Annales Mathematicae et Informaticae
58 (2023) pp. 9–19
DOI: 10.33039/ami.2023.08.010
URL: https://ami.uni-eszterhazy.hu

Static analysis for safe software upgrade*

Dániel Ferenczi, Melinda Tóth

ELTE, Eötvös Loránd University, Budapest, Hungary
{danielf,toth_m}@inf.elte.hu

Abstract. Having applications accessible without downtime is no longer an exclusive requirement of mission-critical applications or traditional domains like communications. Running applications also require changes in the source code and upgrading live systems. Different approaches exist depending on the used technology. Systems implemented in Erlang can take the advantage of the underlying BEAM virtual machine and can be upgraded easily. However, source code has to be developed carefully once an upgrade is needed to not introduce run-time errors during the upgrade. We are developing a method to statically check the source code of Erlang applications for constructs that may lead to upgrading issues.

Keywords: Erlang, static analysis, software upgrade, hot code load

AMS Subject Classification: 68M15 Reliability, testing and fault tolerance of networks and computer systems

1. Introduction

With the ever-increasing use of e-commerce, even the owner of a simple webshop expects her site to operate without incidents all year round. Indeed, users expect high availability in general, and services for banking, commerce, news, and entertainment are expected to operate throughout the year with minimal disruptions. Running applications also require changes, however, the need to fix security issues or add new features and changes may happen at any time. An outage while a change is applied thus results in customer dissatisfaction, or even broken SLAs, and in the end, lost revenue. This presents a demand for seamless, "zero downtime" upgrades. The facilities for such upgrades depend on the stack chosen for

*Application Domain Specific Highly Reliable IT Solutions project has been implemented with the support provided from the National Research, Development and Innovation Fund of Hungary, financed under the Thematic Excellence Programme TKP2020-NKA-06 (National Challenges Subprogramme) funding scheme.

Submitted: July 28, 2023
Accepted: August 7, 2023
Published online: August 23, 2023

the development and operation of the application. These choices also determine what is possible during such an upgrade. Some tools [13] will launch new containers running a new version of the application in question, while slowly removing the previous release and possibly leading to a lost state [1]. Other tools allow for a more fine-grained approach, upgrading only the changed modules, and making state preservation possible [13, 16]. Errors in the use of these tools will however lead to unexpected behaviour or even downtime. This leads to the need to statically analyse the code responsible for the upgrade.

In this work, we demonstrate such a static analysis made for the language Erlang [5]. Erlang is a general-purpose, dynamically typed, functional programming language. It is designed to build distributed, concurrent, fault-tolerant computer systems. Originally designed for writing software in the telecommunications domain, the language and BEAM, the virtual machine it runs on, now see use as general tools for building fault-tolerant, concurrent, reliable software. Erlang was designed with high availability in mind. If we look at Joe Armstrong's, one of Erlang's designer's thesis [2], he notes that the option to "dynamically upgrade code" should be a feature of the language itself. This contrasts with other tools, that do not provide such feature and need additional tools to support code upgrades without downtime.

The goal of our work is to support safe software upgrades for the Erlang programming language by static program analysis. In this paper, we propose a method to identify servers which might crash or produce faulty behaviour after a live upgrade. We base our work on the static source code analyser and transformation tool, RefactorErl [4].

The rest of the paper is structured as follows. In Section 2 we illustrate the problem of upgrading through an example. In Section 3 we introduce RefactorErl and the algorithm used for pattern detection. Section 4 presents a solution for the problem in the context of *gen_server* behaviours. Sections 5 and 6 show possible ways to continue our research based on our existing foundation and related work respectively. We conclude our work in Section 7.

