Acta Cybernetica **27** (2025) 155–174.

Towards Correct Dependency Orders in Erlang Upgrades*

Daniel Ferenczi[ab] and Melinda Tóth[ac]

Abstract

Erlang tooling offers rich options to control the exact tasks to perform during an upgrade. This control aims to allow for zero-downtime upgrades. Upgrades affecting multiple dependent modules must reflect the dependency order in the upgrade's configuration, as an erroneous configuration results in unintended behavior, possibly even downtime. This paper presents two static analysis-based checkers for verifying module-related aspects of upgrades. In our first analysis, we compare the actual dependency order derived from the source code with that expressed in the upgrade configuration. We also analyze the code to find circular dependencies among its modules. These pose a problem during upgrades and are generally good practices to avoid. Both checkers present an argument in favor of using static analysis methods to define upgrade specifications.

Keywords: Erlang, upgrades, static analysis, upgrade safety, dependency order, RefactorErl

1 Introduction

Ensuring continuous operation of IT services is considered the norm in today's software environment. While this was typically a feature of safety-critical systems decades ago, today we can encounter it in many customer-facing applications: retail, banking, and entertainment. The need for this is reasonable considering the global nature of these services. These applications operate around the clock and are changed while running without a noticeable impact on the end user. The state-of-the-art tooling based on containers, serverless services, and other features offered

*Project no. TKP2021-NVA-29 has been implemented with the support provided by the Ministry of Culture and Innovation of Hungary from the National Research, Development and Innovation Fund, financed under the TKP2021-NVA funding scheme. Supported by the EKÖP-KDP-24 University Excellence Scholarship Program Cooperative Doctoral Program of the Ministry for Culture and Innovation from the source of the National Research, Development and Innovation Fund.

[a]Faculty of Informatics, Eötvös Loránd University, Budapest, Hungary
[b]E-mail: danielf@inf.elte.hu, ORCID: 0009-0002-0611-006X
[c]E-mail: tothmelinda@elte.hu, ORCID: 0000-0001-6300-7945

DOI: 10.14232/actacyb.312294

by hyperscalers standardizes some of the associated tasks. However, these solutions require additional effort from operational specialists, for example, load balancers and draining periods have to be configured, and the application state has to be preserved. These update schemes also work on high granularity: even small code changes require the replacement of a unit typically composed of the whole application binary. As larger applications contain multiple such units, upgrades with a broader scope will require care when determining which unit to change at each upgrade step.

Some languages and runtimes allow runtime changes on a small granularity. This allows for preserving the application state during software changes. Erlang is a language that allows for state-preserving code changes. In our work, we research the challenge of changing dependent code units in Erlang-based software stacks [2]. Erlang was developed with built-in features for concurrency, fault tolerance and continuous operation. This thins the software stack Erlang applications require. Consequently, developers maintaining them can create zero-downtime upgrades by solely using the features of the language and its runtime. With regard to upgrades, Erlang allows for live replacement of application modules and has upgrade-related tooling built-in into the language as well. The tooling, the small upgradeable units, and the runtime together allow the developer to reason about changes on a small granularity and declare disruption-free upgrades for her application.

Although upgradeable units are small, the problem of identifying how they depend on each other inside a given application is still applicable. Module dependency structure is also a good candidate for analysis: circular dependencies are best to avoid in general, and the Erlang release handling guidelines even advise against using them, as they might make safe upgrades impossible. We also support the detection of these with our checker, which will aid developers in structuring their code.

As the application modules' code meant to be upgraded and the upgrade specification are both expressed in Erlang, we based our work on existing static analysis tools to inspect the dependencies in the code.

This paper is structured as follows: in the next Section 2 we briefly present the Erlang language and the RefactorErl static analyzer on which we base our work. In Section 3 we introduce how upgrades work in Erlang applications. Section 4 exposes the specific problems we investigated and our developed checkers. We have dedicated a specific subsection for each analyzed problem: dependency discrepancies 4.1 and upgrades of dependency cycles 4.2. We evaluate our work in Section 5. Related work is presented in Section 6. Finally, concluding remarks and future work are described in Section 7.

2 Erlang and RefactorErl

Erlang is a dynamically typed functional programming language. It was developed at Ericsson for use in the telecommunications domain. It contains in-language features for developing highly scalable, fault-tolerant distributed software. These fea-

tures are provided by the runtime and the standard libraries included with Erlang distributions. This contrasts with other languages that require the introduction of other components into the software stack to allow for fault-tolerance or disruption-free code changes. Bundled tooling includes software for defining and managing upgrades on a fine-grained level to ensure disruption-free upgrades. These tools are used to define applications, releases (entities composed of multiple Erlang applications), and respective upgrade files, `appup` and `relup`. These upgrade files are interpreted by the Erlang runtime's Release Handler [10]. The requirements for reliable, upgradeable software have become more general since Erlang's first release. By now, the language has been adopted across several other domains: banking, instant messaging, cloud services [3, 4].

