `webAppFingerprintingTerminate` and `webAppFingerprinting`. Whenever the same pair initiates another bad request, `badHttpRequestExtend` evaluates *true* and `webAppFingerprinting` counts the in-

```
input int Protocol
input bool ResponsePhrase
input int Source
input int Destination

output bool badRequest <>
:= (Protocol = 1) & (ResponsePhrase = false)

output (int,int) badHttpRequestInvoke <>
  extend:   badRequest
:= (Source,Destination)

output bool badHttpRequestExtend <int src, int dst>
  invoke:   badHttpRequestInvoke
:= ((src = Source) & (dst = Destination)) & (ResponsePhrase =
   false)

output bool webAppFingerprintingTerminate <int src, int dst>
  invoke:   badHttpRequestInvoke
:= ((src = Source) & (dst = Destination)) & (ResponsePhrase =
   true)

output int webAppFingerprinting <int src, int dst>
  invoke:   badHttpRequestInvoke
  extend:   badHttpRequestExtend
  terminate:  webAppFingerprintingTerminate
:= webAppFingerprinting(src,dst)[-1,0] + 1

trigger any(webAppFingerprinting > threshold)
```

Figure 4.1: *Lola 2.0* specification web application fingerprinting attack

```
input int SensorId
input int SensorData

output bool action <int id>
  invoke:   SensorId
:= SensorId = id

output int splitData <int id>
  invoke:   SensorId
  extend:   action
:= SensorData

output int windowSum <int id>
  invoke:   SensorId
  extend:   action
:= (windowSum(id)[-1,0] + splitData(id)[0,0]) -
   splitData(id)[-10,0]

output int average <int id>
  invoke:   SensorId
  extend:   action
:= windowSum(id)[0,0] / 10

output bool highValue <int id>
  invoke:   SensorId
:= average(id)[0,0] > threshold1

output int newAlert <int id>
  invoke:   SensorId
  extend:   highValue
:= id

output bool terminAlert <int id>
  invoke:   newAlert
  terminate:  terminAlert
:= average(id)[0,0] < threshold1

output bool Alert <int id>
  invoke:   newAlert
  terminate:  terminAlert
:= average(id)[0,0] > threshold2

trigger any(Alert = true)
trigger count(Alert) > threshold3
```

Figure 4.2: *Lola 2.0* specification sensor data monitoring

cidents. If the client initiates at some point a positive request, the corresponding instance of `webAppFingerprintingTerminate` evaluates to *true* and the monitoring of that pair is terminated.
If the quantity of bad responses induced by `any` instance of the `webAppFingerprinting` template exceeds a certain `threshold`, the trigger generates a notification to indicate a potential web application fingerprinting attack.

4.1.2 Sensor Data Monitoring

Our second specification given in Fig. 4.2 focuses some simple statistical measurements. Given an input stream `SensorId` consisting of sensor IDs and a corresponding input stream `SensorData` of sensor data, we want to monitor the compliance of a specific sensor's data stream with some given requirements. To correctly work on the data of some sensor, we first have to split the input data stream into new `splitData` streams. That means, for every new ID of `SensorId`, a new instance of `splitData` is invoked. Whenever new data for a specific sensor arrives, the corresponding `action` stream evaluates to *true* and the data is written to its `splitData` instance.
With the templates `windowSum` and `average`, some simple statistics on the sensor's data are computed. An instance of `windowSum` computes the sliding window sum of the last 10 values by continuously adding the latest value to, and subtracting the value with offset -10 from the sum. The template `average` determines the average of the last 10 values by dividing the windowed sum by 10.
With `highValue` we check if some sensors average value exceeds a certain `threshold1`. Happening so, `newAlert` is extended by the corresponding sensor ID and induces the invocation of `Alert` and `terminAlert` to start its monitoring. If at some point the average falls below the threshold again, `terminAlert` evaluates to *true* and the monitoring process of the associated sensor is terminated. On the other hand, if the average value exceeds a `threshold2`, a trigger notification is generated by the aggregation operator **any**. Another trigger notification is initiated whenever the amount (operator **count**) of currently monitored sensor IDs exceeds `threshold3`.

4.1.3 Specification Analysis

We briefly analyze the given specifications on main differences so that we can interpret the results better.

