Work on new event sources for detectEr

Guðmundur Stefánsson
Master of Science
May 2017
School of Computer Science
Reykjavík University

M.Sc. Final Report

Work on new event sources for detectEr

by

Guðmundur Stefánsson

Final Report of 30 ECTS credits submitted to the School of Computer Science
at Reykjavík University in partial fulfillment
of the requirements for the degree of
Master of Science (M.Sc.) in School of Computer Science

May 2017

Supervisors:

Luca Aceto, Supervisor
Professor, Reykjavík University, Iceland

Anna Ingólfsdóttir, Supervisor
Professor, Reykjavík University, Iceland

Examiners:

Marcel Kyas, Examiner
Assistant Professor, Reykjavík University, Iceland

Adrian Francalanza, Supervisor
Senior Lecturer, University of Malta, Malta

Work on new event sources for detectEr

Guðmundur Stefánsson

May 2017

Abstract

detectEr is an actor-based runtime verification tool for Erlang programs. It uses the tracing mechanism of the Erlang virtual machine to trace messages sent between Erlang processes, creating events that are matched against correctness properties defined in the logic mHML. This thesis describes work that was done to allow *detectEr* to receive events from new sources, including reading events from a textfile, and listening for events on a TCP port. This change opens the door for *detectEr* to verify non-Erlang programs, and that is demonstrated by showing how *detectEr* can now be used to verify communications between an Erlang client and a Java server.

Unnið að nýjum atburðum fyrir detectEr

Guðmundur Stefánsson

maí 2017

Útdráttur

detectEr er keyrslu-sannreyningartól fyrir Erlang forrit sem byggist á leikarahöguninni. Það notar rekjunarvirkni Erlang sýndarvélarinnar til að rekja skilaboð á milli Erlang ferla, og býr þannig til atburði sem eru bornir saman við réttleikaeigindi skilgreind í mHML. Þessi ritgerð gerir grein fyrir því hvernig *detectEr* var uppfært til að geta fengið atburði á nýja máta, þ.á.m. með því að lesa þá úr textaskrá, og með því að hlusta eftir þeim á TCP porti. Þessi breyting gerir *detectEr* kleift að sannreyna forrit sem ekki eru skrifuð í Erlang, og sýnt er fram á það með því að sýna hvernig hægt er að nota *detectEr* til að sannreyna samskipti á milli Erlang biðlara og Java miðlara.

Work on new event sources for detectEr

Guðmundur Stefánsson

Final Report of 30 ECTS credits submitted to the School of Computer Science
at Reykjavík University in partial fulfillment of
the requirements for the degree of
Master of Science (M.Sc.) in School of Computer Science

May 2017

Student:

.....

Guðmundur Stefánsson

Supervisors:

.....

Luca Aceto

.....

Anna Ingólfsdóttir

Examiners:

.....

Marcel Kyas

.....

Adrian Francalanza

.....
date

.....
Guðmundur Stefánsson
Master of Science

Contents

Contents **xi**

List of Figures **xiii**

1 Introduction **1**
1.1 *detectEr* . . . 2
1.2 Contributions . . . 2

2 Context **3**
2.1 Related work . . . 3
2.2 μHML and mHML . . . 4
2.3 Erlang . . . 5
2.4 How to instrument a system . . . 5

3 Making *detectEr* more versatile **7**
3.1 Common behaviour . . . 7
3.2 Receiving events via tracing . . . 8
3.3 Retrieving events from a file . . . 8
3.4 Retrieving events from TCP . . . 8
3.5 Creating new event sources . . . 8
3.6 Multiple event sources . . . 9
3.7 How to use detectEr . . . 9

4 Case study **11**
4.1 Case study behaviour . . . 11
4.2 Case study behaviour - Erlang . . . 11
4.3 Case study behaviour - Java . . . 12
4.3.1 The tracker . . . 12
4.4 Examples . . . 13
4.4.1 The properties . . . 13
4.4.2 Two Erlang traces . . . 14
4.4.3 Java via TCP . . . 14
4.4.4 Reading a file . . . 14
4.4.5 Reading a long file . . . 16
4.4.6 Discussion . . . 16

5 Conclusions and future work **17**

Bibliography **19**

List of Figures

2.1 Synthesis and operation of a runtime verification monitor 3
2.2 The μHML syntax and semantics 4
2.3 The syntax of mHML 5

3.1 The processes of *detectEr* 7
3.2 The processes of multiple event sources 10

4.1 Output from the first test 15

Chapter 1

Introduction

Concurrency [1] is a term used to describe the behaviour of systems comprising of multiple components whose lifetimes overlap. When concurrent systems are running, a single component executes one or more commands at a time, and then another component takes over (thread interleaving). This way, the various components don't need to wait for each other to finish working, and thus it appears that they are running at the same time when they aren't. One of the benefits of this approach is that it incentivices dividing systems into multiple components, where each component performs a specific task. But it has some drawbacks as well, because it is possible for the multiple components to access shared resources at the same time, which can cause errors. It is also possible for two processes to wait for each other to finish, leading to neither process being able to continue. These kinds of problems can make concurrent systems notoriously hard to debug, but despite that, concurrent systems are still generally preferred to ones with only one component.

Due to the difficulty of debugging concurrent systems, it is necessary to formally verify their correctness. However, static verification techniques like model checking may not always be the most suitable methods for this task. Concurrent systems have state spaces that grow exponentially in size w.r.t. the number of components, due to thread interleaving, thus making it potentially infeasible to check their entire state space. This is called the *state explosion problem* [2]. Moreover, model checking requires a model of the system to be verified, and those are not guaranteed to be accurate [3], especially for large systems. Therefore, it is useful to explore alternative methods to address the correctness problem. One approach that has been proposed to that end is runtime verification.

Runtime verification (RV) is a method to verify the correctness of systems by analyzing a single execution, or a finite subset of possible executions of the system [4]. Thus, the system's state space isn't exhaustively checked, circumventing the state explosion problem. It also eliminates the need for a model of the system, and lets the system be verified in its runtime environment, which may not always be available for model checking. It is similar to testing in the way that it doesn't exhaustively check the system's state space, but the difference is that testing is only performed pre-deployment, whereas RV is performed post-deployment. This means that RV can verify a program for as long as it is used, meaning that potentially, all executions that will ever be executed will be verified. It is also possible to use RV to recover from violations by altering the offending executions. (See, for instance, [5]–[7]). However, runtime verification is not a perfect solution for verifying the correctness of concurrent systems because it is not as expressive as model checking since it judges the system based on a finite subset of possible executions instead of the entire state space.