2. Problem statement

Erlang is distributed alongside the BEAM virtual machine, which compiles and runs Erlang code. Support for doing zero-downtime upgrades (in Erlang terms, "hot code loads") is built into the BEAM environment. It allows for state-preserving code changes on a module basis, in contrast to tools, like Kubernetes [9] that typically route connections to containers running new software versions. BEAM can keep two versions of the same module simultaneously. Code should thus be written with care, as expressions can be written in a way, that make code in the new version point to code in the previous version. Due to the limit in the number of versions BEAM can store, this will eventually lead to calls to versions no longer present in the virtual machine.

Avoiding these pitfalls requires care from the developer. For code meant to be

updated usually the intent is for such expressions to get updated as well during an upgrade and point to code present in the new version of the module.

The code snippet in Figure 1 presents a typical example of problematic code that cannot be upgraded. The module `srv` defines a server that can be started with the `srv:start/0` function call. The function spawns/creates a new process and registers it with the `server` name. In the new process the `init/1` function is evaluated that initialises and starts the server's tail-recursive loop function. This is a standard way to develop a server in Erlang. The `loop/1` process stores an integer value and an updater function in its state. The process handles `{num, ClientPid}` and `upgrade` messages. In the former case, it answers to the requester with the updated counter value (line 19). In the latter one, it simply applies a qualified recursive call (line 16) to handle the code change and upgrades itself.

Having function references in the loop's state is dangerous however, as these references might become outdated during module upgrades. A careful developer will ensure that these references are fully qualified - which results in function calls

```
1  -module(srv).
2  -export([start/0,init/1,loop/1,
3    adder/1,getNum/0]).
4
5  start() ->
6    Pid = spawn(srv, init, [0]),
7    register(server, Pid).
8
9  init(InitNum) ->
10   Adder = fun adder/1,
11   srv:loop({InitNum, Adder}).
12
13 loop({InitNum, Adder}) ->
14   receive
15     upgrade ->
16       srv:loop({InitNum, Adder});
17     {num, ClientPid} ->
18       NewNum = Adder(InitNum),
19       ClientPid ! {num, NewNum},
20       loop({NewNum, Adder})
21   end.
22
23 adder(N) ->
24   N + 42.
25
26 getNum() ->
27   ClientPid = self(),
28   server ! {num, ClientPid},
29   receive
30     {num, Num} -> Num
31   end.
```

```
1  -module(srv).
2  -export([start/0,init/1,loop/1,
3    adder/1,getNum/0]).
4
5  start() ->
6    Pid = spawn(srv, init, [0]),
7    register(server, Pid).
8
9  init(InitNum) ->
10   Adder = fun srv:adder/1,
11   srv:loop({InitNum, Adder}).
12
13 loop({InitNum, Adder}) ->
14   receive
15     upgrade ->
16       srv:loop({InitNum, Adder});
17     {num, ClientPid} ->
18       NewNum = Adder(InitNum),
19       ClientPid ! {num, NewNum},
20       loop({NewNum, Adder})
21   end.
22
23 adder(N) ->
24   N + 42.
25
26 getNum() ->
27   ClientPid = self(),
28   server ! {num, ClientPid},
29   receive
30     {num, Num} -> Num
31   end.
```

Figure 1. Example Erlang server function. Note the differences in line 10 on how the `adder` function is referred.

using the implementations in the latest version of the module. On the left-hand side snippet, the reference at line 10 will keep its original value of the `Adder` function through module upgrades, when execution reaches line 18. This will result in an error as at this point the `Adder` symbol will eventually reference a version of the function no longer present in the virtual machine.

The right-hand snippet presents a fix to this, by using a fully qualified reference to the function in the fun expression. In Erlang using a fully qualified function inside a fun expression has the `fun module:function/arity` syntax. Note the difference in line 10 of the example. This, when called will reference the latest version of the adder function.

Apart from the pattern in the example, other expressions will also make upgrades unsafe in a similar fashion: a reference to an initial version of a function added to the server loop's state during the loop's initialisation. This reference eventually may become obsolete as the applications are upgraded.

Overall we have identified the patterns listed in Figure 2 as being unsafe.

