Saarland University

Faculty of Mathematics and Computer Science

Bachelor's Thesis

Runtime Verification of Critical Web-Based Systems with Lola

Author
Marvin Hofmann

Supervisor
Prof. Bernd Finkbeiner, Ph. D.

Advisor
Maximilian Schwenger

Reviewers
Prof. Bernd Finkbeiner, Ph. D.
Prof. Dr. Jens Dittrich

Submitted: 03rd October 2018

Eidesstattliche Erklärung

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

Statement in Lieu of an Oath

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, 03rd October, 2018

Abstract

A bug in the code base of a given program can expose the system and can cause failures. These may impact not only the availability of the system but also the integrity of the data. Bugs are hard to predict and can cause millions of dollars in damage. Therefore, preventing these bugs is an important objective, principally, but not only, considering critical systems like financial applications.

In web-based systems, automated testing is a common approach to detect bugs before they are deployed into a production environment, preventing the bugs from causing harm to the system. However, testing can only verify code paths explicitly chosen to be tested, and actual usage can differ. This pushes the responsibility to the developers to decide what code paths are critical and require a test. At the same time, tests can also contain bugs and impose a maintenance overhead.

Therefore, a solution is desirable, that can help to protect from damages caused by bugs. This solution needs to be easy to use while imposing as little as possible additional overhead onto the developers. At the same time, it should cover all traversed code paths and be human readable in what is checked. Moreover, it should not itself be a source of bugs.

We introduce Ruby-Lola, a framework for synchronous runtime verification of web-based systems. Ruby-Lola is based on the Lola stream-based specification language, a simple and expressive language that can describe correctness/failure states of a system. The framework provides a domain-specific language to write specifications for runtime verification in an object-oriented way, allowing for references between database tables and a clean syntax that is familiar to developers. Moreover, Ruby-Lola can be used in combination with automated tests, improving their coverage through the specification.

We implemented the framework and applied it to a case study of a web-based system that models an auction house, a practical and critical application. This demonstrates that our framework is sufficient to express complex specifications and that it can detect and mitigate critical bugs in practical applications. We also show that the framework is easy to use and serves as additional documentation. Ruby-Lola imposes minimal overhead related to other costs of processing user requests and is therefore not noticeable by end users in practical usage.

Acknowledgements

I would like to thank Prof. Finkbeiner for his support and trust in my quite practical topic, for the guidance and important previous work, making this thesis possible.

I also want to thank Prof. Dittrich for taking the time and making himself available as second reviewer.

Special thanks goes to my supervisor Maximilian Schwenger for the outstanding support, guidance and patience throughout my thesis.

Lastly I want to thank my proofreaders Marc Schulder, Ray Neiheiser, Constantin Berhard and David Haller for their valuable feedback.

Contents

Chapter 1

Introduction

Development of critical software requires verification to ensure that no dangerous bugs are present. There are multiple methods employed to verify software. Following is an evaluation of their viability in web-based software projects.

In practice, testing is the most frequently used verification method. To test a software project, the developers write test cases that each execute a specific part of the software and compare the result with what is expected from a correct software. It would be computationally expensive or even impossible to test every combination of possible inputs to the software, so tests are generally written for *critical* and *bug prone* parts of the software. The developer writing test cases is responsible for deciding what parts are *sufficiently critical*. This creates the risk of forgetting edge cases or unexpected bugs. Moreover, since tests are code themselves, they can also contain bugs. This can result in false positives, where bugs are detected that are not bugs. Also possible are false negatives where a bug that should have been detected is not detected because the test itself has a bug. Software is often continuously changing and tests have to be adapted to those changes of the code. The amount of work to change a piece of code can easily be doubled when the developer then has to change all the tests and make sure that no new cases were introduced that need new tests. This overhead makes testing a liability in fast-changing environments like the web [15, 2, 22, 33].

Formal verification is the method of proving that a program is correct. Deciding if a program is correct is not easy. All possible inputs and states have to be checked and verified. The more complex a program is, the more resources and time is needed to be used to verify it. State of the art verification software employs many techniques to reduce the amount of work required to verify software. Tools like *Astree* [12], *Cobra* [23] or *Saturn* [38] can verify complex C programs in a matter of hours. However, the techniques employed to achieve this do not work as well for dynamic languages like Ruby, where there are almost no assumptions the tool can rely on, increasing the computation required. Additionally, waiting for hours to get

results can hinder development and become a similar liability as the overhead of testing [15, 2, 22].

A middle ground would be runtime verification. This technique verifies that a program is correct *so far* and makes no claims on overall correctness. Verifying that the current state of a program is correct is computationally inexpensive compared to checking all cases, which makes runtime verification attractive regarding the overhead imposed. It also solves some of the duplications of testing, as there is one specification detailing correct behavior and not many different tests. While the system still does many checks, the developer has one single source of truth for how a program should behave. It can even be combined with testing in that the test is only a scenario and the runtime verification system does the verification of correctness.

There is some effort to employ runtime verification for web-based systems [28, 19, 6, 24, 4] but there are problems that prevent general adoption. Current runtime verification systems take work to integrate into existing projects, as explored in Chapter 6. With unit tests developers are used to things *just working automatically* and having to write code to connect a runtime verification system to their existing software is hindering adoption. Providing a system that integrates itself automatically on installation could help reduce this adoption hurdle.

The goal of this thesis is to ease some of the pain points in integrating runtime verification into web-based systems. To accomplish this, we developed a new tool, Ruby-Lola, a runtime verification system that is easy to use and integrate into existing Ruby on Rails projects. It is based upon Lola 2.0, a stream-based runtime verification language with interesting properties [14, 18]. Accompanying this are case studies of using the tool in real-world use. The tool features a domain specific language (DSL) to write human readable specifications and facilitates automated data collection and instrumentation. It works out of the box for existing systems with a simple installation of the Ruby gem, no setup required.

Ruby-Lola aims to create a bridge between academic runtime verification and industry built applications. This allows for greater adoption of runtime verification and potentially less buggy software. Startup companies developing web-based systems frequently use Ruby on Rails, so providing a tool for that framework can have an impact on software development.

1.1 Motivating Example

There are many legal pitfalls for new companies. One important legal construct are the terms of service that a user has to accept before they use the application.

One envisions a startup developing a website, where customers can log into their accounts. The startup uses Ruby on Rails and has a User model for each user. When a user creates their account, they need to accept the Terms of Service, which

is stored in the model. The model also stores if an account is enabled, which means it fulfills all criteria to use the service.

The code for the User class could look like Figure 1.1

```
class User
  attr_accessor :tos_accepted, :account_enabled
end
```

Figure 1.1: An example User class that has fields for storing Terms of Service acceptance and if the user account is enabled. The keyword *attr_accessor* is used as a shorthand in Ruby to create getters and setters for a parameter, essentially defining it.

Said startup now risks its users finding a bug that allows them to use the service without first accepting the Terms of Service. This could cause expensive legal fees that need to be avoided. So the legal department writes a specification, stating that no enabled account should exist that has not accepted the Terms of Service.