RefactorErl [1] is a static source code analyzer for Erlang that also supports code comprehension and refactoring. It is available for Linux, macOS and Windows and can be used through IDE integration, the command line, or a web interface. The tool analyzes loaded code, generates its abstract syntax tree, and enhances it with output from different analyzers: function, data-flow, etc. The resulting Semantic Program Graph (SPG) [8] allows easy analysis of the loaded program through a query language [17]. It is distributed with several built-in checkers for inspecting OWASP vulnerabilities, dependency structure, and dynamic function calls. It offers a rich framework for semantic analysis, including a rich query language. These features allow the user to develop their own static analyzers. As RefactorErl is also open-source, even more elaborate checkers can be developed and integrated into the tool. Given its features and extensibility, we chose it as our tool to implement our code checkers.

3 Upgrading dependent Modules in Erlang

Erlang source files (modules) may contain references to functions exported in other modules. We say that module `a` depends on module `b` if there is a call in `a` to a function in `b`. We call a module that has dependencies a dependent module. We represent this relationship with an arrow pointing from `a` to `b`: $\mathtt{a} \rightarrow \mathtt{b}$. In this context `a` is a dependent module. Dependency relationships can consequently be represented using directed graphs, and we can analyze the dependencies of application by inspecting such graphs.

Figure 1 shows a simple dependency relationship between modules. Module `a` depends on module `b`, which in turn depends on modules `c` and `d`.

The order upon which modules depend on each other is important during a release's upgrade cycle: as complex upgrades involve changes in multiple modules, if these depend on each other, their dependency has to be reflected in the upgrade steps as well. This is required as the application runs and function calls can happen during the upgrade process. To this result, a developer has to ensure that the version of the dependent module is aligned with that of the dependency in periods when calls can be made from the dependent to the dependency. A call from a different version could result in the dependent assuming a different interface for the

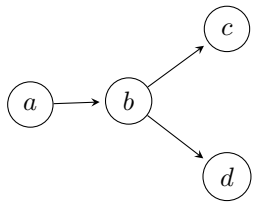


Figure 1: A simple dependency relationship between 4 modules represented as a graph

function in the dependency than what is actually implemented. To solve this, a safe upgrade procedure must ensure that running dependent modules are compatible with their dependencies' interfaces as these are changed.

When using the standard Erlang tooling for managing upgrades, the steps for performing the upgrade are declared in an `appup`, *application upgrade* file by the developer and are specific to the application that is updated. These files contain high-level instructions for declaring the module-specific actions that are to be performed during the application's upgrade. These actions offer control over whether modules are suspended while changed, added, or removed when upgrading the application to a new version or downgrading to a previous one. As a release may consist of multiple applications, `appup` files are combined into a `relup`, *release upgrade* file that must contain lower-level instructions on how to perform the upgrade. These files contain upgrade and downgrade instructions to support changes to different versions. As instructions are executed sequentially, their order must be aligned with the dependency relation of the modules and the interoperability between release versions. The structure of these files is illustrated in Figure 2. In the tuple describing the upgrade `Vsn` refers to the version to which we want to upgrade or downgrade. The second element of the tuple lists the versions we can upgrade from `Vsn` and the required set of instructions for the given upgrade. The last element of the tuple lists similarly downgrades paths and instructions.

```
{Vsn,
  [{UpFromVsn, Instructions}, ...],
  [{DownToVsn, Instructions}, ...]}.
```

Figure 2: Structure of `appup` and `relup` files

`relup` files are interpreted by the Erlang Release Handler and must only contain low-level instructions for the upgrade's definition. Low-level instructions differ from higher-level ones in that they offer control of the lifecycle of running processes, including their suspension, transformation of their state or synchronization of Erlang nodes. `relup` files are typically generated from `appup` files with the help of the release-related tooling offered by Erlang. This automatic conversion to `relup` files assumes however backward compatibility of new modules when determining

the order of module changes. As the interoperability of modules between versions might differ from that assumed by the Release Handler, `relup` files may also be written manually.

To account for both manual and automated workflows, we compare the dependencies derived from the source files with those implicitly expressed in the `relup` files. An example of an `appup` and its derived `relup` file can be seen in Figure 3. The sample's first section, from lines 2 to 11, shows high-level instructions for defining a release. This block includes the identifier of the released version in line 2, and two lists, from lines 3 to 6, and from lines 7 to 10. These lists allow for listing upgrade and downgrade paths respectively, following the structure presented in Figure 2. In our example, we declare the the rules for upgrading from version 1.0 to 1.1, and for downgrading from version 1.1 to version 1.0. Both blocks support the declaration of multiple paths, so, for example we could define the instructions to upgrade from version 0.9 as well.

The second section, from lines 14 to 28, shows a corresponding `relup` file. Although the structure is the same as that of the `appup` file, the commands defined must be low-level instructions which are executed by the Erlang Release Handler. Details of the example are described in Section 4.1.1.