- The web application fingerprinting (WAF) specification is composed of 5 template definitions. `badRequest` and `badHttpRequestInvoke` represent single streams, whereas the other 3 templates support instance invocation. Except

from the `webAppFingerprinting` template, whose instances need to have a buffer of size 2, all templates get by with a single value for their current position. The WAF specification contains only 1 arithmetical expression, occuring in the `webAppFingerprinting` template, beside many comparison and logical operations.

- The sensor data monitoring (SDM) specification is composed of 8 template definitions, all supporting instance invocation. Except from `Alert` and `terminAlert`, which only contain the currently monitored sensor instances, the templates contain all possible instances, since they are invoked by the input stream `SensorId`. Besides the templates with buffers of size 1, the SDM specification consist also of the `windowSum` template, whose buffers are of size 2, and the `splitData` template with buffers of size 11, to be able to store the last 10 values. There are 3 arithmetical expressions and several comparison operations occurring in the SDM specification.

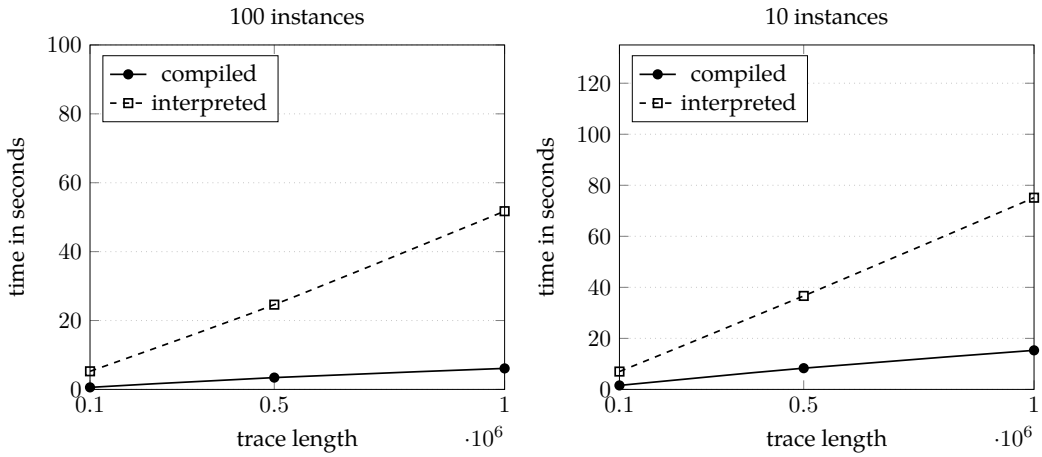
Looking at above recapitulation of our example specifications we can state that SDM is the more expensive one. It is not only composed of more templates with stream invocation, but also needs to store a good deal more values at every position. Additionally the SDM template expressions contain the more costly operations.

4.2 Experiment Conditions

We conducted our experiments on randomly generated data traces of lengths 100 thousand, 500 thousand and 1 million steps. For the WAF specification, we generated appropriate traces with the number of possible invocations limited to 100, 250, 500 and 900 instances. The probability of seeing a bad request in input stream `ResponsePhrase` was set to 30%.
For the SDM specification the maximum number of possible invocations was merely set to 10, 50 and 100, since this specification is more costly as determined in Section 4.1.3. The values of the data stream `SensorData` were set to numbers from 0 to 100.
In our experiment, the data traces were processed by the interpreter and by the compiled C code monitors. Thereby information on total running time as well as its separation into system time and user time was gathered. Furthermore we observed the maximum resident set size and the number of minor page faults. All experiments were executed on a MacBook Pro 2016 with an 3.3 GHz Intel Core i7 Processor and a 16GB 2133 MHz LPDDR3 RAM.