1.1 *detectEr*

detectEr [8] is a runtime verification tool that verifies Erlang [9], [10] programs by tracing the messages that they send and receive, using the tracing mechanism of the Erlang virtual machine. These messages are then used to test the program against the correctness specifications defined in the logic mHML [11]. It does so by generating a system of monitors from a given property specification that run alongside the verified system and notify the user whenever a violation or a satisfaction of a property is detected.

detectEr works well for verifying Erlang programs in terms of the messages sent between processes, but there is room for improvement. For example, there are systems whose behaviour can be described in terms of messages sent between actors that are not programmed in Erlang which could potentially be verified by *detectEr*. But since they aren't, they can't. There are also many systems that have components in different programming languages, like an Erlang server that serves clients written in Java or Python, for example. Therefore, it is a good idea to change *detectEr* so it can receive its events from multiple sources simultaneously, and by using methods other than just the tracing mechanism of the Erlang virtual machine, making it more versatile.

1.2 Contributions

Chapter 2 provides some useful context on runtime verification, and describes *detectEr* in more detail. Readers who are familiar with *detectEr* can safely skip this chapter. Then chapter 3 describes how I made *detectEr* more versatile by replacing the tracing functionality with a more general component: *the event source*. The new component allows *detectEr* to read events from a text file and listen for events on a TCP port, in addition to the already existing tracing functionality. It also makes it easier to extend *detectEr* by adding support for new sources of events, which makes future work on the tool easier. I also made it so that it is possible to monitor multiple systems simultaneously, which can be a great asset when monitoring concurrent systems. After that, chapter 4 demonstrates how the event source works in action, by describing two systems that I used as a case study, one written in Erlang, and the other one written in Java. These two systems are used to show how *detectEr* can now be used to monitor multiple systems in different programming languages. Finally, chapter 5 discusses the event source's impact and how work on it can continue in the future.

Chapter 2

Context

When a system is executed, a (possibly infinite) sequence of events is generated. This sequence is known as a *trace*, and it reflects the behaviour of the system in terms of function invocations or messages sent between processes. In runtime verification, these traces are fed to a software entity known as a *monitor*, which matches the events against a predetermined specification of correctness, known as a *property*, in order to determine if the system is behaving according to the given specification. The monitor can reach one out of three possible verdicts for each execution: the system satisfies the property, the system violates the property, or the trace does not give enough information to reach a conclusive verdict. This process is illustrated in Figure 2.1.

2.1 Related work

There exist a number of runtime verification tools. For example, LARVA [12] is a runtime verification tool which uses properties defined with DATEs (Dynamic communication Automata with Timers and Events) [13] to verify Java programs. It has been extended to monitor Erlang programs as well, by creating a new version, Elarva [14], making it the only other runtime verification tool for Erlang, to my knowledge. Java PathExplorer [15] is another runtime verification tool for Java, developed by NASA, which tests execution traces

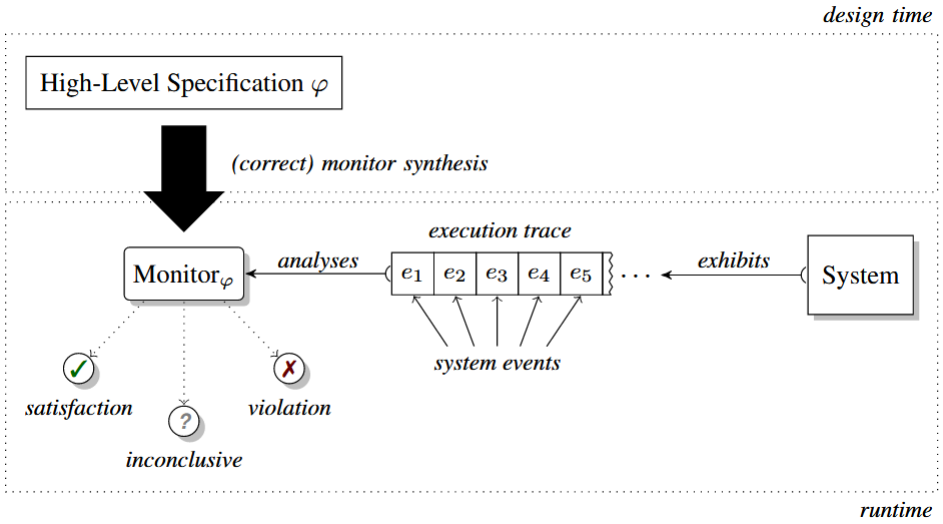

Figure 2.1: Synthesis and operation of a runtime verification monitor

Syntax

$$
\begin{array}{lllll}
\varphi, \phi \in \mu\text{HML} ::= & \mathsf{tt} & \text{(truth)} & \mid \mathsf{ff} & \text{(falsehood)} \\
\mid & \varphi \vee \phi & \text{(disjunction)} & \mid \varphi \wedge \phi & \text{(conjunction)} \\
\mid & \langle\alpha\rangle\varphi & \text{(possibility)} & \mid [\alpha]\varphi & \text{(necessity)} \\
\mid & \mathsf{min}\, X.\varphi & \text{(min. fixpoint)} & \mid \mathsf{max}\, X.\varphi & \text{(max. fixpoint)} \\
\mid & X & \text{(rec. variable)} & &
\end{array}
$$