Multiple developers are working on the service, and there are multiple code paths to account creation, so it could happen that the check for Terms of Service was forgotten in one code path. So how to make sure that future code paths also comply with the specification? Where should one document that it exists? What if the specification changes? All of this is a liability that needs an easy solution.

One solution to this problem is Ruby-Lola. The developers install the library once and start writing a formal specification from the verbal one given by the legal department. If an account is enabled but has not accepted the Terms of Service, it is violating the specification.

With Ruby-Lola installed, every time a part of the code tries to enable a user, the specification checks if all legal criteria are met before allowing the change. Also, the check is easily readable and updating the requirements is done in one place only. Also, should the startup need users to accept their privacy policy in the future, that check can be added with ease.

The aforementioned specification can be formulated as the following Figure 1.2

Instrumenting the system under test and performing the runtime verification are tasks that Ruby-Lola performs automatically, the developer only has to declare a specification.

```
class User
  attr_accessor :tos_accepted, :account_enabled
  lola_specification do
    define :legally_not_enabled, :boolean do
      :account_enabled.and(not(:tos_accepted))
    end
    trigger :legally_not_enabled,
      "TOS acceptance missing!"
  end
end
```

Figure 1.2: The User class from Figure 1.1 extended by a Lola specification. This specification defines a stream of boolean type that computes if an account is enabled but has not accepted the terms of service. The specification adds a trigger to that stream which rejects all changes that make the stream output true, essentially preventing accounts from being enabled without an accepted TOS. The trigger has an error message that a user sees when the specification was violated.

Chapter 2

Background

2.1 Introduction to Runtime Verification

It is almost unavoidable to produce a complex software without bugs [30] In order to prevent their existence or at least reduce it, different techniques are used. A widespread approach to avoid this is using testing and verification.

Software testing is the process of executing a program in specified scenarios and asserting that it behaves correctly. Testing covers a wide field of diverse, often ad hoc, and incomplete methods for showing correctness, or, more precisely, for finding bugs. Testing cannot prove the absence of bugs. Making sure that all execution paths through some code are covered, and all edge cases are accounted for is time-consuming and error prone [30, 25, 37, 28, 33].

Static verification can prove that a software system conforms to a specification. There are techniques like theorem proving [8] and model checking [10]. If the specification is without problems and the program passes the specification, one can derive that the program is without problems too. However, static verification of code is a very complex and computationally expensive endeavor. To prove that some code abides by a specification, every possible execution trace through that code has to be verified [17]. This is possible for simple programs but gets impractical in real-world use very quickly as the space of possible states to check explodes. One would have to code very rigorously and avoid many language features and dynamic languages completely [27, 32, 28].

Runtime verification takes the concept of a specification that a software should abide by. It goes around the exploding state space by letting the program run and check its state while it runs [14]. That way it is no longer possible to assert that the program is bug-free in general, but it can be said that no execution of the program so far has encountered a bug. Also, if a bug is found, it can be reacted to fairly quickly [28].

2.2 Introduction to Synchronous Runtime Verification

There are different ways of how one can monitor a system. Reading program traces independent of the program itself is called completely asynchronous runtime verification. This is often achieved via log files generated by the program. Asynchronous runtime verification of a program is beneficial when there is no time sensitivity involved. If the log files are big and the verification resource intensive, not interrupting the program can be a valuable property.

However, in many cases, time to detection plays an important role in runtime verification. Verification of a system while the system is running, stopping the system execution to do the verification is called completely synchronous. This method leaves no delay between a specification violation happening and the detection of it. One interesting property of completely synchronous runtime verification systems is that if a violation is detected, it is not yet too late to mitigate it. Some such runtime verification systems even allow the system under test to react to the violation, potentially preventing it from even happening in the first place. This property becomes invaluable in critical systems like blockchain businesses, where one mistake can mean the loss of funds [7].

2.3 Introduction to Lola

Lola is a stream-based specification language for the online and offline monitoring of synchronous systems. A Lola specification describes the computation of output streams from a given set of input streams. An in-depth introduction can be found in the original papers [14, 18].

A *Lola specification* is a system of equations of stream expressions over typed *stream variables* of the following form:

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

Each stream expression $e_i(t_1, \ldots, t_m, s_1, \ldots, s_n)$ for $1 < i < n$ is defined over a set of *independent* stream variables $t_1, \ldots, t_n$ and *dependent* stream variables $s_1, \ldots, s_n$.

Independent stream variables refer to input stream values, and dependent stream variables refer to output stream values computed over the values of all streams. All stream variables are typed: the type of an independent stream variable t_i is T_i , the type of an dependent stream variable s_i is T_{m+i}.

A stream expression $e(t_1, \ldots, t_m, s_1, \ldots, s_n)$ is recursively defined as follows:

- A constant c of type T is a stream expression of type T.
- An independent stream variable t of type T is a stream expression of type T.
- A dependent stream variable s of type T is a stream expression of type T.
- Let $f : T_1 \times T_2 \times \cdots \times T_k \mapsto T$ be a k-ary operator. If for $1 \leqslant i \leqslant k$, 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 $ite(b, e_1, e_2)$ is a stream expression of type T; note that ite abbreviates if-then-else.
- If e is a stream expression of type T , c is a constant of type T, and i is an integer, then $e[i, c]$ is a stream expression of type T. Informally, $e[i, c]$ refers to the value of the expression e offset i positions from the current position. The constant c indicates the default value to be provided, in case an offset of i takes us past the end or before the beginning of the stream.

The evaluation model of Lola is recursively defined with j being the evaluation time, and $val(e)(j)$ the evaluation function applied on the stream expression e at the evaluation time step j as follows:

- A constant c evaluates to itself all the time:

$$val(c)(j) = c$$

- An independent stream variable t evaluates to the value of its stream at the point of evaluation:

$$val(t)(j) = t(j)$$

- A dependent stream variable s evaluates to the value of its stream expression at the point of evaluation:

$$val(s)(j) = s(j)$$

- A k-ary operator $f(e_1, \ldots, e_k)$ application evaluates to all parameters evaluated and applied to the operator:

$$val(f(e_1, \ldots, e_k)(j) = f(val(e_1)(j), \ldots, val(e_k)(j))$$

- An if-then-else operator $ite(b, e_1, e_2)$ is evaluated as follows:

$$val(ite(b, e_1, e_2))(j) = \text{ if } val(b)(j) \text{ then } val(e_1)(j) \text{ else } val(e_2)(j)$$

- A look-back-operator $e[i, c]$ with N being the end of the stream e is evaluated as follows:

$$val(e[i, c])(j) = \text{ if } 0 \leqslant j + i \leqslant N \text{ then } val(e)(j + i) \text{ else } c$$

In addition to the stream equations, Lola specifications often contain a list of triggers

$$\text{trigger } \phi_1, \phi_2, \ldots, \phi_k$$

where $\phi_1, \phi_2, \ldots, \phi_k$ are expressions of type boolean over the stream variables. Triggers generate notifications when their value becomes true.

```
input bool loginSuccess
output int attempts :=
    ite(loginSuccess, 0, attempts[1,0] + 1)
output bool bruteforce := attempts > 3
trigger bruteforce
```

Figure 2.1: This Lola specification is tracking the login attempts of a user. If the login attempt was not successful then the attempts counter gets increased. And if the counter tracks more than 3 failed attempts, a bruteforce warning is triggered.