4.3 Experimental Results

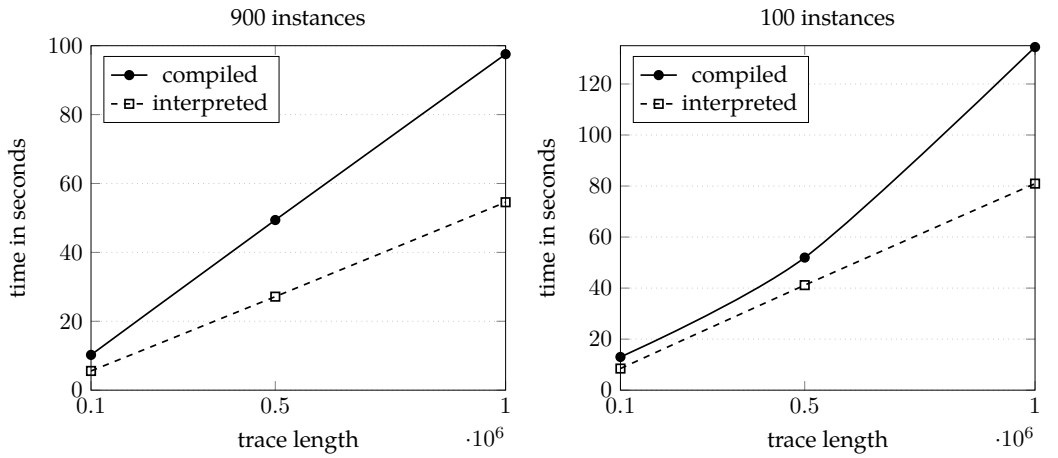
Our experiments show that the runtime increases nearly linear with the length of the input trace, which is illustrated in Fig. 4.3 and Fig. 4.4. The variance of some values, especially the C code runtime on 1 million steps in Fig. 4.4b, can be explained by worse external circumstances like lower CPU time and an extremely high number of involuntary context switches.
As presented in Fig. 4.3, at low numbers of instances the compiled code is faster than the interpreter by a factor of about 4.4 for the SDM specification and about 8 for the WAF specification. However, we have to consider that in general C code is faster than *Swift* by a factor of about 2.5. If the number of instances grows, we can see a dramatical increase of the C code runtime in Fig. 4.4, whereas the runtime of the interpreter rises only slightly. That is illustrated again in Fig. 4.5, where we plotted the runtime depending on the instance count for a fixed trace length of 1 million steps. The high variance of the runtime of WAF with 500 instances in Fig. 4.5a can be explained by an extremely high number of more than 90000 involuntary context switches during its execution, compared to about maximum 20000 for other executions of that trace length. Therefore, this value is not representative, since its spreading is caused by external circumstances.
Obviously, the speed of our compiled code highly depends on the amount of existing template instances. This can be explained by a very optimized template iteration in contrast to our implemented linked list of hash map entries. Additionally, the evaluation algorithm implemented in the interpreter is more efficient, since it directly evaluates a stream when its extension stream evaluates to *true*. Thereby a lot of iterations are stinted compared to our compiled code, which always iterates over the whole template for its evaluation. That means that the C code monitor performs an extremely higher amount of iterations during an evaluation step, which of course slows down its speed the more instances are invoked.
Comparing system time to user time in the SDM example as shown in Fig. 4.6, we can see that they almost equally increase more or less linear with the number of instances. The variance is again induced by external happenings. It behaves the same for the WAF example and is therefore not presented separately. We have to remark, that the system time of our compiled code is even for a large amount of instances much lower than the system time of the interpreter, as illustrated in Fig. 4.6a. This could be explained by a higher quantity of memory allocation calls due to the interpreter's expensive bookkeeping data structures. Nevertheless, it will exceed the system time of the interpreter somewhere along the way, which can be traced back to the missing optimization of our generated code.
Looking at the maximum resident set size (RSS) presented in Fig. 4.7, we recognize a difference by factor 5 for high instances in WAF and factor 9 for SDM, which stays relatively constant over the instance count in the latter case. The higher memory



