Analysing the hierarchical structure of Erlang applications

István Bozó
bozoistvan@caesar.elte.hu

Bence János Szabó
szbence@hotmail.com

Melinda Tóth
tothmelinda@caesar.elte.hu

ELTE Eötvös Loránd University
Faculty of Informatics
Budapest, Hungary

Abstract

Understanding and maintaining legacy software is a well-known problem. Static analysis tools aims to support developers in these tasks. We present a static analysis method for extracting the supervisor hierarchy of Erlang programs, which can support program comprehension and maintenance. Beside the algorithm we introduce two different views of the supervisor graph, a compact supervisor graph with only the essential information, and a more detailed full supervisor graph. We demonstrate the method on an example Erlang application and present the two different views of the supervisor hierarchy. The method has been implemented as an extension of the RefactorErl analyser framework.

1 Introduction

RefactorErl [1, 2] is a static source code analyser, comprehension and refactoring tool for Erlang [3, 4]. The aim of the tool is to support source code comprehension and maintenance. RefactorErl is an extensible framework that provides several analysis features [5] and tools. The extracted information of the source code is stored in a *Semantic Program Graph* [6] (*SPG*) which is available for users through a query language [7].

The tool provides several basic semantic analyses, like function call analysis, variable binding and reference analysis, data flow analysis, etc. There are several advanced analyses built on the basic semantic analyses, for example duplicate code analysis, dead code analysis, and dependency graphs, etc. However, the analysis of supervisors and applications [8] of Erlang programs was not supported. Both concepts are fundamental in Erlang because they describe dependencies and connections between Erlang modules.

A supervisor is a process that monitors and manages other processes and handle errors based on the given strategies. A supervised process can be a worker or an other supervisor process. The hierarchically built structure of supervisor processes are called supervision trees. Using supervisors is a good programming practice to build fault-tolerant applications. Understanding the hierarchy of complex and concurrent or distributed applications is essential in maintenance and support phases.

In this paper we are going to extend the semantic layer of the *SPG* with information about supervisors. We provide formal definitions both for the extensions of the model of the semantic program graph and for the algorithms of the semantic analyses. In addition we define two new graphs, a compact view of the supervision hierarchy and a view of the full supervisor graph including detailed information.

Copyright © by the paper's authors. Copying permitted for private and academic purposes.

In: E. Vatai (ed.): Proceedings of the 11th Joint Conference on Mathematics and Computer Science, Eger, Hungary, 20th – 22nd of May, 2016, published at http://ceur-ws.org

The paper is structured as follows. Section 2 describes the necessary background information for understanding the approach. Section 3 describes the formal extension of the used source code representation (SPG) and the algorithm of the analysis. Section 4 introduces two graphs/views with different granularity level. Section 5 demonstrates the presented algorithm and views on an example. Section 6 surveys the supervisor concept used in other programming languages. Finally, Section 7 summarises our results and presents our conclusions.

2 Background

Before discussing the method in details, it is necessary to survey the Erlang supervisor and the RefactorErl framework used as a basis for the analysis.

2.1 Erlang supervisors

The supervisor behaviour is a library module that contains generic code for process monitoring and managing. The specific code is implemented in the *callback* module. The only mandatory function in the callback module is the function `init/1`. This function defines the restart strategy, the maximum restart intensity and the child specifications of the supervisor process. The child specification describes the identifier of the child, the function to start the process, the restart option, the timeout of the shutdown, the type of child process (worker or supervisor) and its callback module.

The supervisor process can be started with the `supervisor:start_link/2,3` functions. These functions require the name of the supervisor (optional), the callback module and startup arguments. When the function is applied the `init/1` function is invoked from the provided callback module with the specified arguments.

We must note here that besides the static initialising arguments, there are several supervisor manipulating functions such as adding, restarting, deleting a child process, or querying and checking child specifications, etc in runtime.

2.2 RefactorErl

RefactorErl stores the analysed data in a special directed graph called semantic program graph (SPG). This graph includes all the information that can be statically extracted from the source code with different analyses. The SPG consists of three layers. The first is the lexical layer, the second is the syntactic layer and the third is the semantic layer.

The semantic layer is constructed by several analyses which can be divided into two groups. The first is the group of *pre analyses* which are executed when an Erlang source file is being loaded into RefactorErl, and the second is the group of *post analyses* which need to be run manually after the pre analysis phase.

The SPG is a directed graph that can be described as an ordered hextuple:

$$SPG = (V, A_N, A_V, l, E, e)$$

The components of the hextuple are the following:

- $V = (V_{lex} \cup V_{syn} \cup V_{sem} \cup V_r)$: The nodes of the SPG are the union of the lexical, syntactic and semantic nodes, extended with a special singleton set of the root node.
- A_N: The set of attribute names of the nodes.
- A_V: The set of possible attribute values.
- $l : V \times A_N \to A_V$: The node labelling partial function.
- E: The set of edge labels.
- $e : V \times E \times \mathbb{N} \to V$: The partial function that describes labelled, ordered edges between the graph nodes.