2.4 Introduction to Lola 2.0 Streams

Lola 2.0 is an extension to the Lola specification language, adding new features that allow for the precise description of complex security properties in network traffic. This makes Lola 2.0 a good choice for runtime verification on the web. An In-depth introduction can be found in the original paper [18].

Lola 2.0 extends Lola with stream equation templates of the following form:

$$\begin{aligned} \text{output} T_s < hp_1 : T_1, \ldots, p_l : T_l >: inv : s_{inv}; ext : s_{ext}; ter : s_{ter} := \\ e(t_1, \ldots, t_m, s_1, \ldots, s_n, p_1, \ldots, p_l) \end{aligned}$$

Each such stream equation template introduces a template variable s of type T that depends on parameters $p_1, \ldots, p_l$ of types $T_{p_1}, \ldots, T_{p_l}$, respectively. For given values $v_1, \ldots, v_l$ of matching types $T_{p_1}, \ldots, T_{p_l}$ we call

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

an instance of s. The template variables s_{inv}, s_{ext}, and s_{ter} indicate the following auxiliary streams:

- s_{inv} is the invocation template stream variable of s and has type $T_{p_1} \times \ldots \times T_{p_l}$. If some instance of s_{inv} has value $v_1, \ldots, v_l$, then an instance $s < v_1, \ldots, v_l >$ of s is invoked.

- s_{ext} is the extension template stream variable of s and has type bool and parameter of type $T_{p_1} \times \ldots \times T_{p_l}$. If s is invoked with parameter values $\alpha = (v_1, \ldots, v_l)$, then an extension stream s^{α}_{ext} is invoked with the same parameter values. If s^{α}_{ext} is true, then the value of the output stream $s < v_1, \ldots, v_l >$ is computed at the position.

- s_{ter} is the termination template stream variable of s and has type bool and parameters of type $T_{p_1} \times \ldots \times T_{p_l}$. If s is invoked with parameter values $\alpha = (v_1, \ldots, v_l)$, then a terminate stream s^{α}_{ter} is invoked with the same parameter values. If s^{α}_{ter} is true, then the output stream $s < v_1, \ldots, v_l >$ is terminated and not extended until it is invoked again.

2.4.1 Example of Lola 2.0 Streams

Lola 2.0 extends the specification language with template streams. These streams carry input parameters and can, for example, be used to track login attempts of different users in their own streams, all following a general pattern.

These new streams are dynamic, meaning they each run on their own time scale. For this, each template has three new stream expressions, invocation, extension, and termination.

The invocation equation controls when a new stream instance is created. When the invocation equation holds true, a new stream instance gets created, and the parameter for it computed. There can only be one instance with that parameter, so repeated invocations are ignored.

Example: Failed logins. We only want to track users who are failing to login. So our invocation is *failedToLogin: true* and our parameter is the user id. So say user 10 failed to login, then a stream with the parameter 10 gets created. So if he failed to log in again, the stream already exists, so it does not have to be recreated.

The extension equation controls when a stream should compute a new value. Every time this equation is true for a stream that is already created, the stream equation will be evaluated and the stream extended by the resulting value. This means that not all streams have to have the same length or speed. Only streams that were previously invoked can be extended.

Example: Since we want to see how often our users fail to login, we set our extension equation to failedToLogin: true and our stream equation to count the

number of times failedToLogin is true with a look back that defaults to 0. We can then set a trigger to alert us when the amount of failed login attempts are too high.

The termination equation controls when a stream is not needed any more and should be terminated. If a stream has been created and then the termination equations holds true, it will be terminated and completely deleted. Only previously invoked streams can be terminated.

Example: Once a user has a successful login, we want to clear our counter. To save space, we delete the whole thing. So our termination equation is set to failedToLogin: false and will immediately discard failed attempts once one attempt is successful. Our user 10 finally got his password right, and his stream gets deleted immediately. He now has his attempts again.

Figure 2.2 and 2.3 show an example comparison between a Lola 2.0 specification and a Ruby-Lola specification.

```
input bool loginSuccess
input String uid
output bool useraction<u> := (uid=u)
output int attempts<user> :
    inv: uid;
    ext: useraction
    := ite(loginSuccess, 0, attempts(user)[1,0] + 1)
output bool bruteforce<user> :
    inv: uid;
    ext: useraction
    := attempts(user) > 300
trigger any(bruteforce)
```

Figure 2.2: This Lola 2.0 specification is tracking the login attempts of all users. Each user has their own attempts counter. If the login attempt was not successful then the attempts counter gets increased. And if the counter tracks more than 300 failed attempts, a bruteforce warning is triggered.

```
class User
  define_specification do
    define :attempts, :numeric do
      ite(loginSuccess, 0, look_back(:attempts, 1, 0) + 1)
    end
    define :brute_force, :boolean do
      attempts > 300
    end
    trigger :brute_force, "too many login attempts!"
  end
end
```

Figure 2.3: This Ruby-Lola specification is tracking the login attempts of all users. It works the same as Figure 2.2 but the domain-specific language allows for committing of some boilerplate. Since everything happens inside the User class, there is no need to specify it. Also, fields of the class are implicitly defined and can be used with no extra code. One can see how this is more readable.

Chapter 3

Contribution

3.1 Lola Model Streams, Field Streams, and how to Link Them Together

A model is a representation of a table in the database system. One example of a model would be a User. Each model has a set of typed fields, for example, the string name and the integer age. Each instance of a model has all the fields attached to it as properties, each holding a value of that type. These properties can change over time, and not all values are valid. This aligns with Lola 2.0 streams.

To monitor different models in our runtime verification system, we create a template stream for each model, called a model stream. We also create template streams for each field the models have. These field streams are typed by the type of the corresponding field. Both the model stream and all corresponding field streams have the model identifier as a parameter. This groups them into one logical unit that we will later use to extend the model stream with syntactic sugar to make it behave in object-oriented ways.

The model stream and the field stream share their invocation, extension and termination equations. They are invoked when a new instance of the model is created in the database. They are extended when that instance is changed and saved. Also, they are terminated if that instance is deleted. They track the life cycle of the instance matching their identifier parameter and can verify that said instance behaves to the specification given.

Each model can have a Lola specification attached to it. This specification is prepended with the model and field streams corresponding to the model and its fields. Also, every stream expression is extended by the identification parameter and the invocation, extension, and termination equations, converting it into a Lola 2.0 specification.