Defining the steps necessary for an upgrade is a manual, error-prone process that requires a thorough understanding of the application source code and the dependency relations within. It is also unsafe, as `appup` or `relup` instructions inconsistent with the actual dependency relationship can result in errors or even temporary failures which are hard to debug. A developer declaring the upgrade instructions would need to review the dependency relationship of the affected modules, and in case of an inconsistency either redefine the upgrade instructions or change the source of the new release. As an incorrect manual analysis can consequently lead to either a failed upgrade or unnecessary changes, the developer would benefit of static checkers that contrast upgrade steps with the dependency structure.

In order to support the otherwise unsafe task of declaring upgrade definitions, we extended RefactorErl to analyze `relup` files and contrast the specified upgrade steps with the actual dependencies of the application. Although existing tools, such as erlup[1], relflow[2] or the appup plugin for the rebar3 build tool[3] support the generation of `appup` or `relup` files, they work by assuming specific code structures and backward compatibility and do not support validation of custom `relup` files against the actual dependency relationship. Our work is novel in the approach to verifying custom release definitions using effective module relationships. Our main contributions are as follows:

- The development of new features for RefactorErl, to retrieve upgrade-related information from `relup` files
- The development of two checkers for RefactorErl that use existing dependency-related analysis along with the one developed for upgrade specifications

[1] https://github.com/soranoba/erlup
[2] https://github.com/RJ/relflow
[3] https://github.com/lrascao/rebar3_appup_plugin

```
%% appup file for application release_tst
{"1.1",
  [{"1.0", [
    {load_module, depmod},
    {update, servermod, [depmod]}
  ]}],
  [{"1.0", [
    {load_module, depmod},
    {update, servermod, [depmod]}
  ]}]
}.

%% relup file for release consisting of app release_tst
{"1.1",
 [{"1.0",[],
   [{load_object_code,{release_tst,"1.1",[servermod,depmod]}},
    point_of_no_return,
    {suspend,[servermod]},
    {load,{depmod,brutal_purge,brutal_purge}},
    {load,{servermod,brutal_purge,brutal_purge}},
    {resume,[servermod]}]}],
 [{"1.0",[],
   [{load_object_code,{release_tst,"1.0",[servermod,depmod]}},
    point_of_no_return,
    {suspend,[servermod]},
    {load,{servermod,brutal_purge,brutal_purge}},
    {load,{depmod,brutal_purge,brutal_purge}},
    {resume,[servermod]}]}]}.
```

Figure 3: Example of `appup` and `relup` files

4 Supporting Correct Release Definitions

In the following subsections, we present the checkers that we have developed for detecting instruction order-related problems in `relup` files and recognizing updates of circular dependencies. They present the details of their respective domains that define the goals of our analyzer.

4.1 Discrepancy Detection in Upgrade Definitions

Our research aims to verify whether the dependency order expressed in `relup` files is consistent with the actual dependency of the modules. We begin by discussing the details of `relup` files relevant to our checker and how RefactorErl can support

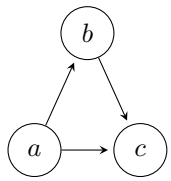
our analysis. We continue with the objectives, implementation, and limitations of our checker algorithm.

4.1.1 Problem Description

To understand how dependency order can be taken into account during upgrades, we can look at the example in Figure 3. In the example's `appup` file, we declared an upgrade from version 1.0 to 1.1 and a downgrade from version 1.1 to 1.0 respectively at lines 3 and 7. Specifically, we tell the Release Handler to load the newer version of `depmod` and update module `servermod`, which depends on `depmod`. The dependency relation is declared in the lists in lines 5 and 9. If we look at the list of instructions, both `load_module` and `update` atoms declare code changes. The difference lies in that `update` takes care of temporarily suspending processes running the target module, and transforming the internal state of the running process if the new version requires it. These additional operations allow for zero-downtime upgrades. `servermod` being a server implementation requires `update` for its code upgrade.

The generated `relup` file is of a similar structure: it contains first the upgrade and then the downgrade instructions. The set of instructions can only contain however lower-level operations executed by Erlang's Release Handler. Without going into detail, we can observe how the dependency relation results in `depmod` being changed before the dependent `servermod` module in lines 19 and 20 for the upgrade, and 26 and 27 for the downgrade. In these files, we are looking for `suspend`, `load` and `resume` instructions to ensure that these dependencies are updated before dependent modules.

The actual dependencies of an application are expressed as function calls in the Erlang source files. For analyzing and retrieving them, we rely on RefactorErl's features to inspect module dependencies. These features allow us to list the set of modules each dependent module depends on. Figure 4 shows an example of how RefactorErl generates the text representation of dependency relationships within an application.




```
c -> []
b -> [c]
a -> [c, b]
```

Figure 4: An example for dependent modules (Left). RefactorErl's textual representation of the dependencies (Right).