Semantics

$$
\begin{array}{ll}
[\![\mathsf{tt}, \rho]\!] \stackrel{\text{def}}{=} \text{PROC} & [\![\mathsf{ff}, \rho]\!] \stackrel{\text{def}}{=} \emptyset \\
[\![\varphi_1 \wedge \varphi_2, \rho]\!] \stackrel{\text{def}}{=} [\![\varphi_1, \rho]\!] \cap [\![\varphi_2, \rho]\!] & [\![\varphi_1 \vee \varphi_2, \rho]\!] \stackrel{\text{def}}{=} [\![\varphi_1, \rho]\!] \cup [\![\varphi_2, \rho]\!] \\
[\![[\alpha]\varphi, \rho]\!] \stackrel{\text{def}}{=} \left\{ p \mid p \stackrel{\alpha}{\Rightarrow} q \text{ implies } q \in [\![\varphi, \rho]\!] \right\} & [\![\langle\alpha\rangle\varphi, \rho]\!] \stackrel{\text{def}}{=} \left\{ p \mid p \stackrel{\alpha}{\Rightarrow} q \text{ and } q \in [\![\varphi, \rho]\!] \right\} \\
[\![\mathsf{min} X.\varphi, \rho]\!] \stackrel{\text{def}}{=} \bigcap \{ S \mid [\![\varphi, \rho[X \mapsto S]]\!] \subseteq S \} & \\
[\![\mathsf{max} X.\varphi, \rho]\!] \stackrel{\text{def}}{=} \bigcup \{ S \mid S \subseteq [\![\varphi, \rho[X \mapsto S]]\!] \} & [\![X, \rho]\!] \stackrel{\text{def}}{=} \rho(X)
\end{array}
$$

Figure 2.2: The μHML syntax and semantics

against high level specifications written in Maude, and can also detect deadlocks and race conditions.

McErlang [16] is a model checker for Erlang programs that verifies them using properties defined in Erlang.

Rebeca [17] is an actor-based modeling language, making it a good choice for creating models of Erlang programs, which provides automatic translation to model checkers like SPIN [18] and SMV [19].

2.2 μHML and mHML

μHML [11] is a branching-time logic for describing correctness properties over *Labeled Transition Systems (LTSs)*. A LTS consists of a set of system states $p, q \in \text{SYS}$, a set of actions $\alpha \in \text{ACT}$, and a transition relation between states and actions which contains a set of moves on the form, $p \xrightarrow{\alpha} q$, meaning that it is possible to transition from state p to state q by performing an a - move. The syntax and the semantics of the logic are shown in Figure 2.2 [11] and the semantics is defined in terms of the mapping between μHML formulae φ and the set of LTS states $S \subseteq \text{SYS}$ which satisfy them. The truth value **tt** is satisfied by all processes and the falsity value **ff** is satisfied by none. Possibility formulae <α>φ mean that there must be at least one system transition with the event α where the subsequent transition satisfies φ. On the other hand, necessity formulae $[\alpha]\varphi$ mean that all system executions (if any) with the event α must lead to states satisfying φ.

Here are a few examples of μHML properties:

- φ_1 = <α>tt
- φ_2 = [α]ff
- φ_3 = max X.([β]X $\wedge$ [α][α]ff)
- φ_4 = min X.(<α>X $\vee$ <β>tt)

$$\psi \in \text{mHML} \stackrel{\text{def}}{=} \text{sHML} \cup \text{cHML} \text{ where:}$$

$$\begin{array}{llllllll} \theta, \vartheta \in \text{sHML} ::= & \mathbf{tt} & \mid \mathbf{ff} & \mid \theta \wedge \vartheta & \mid [\alpha]\theta & \mid \mathbf{max}\, X.\theta & \mid X \\ \pi, \varpi \in \text{cHML} ::= & \mathbf{tt} & \mid \mathbf{ff} & \mid \pi \vee \varpi & \mid \langle\alpha\rangle\pi & \mid \mathbf{min}\, X.\pi & \mid X \end{array}$$

Figure 2.3: The syntax of mHML

Property φ_1 is satisfied by all processes that can perform an α-action, and φ_2 is satisfied by all processes that can not make an α-action. φ_3 is violated by processes that via a sequence of β - moves can reach a state that can perform two α - moves in a row, and φ_4 is satisfied by processes that can perform a β - move after zero or more α - moves.

In μHML, there are properties that can be shown to be satisfied or violated by examining a single system computation. For example, φ_1 requires the system to be able to perform a single α - move to be satisfied, so if the system performs an α - move, it is enough information to determine that the system satisfies φ_1. Similarly, a system only needs one α - move for it to be clear that it violates φ_2. But for some properties, particularly ones describing infinite or branching executions, a single finite runtime trace is not enough to reach a conclusive verdict. The work in [11] explores the monitorable limits of μHML, and defines a logical subset called mHML, which is monitorable and maximally-expressive w.r.t. the constraints of runtime monitoring. Its syntax is described in Figure 2.3 [11] and it consists of two parts: *safety* HML (sHML) and *co-safety* HML (cHML). sHML describes invariant properties that state that violations do not occur and cHML describes properties that will eventually be satisfied after a finite number of events.

2.3 Erlang

Erlang [9], [10] is a functional programming language that uses the actor-model to model concurrency. In the actor model, each component can only communicate with other components by sending messages to them. In Erlang, each process is an actor and there are no shared resources, thus the problem of mutial exclusion is circumvented. Each process can communicate with other processes by sending asynchronous messages with the `!` operator, and the messages are then stored in the receiving process' message queue. The receiving process can then use the `receive` keyword to dequeue messages from the queue. It is also possible to use the tracing mechanism of the Erlang virtual machine (EVM) to trace these messages without changing the source code or altering the behaviour of the program.

2.4 How to instrument a system

detectEr is an asynchronous actor-based runtime verification tool for Erlang programs that uses properties defined in mHML. It is available online on bitbucket.org [20] and the work presented in this thesis is on the branch `detecter-lite-1.0-event-source`. It requires a working installation of Erlang, and the GNU `make` utility. OSX users can acquire it by installing the XCode Command Line Tools, and Windows users can install MinGW, Cygwin or GnuWin.

In order to use *detectEr* to verify a system, the system must go through the instrumentation process. That is done by invoking the `instrument` target of *detectEr*'s makefile. During the instrumentation process, *detectEr* is compiled, and the mHML property that will

be used to verify the system is parsed and used to generate a monitor. The monitor is a process that listens for events generated by the system, and then matches them against the mHML property, checking if the system violates or satisfies it.

The arguments needed to instrument a system are:

- `hml`: The relative or absolute path that leads to the formula script file
- `app-bin-dir`: The target application's binary base directory
- `MFA`: The target application's entry point function, encoded as a `{Module, Function, [Arguments]}` tuple

However, following the work described in this paper, these arguments were changed, and the changes will be described later.

When the instrumentation process is done, the output is a *launcher.beam* binary file, which will be located in the `app-bin-dir` directory. It contains the `launcher:start/0` function, which can be invoked to start *detectEr*.