The SPG has several type of nodes with different attributes describing the properties of the nodes. The nodes of the SPG and their attributes that are important from the aspect of the supervisor analysis are as follows:

Root: A special singleton set that includes the root node of the SPG. Symbol: V_r

File: Set of syntactic nodes representing the files in the SPG. Symbol: $V_{fi} \subseteq V_{syn}$

Expr: Set of syntactic nodes representing the expressions in the *SPG*. The expression nodes have a `type` attribute which describes the type of the expression. The supervisor analysis uses expression nodes with type `application`, which means that the expression is a function invocation. Symbol: $V_e \subseteq V_{syn}$

Module: Set of semantic nodes representing an Erlang module in the *SPG*. The module nodes have a `name` attribute which describes the name of the module. Symbol: $V_m \subseteq V_{sem}$

Behaviour: Set of semantic nodes representing the behaviours in the *SPG*. A behaviour node is present in the *SPG* if the analysed module implements a behaviour. The `type` attribute of the node describes the name of the implemented behaviour. Symbol: $V_b \subseteq V_{sem}$

Func: Set of semantic nodes representing the functions in the *SPG*. The functions are identified by the triple of the name of the defining module, and the name and arity of the function (`(M,F,A)`). Symbol: $V_f \subseteq V_{sem}$

3 Supervisor analysis

The analysis defined in this paper extends the *SPG* with the supervisor semantic node and new labelled links between semantic and syntactic nodes of the graph according to the semantics of the supervisor behaviour.

3.1 *SPG* extension

We formally specify how do we extend the previously described set of semantic nodes with a set of *Supervisor* nodes (V_{sup}) and the directed edges between the nodes of the graph.

Attributes of the Supervisor node:

- `name`: The name of the supervisor. Type: `atom()`
- `type`: The type of the supervisor. Type: `[supervisor,worker]`
- `registered_name`: This attribute contains the possible registered names of the supervisor.
  Type: `supRegName() = [{[via], [Module :: atom()], [Name :: atom()]}], {[Scope :: atom()], [Name :: atom()]}`.
- `data`: If the supervisor is a worker then this attribute contains every information about the worker. Type: `supPropList() = [{child_id, [atom()]}, {restart, [atom()]}, {shutdown, [integer()]}, {modules, [atom()]}]`.
- `strategy`: The restart strategy of the supervisor.
  Type: `supStrategy() = [{[Strategy :: strategy()], [MaxTime :: integer()], [MaxRestart :: integer()]}]` where the `strategy()` is the type of restart strategies (one_for_all, one_for_one, rest_for_one, simple_one_for_one), `MaxTime` and `MaxRestart` describe the maximum intensity of restarts within the given time period.
- `start_args`: Contains the last argument of the `supervisor:start_link/2,3` function calls which can start the actual supervisor. The last argument is passed to the `init/1` *callback* function. Type: `[term()]`

New edges from the Supervisor node:

- $sup_cb_module : V_{sup} \rightarrow V_m$: Points to the node of the *callback* module of the supervisor.
- $sup_start : V_{sup} \rightarrow V_f$: Points to the function nodes that apply the functions `supervisor:start_link/2,3` in their definition (potentially start the supervisor process).
- $sup_init : V_{sup} \rightarrow V_f$: Points to the function node `init/1` in the *callback* module.
- $sup_worker : V_{sup} \rightarrow V_{sup}$: Points from a supervisor to its direct successor supervisor nodes (in the supervision tree).
- $w_def : V_{sup} \rightarrow V_f$: Points from a worker supervisor node to the function node that defines the worker.
- $sup_def : V_{sup} \rightarrow V_{sup}$: Points from a supervisor to its defining supervisor node.

- $sup_ref : V_{sup} \to V_e$: Points to the function calls (expressions) that can modify or query the supervisor in runtime.

New edges to the Supervisor node:

- $beh_sup : V_b \to V_{sup}$: This edge points from the behaviour node to the supervisor node if its `type` attribute is *supervisor*.