4.1.2 Detection Methodology

Our task is to determine if the upgrade steps declared in `relup` files are aligned with the actual dependencies of the application's modules. A *fitting* upgrade defini-

tion ensures that dependent modules use dependencies of the corresponding release version. If the versions between the dependencies are not aligned, modules might attempt to use non-existing functions from their dependencies. Using implementations from other versions can also be dangerous if they contain side effects. An example of `relup` instructions that can result in a runtime problem is shown in Figure 5.

```
...
    {load,{a,brutal_purge,brutal_purge}},
    {load,{b,brutal_purge,brutal_purge}},
...
```

Figure 5: A simple sequence of loaded modules

Here, assuming that module `a` depends on module `b`, we load the new version of the dependent before that of its dependency. As the load instruction simply replaces the running code without suspending processes, for a brief time window, between the two steps, code in module `a` can call functions in module `b` that do not yet exist. If we load the dependency before the dependent, the period between the two steps will allow `a` to attempt to use functions in `b` that were present in the previous version. If `b`'s new release is backward compatible with the old one, this will not result in a runtime error.

If the dependency's new version is not backward compatible, we need to ensure that calls only happen between modules of the same release version. This can be achieved by suspending the dependent module and replacing it and its dependencies during the suspension. Once all affected modules are replaced, the dependent module can be resumed. This process ensures that all affected modules are of compatible versions during active periods of the dependent module. Although execution of code halts during suspension, this still does not cause a disturbance in the application's availability, as the Erlang runtime will take care of processing any requests on the dependent once it is running again. An example of loading changing code during the dependent's suspension can be seen in Figure 6.

```
...
    {suspend,[a]},
    {load,{b,brutal_purge,brutal_purge}},
    {load,{a,brutal_purge,brutal_purge}},
    {resume,[a]}
...
```

Figure 6: Loading a module during suspension

In our research, we look at how dependencies are updated concerning the suspension of their dependents. In terms of the instructions in a `relup` file, a module's suspension period is the set of instructions between the module's `suspend` and `resume` instructions. An upgrade can be either a `load` instruction by itself or surrounded by a pair of `suspend` and `resume` instructions. Suspension periods can be

nested, as dependencies might also require suspension when updating them or their own set of dependencies.

Our work does not consider the different versions of a module across releases and hence we do not attempt to reason about interface compatibility of dependencies and the flexibility this offers. However, if a dependent is updated while suspended along with its dependencies, we can argue about the correctness of the order of instructions inside the `relup` file describing the update. Therefore, we analyze upgrade sequences where dependent modules are suspended and identify the update of a dependency with its `load` instruction.

We can summarize our goals with the following rules:

- A dependency does not have to be loaded if the dependent is suspended
- A dependency must only be loaded during the suspension of its dependent

A release does not have to include changes for all source modules, and the lack of change in a module does not impact the reliability of the software. With regard to the second observation, if a dependency were to be upgraded outside the suspension of the dependent, there could be room for discrepancies between what the dependent expects and the source of the dependency. Consequently, we assume that in releases where a dependent is suspended, changed dependencies are to be modified during the dependent's suspension period. To do so, we iterate through the list of dependency relationships generated by RefactorErl (see the example in Figure 4), identifying suspension periods of dependents and verifying whether dependencies are updated exclusively in these segments.

4.1.3 Algorithm

Our algorithm for detecting issues in `relup` files is presented in Algorithm 1.

The algorithm receives as input the `relup` file and the dependency relationships generated by RefactorErl and presented in Figure 4. In line 1 we retrieve the set of release definitions from the `Relup` file. This includes the set of instructions for both upgrade and downgrade releases. Recall from Figure 3 that a single file may contain multiple instruction lists, one for each upgrade and downgrade path. To group our results on a release definition basis, we iterate through these sets in line 2, and through the dependents in a nested loop in line 3. Next, we iterate through the individual instructions of the current release definition. For each instruction, we are interested in whether it affects the suspension state of the actual dependent, or if it relates to a dependency of the dependent. Throughout the loop, in line 6 we observe whether the dependent is currently suspended. In line 7 we verify if the current instruction relates to an update of a dependency. If so, the dependency is upgraded outside the suspension period of the dependent, and we add the instruction along with the name of the dependent module and the version identifier to a list in line 8. Storing the name and version is important so that the developer can find the instruction in the `relup` file more easily. The loops will perform the same analysis through the different sets of instructions described in the `relup` file. Finally, we return the list of unsafe instructions.

Algorithm 1 Finding discrepancies between `relup` instructions and actual dependencies

Funct FindUpdateDiscrepancy(*Relup*, *dependents*)

```
1:  Release Definitions ← Relup
2:  for all Release Definition ∈ Release Definitions do
3:    for all dependent ∈ dependents do
4:      Dependencies are determined along with dependent
5:      for all Instruction ∈ Release Definition do
6:        IsdependentSuspended is determined based on dependent and processed
          Instruction
7:        if not IsdependentSuspended and Instruction updates a Dependency of
          dependent then
8:          Store Instruction and dependent pair in UnsafeInstructionList
9:        end if
10:     end for
11:   end for
12: end for
13: return UnsafeInstructionList
```

A developer should treat this list as a warning, as in practice she knows best if a module can be changed while the ones using it are still running.