Chapter 3

Making *detectEr* more versatile

Some drawbacks of *detectEr* are that it is only able to monitor Erlang applications, and only by running a program and tracing it. Ideally, it should be able to monitor any type of system, so long as its behaviour can be expressed by a finite sequence of events. In order to make that happen, I replaced *detectEr*'s tracing functionality with a more general component: the *event source*.

3.1 Common behaviour

When detectEr is launched, the event source is started on a new process, *es*, and the monitor is started on another process, *sup*. The event source process then acts as a medium between the monitored system, *sys*, and the monitor *sup*, as shown in Figure 3.1.

Currently, there are 3 implementations of the event source, and each is defined by an Erlang module. These are `trace_event_source`, which receives events by tracing an Erlang program, `file_event_source`, which reads events from a text file, and `tcp_event_source`, which listens for events on a TCP socket.

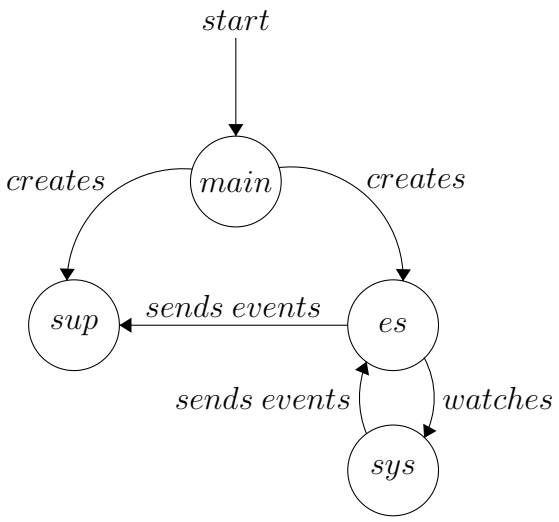

Figure 3.1: The processes of *detectEr*

3.2 Receiving events via tracing

Originally *detectEr* could only receive events by tracing an Erlang program. I relocated that functionality to the module `trace_event_source`. As an argument, it takes a tuple containing the module, function and arguments of the program to be monitored. This tuple is used to start the program, and then the event source traces the program, and passes the events forward to the monitor.

3.3 Retrieving events from a file

In order to read events from a file, I created the module `file_event_source`. The only argument that it needs is the path to the file from which the events should be read. However, it should be noted that the given filepath needs to be absolute or relative to the output *launcher.beam* file, unlike the other arguments of the instrumentation process. That is because the filepaths of the `instrument` target are used at compile-time, but the filepath of `file_event_source` is used at runtime.

The supplied file should have a number of events written as Erlang terms, separated by periods. The event source expects two types of events; namely send and recv, in the following forms:

- `{recv, Receiver, Message}`
- `{send, Sender, Receiver, Message}`

The atoms `recv` and `send` determine whether the event is a receive(?) or a send(!) operation. *Receiver* and *Sender* are identifiers for the systems receiving and sending the messages, respectively, and *Message* is the payload of the message. An example of events from a simple echo server would be:

```
{recv, server, {request, client, 1}}.
{send, server, client, {response, 1}}.
```

After the file has been read, the events are sent to the monitoring process by sending `{read_line, N}` messages to the event source's process, where `N` is either a positive integer, determining how many lines should be read, or the atom `all`, meaning that all remaining lines should be read.

3.4 Retrieving events from TCP

In order to read events from a TCP port, I created the module `tcp_event_source`. The only argument that it needs is the port that should be used. When the launcher is started, it will listen for a connection to that port, and then receive TCP messages on that port. The messages should contain events of the same form as previously described for `file_event_source`.

3.5 Creating new event sources

Creating support for new event sources becomes very easy once the communications between the monitor and the system to be monitored have been isolated within a single module.

Indeed, in order to introduce a new event source, all the programmer has to do is to create a new module, using the behaviour `event_source_behaviour`. The only function that the behaviour has is `setup/2`, which takes an object and a Pid as arguments. The object should contain whatever information is required to set up the event source (filename, port, et.c.), and the Pid will be the Pid of *sup*, which will receive the events generated by the event source. The function should create a new process and return its Pid. That process then waits for the `mon_start` message, and then performs the necessary setup for the events. For example, in order to create an event source that reads events from a database, the new process should start a connection to the database after receiving `mon_start`.

3.6 Multiple event sources

Since concurrent systems consist of multiple components working concurrently, it seems logical that verification tools should be capable of handling multiple systems simultaneously. Therefore, I changed *detectEr* to support multiple event sources, each being checked against a single property. For each event source, the same processes are created, just like described in Figure 3.1. However, in order to identify them, a number is appended to their names, as shown in Figure 3.2.

There is still much to be done with this new feature, though. One limitation is that each event source can only check properties using events from only one event source. If that was changed, so that properties could be checked against events from multiple event sources, the resulting version of *detectEr* would allow us to check properties involving multiple systems.

Another limitation is that each event source creates only one monitor, and thus only checks for only one property at a time. But most systems have to satisfy multiple properties at the same time, and it would therefore be ideal to be able to have each event source verify its system for multiple properties. That can be done by creating multiple monitors for each event source, where each monitor corresponds to a property that the system has to satisfy.

However, it is possible to combine multiple mHML properties into one big property which can be used to create a monitor, but that approach has its drawbacks. If we have two properties φ_1 and φ_2, and we create a new property $\varphi_3 = \varphi_1 \wedge \varphi_2$, then φ_3 is violated iff either φ_1 or φ_2 is violated. But if we use φ_3 to create a monitor, and then the monitor detects a violation, it is not obvious which property is being violated. The violated property can be deduced by looking at the events and the output of the monitor, but it would be better to have them separate. Furthermore, a bigger problem is that something as simple as conjoining or disjoining multiple properties may not work well enough when we have multiple properties, some of which detect satisfactions, others detect violations, and yet others may detect either. Creating a big property from smaller ones is something that can be done, but it has to be done carefully.