The choice of giving specification writers only the standard Lola capabilities

was made for reasons of simplicity. It is easier to reason about less complex specification languages and reduces the barrier of entry into runtime verification with Lola while conserving the needed parts of Lola 2.0 for the implementation doing verification in the background. This way it is also possible to introduce syntactic sugar and abstractions that further simplify writing the specification. Specification writers do not have to think much about Lola and can reason in object-oriented ways, matching the thinking that is needed to program the models themselves. This reduces cognitive load and could help with adoption.

3.2 The Rails Way of Convention Over Configuration

Ruby on Rails is a framework for developing websites in Ruby. It follows the mantra of convention over configuration. If the default path is followed, everything automatically works. This reduces mundane, repetitive work and increases developer productivity in many general cases. Ruby-Lola does the same for instrumenting the application. Rails uses models as a way to communicate with the database, it is an abstraction layer over a database table and adds useful functionality to it. This means that database calls are in one central place we can tap into. Moreover, as long as the user uses models, we can do most of the work setting Ruby-Lola up for him.

The way model streams and field streams are designed, they map Rails models directly with no translation required. The specification is written in Lola and can reference all model fields without defining them beforehand. Ruby-Lola translates these specifications as previously specified, adding model fields as input streams in a conventional way. The result is a system that *just works* where the heavy lifting is done in the background.

Ruby-Lola also allows extra information to be added to triggers. Specification writers can add an error message to each trigger that will in most cases be directly passed on to the user who triggered the change. It is also possible to have the system notify developers of the violation, giving them critical information over what exactly happened. These notifications make it easier to fix the problem that caused a change to violate the specification. Optionally one might want to try to rescue the operation and can do so with a callback added to a trigger. That callback can try to fix the problems that made the model change violate the specification. Should a specification be violated in a unit test, Ruby-Lola can fail the test and give out debugging information, saving developers valuable time in repeatedly checking for common conditions in each test anew.

3.3 ActiveRecord Callbacks

Ruby on Rails uses ActiveRecord as a database abstraction layer. ActiveRecord provides callbacks for database actions that we can use to detect changes. Ruby-Lola uses an *around_update* callback to first check if the specification holds, then submit the changes to the database and when the database accepts it, the new now persistent values are added to the streams. This way it is ensured that there was no problem saving data to the database, keeping consistency. It also allows other callbacks to run before Ruby-Lola, allowing developers to run simple sanitization checks before the verification.

Using ActiveRecord callbacks has the benefit of being sure that verification always happens when the database is about to be changed, no matter what code calls it. There are ways to disable or skip these verifications in ActiveRecord, so the developers still have full control.

3.4 Rails Integration of Ruby-Lola

When the Rails application is starting up, the model is loaded to setup the database table structure. After that, the specification gets loaded. This happens implicitly as part of Ruby-Lolas domain specific language. Since the database table structure is already loaded, Ruby-Lola collects all database fields and makes them available for use by the specification. With type information available in the database schema and provided through Rails, Ruby-Lola can perform a type check on the specification. Additional checks are done to ensure that the specification is well defined. All these checks will fail at application startup if the specification does not fit the underlying structure of the model. This is the runtime equivalent of a compiler step to check for correct types and references and ensures that the specification is always valid and working.

This is an important step, as in a dynamic, interpreted language it would be possible to reference undefined streams, crashing the application when the first model instance is created. The startup consistency checks ensure that if the Rails application can start, the Ruby-Lola specifications are all well defined and will not crash at runtime. Having a specification crash the application would produce the same problem runtime verification aims to protect against and would defeat the benefits gained from using a runtime verification system.

3.5 Ruby-Lola Stream Implementation

The streams of Ruby-Lola are implemented as a ring buffer of fixed size. However, with the dynamic nature of Ruby arrays, they are auto-extending, so even if the size

is capped, the arrays start smaller. It would be possible to have unbound arrays, but since, with fixed look back values, we know how many values we maximally need, we can discard excess valued from the array to save memory.

Since we are using arrays internally, lookups are very fast. This is important to ensure that application performance does not suffer with the use of runtime verification.

Updates to streams are committed atomically, so it is not possible to see intermediate results. The streams are always *read consistent*. This is an essential property as otherwise, a trigger could leave streams in an intermediary state that could cause problems. The intermediary state can be discarded if it is not needed.

Streams are also very modular, so they allow for different back-end solutions like a Redis database to be added. This is important for future work and scalability, as external storage solutions could allow for multiple servers and concurrent request support, which Ruby-Lola does not currently support.

3.6 Ruby-Lola Domain Specific Language (DSL)

To write a specification, developers can use a domain specific language provided by Ruby-Lola. The goal of this language is to allow specifications to be human readable, easily understandable and simple to modify.

A specification is appended as shown in Figure 3.1 with the *define_specification* function that takes a Ruby block containing specification definitions as its argument. An empty specification like this one will create a model stream for *Example* and field streams for all fields of the *Example* model implicitly. Since Ruby on Rails already knows all fields of the model, they do not need to be specified.

```
class Example
  define_specification do
  end
end
```

Figure 3.1: A class called *Example* with an empty specification.

Defining additional streams is done inside the specification with the *define* function, as shown in Figure 3.2. This function takes the name of the newly created stream variable, the type it should have and a Ruby block containing the stream expression associated with it. Types of stream variables have to be explicitly stated for clarity and safety reasons. This way the type checker can protect the developer from type errors while writing the specification. Defining a constant stream expression is done by passing the corresponding Ruby constant.

```
class Example
  define_specification do
    define :constant, :numeric do
      0
    end
  end
end
```

Figure 3.2: A class called *Example* with a specification that defines a constant stream variable.

Using existing stream variables can be done with a Ruby symbol named after the referenced stream variable, as shown in Figure 3.3. The symbol is matched against the list of stream variables, and if found, the stream variable is used at the current time step. Stream variables can be operated on as per Lola specification.

```
class Example
  define_specification do
    define :output, :numeric do
      :input + 1
    end
  end
end
```

Figure 3.3: A class called *Example* with a specification that defines an output stream variable from an input stream variable.

Looking up previous values of stream variables can be done with the *look_up* function , as shown in Figure 3.4. It takes a stream variable, an amount of steps to look back and a default value and evaluates them as per Lola specification.

Conditional statements can be used with the *ite* function, as shown in Figure 3.5. It takes a stream expression of boolean type and two stream expressions of the same type and applies a conditional as per Lola specification.