4.1.4 Limitations

As mentioned, our checker does not take into account the interoperability of modules across releases. Backward compatibility allows flexibility in organizing upgrade instructions. Therefore, our approach overapproximates and we could produce a more exact analysis by taking into account actual changes in the source code.

Another limitation is present when analyzing modules that have unrelated dependents. Figure 7 presents such an example.

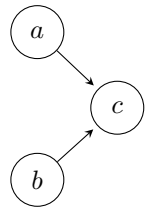


Figure 7: Multiple independent modules (`a` and `b`) depending on a single dependency (`c`)

In such scenarios, it is difficult to argue in which dependent's suspension period should the dependency be upgraded. Again, the correct way to upgrade such an application depends on the actual details in the source code.

4.2 Circular Dependency Detection

Circular dependencies amongst modules may also put upgrades at risk. We look at the information present in `relup` files and RefactorErl reports and identify the exact patterns that we wish to detect. We follow by presenting an algorithm for this purpose and discuss opportunities for improvement.

4.2.1 Problem Description

As established in Section 3, two modules depend on each other if one module's code calls functions implemented in other modules. Module `a` and `b` depend on each other if both `a` → `b` and `b` → `a` hold true. A circular dependency between two modules might be either direct, when the modules call each other's functions explicitly, or indirect if there are additional modules in the dependency circle. Examples of these are presented in Figure 8. A code sample with two circularly dependent modules is shown in Figure 9. In the examples modules `a` and `b` directly depend on each other.

Figure 8: Direct (left) and indirect (right) circular dependencies

The difficulties in determining the correct order for defining an upgrade are noted in the Release Handling Section of the Erlang manual. In most cases, it is best to avoid them altogether for code meant to be upgraded. For this checker, we also rely on the graph analysis feature of RefactorErl. For the analysis, we use a graph model of the dependency structure, where the modules will be the nodes, and the dependency relations the edges. Our task will be to determine if such a graph, excluding modules not being subject to an upgrade, has a topological ordering. We base our work on RefactorErl's features for analyzing dependency graphs. To retrieve the list of modules changed in a release from a `relup` file, as in our work presented in Section 4.1.

```
%% code of module a
-module(a).
-export([sum/0, number/0]).

sum() ->
  b:number().
number() ->
  42.

%% code of module b
-module(b).
-export([number/0]).

number() ->
  a:number().
```

Figure 9: Sample code with two modules depending on each other

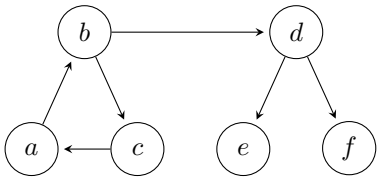
4.2.2 Detection Methodology

Our checker aims to warn developers if their `relup` contains `load` instructions for modules that are part of a dependency cycle. We do not wish to raise warnings because of the general presence of a cycle, as in such an application it is still possible to define upgrades of modules that are not part of any cycle.

However, we do want to raise a warning if a module is part of a cycle, as reasoning about the safe way to release a change would depend on the code. We present these scenarios in Figure 10. The application consists of six modules, three of which, `a`, `b` and `c` depend on each other. The top instructions in the `relup` file relate to upgrading modules `a` and `b` and could as a consequence be unsafe, as there is no clear dependency order by which the upgrade should be performed. In contrast, the `load` instructions for modules `f` and `d` (assuming that `f` is backward compatible) are safe as a dependency order can be determined from the graph. An application might also contain multiple cycles. As a result, the set of all cycles should be an input of our checker. As output, we expect those *load* instructions that relate to modules present in circular dependencies.

RefactorErl already provides a method for retrieving all dependency cycles present in an application. This functionality also detects dynamic function calls and includes them in the dependency graph. In dynamic function calls, called modules are not explicitly invoked, but rather the target module is referred to using a variable. The variable can be defined in other parts of the source code, making manual dependency analysis more difficult. An example of a module using a dynamic function call is presented in Figure 11. This code snippet shows a call retrieving the name of module `b` in line 5, and the invocation of `b`'s `my_function` in line 6. RefactorErl's dependency checker can detect dynamic dependencies through data-flow analysis [16, 7]. Detecting dependencies stemming from dynamic function

calls increases our checkers precision, as we can find dependency relationships typically only detectable during runtime. We base our work on RefactorErls data-flow analysis-based dependency checker.




```
...
%% possibly unsafe instruction sequence
{load,{a,brutal_purge,brutal_purge}},
{load,{b,brutal_purge,brutal_purge}},
...
%% safe instruction sequence
{load,{f,brutal_purge,brutal_purge}},
{load,{d,brutal_purge,brutal_purge}},
...
```

Figure 10: The dependency relationship of six modules contains a cycle (left). The release instructions include changes to the modules not present in the circular relationship (right)

```
1  -module(a).
2  -export([fun/0]).
3
4  fun() ->
5    Mod = get_mod(),
6    Mod:my_function().
7
8  get_mod() ->
9    b.
```

Figure 11: A dynamic function call

4.2.3 Algorithm

Our algorithm for detecting changes in dependency cycles is presented in Algorithm 2.