Formal extension of the SPG

Let SPG' denote the extended semantic program graph:

$$SPG' = (V', A'_N, A'_V, l', E', e')$$

The set of semantic nodes (V_{sem}) is extended with the set of *Supervisor* nodes (V_{sup}):

$$V'_{sem} = V_{sem} \cup V_{sup}$$

$$V' = V \cup V'_{sem}$$

The sets of attribute names (A_N) and values (A_V) are extended with the attributes and possible values of attributes of the *Supervisor* node:

$$A_N^{sup} = \{name, type, registered_name, data, strategy, start_args\}$$

$$A_V^{sup} = \{atom() \cup [supervisor|worker] \cup supRegName() \cup supPropList() \cup supStrateg() \cup [term()]\}$$

$$A'_N = A_N \cup A_N^{sup}$$
$$A'_V = A_V \cup A_V^{sup}$$

The set of edge labels (E) is extended with the new labels:

$$E_{sup} = \{sup_def, sup_cb_module, sup_start, sup_init, sup_worker, w_def, sup_ref, beh_sup\}$$

$$E' = E \cup E_{sup}$$

The node labelling partial function l is extended with the following cases for an arbitrary $x \in V_{sup}$ node:

$$l' : x \times name \to atom()$$

$$l' : x \times type \to [supervisor|worker]$$

$$l' : x \times registered_name \to supRegName()$$

$$l' : x \times data \to supPropList()$$

$$l' : x \times strategy \to supStrategy()$$

$$l' : x \times start_args \to [term()]$$

The partial function e is extended with the following cases for an arbitrary $x \in V_{sup}$ node and an arbitrary $n \in \mathbb{N}$ natural number:

$$e' : x \times sup_cb_module \times n \to y \in V_m$$

$$e' : x \times \{sup_start, sup_init, w_def\} \times n \to y \in V_f$$

$$e' : x \times \{sup_worker, sup_def\} \times n \to y \in V_{sup}$$

$$e' : x \times sup_ref \times n \to y \in V_e$$

For an arbitrary $z \in V_b$ node:

$$e' : z \times beh_sup \times n \to y \in V_{sup}$$

3.2 The algorithm

The analysis is performed in two phases. The first step is the pre analysis which is performed during the basic semantic analysis. The second step is the post analysis, which extracts all the statically available information about the supervisors. We separated our analysis according to the asynchronous, incremental analyser framework of RefactorErl. The first phase of the analysis (pre analysis) is the syntax based node identification part that can be performed incrementally form by form asynchronously. In this phase we have only a partial view on the semantic layer. The second phase (post analysis) uses the information calculated by the first phase of the supervisor analysis and other semantic analyses performed on the full *SPG* in an interfunctional way (such as the data-flow reaching). The pseudo code of the analysis is presented in Algorithm 1.

Pre analysis

In the pre analysis phase a new *Supervisor* semantic node is created and inserted into the *SPG* if the module implements the supervisor behaviour. That is, the module contains the `-behaviour(supervisor)` attribute. This module is the *callback* module of the behaviour. A new edge with label *sup_cb_module* is inserted between the *Supervisor* node and the module and the `name` attribute of the *Supervisor* is set to the name of the *callback* module. Finally, a new edge with label *beh_sup* is inserted between the corresponding behaviour node and the *Supervisor* node.

Post analysis

The pre analysis phase did the initialisation, new *Supervisor* nodes have been created, initialised and inserted to the *SPG*. The post phase of the algorithm extracts the most possible information about the supervisors. The algorithm determines the starting arguments, workers, hierarchy, etc. The post phase creates new semantic nodes and extends the semantic information available in the *SPG*. The Algorithm 1 shows the pseudo code of the post analysis phase for a given *Supervisor* node.

First of all, the algorithm finds the `init/1` *callback* function of the supervisor using the $find_init_fun$ routine. Once the *callback* function node is found we insert an edge with label *sup_init* between the examined *Supervisor* node and the function node.

The second step of the algorithm is to determine the functions that potentially start the supervisor process. The algorithm searches for `supervisor:start_link/2,3` function applications. For each function application we get its arguments with the *get_args* routine, then we apply data-flow reaching by invoking the $flow$ routine to determine the possible values of the arguments. The $Module$ variable contains the name of the *callback* module of the supervisor that is started with the examined application. We filter the result for the actual supervisor (*Supervisor*) and we update its `registered_name` and `args` attributes. Finally, we create a link with label *sup_start* between the *Supervisor* and the node of the function that applies the `supervisor:start_link/2,3` functions.

The third step of the algorithm is the analysis of restart strategies and child specifications of the *Supervisor*. This information can be found in the return value of the `init/1` *callback* function or the `supervisor:start_-child/2` function. At first, we examine the return points of the `init/1` *callback* function. The return points of the function are queried by the *return_points* routine. For each return point we apply data-flow reaching to determine the possible strategies and child specifications. We update the attribute `restart_strategy` with the newly found restart strategies (*Strategies*) and collect the child specifications in the set *GoodChildSpecs*. Next, we need to get the child specifications from the `supervisor:start_child/2` function applications. We query the function applications and for each function application we try to determine the values of its arguments with data-flow reaching. The gathered child specifications are inserted to the *GoodChildSpec* set if the set of registered names of the examined *Supervisor* and the set $SupRefNames$ has a non empty section. Next we create a new worker *Supervisor* node using the *create_worker* routine for every child specification from the *GoodChildSpec* and create a link with label *sup_worker* between every newly created worker and the examined `Supervisor`. Finally, based on the type of the worker process we use a *sup_def* or *w_def* edge to create a link between the worker and its definition.