3.7 How to use detectEr

In order to facilitate the changes made to *detectEr*, I changed the arguments of the `instrument` target to the following ones:

- `app-bin-dir`: The target application's binary base directory
- `args`: A string containing a list of event source objects. Each event source should be formatted like so: `{hml, event-source, args}`

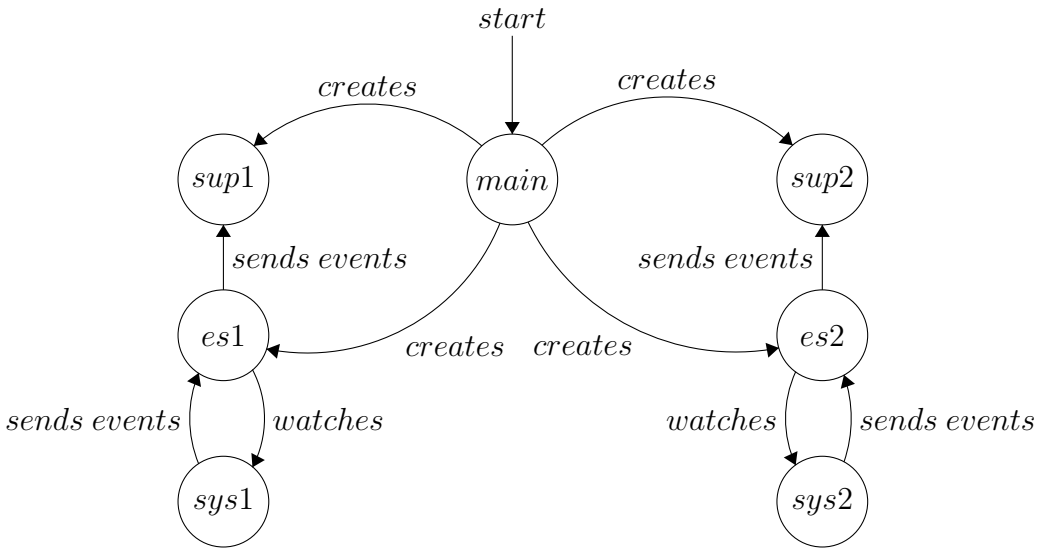

Figure 3.2: The processes of multiple event sources

- `hml`: The relative or absolute path that leads to the formula script file
- `event-source`: The module defining the event source to be used
- `args`: The arguments for the event source

Listing 3.1: An example of arguments for the instrumentation process

```
1 make instrument args="[{'priv/case_study/prop1_server.txt', trace_event_source, {↩
    ↪client_server_erlang, start, [server, echo, nul]}}]" app-bin-dir="ebin"
```

An example would be the arguments shown in Listing 3.1, which indicate that a monitor will be created using only one event source which uses the module `trace_event_source` with the arguments `{client_server_erlang, start, [server, echo, nul]}}` The monitor will verify the system against the property defined in 'priv/case_study/prop1_server.txt' and the resulting *launcher.beam* file will go into the folder *ebin*.

Chapter 4

Case study

One of the benefits of the new event source is that *detectEr* can receive events from various sources, and can therefore, in theory, monitor any type of system regardless of programming language. To demonstrate that, I created two programs as a case study: The Erlang module `client_server_erlang`, and the Java program `client_server_java` [21]. Both programs behave the same way for the most part; the biggest difference is that one is written in Erlang and the other is written in Java.

4.1 Case study behaviour

The case study can be run either as a client or as a server. When run as a client it sends requests to the server and waits for responses. The requests it sends are on the form `{request, Self, Num}`, where `Self` is the Pid of the client, and `Num` is an integer. When run as a server, it waits for requests and then returns responses on the form `{response, Other}`, where `Other` is a number that is determined by the server's configuration and the number that the server receives. For the purposes of testing, the correct response to a request `{request, Self, Num}` is viewed to be `{response, Num + 1}`.

4.2 Case study behaviour - Erlang

The case study written in Erlang has 6 different configurations and it is run by invoking the `client_server_erlang:start/3` method. The first and second parameters define how the program should behave, whether it should be a server or a client, and how it acts. The possible combinations of the first and second arguments are:

- `server success`: Runs a server that returns the correct response (Num + 1)
- `server wrong_response`: Runs a server that returns a wrong response (Num - 1)
- `server echo`: Runs a server that returns a wrong response (Num)
- `client success`: Runs a client that sends a request and waits for a response
- `client success_multiple`: Runs a client that sends two subsequent request, waiting for responses in between
- `client no_request`: Runs a client that does nothing

The third argument is only necessary if the program is being run as a client. It is the identifier of the server, which can be anything that can be on the left side of a ! operator in Erlang. But if the program is being run as a server, it doesn't matter what it is, since it will never be used.

4.3 Case study behaviour - Java

The case study in Java uses the jinterface [22] package in order to create an Erlang node, which it uses to send and receive messages from Erlang programs. It is run by invoking the main function in the `client_server_java` class and it has 5 different configurations, depending on the arguments provided from the command line.

The first two arguments are the names of the Erlang node and mailbox, respectively, used for communication with Erlang programs. The third and fourth arguments define how the program should behave, whether it should be a server or a client, and how it acts. The possible combinations of the third and fourth arguments are:

- `server success`: Runs a server that returns the correct response (Num + 1)
- `server wrong_response`: Runs a server that returns a wrong response (Num - 1)
- `server echo`: Runs a server that returns a wrong response (Num)
- `client success`: Runs a client that sends a request and waits for a response
- `client no_request`: Runs a client that does nothing

If the case study is run as a client, the fifth and sixth arguments need to be the name of the node and the process that will receive the message, respectively.

As an example, when provided with the following arguments:

```
client_server java server success
```

the case study is run as a successful server, and an Erlang program can send a message to it with the following syntax:

```
{java, client_server@User} !  {request, self(), Num}
```

Conversely, when provided with the following arguments:

```
client_server java client success monitor@User sys2
```

the case study is run as a client that sends a single request to the process `sys2` on the Erlang node `monitor@User`.

4.3.1 The tracker

In order to test the usefulness of `file_event_source` and `tcp_event_source`, I equipped the Java case study with an additional component; the *tracker*. The purpose of the tracker is to generate events that the new event sources can understand, thus allowing *detectEr* to monitor the Java case study.

The tracker is an interface with two methods;

- `send(String to, Object[] params)`
- `recv(Object[] params)`

They correspond to the two messages that detectEr expects, ! and ? respectively. In order to monitor the Java case study, these methods are called before messages are sent or received from the Erlang case study.