The algorithm receives the `relup` contents and set of dependencies as input. In line 1 we retrieve all dependency cycles from the dependencies. Next, in line 2 we gather the release definitions from the Relup file. We start iterating through these definitions in line 3, and their individual instructions in line 4. We inspect each instruction to see if it `load`s a module present in any of the cycles identified previously. If the instruction relates to such a module, we store it in a list of unsafe instructions in line 7, together with an identifier of the release definition so that the developer can place the problematic instruction more easily. Finally, we return the List of Unsafe Instructions.

Algorithm 2 Finding changes in dependency cycles expressed in *relup* instructions

Funct FindChangeInCycle(*Relup*, *Dependencies*)

```
1: Dependency Cycles ← dependency cycles from Dependencies
2: Release Definitions ← Relup
3: for all Release Definition ∈ Release Definitions do
4:   for all Instruction ∈ Release Definition do
5:     Changed Module ← Instruction
6:     if Changed Module ∈ Dependency Cycles then
7:       Store Instruction in UnsafeInstructionList
8:     end if
9:   end for
10: end for
11: return UnsafeInstructionList
```

4.2.4 Limitations

RefactorErl provides a solid basis for retrieving all dependency cycles present in the application. There exists the possibility of safe upgrades of dependency cycles if the affected modules are backward compatible. Therefore, our method might mark instructions as unsafe that are safe in practice, but to be more precise we would have to be aware of implementation details of the application being analyzed.

5 Evaluation

`appup` and `relup` files are not typically put under version control and published. Regardless, we aimed to assess the value of our checkers by inspecting the dependency structure of publicly available sources, made available on GitHub[4]. We analyzed 5 popular Erlang applications from several domains: instant messaging (MongooseIM[5]), MQTT (emqx[6]), web servers (Cowboy[7], Yaws[8]) and databases (couchdb[9]).

For our evaluation, we used RefactorErl's same dependency analysis features applied in the checkers we developed. They identified dependency relations are include dynamic dependencies in separate rows. These are determined with data-flow analysis and include dynamic function calls and invocation of the `apply` function that allows calling functions set as its arguments. The details and limitations of combining dependency and static analysis was presented by the authors formerly [7].

[4] https://github.com
[5] https://github.com/esl/MongooseIM
[6] https://github.com/emqx/emqx
[7] https://github.com/ninenines/cowboy
[8] https://erlyaws.github.io
[9] https://couchdb.apache.org

All analyzed applications are designed for implementing highly scalable services, and fault-tolerant services, where the operator can expect to perform disruption-free upgrades. For this task, they would have to create the `appup` or `relup` files necessary for their release tooling. In Table 5 we show the number of dependent modules these projects have, the number modules taking part in a dependency cycle, the highest number of dependencies a module has and whether this module and its dependencies has changed in the latest minor release. For contrast, we have also made these measurements counting dynamic dependencies as well.

Table 1: Dependency complexity in six popular projects: couchdb, MongooseIM, Cowboy, emqx and Yaws

Metric	couchdb	MongooseIM	Cowboy	emqx	Yaws
# of Dependents	378	496	18	115	35
# of Dependents (including Dynamic Dependencies)	383	501	20	117	37
# of Dependents in Dependency Cycles	167	83	2	61	15
# of Dependents in Dependency Cycles (including Dynamic Dependencies)	224	257	14	68	19
Highest Number of Dependencies	31	50	12	35	32
Latest Update Affects Most dependent Modules	Yes	Yes	Yes	Yes	Yes

As the table shows, all releases contain dependency cycles that make reasoning about release correctness more difficult. Additionally, about half of the dependent modules are also present in a dependency cycle if we take into account dynamic dependencies. Thus, including dynamic dependencies leads to a significant difference in the number of modules present in cycles. This shows the value of applying a broader set of static analysis techniques not only for planning upgrades, but for analyzing and improving code structure as well. Changing code of such complexity manually would be unsafe to manage, consequently we find that our tools would help with these tasks, especially with upgrades where a large number of dependencies is changed. Finally, we have found that all projects had their most dependent module have its dependencies changed since the last minor release.

6 Related work

Circular dependencies. Li and Thompson in their previous research [9] have analyzed the issue of circular dependencies in Erlang as part of their work on the refactoring tool, Wrangler. The authors' analyzer focuses on refactoring problematic patterns into clean code. However, it does not analyze upgrade specifications and can not reason about upgrade safety. Also, Wrangler does not include dynamic dependencies in its cycle analysis and does not feature analysis of `relup` files.

Upgrade safety. Naseer, Noccolini, Pain, Frindell, Dasineni and Benson have researched upgrade safety emphasizing runtime facets typically unrelated to an application's implementation language, like connection migration [13] between application versions. Being able to migrate connection is important, as reestablishing them would not only impact the user, but perhaps even require an unavailable amount of resources. Erlang is singular in the regard, that its runtime provides facilities for handling state preservation during code changes, without the need for introducing new tools or bespoke solutions.