Specification violations can be specified with a trigger, as shown in Figure 3.6. The *trigger* function takes a boolean stream variable and optionally a string with an error message. A specification violation is raised if the stream variable evaluates to true. Specification violations can have different effects depending on in what context they happen. In a unit test scenario, the test case will be failed. In production environments, the database change will be rejected, and error messages will be passed on to the user requesting the change. Notifications can be attached to

```
class Example
  define_specification do
    define :output, :numeric do
      :input + look_up(:output, 1, 1)
    end
    define :fib, :numeric do
      look_up(:fib, 2, 1) + look_up(:fib, 1, 1)
    end
  end
end
```

Figure 3.4: A class called *Example* with a specification that defines an output stream variable from an input stream variable and a previous value of the output. For this the *look_up* function is used. The *look_up* function can be used to construct the Fibonacci sequence.

```
class User
  define_specification do
    define :faulty_logins, :numeric do
      ite(:login_success, 0,
          1 + look_up(:faulty_logins, 1, 0))
    end
  end
end
```

Figure 3.5: A class called *User* with a specification that counts how many consecutive faulty logins a user has. If a faulty login is detected then the counter is incremented, otherwise it is reset.

triggers, alerting the development team of potential problems.

```
class User
  define_specification do
    define :faulty_logins, :numeric do
      ite(:login_success, 0,
          1 + look_up(:faulty_logins, 1, 0))
    end
    define :too_many_logins, :boolean do
      :faulty_logins > 300
    end
    trigger :too_many_logins, 'Error: too many logins!'
  end
end
```

Figure 3.6: A class called *User* with a specification that counts how many consecutive faulty logins a user has. If a faulty login is detected then the counter is incremented, otherwise it is reset. If the amount of consecutive faulty logins exceeds three a trigger is raised.

Ruby-Lola provides syntactic sugar to make writing specifications fast and understanding them later easy. Each stream expression can be applied to a function via the dot-notation. This works in line with how Ruby objects evaluate function calls on them. The dot-notation *:name.length* is expanded to *length*(*:name*) and then evaluated, as shown in Figure 3.7. Defining such functions for different types could be done, but Ruby-Lola already brings many different functions and makes native Ruby functions available for standard types which should be enough for most cases.

```
class User
  define_specification do
    define :name_too_long, :boolean do
      :name.length > 30
    end
    trigger :name_too_long, 'Error: Username too long!'
  end
end
```

Figure 3.7: A class called *User* with a specification that checks the length of the users name, ensuring that it does not exceed 30 characters with a trigger. This makes use of object oriented syntactic sugar, applying a function on a stream variable.

Ruby on Rails allows database references as a data type, which Ruby-Lola exposes to the specification as shown in Figure 3.8. A *references* type is introduced that can be interacted with. There are different functions available for references. One example is the *count* function that returns a numeric with the amount of references counted. A different type of function is the *sum* function. It does a sum over one field stream of all referenced model streams. If the dot-notation is used for a field stream of the referenced model, execution is chained, making different functions available. This allows for rich specifications and complicated database designs.

```
class User
  has_many :items
  define_specification do
    define :amount_of_items, :numeric do
      :items.count
    end
    define :total_value, :numeric do
      :items.price.sum
    end
  end
end
```

Figure 3.8: A class called *User* with a specification that expands a has-many relation. One *User* can have many *Items*. This specification counts the number of items associated with this user and calculates their total value.

3.6.1 Technical Details of the DSL Implementation

The domain-specific language makes heavy use of operator overloading to improve readability over conventional methods.

Example, comparing the code for:

```
:age + 2
```

with:

```
add(stream_of(:age), constant_stream(2))
```

So one can see how this improves readability. The first version has almost no noise to it.

This works by monkey patching[1] the symbol and numeric type to add extra

[1]Monkey patching is changing (often internal) classes after they are defined. In this case we define a + operator on the Symbol type, which does not exist. Monkey patching can be quite dangerous, so one needs to take care to not break contracts of patched classes.

capability. Since in Ruby, the + operator on symbols is not defined, it is possible to define it and have Ruby-Lola create a query in such cases. Numeric types like integers are problematic as they already react to the operator + and changing that behavior would create fundamental problems in most applications, mostly breaking how math works. So we added our method over the original behavior and checked that we only create a query when otherwise there would be type clashes with for example symbols. Also, when the original call would have worked, we assume it was intended and hand it through to the original function.

There are some problems with operator overloading as it is not possible to overwrite logical operators like || or *&&* or *if then else*. So for *and* and *or* we can use custom made functions that are confusing to read or use the single | or *&*, which are bitwise operators and can be overloaded. For *if then else* we have to use a custom made function named *ite* that may be hard to read, but at least works.

We can also define custom operators that have new meaning, as long as we accept that the syntax is not as clean as with build in operators.

```
:age + 2
:age.~ 2
```

The . used is needed for the Ruby interpreter to parse the code correctly. There is no way to define new infix operators in Ruby, so this is an acceptable limitation. One interesting approach would be to add logical operators in an unconventional way that works with what Ruby allows and is still clean.

```
ite(:male, :age, :age + 2)
:male.?(:age).!(:age + 2)
```

Our library offers these methods as alternatives to the default *ite* and that way we can monitor which version is preferred. Writing a specification is a lot about it being readable years later by someone who does not know the internal workings of Ruby-Lola. Offering choice in how one writes the specification can help in this regard.

Chapter 4

Example

To put Ruby-Lola to the test, we have created an auction system that is then verified. There is a variety of *Items* stocked in warehouses, and there are *Auctions* for these items. In Figure 4.1 one can see a simplified version of said auction system. The current system is without any restrictions and contains many bugs and problems. Over the next pages, we will explore many of these bugs and how to prevent them using Ruby-Lola.

```
class Item
  attr_accessor :amount
  has_many :auctions
end
class Auction
  attr_accessor :amount, :bid, :bidder
  belongs_to :item
end
```

Figure 4.1: A bidding system where users can bid on items in an auction. This showcases an Item model which stores an amount of items left and a reference list of auctions for this item. It also has an Auction model which has a reference to the item being auctioned. Each auction has a highest bid and the bidder and an amount of the item being auctioned. The code for this example is simplified to ease reading.

To get to a minimal running example of a verification with Ruby-Lola, one has to install the respective gem with *gem 'ruby-lola'*. Now we need to define a specification. We start with an empty specification, which will not do anything yet. Installing the gem is sufficient to set up Ruby-Lola for a Ruby on Rails project, it will automatically install itself and expose the *define_specification* environment, as

seen in Figure 4.2.

```
class Item
  attr_accessor :amount
  has_many :auctions
  define_specification do
  end
end
class Auction
  attr_accessor :amount, :bid, :bidder
  belongs_to :item
  define_specification do
  end
end
```

Figure 4.2: The example from Figure 4.1 with an empty specification attached. No additional setup is required.

Each *Item* in the auction system has an amount field that displays how many of this item are still in stock. It would not make sense to have a negative amount, so to verify that this does not happen, one can write a specification like in Figure 4.3. For this, a stream is created that computes if amount is negative. And then a trigger is set to alert if the condition gets true. Now, every time an *Item* is created or updated, Ruby-Lola verifies that the change is valid before allowing it. If a user tries to edit an item and set a negative amount, an error message "Amount cannot be negative!" would be shown to them.

Similar sanitization checks can be performed for the *Auction* model, as seen in Figure 4.4.