(a) Web Application Fingerprinting (b) Sensor Data Monitoring

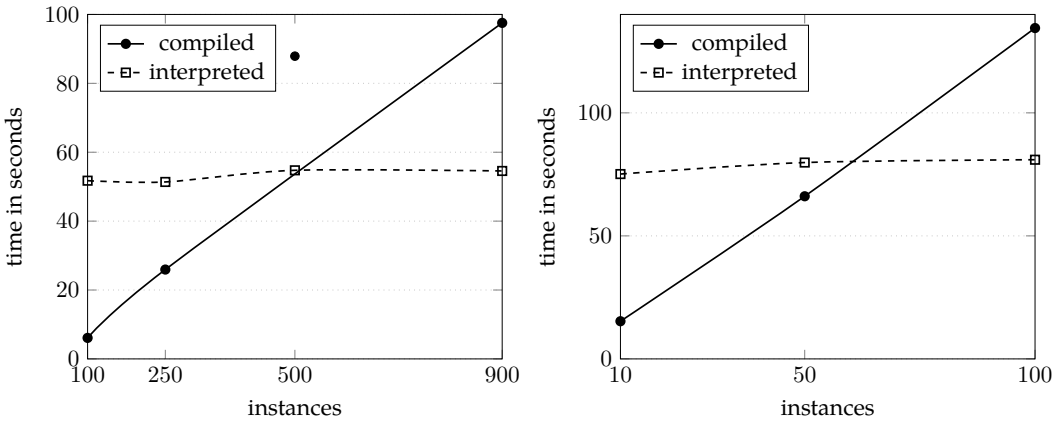
Figure 4.3: Small number of instances

usage of the interpreter can be explained by its big amount of bookkeeping structures to achieve higher performance, as well as the general overhead it brings along. Due to the higher amount of memory, the number of minor page faults is also higher for the interpreter and in general behaves according to the RSS. Here we have a factor of about 4.5 to 5.5 in the WAF and about 6.5 in the SDM specification.



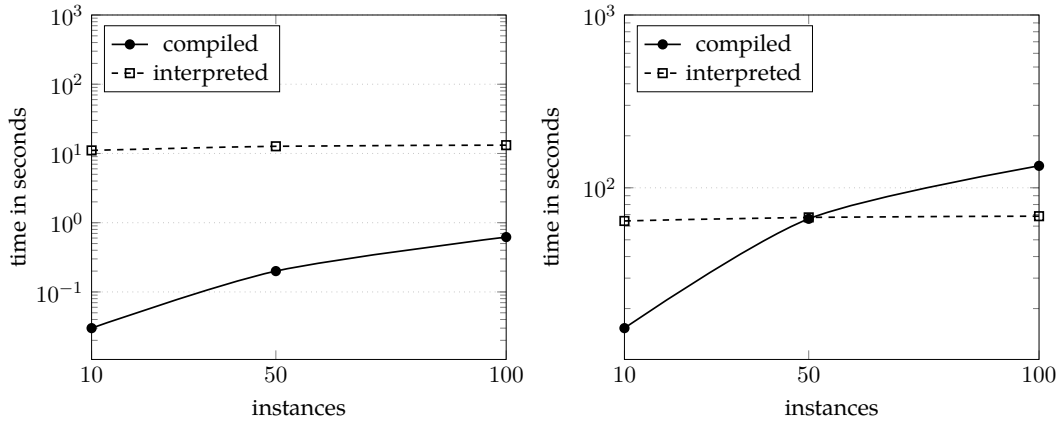
(a) Web Application Fingerprinting (b) Sensor Data Monitoring

Figure 4.4: High number of instances



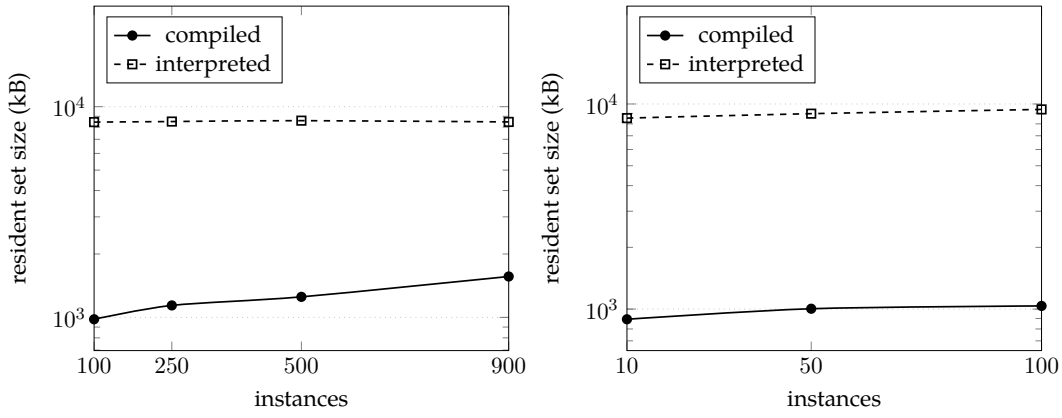
(a) Web Application Fingerprinting (b) Sensor Data Monitoring