The last step is to analyse the supervisor references. We examine the function from the *supervisor* module listed in variable $FunsWithArity$. For each function we query the function applications, then with data-flow reaching we try to determine the referred supervisor from its arguments. If the function application refers the examined *Supervisor*, then a link with label *sup_ref* is created between the function application and the *Supervisor* node.

Algorithm 1 *analyse_supervisor*(*Supervisor*)

1: $InitFun \leftarrow find_init_fun(Supervisor)$
2: $create_link(Supervisor, InitFun, sup_init)$
3: %% *Analyse supervisor* : *start_link*/2, 3 *functions* %%
4: $StartFunCalls \leftarrow get_fun_calls(supervisor, start_link, \{2, 3\})$
5: **for** $FunCall \in StartFunCalls$ **do**
6: $(SupName, Module, Args) \leftarrow flow(get_args(FunCall))$
7: **if** $Supervisor_{name} \cap Module \neq \varnothing$ **then**
8: $Supervisor_{registered_name} \leftarrow Supervisor_{registered_names} \cup SupName$
9: $Supervisor_{args} \leftarrow Supervisor_{args} \cup Args$
10: $create_link(Supervisor, get_fun(FunCall), sup_start)$
11: **end if**
12: **end for**
13: %% *Analyse child specifications* %%
14: $RetPoints \leftarrow return_points(InitFun)$
15: $GoodChildSpecs \leftarrow \varnothing$
16: **for** $RetPoint \in RetPoints$ **do**
17: $(ok, (Strategies, ChildSpecs)) \leftarrow flow(RetPoint)$
18: $Supervisor_{strategy} \leftarrow Supervisor_{strategy} \cup Strategies$
19: $GoodChildSpecs \leftarrow GoodChildSpecs \cup ChildSpecs$
20: **end for**
21: $StartChildFunCalls \leftarrow get_fun_calls(supervisor, start_child, 2)$
22: **for** $FunCall \in StartChildFunCalls$ **do**
23: $(SupRefNames, ChildSpecs) \leftarrow flow(get_args(FunCall))$
24: **if** $Supervisor_{registered_name} \cap SupRefNames \neq \varnothing$ **then**
25: $GoodChildSpecs \leftarrow GoodChildSpecs \cup ChildSpecs$
26: **end if**
27: **end for**
28: **for** $ChildSpec \in GoodChildSpecs$ **do**
29: $NewWorker \leftarrow create_worker(ChildSpec)$
30: $WorkerDef \leftarrow get_worker_def(ChildSpec)$
31: $create_link(Supervisor, NewWorker, sup_worker)$
32: **if** $supervisor \in NewWorker_{type}$ **then**
33: $create_link(NewWorker, WorkerDef, sup_def)$
34: **else if** $worker \in NewWorker_{type}$ **then**
35: $create_link(NewWorker, WorkerDef, w_def)$
36: **end if**
37: **end for**
38: %% *Analyse supervisor references* %%
39: $FunsWithArity \leftarrow \{(start_child, 2), (terminate_child, 2), (delete_child, 2),$
40: $(restart_child, 2), (which_children, 1), (count_children, 1)\}$
41: **for** $(Fun, Arity) \in FunsWithArity$ **do**
42: $FunCalls \leftarrow get_fun_calls(supervisor, Fun, Arity)$
43: **for** $FunCall \in FunCalls$ **do**
44: $(SupRefNames, _) \leftarrow flow(get_args(FunApp))$
45: **if** $Supervisor_{registered_name} \cap SupRefNames \neq \varnothing$ **then**
46: $create_link(Supervisor, FunCall, sup_ref)$
47: **end if**
48: **end for**
49: **end for**

4 The supervisor graph

In Section 3 we extended the *SPG* with additional semantic information about supervisors. It is possible to visualise the *SPG*, however it is too extensive and it is hard to find relevant information about supervisors. In

this section we define a supervisor graph which is a view of the SPG that contains information only about the supervisors.

The introduced graphs contain explicit information about the supervisor structure that can be used for code comprehension, debugging or checking the hierarchy of the software.