Until now, the specification only checked for current values. However, Lola can also retrieve previous values inside a stream with a look back operator. This is needed to ensure that each bid is bigger than the previous one. The example in Figure 4.5 shows how to use the look-back operator.

Sometimes one needs to reference a different model while writing a specification. This is where the powers of Ruby-Lola come into play. Using existing references, on can reference and access other models inside a specification. In this case, an auction should never put up more items than are in stock. The example in Figure 4.6 shows how to reference other models.

Since each *Item* can have multiple auctions running at the same time, there needs to be a check in place to ensure that the sum of all items up for auction is available in stock. The reference, in this case, is a many reference, so aggregations can be used on it, as shown in Figure 4.7.

```
class Item
  attr_accessor :amount
  has_many :auctions
  define_specification do
    define :amount_negative, :boolean do
      :amount < 0
    end
    trigger :amount_negative, 'Amount cannot be negative!'
  end
end
```

Figure 4.3: This example showcases the Item model which stores an amount of items left and a reference list of auctions for this item. It shows how to verify that the amount will never be negative. Since *amount* is a field of *Item*, an input stream for it is implicitly created an can just be used. Each Item created will have its individual stream for amount, Ruby-Lola takes care of the parameter.

```
class Auction
  attr_accessor :amount, :bid, :bidder
  belongs_to :item
  define_specification do
    define :amount_too_small, :boolean do
      :amount < 1
    end
    trigger :amount_too_small,
      'Amount needs to be positive!'
    define :bid_negative, :boolean do
      :bid < 0
    end
    trigger :bid_negative, 'Bid cannot be negative!'
  end
end
```

Figure 4.4: This example showcases the Auction model which has an amount and a bid. Here each auction should put at least one item up for bidding, but the starting bid is allowed to be zero. Negative bids are excluded. A Lola specification ensures that these conditions are not violated.

```
class Auction
  attr_accessor :amount, :bid, :bidder
  belongs_to :item
  define_specification do
    define :bid_not_bigger, :boolean do
      :bid <= look_back(:bid, 1, -1)
    end
    trigger :bid_not_bigger,
      'Bid needs to be bigger than previous bid!'
  end
end
```

Figure 4.5: This example showcases the Auction model which has an amount and a bid. Here each new bid should be more than the previous bid. A Lola specification ensures that these conditions are not violated. In this case, only the previous value is needed but one can look back as much as needed and Ruby-Lola will automatically keep that many records in memory. Memory usage is therefore capped by the biggest look back.

```
class Auction
  attr_accessor :amount, :bid, :bidder
  belongs_to :item
  define_specification do
    define :not_in_stock, :boolean do
      :item.amount < :amount
    end
    trigger :not_in_stock,
      'This item has not enough stock for this auction!'
  end
end
```

Figure 4.6: This example again showcases the Auction model which has an amount and a bid. The amount put up for auction should always be available in stock. This is checked by using the reference to item. Here one can see the benefits of the object-oriented way of writing such a specification, as accessing the reference works similar to how one is used to work with a reference in Rails.

```
class Item
  attr_accessor :amount
  has_many :auctions
  define_specification do
    define :not_in_stock, :boolean do
      sum(:auctions.amount) < :amount
    end
    trigger :not_in_stock,
      'This item has not enough stock for its auctions!'
  end
end
```

Figure 4.7: This example showcases the Item model which stores an amount of items left and a reference list of auctions for this item. Here a sum aggregation is done on the auctions reference, specifically the amount field. The object-oriented way of writing makes it easy to express complex queries like this one while still staying readable for humans.

In the end, this example shows how to do a variety of verification checks inside a human-readable specification. The full example is shown in Figure 4.8.

4.1 Benchmarks

To measure the overhead introduced by this specification into the system, we conducted tests simulating common user actions repeatedly and compared the time spend inside verification code with the time spent storing the change into the database.

Our first benchmark does this with a local database and a simple specification, results are shown Figure 4.9. This scenario ignores many factors that add time to a request, environmental factors such as internet connection, technical factors such as the clients hardware, Rails routing and business logic. In this offline environment, Ruby-Lola adds a 10 percent overhead per request.

With a second benchmark, we tried to produce a very inefficient specification that does a look back over the last 10000 previously seen values, with results in Figure 4.10. This benchmark increases the overhead to about 15 percent.

In a practical setting these overheads will not be noticeable by the user. Normal requests for Ruby on Rails applications take approximately between 10ms and 200ms, depending on internet connectivity and other factors. From our offline benchmarks, we estimate the overhead of Ruby-Lola to be between 0.01ms to 0.1ms per request, which is less than one percent of request time, even with complicated queries.

```
class Item
  attr_accessor :amount
  has_many :auctions
  define_specification do
    define :amount_negative, :boolean do
      :amount < 0
    end
    trigger :amount_negative,
      'Amount can not be negative!'
    define :not_in_stock, :boolean do
      sum(:auctions.amount) < :amount
    end
    trigger :not_in_stock,
      'This item has not enough stock for its auctions!'
  end
end
class Auction
  attr_accessor :amount, :bid, :bidder
  belongs_to :item
  define_specification do
    define :amount_too_small, :boolean do
      :amount < 1
    end
    trigger :amount_too_small,
      'Amount needs to be positive!'
    define :bid_negative, :boolean do
      :bid < 0
    end
    trigger :bid_negative, 'Bid can not be negative!'
    define :bid_not_bigger, :boolean do
      :bid <= look_back(:bid, 1, -1)
    end
    trigger :bid_not_bigger,
      'Bid needs to be bigger than previous bid!'
    define :not_in_stock, :boolean do
      :item.amount < :amount
    end
    trigger :not_in_stock,
      'This item has not enough stock for this auction!'
  end
end
```

Figure 4.8: The bidding system where users can bid on items in an auction. This part of the specification shows simple validations that ensure no faulty data is entered. Items are validated not to allow negative stock. Auctions are validated to always auction some items. Bids need to be positive and always bigger than the previous bid. Moreover, the amount of items put up for auction should always be in stock. This is done through a reference to a different model.

Measuring the overhead in actual online requests posed to be difficult, as the time spend in Ruby-Lola code each request was less than one millisecond. So we concluded that overhead in practical scenarios would not be noticed by application developers or users.

```
                          user     system      real
lola prev x 10000:   0.551000   0.000000 (  0.427589)
db stores x 10000:   2.639000   0.363000 (  3.749787)
lola post x 10000:   0.016000   0.000000 (  0.036436)
Finished in 8.14649s
```

Figure 4.9: A simple benchmark with a small specification. We simulated 10000 model changes and summed up the time spend inside Ruby-Lola code and in requests to the database. The label *lola prev* indicates Ruby-Lola verification code that evaluates the specification, *db stores* indicates the time spend in code that saves the Rails model to a local PostgreSQL database, and *lola post* indicates the time Ruby-Lola spends storing new stream values and state.

```
                          user     system      real
lola prev x 10000:   0.296000   0.030000 (  0.451939)
db stores x 10000:   2.142000   0.406000 (  4.372816)
lola post x 10000:   0.108000   0.031000 (  0.129353)
Finished in 8.91038s
```

Figure 4.10: A simple benchmark with an inefficient look back, checking 10000 values each time the specification is evaluated. We simulated 10000 model changes and summed up the time spend inside Ruby-Lola code and in requests to the database. The label *lola prev* indicates Ruby-Lola verification code that evaluates the specification, *db stores* indicates the time spend in code that saves the Rails model to a local PostgreSQL database, and *lola post* indicates the time Ruby-Lola spends storing new stream values and state.