Figure 4.5: Fixed trace length of 1 million steps



(a) system time (b) user time

Figure 4.6: Sensor Data Monitoring with 1 million steps



(a) Web Application Fingerprinting (b) Sensor Data Monitoring

Figure 4.7: Maximum resident set size with 1 million steps

Chapter 5

Future Work

Since we have implementation-specific limitations and do not use perfectly optimized data structures, there are several approaches to future work. In Section 3.2.3 we imposed two restrictions that would be of interest to remove. One restriction, which is relatively easy to solve, has been induced by generating a strict evaluation order for the templates. That precluded the evaluation of a new invoked instance during the same step, if the evaluation of the corresponding template has already been finished. A solution to this problem can be achieved by implementing a fixed-point algorithm for an evaluation step. That means, we iterate the evaluation order until there is no stream left to be evaluated.
The ban on positive offset references in *Lola 2.0* was a major restriction, which allowed for the consideration of only efficiently monitorable specifications. Dealing with positive offsets would necessitate the use of more flexible data structures, since the evaluation of a value could be delayed indefinitely, by a not yet evaluable extension stream.
Not only these possible extensions of the compiler, but also the removal of inefficiencies in the generated code will be part of future work. One optimization could be done by a better memory management. Instead of freeing no longer used memory, we could retain it and stint time-consuming reallocations. A more specific example is the data structure used for templates. Managing integer invoked streams would be faster using multi-level arrays, or even preallocated arrays of fixed size, if the range of possible invokes is given. Moreover, the iterator of the hash map could be optimized to speed up the evaluation process of the template. That could be accomplished by additionally storing all instances of the hash map in an array, and using it for iterations.
To decrease iterations in general, we could change the evaluation method by better using the known dependencies. Since an instance of a template is merely evaluated if its extension stream evaluates to true, we could directly link its evaluation in sequence to its extension stream. This would stint the iteration for evaluating the template.

Chapter 6

Conclusion

In this thesis we presented an approach for the compilation of *Lola 2.0* specifications to C code monitors as well as the implementation of the compiler. After describing the abstract syntax tree of a *Lola 2.0* specification, we introduced the dependency graph (DG) as intermediate representation and briefly explained its implementation. We restricted our considerations on efficiently monitorable specifications and proposed their definition by means of the DG. Since most applications do not require positive offsets, the restriction is feasible and greatly simplifies the compilation approach. Thus, we were able to present an algorithm to generate a strict evaluation order which enabled a straightforward implementation of the monitor. We gave an example of the general `main.c` file setup and explained further generated data structures, representing the templates and stream instances of a specification. Since we implemented the compiler, we could compare its performance to the interpreter. Therefore we presented two example specifications with slightly different characteristics and employed them for our experiments. The interpreter and the C code monitors were fed with the same randomly generated input data of three different lengths and a variable amount of possible invocations. The results showed that for fewer instance counts, the generated monitor is able to beat the interpreter. Unfortunately its runtime rapidly grows with the amount of instances, since the data structures of the monitor are rather simple and easy to understand than optimized for performance. Looking at the maximum resident set size, we saw that for our examples the interpreter requires up to 9 times more memory, which is a nice result for our C code monitors.
All in all, the experiments provided better results than expected, since it was not our primary goal to implement a perfect evaluation algorithm and to generate optimized data structures for achieving high-performance C code monitors. We take it as a success, that we could beat the interpreter at least for the lower number of instances. Moreover, we are actually able to generate fairly clear and understandable C code from a given *Lola 2.0* specification, which was our main goal for this thesis. In comparison to the interpreter, this represents a great advantage in terms of higher flexibility and increased maintainability, and greatly promotes the use of *Lola 2.0* for industrial deployment.
As we have seen, our compiler and the generated monitor code are not perfect, but leave room for interesting future work to beat the interpreter in all respects.

Bibliography

[1] Swift programming language. https://swift.org/. Accessed: 2019-01-30.