4.1 Definition of the supervisor graph

Let the supervisor graph G^{sup} be an ordered pair. The first component of the pair is the set of nodes of the supervisor graph. The second component of the pair is the set of directed edges represented as ordered triples.

$$G^{sup} = (V^{sup}, E^{sup})$$

The set of nodes is the union of supervisor nodes, worker nodes, function nodes and expression nodes, formally:

$$V^{sup} = (V_s^{sup} \cup V_w^{sup} \cup V_f^{sup} \cup V_e^{sup})$$

The set of edges is represented by ordered triples. The components of a triple are the head node, the tail node and the label of the edge:

$$E^{sup} \subseteq V^{sup} \times V^{sup} \times LABEL^{sup}$$

The set of edge labels of the supervisor graph is:

$$LABEL^{sup} = \{start, init, supervisor, worker, sup_def, worker_def, reference\}$$

Nodes of the supervisor graph

- The definition of the supervisor node is the following:

$$\textit{Sup node} = \{\textit{name} :: \textit{string}(),\ \textit{strategy} :: \textit{supStrategy}(),\ \textit{registered_name} :: [\textit{term}()],\ \textit{start_args} = [\textit{term}()]\}$$

  The *Sup nodes* are representing the supervisors in the supervisor graph. The *Sup node* has the following attributes:

  - *name*: The name of the supervisor. This will be the caption of the node in the supervisor graph.
  - *strategy*: The restart strategy of the supervisor.
  - *registered_name*: If the supervisor is registered, this attribute contains the type of the registration along with the registered name.
  - *start_args*: The arguments which are passed to the `init/1` callback function.

- The definition of the worker node is the following:

$$\textit{SupWorker node} = \{\textit{name} :: \textit{string}(),\ \textit{data} :: \textit{supPropList}()\}$$

  The *SupWorker nodes* are representing the workers in the supervisor graph. The *SupWorker node* has the following attributes:

  - *name*: The name of the worker. This name will be the caption of the node in the supervisor graph.
  - *data*: This attribute contains every information available about the worker, for example: the type of the worker.

- The definition of function node is the following:

$$\textit{SupFunction node} = \{\textit{name} :: \textit{string}()\}$$

  The function nodes represent the *callback* functions in the supervisor graph. The *SupFunction node* has the following attributes:

  - *name*: The name of the function. This will be the caption of the node in the supervisor graph.

- The definition of the expression node is the following:

$$SupExpression\ node = \{name :: string(),\ func :: \{module(), fun(), arity()\},\ args : [term()]\}$$

The expression nodes represent the supervisor references in the supervisor graph. Supervisor references are function applications, which might modify or query the supervisor arguments during runtime. The *SupExpression node* has the following attributes:

  - *name*: The text of the expression node which will be the caption of the node in the supervisor graph.
  - *func*: A module, function and arity triple ({`M,F,A`}) that identifies the function which is referred by the application expression.
  - *args*: The arguments of the function application.

Notations

In the previous section we introduced the V_{sup} set of *Supervisor* semantic nodes in the *SPG*. In this section we use the V^{sup} notation for the set of supervisor nodes in the new supervisor graph. We define two subsets of set V_{sup}:

- $V_{sup_s} \subseteq V_{sup}$ – set of supervisor nodes which type is *supervisor*. Formally:

$$V_{sup_s} = \{\ x \mid x \in V_{sup},\ supervisor \in l(x, type)\}$$

- $V_{sup_w} \subseteq V_{sup}$ – set of supervisor nodes which type is *worker*. Formally:

$$V_{sup_w} = \{\ x \mid x \in V_{sup},\ worker \in l(x, type)\}$$

Auxiliary functions

We introduce some auxiliary functions to shorten the supervisor graph building rules:

- $edges(x, y)$: The function takes two nodes from the *SPG* and returns the set of labels of directed edges with head x and tail y.
- $updateSup(Sup, Sup')$:

$$\frac{Sup \in V_{sup_s} \land Sup' \in V_s^{sup}}{\begin{array}{c} Sup'[name = Sup_{name},\ strategy = Sup_{strategy}, \\ registered_name = Sup_{registered_name},\ start_args = Sup_{start_args}] \end{array}}$$

The rule says that the Sup' node (type of *Sup node*) shall have the same name, strategy, registered name and start arguments as the *Sup Supervisor* node from the *SPG*.

- $updateSupWorker(Sup, Worker)$:

$$\frac{Sup \in V_{sup_w} \land Worker \in V_w^{sup}}{Worker[name = Sup_{name},\ data = Sup_{data}]}$$

The rule says that the *Worker* node (type of *SupWorker node*) shall have the same name and the same data as the *Sup Supervisor* node (which type is worker) from the *SPG*.

- $updateSupFunction(Fun, Fun')$: The function updates the *name* attribute of Fun' node (type of set *SupFunction node*) with the `(M,F,A)` of the Func node from the *SPG*.
- $updateSupExpression(Expr, Expr')$: The function updates the attributes of $Expr'$ node (type of *SupExpression node*). The `name` attribute shall be the string created from `(M,F,A)` of the function that contains the *SPG* expression *Expr*. The `args` attribute contains the arguments of the function application which information is gathered from. Finally, the `func` attribute contains the ordered triple {`M,F,A`} of the function which the supervisor reference refer. E.g.: {*supervisor,start_child,2*}.