Chapter 5

Conclusion

We created Ruby-Lola, a domain specific language and framework for the monitoring of critical web applications. It streamlines the usage of the Lola runtime verification language in Ruby on Rails applications.

It introduces an expressive domain specific language that is based on the Lola stream-based specification language. The language is simple to read, but also powerful enough to express temporal logics and aggregations, while restricting specifications that are expensive to use, like future lookups. This allows for completely synchronous verification of user requests, detecting bugs and rejecting changes caused by them.

The domain-specific language reduces noise by implicitly making all model fields available as input streams and automatically handling model stream life cycles. It specification can also serve as a form of documentation for the model.

Ruby-Lola specifications are verified automatically with each new change to a model. Violations are handled depending on the execution context. In automated tests, a specification violation will fail the test, automatically enhancing the test suit through the specification. In user requests, error messages can be passed on as form validation errors, or they can silently alert developers with the problem. Changes to the database are automatically rejected, should the specification be violated.

This allows to integrate Ruby-Lola into existing tested applications with minimal effort. Even small specifications can amplify existing test suits to cover a wide range of cases. And when a specification grows bigger, the overhead imposed is small enough that it is not noticeable for the end user.

We conclude that Ruby-Lola can provide real value to existing projects that want to adopt runtime verification.

Chapter 6

Related Work

There are different design decisions for runtime verification systems that make them fit or unfit for certain use cases. For our use case, the synchronous monitoring of critical web applications, we evaluated multiple runtime verification systems and their fitness for the use case at hand.

First some general reading material for the topic of runtime verification. Sokolsky et al. give an overview of different approaches to runtime verification and different design decisions that runtime verification systems have to make [36]. A good first read into the topic.

For our chosen runtime verification system, we base our work on the papers introducing Lola. Lola 1.0 [14] and Lola 2.0 [18] introduced the Lola language, which we implemented with Ruby-Lola. Lola 2.0 is based on Lola 1.0, and we use language features from Lola 2.0, so both papers are cornerstones of our work. We depend on the proofs and guarantees of Lola 1.0 and 2.0 in our work on Ruby-Lola and ensure by design that they still hold for our implementation. Since Lola is a synchronous runtime verification system that allows complex operations over past values and gives safety guarantees on the specification, it became the runtime verification system of choice for our use case.

Cassar et al. compared different runtime monitoring solutions and put them on a spectrum of how coupled they were with the system under test [7]. Our runtime verification system Ruby-Lola falls into the *Completely Synchronous Monitoring* category as it allows the system under test to react to violations and therefore the monitor code stops execution completely until it is verified to comply to the specification. This distinction alone makes many runtime verification systems unfit for our use case.

One common design decision was to patch the executable after compilation [29, 9, 15, 13, 20]. The technique falls into the synchronous category. Aspect-Oriented Programming in Java and AspectJ are popular tools for this technique. While this approach is convenient for the user, it has drawbacks in our use case. Patching

binary files mixes user code with monitoring code, which can lead to hard to spot errors and undefined behavior. These would be a security risk and monitoring a system would not be worth much if the monitor produces new errors. Ruby-Lola uses the Rails framework for instrumentation to have a strong separation between user code and monitoring code.

A different design decision was to process log files after the fact [3, 26, 35]. This is called asynchronous monitoring. It has the benefit of having future data readily available and the time to do complex processing operations. However, for our use case, we need to react to violations before they are committed, for example, to reject faulty transactions.

Other runtime verification systems are intended for completely different domains. Colombo et al. [11] investigated runtime verification in highly dynamic systems where components are not always known beforehand. In our case, we know all the components and specifications before we test them. Distefano et al. [16] explored monitoring via register automata as a means to restrict resource usage for runtime verification of systems with unbounded resource generation. In our case, the bounds are established by the database model. Should we want to monitor big datasets for properties then that could be an interesting approach. Other works explored real-time monitoring for runtime verification systems[34, 31]. Achieving real-time is not necessary for web-based systems and certainly not an easy feat. Havelund et al. [21] explored how hierarchical state machines can be used for model testing. The general idea of the testing is similar to our scenario tests, but hierarchical state machines do not map as easily to database entries as Lola streams do. Armbrust et al.[1] explored runtime verification in the spark system.

We also found some ideas for further improvements of Ruby-Lola that could be explored in future work. Artho et al. [2] explored how to leverage a runtime verification specification for unit test generation. This could be interesting for future work. Generating test cases from code analysis in a dynamic language can be an interesting challenge. Utilizing Rails controller route data to achieve some coverage of possible user actions could also be worth exploring. Tools like Zapata[2] could assist in this process. Calinescu et al. [5] explored adapting the software at runtime, which is something Ruby-Lola could make use of in the future. It is not in the scope of current work, but the design of Ruby-Lola would allow for such modifications.

[2] `https://github.com/Nedomas/zapata`

Bibliography

[1] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1383–1394, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-2758-9. doi: 10.1145/2723372.2742797. URL `http://doi.acm.org/10.1145/2723372.2742797`.

[2] Cyrille Artho, Howard Barringer, Allen Goldberg, Klaus Havelund, Sarfraz Khurshid, Mike Lowry, Corina Pasareanu, Grigore Rosu, Koushik Sen, Willem Visser, and Rich Washington. Combining test case generation and runtime verification. *Theor. Comput. Sci.*, 336(2-3):209–234, May 2005. ISSN 0304-3975. doi: 10.1016/j.tcs.2004.11.007. URL `http://dx.doi.org/10.1016/j.tcs.2004.11.007`.

[3] David A. Basin, Matús Harvan, Felix Klaedtke, and Eugen Zalinescu. Monpoly: Monitoring usage-control policies. In *RV*, 2011.

[4] Andreas Bauer, Martin Leucker, and Christian Schallhart. Runtime verification for ltl and tltl. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 20(4):14, 2011.

[5] Radu Calinescu, Carlo Ghezzi, Marta Kwiatkowska, and Raffaela Mirandola. Self-adaptive software needs quantitative verification at runtime. *Commun. ACM*, 55(9):69–77, September 2012. ISSN 0001-0782. doi: 10.1145/2330667.2330686. URL `http://doi.acm.org/10.1145/2330667.2330686`.