Static analysis. Tools to support schema changes in backing databases have been researched by Maule, Andy and Emmerich [11]. The authors developed an approach for verifying whether a database schema change is consistent with the application's source code, improving on existing string-based checkers with static analysis. They argue for improving the accuracy of impact analysis by introducing further methods from static analysis. Meurice, Nagy and Cleve in their work [12] used static analysis as well to locate source code affected by database schema changes. They also assess whether a given change affects the developed application. It would be worth to investigate whether other upgrade-related static properties can be defined for Erlang using RefactorErl.

Microservice changes. Sampiao, Kadiyala, Hu, Steinbacher, Erwin, Rosa, Beschastnikh and Rubin have investigated challenges with regards to support upgrades in running microservice systems [15]. The research proposes modelling the software's evolution by analyzing static and dynamic information obtained from the system. Such models would help developers design their upgrade schemes and plan the evolution of their software in a consistent manner. Erlang developers design the scale, distribution and upgrades of their application in the same codebase. Our work could be extended by researching a broader set of changes between application releases.

RefactorErl. Tóth and Bozó have used RefactorErl for analyzing other static properties of source code, including the presence of common vulnerabilities [18]. These checkers can be used to assess the security of new releases. RefactorErl's database and queries can be also used to verify further upgrade-related properties and data-flow analysis allows inspecting behavior that could typically be only observed during execution.

Code upgrades and downtime. Neamtiu and Dumitra analyzed the relationship between upgrades and downtime [14]. The authors look at challenges across the stack: database schema migrations, infrastructure changes, mixed-version race conditions and protocol changes. They highlight the value of upgrade schemes that provide more control than rolling upgrades and of expressing the details of an upgrade explicitly. Erlang allows for expressing the details of an upgrade and the application's operation in it's language. This also allows for using existing static analysis methods for inspecting the details of an upgrade and is the motivation of our current and future work.

Safe Upgrades for Erlang Software. In our previous work, we have also analyzed other conditions for upgrading Erlang software without disruptions. For example, code meant to be upgraded must consistently call functions that are still present in the runtime. State transitions constitute another example: Erlang allows for changing a running application's state during its upgrade, but it is the developer's responsibility to change and use the state consistently. In previous research, we looked into these two problems and identified several coding patterns that would disrupt safe upgrades. We have also implemented code checkers based on RefactorErl to aid developers write upgradable software.

We began by researching [5] if applications contain any references to functions that would expire as the code is upgraded. The Erlang runtime only holds two versions of a given runtime at the same time. As consequence, local function references that remain unchanged during upgrades are unsafe as they become obsolete. By using fully qualified references, the functions used will be of the module version loaded last. Our checker helps the developer identify places where fully qualified references should be used.

In our second work [6] we investigated if state uses in a new application version are consistent with the state transformations performed during code changes. Erlang allows the developer to modify their application's state during an upgrade. To this effect they must implement `code_change` functions that specify a state transformation logic for the different upgrade paths. Researching additional unsafe patterns and a generic approach to support upgrade safety are further areas worth exploring.

7 Conclusions and Future Work

Erlang offers the tools necessary to create application releases with fine-grained instructions to ensure disruption-free upgrades of modules. To achieve this, code has to be structured in an upgradeable manner, and the release's descriptor file should also reflect this structure correctly. We researched how specific upgrade-related instructions should be ordered to be in line with the actual code structure, identified two problem categories, and developed checkers for them using the RefactorErl framework. Our first checker identifies if dependencies are upgraded during their dependent's suspension period. This checker does not cover the flexibility

that interfaces compatible between releases would allow. For example, assuming that the new release of a dependency is capable of receiving new calls, its new version can be `load`ed before we `load` the new version of the dependent, without requiring any sort of suspension. Our research can be extended in two steps: analyzing if modules are `load`ed in the correct order assuming that their interface changes are backward compatible; and analyzing actual backward compatibility as well between code releases. Of course, interface compatibility would not guarantee a well-working application. Upgrades can introduce domain-specific discrepancies into application code that retain interface-compatibility but will result in runtime problems. Such issues are hard to detect with static code analysis, and thus remain outside the scope of our work.

Our second checker investigates if changed modules are part of dependency cycles. Upgrading such structures is not recommended, as it is difficult to reason about the correct instruction order to implement an upgrade. Our analysis' precision can again be improved by taking into account the interoperability between code releases.

In conclusion, our research offers two checkers for developers to evaluate the correctness of their upgrade definitions and covers the directions to improve this analysis.

References

[1] Bozó, I., Horpácsi, D., Horváth, Z., Kitlei, R., Köszegi, J., M., T., and Tóth, M. RefactorErl — Source code analysis and refactoring in Erlang. In *Proceedings of the 12th Symposium on Programming Languages and Software Tools*, pages 138–148, Tallin, Estonia, 2011. ISBN: 978-9949-23-178-2.