COMPACT supervisor graph

We introduce two types of supervisor graphs. The full supervisor graph contains every available information about the supervisors and workers, like *callback* functions, supervisor references, worker definitions, etc. The *COMPACT* supervisor graph only contains information about the hierarchy of the supervisors and workers. We use the *COMPACT* flag to distinguish the *COMPACT* and full supervisor graphs in the rules.

Rules

In the followings we define the rules which are used to build the full and *COMPACT* supervisor graphs.

1. Supervisor node

$$\frac{x \in V_{sup_s}}{x' \in V_s^{sup} \land updateSup(x, x')}$$

If an x *Supervisor* node is present in the *SPG* then the x' node which is synthesised from x using the *updateSup* function will be in the supervisor graph. This x' will be a member of the *Sup nodes* of the supervisor graph.

2. Supervisor edge (*COMPACT*)

$$\frac{\{x, y, z\} \subseteq V_{sup_s} \land sup_worker \in edges(x, y) \land sup_def \in edges(y, z) \land COMPACT}{\begin{array}{c}x' \in V_s^{sup} \land z' \in V_s^{sup} \land updateSup(x, x') \land updateSup(z, z') \land \\ (x', z', supervisor) \in E^{sup}\end{array}}$$

Let the x, y and z be *Supervisor* nodes from *SPG*. If there is a *sup_worker* edge between x and y and a *sup_def* edge between y and z, then there will be a *supervisor* edge between the x' and z' synthesised *Sup node* nodes in the *COMPACT* supervisor graph. The not *COMPACT* (full) supervisor edge rule can be analogously defined.

3. Worker edge

$$\frac{x \in V_{sup_s} \land y \in V_{sup_w} \land sup_worker \in edges(x, y)}{\begin{array}{c}x' \in V_s^{sup} \land y' \in V_w^{sup} \land updateSup(x, x') \land updateSupWorker(y, y') \land \\ (x', y', worker) \in E^{sup}\end{array}}$$

Let x and y be *Supervisor* nodes in the *SPG* and the type of y is `worker`. If there is a *sup_worker* edge between x and y, then there will be a *worker* edge between the x' and y' synthesized *Sup node* nodes in the supervisor graph.

4. Supervisor init

$$\frac{x \in V_{sup_s} \land f \in V_f \land sup_init \in edges(x, f) \land \neg COMPACT}{\begin{array}{c}x' \in V_s^{sup} \land f' \in V_f^{sup} \land updateSup(x, x') \land updateSupFunction(f, f') \land \\ (x', f', init) \in E^{sup}\end{array}}$$

If a *Supervisor* node is connected to a function node with a *sup_init* edge in the *SPG*, then there will be an *init* edge between the synthesised x' and f' nodes in the supervisor graph (if the supervisor graph is not *COMPACT*). This rule describes that the supervisor shall be connected to its *init callback* function.

5. Supervisor start

$$\frac{x \in V_{sup_s} \land f \in V_f \land sup_start \in edges(x, f) \land \neg COMPACT}{\begin{array}{c}x' \in V_s^{sup} \land f' \in V_f^{sup} \land updateSup(x, x') \land updateSupFunction(f, f') \land \\ (x', f', start) \in E^{sup}\end{array}}$$

If a *Supervisor* node is linked to a function node with a *sup_start* edge in the *SPG*, then there will be an *start* edge between the synthesised x' and f' nodes in the supervisor graph (if the supervisor graph is not *COMPACT*). This rule describes that the supervisor shall be connected to the function that can start the supervisor with an application of the `supervisor:start_link/2,3` functions.

6. Supervisor reference

$$\frac{x \in V_{sup_s} \wedge e \in V_e \wedge sup_ref \in edges(x,e) \wedge \neg COMPACT}{x' \in V_s^{sup} \wedge e' \in V_e^{sup} \wedge updateSup(x,x') \wedge updateSupExpression(e,e') \wedge (x',e',reference) \in E^{sup}}$$

If the x *Supervisor* node is linked to an expression node x with a *sup_ref* edge in the *SPG*, then there will be a *reference* edge between the synthesised x' and e' nodes in the supervisor graph (if the supervisor graph is not *COMPACT*).

5 Demonstrating Example

In this section we illustrate the algorithm of the analysis with an example implementation of a supervisor. First we introduce the example implementation, then we discuss the algorithm step by step focusing mostly on the post analysis phase of the *SPG* extension. Finally we present the supervisor graph and the compact supervisor graph.

5.1 Example Supervisor

The example supervisor can be seen in Figure 1. This supervisor is a simplified model of a theatre. The theatre has a director, a technician, and a bandmaster. The bandmaster supervises the musicians as can be seen in Figure 2.