- `fun(Arg) -> Body end.`
- `fun function/Arity`
- `fun(Arg) -> module:function(Arg) end.`

Figure 2. List of unsafe patterns.

In Erlang terms, we can say, that explicit fun expressions and non-fully qualified implicit fun expressions are unsafe as they cannot be upgraded. These patterns can also be present directly in state of the server, e.g. `loop{InitNum, fun adder/1}`, which also lead to undesired behaviour. Our method detects this, direct use as well. Finally, it is important to note, that this risk is present all throughout the code, e.g. clauses in receive blocks may also change the state, and can add references that are unsafe for code upgrades. Our present work offers a detection method for such patterns in RefactorErl. Other unsafe patterns also exist and are the scope of our future work, details of which are described in Section 5.

3. Detection methodology

The mentioned upgrade-related issues can be detected before the execution of the code, and during the development phase using static analysis techniques. In our particular case, we want to analyse the contents of the state passed to server loops. If the state contains function calls, they should be fully qualified, otherwise, code upgrades during the operation of the program may lead to errors. For this analysis, we are using RefactorErl [4] as a framework.

3.1. RefactorErl

RefactorErl is an open-source static analysis framework for Erlang. RefactorErl allows for analysing source code represented and stored in its Semantic Program

Graph (SPG) [8]. The SPG is a three-layered rooted graph, containing lexical, syntactic and semantic information about the analysed source code. Once the source code is stored, the tool offers different options for analysis, through its querying interfaces. The semantic query language provides an expressive language for the programmer to analyse her code [15], but RefactorErl also offers the option to run queries through a graph representation of the program. The more accessible semantic queries are also implemented using these graph queries. In our work, we have developed our algorithm in the graph query language and then exposed it as a semantic query term for easier use.

RefactorErl also provides other functions: code transformations, static analysis of security-related issues [3] and features for code comprehension but for the present work, we will focus on targeted static analysis.

3.2. Detection algorithm

To find the code fragments that are not safe during an upgrade (such as the one presented on Figure 1), we need to identify the arguments of the server function (the state of the server) and point the local function references in the state. These patterns might not be visible at the point of the server call. The state argument could easily consist of variables having their values returned by other functions. Figure 3 shows an example for this: in line 3 we do not know the exact values of the state but the server loop is still initialised with an unsafe reference from line 6. In such cases considering solely syntactic information will not help us find the possible value of the state.

```
1 init(InitNum) ->
2   Adder = srv:getAdder(),
3   srv:loop({InitNum, Adder}).
4
5 getAdder() ->
6   A = fun(N) -> N + 42 end,
7   A.
```

Figure 3. Example of a function returning a reference to a fun expression.

To overcome this obstacle we rely on the intraprocedural dataflow analysis [14]. Using the first order dataflow relation we are able to calculate possible values of an expression. RefactorErl builds the Dataflow Graph during the initial analysis, and stores the direct dataflow edges in the Semantic Program Graph. The first order dataflow reaching relation is implemented on the top of this graph. It allows us to track information between functions, and provides a way to determine the possible values of the state for our analysis.

When using our detection function, the developer is expected to provide a function as an input for our query to check its arguments across every instance

where it is called. The algorithm will return the list of expressions where unsafe (as in Figure 2) arguments are defined.

Algorithm 1 Finding unsafe expressions in state.

```
1: function FIND(F)
2:     function_applications ← find_applications(F)
3:     for all application ∈ function_applications do
4:         argument_list ← get_arguments(application)
5:         expression_list ← get_subexpressions(argument_list)
6:         originating_exprs ← find_orig_exprs(expression_list)
7:         unsafe_exprs ← filter_to_unsafe(originating_exprs)
8:     end for
9: end function
```

Our detection algorithm finds these expressions as it is defined in Algorithm 1. At first, we gather all instances of application of our function. Our goal is to determine the 'safety' of the arguments for each application. Thus, for each application we gather the expressions used as arguments and find the originating expressions of these with dataflow reaching. Finally, we determine whether individual originating expressions are safe, and return them if they are not.