The tracker has two different implementations. One generates events by writing events to a file which can then be read by `file_event_source`, and the other one sends the events over TCP to be received by `tcp_event_source`.

The tracker is rather simple, and can be replicated easily in other programming languages. Its main limitation is that the only way to use it is to alter the source code of the monitored program manually. Despite this limitation, it does allow *detectEr* to monitor the Java case study, and it is possible to make a tracker or a similar component in any programming language, making it possible to use *detectEr* to monitor any system regardless of programming language.

4.4 Examples

I used the case study to verify that the new event source component works as intended. The following examples all have a simple client-server behaviour, where the client sends one or more requests to the server and the server responds with either the correct response (Num + 1) or an incorrect one.

4.4.1 The properties

I created 4 mHML properties in order to test the case study. They are: *client_vio*, *client_sat*, *server_vio* and *server_sat* [1], and they describe the correct behaviour of the program.

The properties *client_vio* (Listing 4.1) and *server_vio* (Listing 4.2) look for violations in the behaviour of the client and the server, respectively. Both of them detect a violation if the server responds with anything but Num + 1, and they will keep checking until there is a violation.

Listing 4.1: As long as the client lives, it does not receive an incorrect response

```
1 max('X',
2 [Server ! {request, Client, Num}]
3     ([Client ? {response, Other}] iff (Other =/= Num + 1) implies ff else 'X'))
```

Listing 4.2: As long as the server lives, it never sends incorrect responses

```
1 max('X',
2 [Server ? {request, Client, Num}]
3     ([Client ! {response, Other}] iff (Other =/= Num + 1) implies ff else 'X'))
```

Conversely, the properties *client_sat* (Listing 4.3) and *server_sat* (Listing 4.4) look for a satisfaction in the behaviour of the client and the server, respectively. They only check the first request made by the client to the server, and if the request is answered with a Num + 1, then a satisfaction is detected.

Listing 4.3: The client gets a single correct response

[1] The possibility formulae are written with /α\ instead of <α>. This will be changed in a later version of *detectEr*

```
1 /Server ! {request, Client, Num}\/Client ? {response, Other}\ iff (Other == Num + 1) ↩
    ↪implies tt else ff
```

Listing 4.4: The server sends a single correct response

```
1 /Server ? {request, Client, Num}\/Client ! {response, Other}\ iff (Other == Num + 1) ↩
    ↪implies tt else ff
```

4.4.2 Two Erlang traces

To test two trace event sources together, I used the command shown in Listing 4.5. The resulting monitor runs and traces both the client and the server in Erlang, and tests them against the properties *client_sat* and *server_sat*, which they should satisfy, because they are both run with the correct configuration, as described in section 4.2. Figure 4.1 shows the output of the monitor. At the end, both the server, *sys1*, and the client, *sys2*, reach a satisfaction, as expected.

It should be noted that the output of the program tends to be about as long, or even longer than the output shown in Figure 4.1. Therefore, the output will not be shown for the rest of the test cases.

Listing 4.5: The make for the first test

```
1 make instrument args="[{'priv/case_study/server_sat.txt',␣trace_event_source,␣{↩
    ↪client_server_erlang,␣start,␣[server,␣success,␣nul]}},{'priv/case_study/↩
    ↪client_sat.txt',␣trace_event_source,␣{client_server_erlang,␣start,␣[client,␣↩
    ↪success,␣sys1]}}]" app-bin-dir="ebin"
```

4.4.3 Java via TCP

I used the command shown in Listing 4.6 to create a monitor that monitored the case studies in Erlang and Java at the same time. The Erlang case study, which acts as a server returning an incorrect response (Num) in this case, was monitored via tracing, but the Java case study, acting as a client in this case, was monitored via TCP over port 3456. As expected, the Erlang server sends an incorrect response to the client's request, and a violation is detected. This causes the Java client to receive an incorrect response, causing another violation, which is also detected.

Listing 4.6: The make for the second test

```
1 make instrument args="[{'priv/case_study/server_vio.txt',␣trace_event_source,␣{↩
    ↪client_server_erlang,␣start,␣[server,␣echo,␣nul]}},{'priv/case_study/client_sat.↩
    ↪txt',␣tcp_event_source,{3456}}]" app-bin-dir="ebin"
```