Figure 1: Theatre supervisor

```
1  -module(theatre).
2  -behaviour(supervisor).
3
4  -export([start_link/1, add_crew/1]).
5  -export([init/1]).
6
7  start_link(Args) ->
8      supervisor:start_link({local,?MODULE}, ?MODULE, Args).
9
10 init(Args) ->
11     Children = [
12         {director,
13         {director, start_link, []},
14          transient, 100, worker, [director]},
15         {tech,
16         {tech, start_link, []},
17          transient, 100, worker, [tech]},
18         {bandmaster,
19         {bandmaster, start_link, [Args]},
20          transient, 100, supervisor, [bandmaster]}],
21     {ok, {{one_for_all, 3, 500}, Children}}.
22
23 add_crew(ChildSpec) ->
24     supervisor:start_child({local,?MODULE}, ChildSpec).
```

Figure 2: Bandmaster supervisor

```
1  -module(bandmaster).
2  -behaviour(supervisor).
3
4  -export([start_link/1]).
5  -export([init/1]).
6
7  start_link(Args) ->
8      supervisor:start_link({local,?MODULE}, ?MODULE, Args).
9
10 init(lenient) ->
11     init({one_for_one, 3, 60});
12 init(angry) ->
13     init({rest_for_one, 2, 60});
14 init(jerk) ->
15     init({one_for_all, 1, 60});
16
17 init({RestartStrategy, MaxRestart, MaxTime}) ->
18     Mod = musicians,
19     Type = worker,
20     Children = [
21         {guitar,
22         {Mod, start_link, [guitar, good]},
23          transient, MaxRestart, Type, [Mod]},
24         {drummer,
25         {Mod, start_link, [drummer, average]},
26          transient, MaxRestart, Type, [Mod]}],
27     {ok, {{RestartStrategy, MaxRestart, MaxTime}, Children}}.
```

Pre analysis

The pre analysis phase looks for "`-behaviour(supervisor).`" attribute in each module. In this case both `theatre` (in line 2) and `bandmaster` module (in line 2) has such attribute. Two new *Supervisor* nodes will be inserted into the *SPG*. One for `theatre` module and one for `bandmaster` module, and two *sup_cb_module* links will be inserted between the *Supervisor* nodes and the module nodes. The name attribute of the *Supervisor* nodes will be equal to the names of the module which they are connected to. Also two *beh_sup* links will be inserted between the *Behaviour* nodes and the *Supervisor* nodes.

Post analysis

We discuss only the analysis of the *theatre* supervisor since the analysis of the *bandmaster* can be executed analogously. The only difference will be the amount of extracted information about the restart strategy and maximal restart intensity as the information is passed as a parameter to function `init/1`. This additional information is not always trivially available statically, but the data-flow analysis can calculate the possible values of variables statically.

The first step of the post analysis is to find the definition of the function `init/1` from *theatre callback* module in the *SPG*. It can be found in the line 10 of Figure 1. We insert a link *sup_init* between the supervisor node and the function node.

The second step of the analysis is the examination of the `supervisor:start_link/2,3` function applications. We query the references of the function and we get two function applications, one application in the definition of function `theatre:start_link/1` (in line 8 in Figure 1) and one other one in the definition of function `bandmaster:start_link/1` (in line 8 in Figure 2). The next step is the filtering of the relevant applications. To do that we query the possible values of arguments for each application (using data flow analysis) and we look for the second argument which describes the started supervisor. We find that the function application from the `theatre` module refers to the *theatre* supervisor because the `?MODULE` macro is substituted to the name of the containing module. We update the attributes of the *theatre* supervisor node with the gathered information. The

attributes `registered_name` and `args` are updated with the possible values of the first and the third arguments of the application. Finally we insert a link *sup_start* between the node of the function `theatre:start_link/1` (in line 7 in Figure 1) and the node of the supervisor *theatre*.

The third step is the analysis of restart strategies and child specifications. To find the child specifications we need the possible return values of the `init/1` *callback* function. We query the possible return values and extract the information about the restart strategy and maximum intensity of restarts from the result. In our case it is the tuple {`one_for_all, 3, 500`} (in line 21 in Figure 1). We update the `strategy` attribute of the *theatre* node with this information. Next we try to extract the child specifications from the return values using backward data-flow reaching. The `Children` variable (in line 11 in Figure 1) contains the child specifications, and we need to determine the possible values of this variable using data-flow analysis. There are three different children of the *theatre* supervisor. For each children we create a new *Supervisor* node of type worker and update the `data` attributes with the information can be found about each supervisor. We link the `theatre` supervisor to the workers with *sup_worker* edges. We insert *worker_def* links between the workers and their function definition. The name of the worker function will be the second component of the child specification. For example, {`director, start_link, []`} (in line 13 in Figure 1) means, that the children will be started using the `director:start_link/0` function. We also examine the child specifications found in the applications of `supervisor:start_child/2` function, but in our case there is no information about them.