4. Analysing *gen_server* applications

Erlang distributions come bundled with a library called Open Telecom Platform. This library provides a construct called behaviour, which allows for abstracting away the complex, generic details of a pattern so that the developer only has to take care of implementing the specific parts of her application. It allows for code to follow common patterns which helps during the development life-cycle. OTP comes bundled with some built-in behaviours which can be suited for distributed applications: servers, state machines, event managers, and supervisors [6, 11]:

- gen_server: for implementing server applications
- gen_statem: for implementing state machines
- gen_event: for managing events and triggered actions
- supervisor: for adding fault tolerance

For the purposes of our research, it is worth looking at the *gen_server* behaviour, as server applications usually are implemented using it, rather than in a way akin to the example shown before. A developer implementing her server as a gen_server behaviour will have to write an Erlang module called in Erlang terms a *callback module*. This module needs to implement the behaviour's callback functions. The details of the implementation will of course depend on the

problem the developer is working on. The structure and purpose of a behaviours callback functions is fixed however. For example, a server can be started with the `gen_server:start_link/4` function call that triggers a call to the callback module's `init/1` function that returns the initialised state of the server. Sending synchronous messages can be done through `gen_server:call/2` function calls that trigger a call to a `handle_call/3` callback function that returns the reply to the message and the modified state.

As *gen_server* behaviours split the implementation code to separate functions that deal with initialisation and have their own code running between interface and callback functions our solution will not work out of the box. An easy solution would be to simply add the entirety of the OTP library to the RefactorErl database. This would allow our dataflow analysis to traverse all relevant code, however, this comes with the cost of having to analyse the OTP library itself unnecessarily, as it will not contain code adding unsafe functions to the server state.

4.1. Example

A cleaner, more efficient solution would be to look at the behaviour functions and their callbacks and determine the points where the state is set, use these points as a basis for our analysis. This is possible, as behaviours enforce a structure, the points where the state is set are defined. Figure 4 shows an abridged *gen_server* implementation of the example from Figure 1.

```
1  -module(srvg).
2  -export([start_link/0, get_number/1]).
3  -export([init/1, handle_call/1]).
4  -behaviour(gen_server).
5  start_link() ->
6      gen_server:start_link({local, srvg}, srvg, [], []).
7
8  get_number() ->
9     gen_server:call(srvg, get_num).
10
11 init([]) ->
12     Adder = fun srvg:adder/1,
13     InitNum = 0,
14     {ok, {Adder, InitNum}}.
15
16 handle_call(get_num, _From, {Adder, Num}) ->
17     NewNum = Adder(Num),
18     {reply, NewNum, {Adder, NewNum}};
19 ...
```

Figure 4. Example of a behaviour. Note the definition of a "safe" Adder in line 12.

This server can be started by calling the `srvg:start_link/0` function that calls the `gen_server:start_link/4` function. The first argument is the type and name

of the server. The second argument of the call (`srvg`) defines the callback module. The started server evaluates the `init` function from the `srvg` callback module. In our example, the `init` function at line 11 initialises the server with the state in line 14. The `get_number/0` function at line 8 forwards the `get_num` request to the server and it waits for the result. The behaviour eventually processes the request with the `handle_call/3` function in line 16, setting the new server state to the `{Adder, NewNum}` tuple in line 18, and sends the `NewNum` value back to the caller process. This value will be the return value of the `gen_server:call/2` function call in line 9.

4.2. Modified algorithm

Behaviours can set the state for example in the return expression of initialisation or callback functions. In general, the values worth inspecting are those returned by the callback functions required by the *gen_server* behaviour. In practice we would have to change the input part defined in line 2 of Algorithm 1: server state is no longer defined in the applications of a server loop, but in the return value of the behaviour's callback functions. Thus we expect the user to provide a behaviour implementation, where we look for unsafe patterns in the return value of the callback functions that are manipulating the state, such as `handle_call/3, handle_cast/2, handle_info/2, code_change/2` function definitions. The new state is usually the last element of the returned tuple and we can use dataflow reaching to calculate the possible values. Once we have these expressions, we can filter out the non qualified function references.