4.4.4 Reading a file

I used the command shown in Listing 4.7 to create a monitor that reads events from a file. The resulting monitor has only one event source, and it tests the events read against the property *server_sat*. The file to be read has only two lines, shown in Listing 4.8. After reading one line at a time, by sending `{read_line, 1}` messages to *sys1* twice, a violation is detected.

```
C:\detectEr\detecter-lite\ebin>erl -sname monitor
Eshell V8.2  (abort with ^G)
(monitor@Gvendurst)1> launcher:start().
[11/5/2017 23:15:25, INFO - <0.61.0> - main_mon:27] - Args for event source trace_event_source: {cli
ent_server_erlang,start,
                                        [server,success,null]}
[11/5/2017 23:15:25, INFO - <0.61.0> - main_mon:28] - EventSource: trace_event_source

[11/5/2017 23:15:25, INFO - <0.61.0> - main_mon:29] - Args: {client_server_erlang,start,[server,succ
ess,null]}

[11/5/2017 23:15:25, INFO - <0.61.0> - main_mon:30] - Property: #Fun<server_sattxt.0.18867451>

[11/5/2017 23:15:25, INFO - <0.61.0> - main_mon:31] - ProcsNames: []

Receiving events from trace
Args: {client_server_erlang,start,[server,success,null]}
[11/5/2017 23:15:25, INFO - <0.61.0> - main_mon:27] - Args for event source trace_event_source: {cli
ent_server_erlang,start,
                                        [client,success,sys1]}
[11/5/2017 23:15:25, INFO - <0.63.0> - trace_event_source:31] - Started <0.63.0>, monitoring process
: sup1. Waiting for mon_start
[11/5/2017 23:15:25, INFO - <0.61.0> - main_mon:28] - EventSource: trace_event_source

[11/5/2017 23:15:25, INFO - <0.64.0> - main_mon:49] - Starting monitor for sys1
[11/5/2017 23:15:25, INFO - <0.61.0> - main_mon:29] - Args: {client_server_erlang,start,[client,succ
ess,sys1]}

[11/5/2017 23:15:25, INFO - <0.64.0> - main_mon:50] - Started main monitor for processes/PIDs [].
[11/5/2017 23:15:25, INFO - <0.61.0> - main_mon:30] - Property: #Fun<client_sattxt.0.87598438>

[11/5/2017 23:15:25, INFO - <0.61.0> - main_mon:31] - ProcsNames: []

Receiving events from trace
Args: {client_server_erlang,start,[client,success,sys1]}
[11/5/2017 23:15:25, INFO - <0.65.0> - trace_event_source:31] - Started <0.65.0>, monitoring process
: sup2. Waiting for mon_start
[11/5/2017 23:15:25, INFO - <0.66.0> - main_mon:49] - Starting monitor for sys2
[11/5/2017 23:15:25, INFO - <0.66.0> - main_mon:50] - Started main monitor for processes/PIDs [].
ok
Running a server that returns the correct response (Num + 1)
Running a client that sends a correct request
(monitor@Gvendurst)2> Received message: {request,<0.65.0>,1}
(monitor@Gvendurst)2> [11/5/2017 23:15:25, INFO - <0.64.0> - main_mon:82] - Resolved procs [].
(monitor@Gvendurst)2> [11/5/2017 23:15:25, INFO - <0.66.0> - main_mon:82] - Resolved procs [].
(monitor@Gvendurst)2> [11/5/2017 23:15:26, INFO - <0.64.0> - main_mon:104] - Starting main monitor l
oop.
(monitor@Gvendurst)2> [11/5/2017 23:15:26, TRACE - <0.68.0> - formula:136] - mon_pos evaluating acti
on: {recv,<0.63.0>,{request,<0.65.0>,1}}.
(monitor@Gvendurst)2> [11/5/2017 23:15:26, TRACE - <0.68.0> - formula:136] - mon_pos evaluating acti
on: {send,<0.65.0>,{response,2}}.
(monitor@Gvendurst)2> [11/5/2017 23:15:26, DEBUG - <0.68.0> - formula:195] - mon_guard guard evaluat
ed to true.
(monitor@Gvendurst)2> [11/5/2017 23:15:26, DEBUG - <0.68.0> - formula:76] - mon_tt matched 'tt' acti
on.
(monitor@Gvendurst)2> [11/5/2017 23:15:26, INFO - <0.64.0> - main_mon:130] -


Monitor for sys1 received 'tt' - *** Satisfaction detected! ***


(monitor@Gvendurst)2> [11/5/2017 23:15:26, INFO - <0.66.0> - main_mon:104] - Starting main monitor l
oop.
(monitor@Gvendurst)2> [11/5/2017 23:15:26, TRACE - <0.69.0> - formula:136] - mon_pos evaluating acti
on: {send,sys1,{request,<0.65.0>,1}}.
(monitor@Gvendurst)2> [11/5/2017 23:15:26, TRACE - <0.69.0> - formula:136] - mon_pos evaluating acti
on: {recv,<0.65.0>,{response,2}}.
(monitor@Gvendurst)2> [11/5/2017 23:15:26, DEBUG - <0.69.0> - formula:195] - mon_guard guard evaluat
ed to true.
(monitor@Gvendurst)2> [11/5/2017 23:15:26, DEBUG - <0.69.0> - formula:76] - mon_tt matched 'tt' acti
on.
(monitor@Gvendurst)2> [11/5/2017 23:15:26, INFO - <0.66.0> - main_mon:130] -


Monitor for sys2 received 'tt' - *** Satisfaction detected! ***


(monitor@Gvendurst)2>
```

Figure 4.1: Output from the first test

Listing 4.7: The make for the third test

```
1 make instrument args="[{'priv/case_study/server_sat.txt',␣file_event_source,{'../priv/↩
    ↪case_study/fileTrace1.txt'}}]" app-bin-dir="ebin"
```

Listing 4.8: The file read in the third test

```
1 {recv, server, {request, client, 19}}.
2 {send, server, client, {response, 19}}.
```

4.4.5 Reading a long file

Finally, I created a monitor to read events from a very long file. Like in the previous test, the resulting monitor only has one event source, but this time the events read are tested against the property *server_vio*. The file to be read has 582 lines, the first 580 describe the server receiving requests and responding to them correctly (Num + 1), but at the end the server receives a request to which it responds incorrectly (Num). As expected, the monitor detects a violation from the last message.

4.4.6 Discussion

As the preceding test cases have shown, creating different kinds of event sources is as simple as changing the command line arguments during the instrumentation process. It is possible to trace multiple Erlang programs simultaneously, and it is possible to verify a Java application by changing the source code to send system events over TCP. Ideally it would be possible to verify Java applications without tampering with the source code at all, but this is a good first step.

It is also possible to read events from a file, which is a very useful feature. Sometimes it can be infeasible to monitor a system at runtime, due to environmental or time constraints on the system. Then it could be a better idea to log the system's events to a text file, and to use the file to verify the correctness of the system. Like listening for events via TCP, reading events from a text file is not dependent on the programming language of the system. Therefore, in theory, it is now possible for *detectEr* to verify the correctness of any system, regardless of programming language.

Chapter 5

Conclusions and future work

As the case study demonstrated, the event source has made *detectEr* a more versatile tool, by decoupling it from the Erlang tracing mechanism. The tool is now capable of reading events from files, and listening for events via TCP, and it is easier than ever to add support for other ways to generate events. It is also possible to have multiple event sources at the same time, which can be very useful when verifying concurrent systems. There is still room for improvement, though. For example, each event source only checks a single property, but systems usually have to satisfy more than one property. Therefore, it could be a good idea to enable each event source to check multiple properties simultaneously. That could be achieved simply by creating multiple monitor processes for each event source, but the arguments for the makefile would need to be altered to reflect that change.

Creating new event source modules is something that can make *detectEr* even more versatile. It could for example be a good idea to create an event source that can trace function calls of Erlang programs, and if that is successful, create a similar event source for other programming languages. As the Java case study shows, it is possible to use the event source to monitor systems in languages other than Erlang. In order to do so, one needs the right event source, and some connective tissue between the event source and the monitored system, like the tracker of the Java case study.