[2] Florian-Michael Adolf, Peter Faymonville, Bernd Finkbeiner, Sebastian Schirmer, and Christoph Torens. Stream runtime monitoring on uas. In Shuvendu Lahiri and Giles Reger, editors, *Runtime Verification*, pages 33–49, Cham, 2017. Springer International Publishing. ISBN 978-3-319-67531-2.

[3] Ben D'Angelo, Sriram Sankaranarayanan, César Sánchez, Will Robinson, Bernd Finkbeiner, Henny B. Sipma, Sandeep Mehrotra, and Zohar Manna. Lola: Runtime monitoring of synchronous systems. In *12^{th} International Symposium on Temporal Representation and Reasoning (TIME'05)*, pages 166–174. IEEE Computer Society Press, June 2005.

[4] Alexandre Duret-Lutz, Alexandre Lewkowicz, Amaury Fauchille, Thibaud Michaud, Etienne Renault, and Laurent Xu. Spot 2.0 — a framework for LTL and ω-automata manipulation. In *Proceedings of the 14th International Symposium on Automated Technology for Verification and Analysis (ATVA'16)*, volume 9938 of *Lecture Notes in Computer Science*, pages 122–129. Springer, October 2016. doi: 10.1007/978-3-319-46520-3_8.

[5] Peter Faymonville, Bernd Finkbeiner, Sebastian Schirmer, and Hazem Torfah. A stream-based specification language for network monitoring. In Yliès Falcone and César Sánchez, editors, *Runtime Verification - 16th International Conference, RV 2016, Madrid, Spain, September 23-30, 2016, Proceedings*, volume 10012 of *Lecture Notes in Computer Science*, pages 152–168. Springer, 2016. doi: 10.1007/978-3-319-46982-9_10. URL http://dx.doi.org/10.1007/978-3-319-46982-9_10.

[6] Carsten Fritz. Constructing büchi automata from linear temporal logic using simulation relations for alternating büchi automata. In Oscar H. Ibarra and Zhe Dang, editors, *Implementation and Application of Automata*, pages 35–48, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. ISBN 978-3-540-45089-4.

[7] Paul Gastin and Denis Oddoux. Fast ltl to büchi automata translation. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *Computer Aided Verification*, pages 53–65, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg. ISBN 978-3-540-44585-2.

[8] Sylvain Hallé. When rv meets cep. In Yliès Falcone and César Sánchez, editors, *Runtime Verification*, pages 68–91, Cham, 2016. Springer International Publishing. ISBN 978-3-319-46982-9.

[9] Mark Timon Hüneberg. Optimizing lola specifications, 2015.

[10] D. Johnson. Finding all the elementary circuits of a directed graph. *SIAM Journal on Computing*, 4(1):77–84, 1975. doi: 10.1137/0204007. URL https://doi.org/10.1137/0204007.

[11] Martin Leucker, César Sánchez, Torben Scheffel, Malte Schmitz, and Alexander Schramm. Tessla: Runtime verification of non-synchronized real-time streams. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, SAC '18, pages 1925–1933, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5191-1. doi: 10.1145/3167132.3167338. URL http://doi.acm.org/10.1145/3167132.3167338.

[12] Jeremy Morse, Lucas Cordeiro, Denis Nicole, and Bernd Fischer. Model checking ltl properties over ansi-c programs with bounded traces. *Software & Systems Modeling*, 14(1):65–81, Feb 2015. ISSN 1619-1374. doi: 10.1007/s10270-013-0366-0. URL https://doi.org/10.1007/s10270-013-0366-0.

[13] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, SFCS '77, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society. doi: 10.1109/SFCS.1977.32. URL https://doi.org/10.1109/SFCS.1977.32.

[14] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972. doi: 10.1137/0201010. URL https://doi.org/10.1137/0201010.

[15] Christoph Torens, Florian Adolf, Peter Faymonville, and Sebastian Schirmer. Towards intelligent system health management using runtime monitoring. In *AIAA Information Systems-AIAA Infotech @ Aerospace*. American Institute of Aeronautics and Astronautics (AIAA), jan 2017. doi: 10.2514/6.2017-0419. URL https://doi.org/10.2514%2F6.2017-0419.

[16] Nathalie Zeller. Comparing lola 2.0 with quantitative regular expressions, 2017.