The modified algorithm is preseneted in Algorithm 2. These modifications will allow our method to be used for analysing unsafe fun expressions in *gen_server* implementations.

Algorithm 2 Finding unsafe expressions in *gen_server* states.

```
1: function FIND(M)
2:     callbacks ← find_callback_functions(M)
3:     for all callback ∈ callbacks do
4:         state ← get_server_state(callback)
5:         expression_list ← get_subexpressions(state)
6:         originating_exprs ← find_orig_exprs(expression_list)
7:         unsafe_exprs ← filter_to_unsafe(originating_exprs)
8:     end for
9: end function
```

5. Further work

In addition to unsafe fun expressions, other patterns might also introduce risk to upgrades and are worth inspecting. Upgrades involving a change in the state

structure are a clear example, where we could analyse if function applications still use the old state structure, or if there are references to elements removed from the state. This can be also examined in *gen_server* implementations, where state transformations for upgrades are handled by the `code_change` functions.

Analysing how modules depend on each other is another example. In order for upgrades to be safe in such a setup, upgrades have to respect the order of dependencies. Additionally, we could verify the existence of fully qualified self-references in server loops, which are required for upgrades in the first place.

Upon identification of further patterns, these additional checkers can be implemented using RefactorErl.

6. Related work

Apart from safe upgrades, code can be analysed for other properties that can present issues during operation. RefactorErl itself can be used to check the code for common vulnerabilities [3]. Ensuring safe upgrades is however a general problem, present across technologies.

Past research [1] has demonstrated challenges and solutions to upgrading distributed, multi-version systems and reasoning about their correctness, although their approach is not suited for preserving connection state.

HotSwap [16] presents a solution for software defined networks. Apart from upgrading without disruptions, the authors also ensure that rules, blacklists stay in effect by replaying events on the new version.

In the domain of modern container orchestration, Kustomize [10] includes tools for ensuring correct configurations for the popular orchestration system, Kubernetes [7]. These system's configuration is typically done in languages that lack type safety, and present a risk for updates when changing several configuration files.

Work has also been published as well on supporting zero-downtime releases from a kernel level [12] on different protocols to allow fast and frequent updates.

It would be worth investigating how general these solutions are, their caveats, and whether they can be applied to software stacks running Erlang.

7. Conclusion

In this work, we have looked at the importance of zero-downtime releases and summarised different approaches for achieving them. We presented how it is achieved in Erlang applications, and showed a potential cause for upgrade failures. To identify such problems before production we have extended the RefactorErl tool with a checker for unsafe use of local fun expressions in server loops in Erlang. We have also demonstrated how unsafe implementations can put upgrades at risk even in *gen_server* implementations. We also have shown a way to efficiently search for unsafe patterns when using behaviours.

Apart from local fun expressions in the state, other problems in the code might also impede safe upgrades. Investigating further unsafe patterns and methods for zero-downtime upgrades could be the topic of further studies, along with the analysis of other software stacks.

References

[1] S. Ajmani, B. Liskov, L. Shrira: *Modular software upgrades for distributed systems*, in: ECOOP 2006–Object-Oriented Programming: 20th European Conference, Nantes, France, July 3-7, 2006. Proceedings 20, Springer, 2006, pp. 452–476.

[2] J. Armstrong: *Making reliable distributed systems in the presence of software errors*, PhD thesis, 2003.

[3] B. Baranyai, I. Bozó, M. Tóth: *Supporting Secure Coding with RefactorErl*, Submitted to the ANNALES Universitatis Scientiarum Budapestinensis de Rolando Eötvös Nominatae Sectio Computatorica (2020).