[6] Tien-Dung Cao, Trung-Tien Phan-Quang, Patrick Felix, and Richard Castanet. Automated runtime verification for web services. In *Web Services (ICWS), 2010 IEEE International Conference on*, pages 76–82. IEEE, 2010.

[7] I. Cassar, A. Francalanza, L. Aceto, and A. Ingólfsdóttir. A Survey of Runtime Monitoring Instrumentation Techniques. *ArXiv e-prints*, August 2017.

[8] Pierre Castéran and Yves Bertot. Interactive theorem proving and program development. coq'art: The calculus of inductive constructions., 2004.

[9] Feng Chen and Grigore Roşu. Mop: An efficient and generic runtime verification framework. *SIGPLAN Not.*, 42(10):569–588, October 2007. ISSN 0362-1340. doi: 10.1145/1297105.1297069.

[10] Edmund M Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT press, 1999.

[11] Christian Colombo, Gabriel Dimech, and Adrian Francalanza. Investigating instrumentation techniques for esb runtime verification. In *SEFM*, 2015.

[12] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, and Xavier Rival. Why does astrée scale up? 35, 12 2009.

[13] Marcelo d'Amorim and Klaus Havelund. Event-based runtime verification of java programs. *SIGSOFT Softw. Eng. Notes*, 30(4):1–7, May 2005. ISSN 0163-5948. doi: 10.1145/1082983.1083249. URL `http://doi.acm.org/10.1145/1082983.1083249`.

[14] B. D'Angelo, S. Sankaranarayanan, C. Sanchez, W. Robinson, B. Finkbeiner, H. B. Sipma, S. Mehrotra, and Z. Manna. Lola: runtime monitoring of synchronous systems. In *12th International Symposium on Temporal Representation and Reasoning (TIME'05)*, pages 166–174, June 2005. doi: 10.1109/TIME.2005.26.

[15] Normann Decker, Martin Leucker, and Daniel Thoma. junitrv - adding runtime verification to junit. In *NASA Formal Methods*, volume LNCS 7871. Springer-Verlag Berlin Heidelberg, Springer-Verlag Berlin Heidelberg, 2013.

[16] Dino Distefano, Radu Grigore, Rasmus Lerchedahl Petersen, and Nikos Tzevelekos. Runtime verification based on register automata. *CoRR*, abs/1209.5325, 2012. URL `http://arxiv.org/abs/1209.5325`.

[17] Vijay D'silva, Daniel Kroening, and Georg Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(7):1165–1178, 2008.

[18] 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`.

[19] Sylvain Hallé and Roger Villemaire. Runtime verification for the web. In *International Conference on Runtime Verification*, pages 106–121. Springer, 2010.

[20] Klaus Havelund. Runtime verification of c programs. In *Proceedings of the 20th IFIP TC 6/WG 6.1 International Conference on Testing of Software and Communicating Systems: 8th International Workshop*, TestCom '08 / FATES '08, pages 7–22, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-68514-2. doi: 10.1007/978-3-540-68524-1_3. URL `http://dx.doi.org/10.1007/978-3-540-68524-1_3`.

[21] Klaus Havelund and Rajeev Joshi. Modeling and monitoring of hierarchical state machines in scala. In *SERENE*, 2017.

[22] Klaus Havelund and Grigore Roşu. Monitoring java programs with java pathexplorer. *Electronic Notes in Theoretical Computer Science*, 55 (2):200 – 217, 2001. ISSN 1571-0661. doi: https://doi.org/10.1016/S1571-0661(04)00253-1. URL `http://www.sciencedirect.com/science/article/pii/S1571066104002531`. RV'2001, Runtime Verification (in connection with CAV '01).

[23] Gerard J Holzmann. Cobra: fast structural code checking (keynote). In *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*, pages 1–8. ACM, 2017.

[24] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. Securing web application code by static analysis and runtime protection. In *Proceedings of the 13th international conference on World Wide Web*, pages 40–52. ACM, 2004.

[25] Manfred Broy Bengt Jonsson, Joost-Pieter Katoen Martin Leucker, and Alexander Pretschner. Model-based testing of reactive systems, 2005.

[26] Sean Kauffman, Klaus Havelund, and Rajeev Joshi. nfer - a notation and system for inferring event stream abstractions. In *RV*, 2016.

[27] Matt Kaufmann and J.Stroother Moore. Some key research problems in automated theorem proving for hardware and software verification. 98, 01 2004.

[28] Martin Leucker and Christian Schallhart. A brief account of runtime verification. *The Journal of Logic and Algebraic Programming*, 78(5):293–303, 2009.

[29] Patrick O'Neil Meredith, Dongyun Jin, Dennis Griffith, Feng Chen, and Grigore Rosu. An overview of the mop runtime verification framework. *International Journal on Software Tools for Technology Transfer*, 14:249–289, 2011.

[30] Glenford J Myers, Corey Sandler, and Tom Badgett. *The art of software testing*. John Wiley & Sons, 2011.

[31] Samaneh Navabpour, Chun Wah Wallace Wu, Borzoo Bonakdarpour, and Sebastian Fischmeister. Efficient techniques for near-optimal instrumentation in time-triggered runtime verification. In *Proceedings of the Second International Conference on Runtime Verification*, RV'11, pages 208–222, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 978-3-642-29859-2. doi: 10.1007/978-3-642-29860-8_16. URL `http://dx.doi.org/10.1007/978-3-642-29860-8_16`.

[32] Martin Ouimet and Kristina Lundqvist. Formal software verification: Model checking and theorem proving. *Embedded Systems Laboratory Technical Report ESL-TIK-00214, Cambridge USA*, 2007.

[33] Jiantao Pan. Software testing. *Dependable Embedded Systems*, 5:2006, 1999.

[34] Lee Pike, Alwyn Goodloe, Robin Morisset, and Sebastian Niller. Copilot: A hard real-time runtime monitor. In *Proceedings of the 1st Intl. Conference on Runtime Verification*, LNCS. Springer, November 2010. Preprint available at `http://www.cs.indiana.edu/~lepike/pub_pages/rv2010.html`.

[35] Martin Roesch. Snort - lightweight intrusion detection for networks. In *Proceedings of the 13th USENIX Conference on System Administration*, LISA '99, pages 229–238, Berkeley, CA, USA, 1999. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1039834.1039864`.

[36] Oleg Sokolsky, Klaus Havelund, and Insup Lee. Introduction to the special section on runtime verification. *Int. J. Softw. Tools Technol. Transf.*, 14(3):243–247, June 2012. ISSN 1433-2779. doi: 10.1007/s10009-011-0218-6. URL `https://doi.org/10.1007/s10009-011-0218-6`.

[37] J. A. Whittaker. What is software testing? and why is it so hard? *IEEE Software*, 17(1):70–79, Jan 2000. ISSN 0740-7459. doi: 10.1109/52.819971.

[38] Yichen Xie and Alex Aiken. Saturn: A sat-based tool for bug detection. In *International Conference on Computer Aided Verification*, pages 139–143. Springer, 2005.