Acknowledgements

I would like to thank my supervisors: Luca Aceto and Anna Ingólfsdóttir, for their help and support during the work on this project. I would also like to thank Adrian Francalanza, for his guidance, and Marcel Kyas, for his honest feedback. And finally, I would like to thank Duncan Paul Attard and Ian Cassar for helping me understand *detectEr*. The work presented in this thesis was supported by the project "TheoFoMon: Theoretical Foundations for Monitorability" (grant number: 163406-051) of the Icelandic Research Fund.

Bibliography

[1] L. Lamport, "The computer science of concurrency: The early years", *Commun. ACM*, vol. 58, no. 6, pp. 71–76, May 2015, ISSN: 0001-0782. DOI: 10.1145/2771951.

[2] E. M. Clarke, W. Klieber, M. Novacek, and P. Zuliani, "Model checking and the state explosion problem", *Lecture notes in computer science*, vol. 7682, pp. 1–30, 2012.

[3] C. Baier and J.-P. Katoen, *Principles of model checking*. The MIT Press, 2008.

[4] M. Leucker and C. Schallhart, "A brief account of runtime verification", *The Journal of Logic and Algebraic Programming*, vol. 78, pp. 293–303, 5 2009.

[5] Y. Falcone, "You should better enforce than verify", in *Runtime Verification - First International Conference, RV 2010, St. Julians, Malta, November 1-4, 2010. Proceedings*, H. Barringer, Y. Falcone, B. Finkbeiner, K. Havelund, I. Lee, G. J. Pace, G. Rosu, O. Sokolsky, and N. Tillmann, Eds., ser. Lecture Notes in Computer Science, vol. 6418, Springer, 2010, pp. 89–105, ISBN: 978-3-642-16611-2. DOI: 10.1007/978-3-642-16612-9.

[6] J. Ligatti, L. Bauer, and D. Walker, "Run-time enforcement of nonsafety policies", *ACM Trans. Inf. Syst. Secur.*, vol. 12, no. 3, 19:1–19:41, 2009. DOI: 10.1145/1455526.1455532.

[7] F. B. Schneider, "Enforceable security policies", *ACM Trans. Inf. Syst. Secur.*, vol. 3, no. 1, pp. 30–50, 2000. DOI: 10.1145/353323.353382.

[8] D. P. Attard, I. Cassar, A. Francalanza, L. Aceto, and A. Ingólfsdóttir. (May 2016). A runtime monitoring tool for actor-based systems, [Online]. Available: http://icetcs.ru.is/theofomon/BettyBookSubmission.pdf (visited on 10/05/2017).

[9] (May 2017). Erlang programming language, [Online]. Available: https://www.erlang.org/ (visited on 10/05/2017).

[10] J. Armstrong, *Programming erlang: Software for a concurrent world*, Second Edition. Pragmatic Bookshelf, 2013, ISBN: 978-1-937785-53-6.

[11] A. Francalanza, L. Aceto, and A. Ingólfsdóttir, "On verifying hennessy-milner logic with recursion at runtime", in *Runtime Verification: 6th International Conference, RV 2015, Vienna, Austria, September 22-25, 2015. Proceedings*, E. Bartocci and R. Majumdar, Eds. Cham: Springer International Publishing, 2015, pp. 71–86, ISBN: 978-3-319-23820-3. DOI: 10.1007/978-3-319-23820-3_5.

[12] C. Colombo, G. J. Pace, and G. Schneider, "Larva — safer monitoring of real-time java programs (tool paper)", in *Proceedings of the 2009 Seventh IEEE International Conference on Software Engineering and Formal Methods*, ser. SEFM '09, Washington, DC, USA: IEEE Computer Society, 2009, pp. 33–37, ISBN: 978-0-7695-3870-9. DOI: 10.1109/SEFM.2009.13.

[13] ——, "Dynamic event-based runtime monitoring of real-time and contextual properties", in *Formal Methods for Industrial Critical Systems: 13th International Workshop, FMICS 2008, L'Aquila, Italy, September 15-16, 2008, Revised Selected Papers*, D. Cofer and A. Fantechi, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 135–149, ISBN: 978-3-642-03240-0. DOI: 10.1007/978-3-642-03240-0_13.

[14] C. Colombo, A. Francalanza, and R. Gatt, "Elarva: A monitoring tool for erlang", in *Runtime Verification: Second International Conference, RV 2011, San Francisco, CA, USA, September 27-30, 2011, Revised Selected Papers*, S. Khurshid and K. Sen, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 370–374, ISBN: 978-3-642-29860-8. DOI: 10.1007/978-3-642-29860-8_29.

[15] K. Havelund and G. Roşu, "An overview of the runtime verification tool Java PathExplorer", *Formal Methods in System Design*, vol. 24, no. 2, pp. 189–215, 2004, ISSN: 1572-8102. DOI: 10.1023/B:FORM.0000017721.39909.4b.

[16] L.-Å. Fredlund and H. Svensson, "McErlang: A model checker for a distributed functional programming language", in *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '07, Freiburg, Germany: ACM, 2007, pp. 125–136, ISBN: 978-1-59593-815-2. DOI: 10.1145/1291151.1291171.

[17] M. Sirjani, A. Movaghar, A. Shali, and F. S. de Boer, "Modeling and verification of reactive systems using Rebeca", *Fundam. Inf.*, vol. 63, no. 4, pp. 385–410, Jun. 2004, ISSN: 0169-2968. [Online]. Available: http://dl.acm.org/citation.cfm?id=1227079.1227084.

[18] G. Holzmann, *Spin model checker, the: Primer and reference manual*, 1st ed. Addison-Wesley Professional, 2003, ISBN: 0-321-22862-6.

[19] K. L. McMillan, *Symbolic model checking*. Kluwer Academic Publ, 1993.

[20] D. P. Attard. (May 2017). duncanatt / detecter-lite - bitbucket, [Online]. Available: https://bitbucket.org/duncanatt/detecter-lite (visited on 10/05/2017).

[21] G. Stefánsson. (May 2017). gudmundurste12 / detecter-monitor-java, [Online]. Available: https://bitbucket.org/gudmundurste12/detecter-monitor-java (visited on 10/05/2017).

[22] (May 2017). Erlang - the jinterface package, [Online]. Available: http://erlang.org/doc/apps/jinterface/jinterface_users_guide.html (visited on 10/05/2017).

School of Computer Science
Reykjavík University
Menntavegur 1
101 Reykjavík, Iceland
Tel. +354 599 6200
Fax +354 599 6201
www.ru.is