[4] I. Bozó, D. Horpácsi, Z. Horváth, R. Kitlei, J. Köszegi, T. M., M. Tóth: *RefactorErl - Source Code Analysis and Refactoring in Erlang*, in: Proceedings of the 12th Symposium on Programming Languages and Software Tools, ISBN 978-9949-23-178-2, Tallin, Estonia, Oct. 2011, pp. 138–148.

[5] F. Cesarini, S. Thompson: *Erlang Programming: A Concurrent Approach to Software Development*, O'Reilly Media, 2009, isbn: 9780596555856.

[6] F. Cesarini, S. Vinoski: *Designing for scalability with Erlang/OTP: implement robust, fault-tolerant systems*, " O'Reilly Media, Inc.", 2016.

[7] B. Copy, M. Bräger, A. P. Koufidis, E. Piselli, I. P. Barreiro: *Integrating IoT Devices Into the CERN Control and Monitoring Platform*, in: Proc. ICALEPCS'19 (New York, NY, USA), International Conference on Accelerator and Large Experimental Physics Control Systems 17, JACoW Publishing, Geneva, Switzerland, Aug. 2020, pp. 1385–1388, isbn: 978-3-95450-209-7, doi: 10.18429/JACoW-ICALEPCS2019-WEPHA125.

[8] Z. Horváth, L. Lövei, T. Kozsik, R. Kitlei, A. N. Víg, T. Nagy, M. Tóth, R. Király: *Modeling semantic knowledge in Erlang for refactoring*, in: Knowledge Engineering: Principles and Techniques, Proceedings of the International Conference on Knowledge Engineering, Principles and Techniques, KEPT 2009, vol. 54(2009) Sp. Issue, Studia Universitatis Babeş-Bolyai, Series Informatica, Cluj-Napoca, Romania, July 2009, pp. 7–16.

[9] *Kubernetes Documentation*, Accessed: 2023-01-12, url: https://kubernetes.io/docs/home/.

[10] *Kustomize Documentation*, Accessed: 2023-01-12, url: https://kubectl.docs.kubernetes.io/references/kustomize/.

[11] M. Logan, E. Merritt, R. Carlsson: *Erlang and OTP in Action*, 1st, USA: Manning Publications Co., 2010, isbn: 1933988789.

[12] U. Naseer, L. Niccolini, U. Pant, A. Frindell, R. Dasineni, T. A. Benson: *Zero Downtime Release: Disruption-Free Load Balancing of a Multi-Billion User Website*, in: Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication, SIGCOMM '20, Virtual Event, USA: Association for Computing Machinery, 2020, pp. 529–541, isbn: 9781450379557, doi: 10.1145/3387514.3405885.

[13] I. Neamtiu, T. Dumitraş: *Cloud software upgrades: Challenges and opportunities*, in: 2011 International Workshop on the Maintenance and Evolution of Service-Oriented and Cloud-Based Systems, IEEE, 2011, pp. 1–10.

[14] M. Tóth, I. Bozó: *Static Analysis of Complex Software Systems Implemented in Erlang*, Central European Functional Programming Summer School – Fourth Summer School, CEFP 2011, Revisited Selected Lectures, Lecture Notes in Computer Science (LNCS), Vol. 7241, pp. 451-514, Springer-Verlag, ISSN: 0302-9743, 2012.

[15] M. Tóth, I. Bozó, J. Kőszegi, Z. Horváth: *Static Analysis Based Support for Program Comprehension in Erlang,* In Acta Electrotechnica et Informatica, Volume 11, Number 03, October 2011. Publisher: Versita, Warsaw, ISSN 1335-8243 (print), pages 3-10.

[16] L. Vanbever, J. Reich, T. Benson, N. Foster, J. Rexford: *HotSwap: Correct and Efficient Controller Upgrades for Software-Defined Networks*, in: Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking, HotSDN '13, Hong Kong, China: Association for Computing Machinery, 2013, pp. 133–138, isbn: 978-1-45032-178-5, doi: 10.1145/2491185.2491194.