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

[3] Cesarini, F. *Which companies are using Erlang, and why?* Erlang Solutions, 2019. URL: https://www.erlang-solutions.com/blog/which-companies-are-using-erlang-and-why-mytopdogstatus/.

[4] Erlang Solutions. *20 Years of Open Source Erlang: OpenErlang Interview with Anton Lavrik from WhatsApp*, 2018. URL: https://www.erlang-solutions.com/blog/20-years-of-open-source-erlang-openerlang-interview-with-anton-lavrik-from-whatsapp/.

[5] Ferenczi, D. and Tóth, M. Static analysis for safe software upgrade. In *Annales Mathematicae et Informaticae*, Volume 58, pages 9–19, 2023. DOI: 10.33039/ami.2023.08.010.

[6] Ferenczi, D. and Tóth, M. Safe process state upgrades through static analysis. In *Proceedings of the 2024 IEEE 18th International Symposium on Applied Computational Intelligence and Informatics*, pages 000351–000356. IEEE, 2024. DOI: 10.1109/SACI60582.2024.10619854.

[7] Horpácsi, D. and Koszegi, J. Static analysis of function calls in erlang. refining the static function call graph with dynamic call information by using data-flow analysis. *e-Informatica Software Engineering Journal*, 7(1), 2013. DOI: 0.5277/e-Inf130107.

[8] Horváth, Z., Lövei, L., Kozsik, T., Kitlei, R., Víg, A. N., Nagy, T., Tóth, M., and Király, R. Modeling semantic knowledge in Erlang for refactoring. In *Knowledge Engineering: Principles and Techniques, Proceedings of the International Conference on Knowledge Engineering, Principles and Techniques*, Volume 54(2009) Special Issue of *Studia Universitatis Babeş-Bolyai, Series Informatica*, pages 7–16, Cluj-Napoca, Romania, 2009. URL: http://www.studia.ubbcluj.ro/arhiva/abstract_en.php?%20editie=INFORMATICA&nr=Sp.Issue%201&an=2009&id_art=6521.

[9] Li, H. and Thompson, S. Refactoring support for modularity maintenance in Erlang. In *Proceedings of the 2010 10th IEEE Working Conference on Source Code Analysis and Manipulation*, pages 157–166, 2010. DOI: 10.1109/SCAM.2010.17.

[10] Logan, M., Merritt, E., and Carlsson, R. *Erlang and OTP in Action*. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2010. ISBN: 9781933988788.

[11] Maule, A., Emmerich, W., and Rosenblum, D. S. Impact analysis of database schema changes. In *Proceedings of the 30th International Conference on Software Engineering*, pages 451–460, New York, NY, USA, 2008. Association for Computing Machinery. DOI: 10.1145/1368088.1368150.

[12] Meurice, L., Nagy, C., and Cleve, A. Detecting and preventing program inconsistencies under database schema evolution. In *Proceedings of the 2016 IEEE International Conference on Software Quality, Reliability and Security*, pages 262–273, 2016. DOI: 10.1109/QRS.2016.38.

[13] Naseer, U., Niccolini, L., Pant, U., Frindell, A., Dasineni, R., and Benson, T. A. 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*, pages 529–541, New York, NY, USA, 2020. Association for Computing Machinery. DOI: 10.1145/3387514.3405885.

[14] Neamtiu, I. and Dumitraş, T. Cloud software upgrades: Challenges and opportunities. In *Proceedings of the 2011 International Workshop on the Maintenance and Evolution of Service-Oriented and Cloud-Based Systems*, pages 1–10. IEEE, 2011. DOI: 10.1109/MESOCA.2011.6049037.

[15] Sampaio, A. R., Kadiyala, H., Hu, B., Steinbacher, J., Erwin, T., Rosa, N., Beschastnikh, I., and Rubin, J. Supporting microservice evolution. In *Proceedings of the 2017 IEEE International Conference on Software Maintenance and Evolution*, pages 539–543. IEEE, 2017. DOI: 10.1109/ICSME.2017.63.

[16] Tóth, M. and Bozó, I. Static analysis of complex software systems implemented in Erlang. In *Proceedings of the Fourth Central European Functional Programming School*, Volume 7241 of *Lecture Notes in Computer Science*, pages 440–498. Springer-Verlag, 2012. DOI: 10.1007/978-3-642-32096-5_9.

[17] Tóth, M., Bozó, I., Kőszegi, J., and Horváth, Z. Static analysis based support for program comprehension in Erlang. *Acta Electrotechnica et Informatica*, 11(3):3–10, 2011. DOI: 10.2478/v10198-011-0022-y.

[18] Tóth, M. and Bozó, I. Supporting secure coding for Erlang. In *Proceedings of the 39th ACM/SIGAPP Symposium on Applied Computing*, page 1307–1311, New York, NY, USA, 2024. Association for Computing Machinery. DOI: 10.1145/3605098.3636185.