In the final step of the analysis we examine the supervisor references that potentially query or change the attributes of the *theatre* supervisor. There is a reference to our supervisor in the `theatre:add_crew/1` function (in line 23 in Figure 1) . We insert a link *sup_ref* between the supervisor node an the reference.

The post analysis of the bandmaster supervisor can be analogously executed. The only important difference is that the data flow analysis will have heavier impact on the data extracted, because the `init/1` *callback* function of the `bandmaster` module have four clauses. Thus, the restart strategy and the maximal restart intensity can be given as parameters to the supervisor. For example the data-flow analysis will determine that the supervisor can have three different restart strategies (`one_for_one, rest_for_one, one_for_all`.

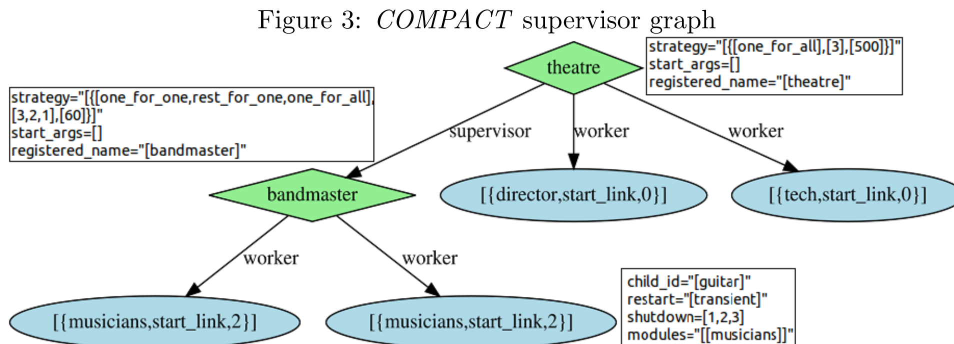
5.2 Results

Using the rules defined in Section 4 the *COMPACT* and full supervisor graphs can be constructed from the *SPG* after the supervisor analysis is completed. We show the *COMPACT* and full supervisor graphs for our example supervisor implementation presented in subsection 5.1.

COMPACT supervisor graph

The Figure 3 shows the *COMPACT* supervisor graph for the *theatre* supervisor. The graph shows only the hierarchy of the supervisors and the workers. The green diamond nodes represent the supervisors and the blue oval nodes are the workers of the supervisors. The attributes of the nodes contains every information that can be extracted statically about the supervisors or the workers as it can be seen in the picture, we highlighted the attributes of three nodes.

Figure 3: *COMPACT* supervisor graph

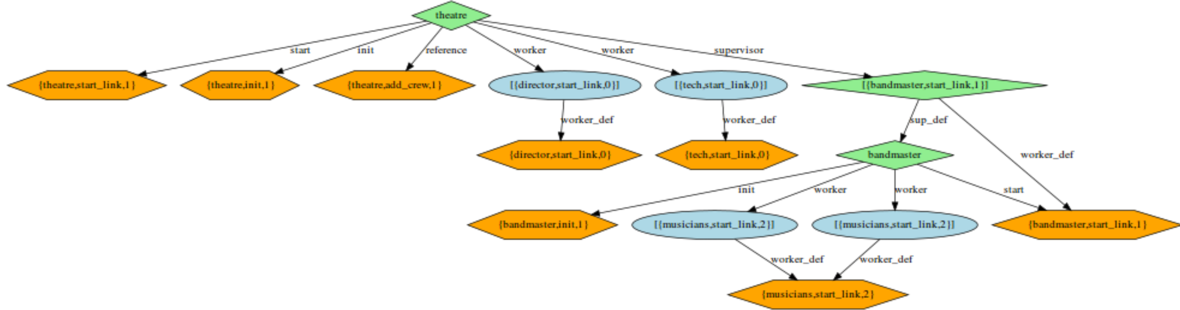


Full supervisor graph

The Figure 4 shows the full supervisor graph for the *theatre* supervisor. We can see that the structure of this graph is more complex than in the case of the *COMPACT* supervisor graph. We can find additional information

in the graph like the functions of the *callback* module, the function nodes of the worker definitions and the supervisor references. The orange hexagons represent the function nodes.

Figure 4: Full supervisor graph



6 Related work

The supervisor concept is present in other programming languages as well. In this section we examine some tools which are capable of analysing supervisors in Erlang and in other programming languages as well.

Percept2

Percept2 [9] is a profiler tool for Erlang programs with focus on processes and process communication. Percept2 is a dynamic analyser tool which analyses the program in runtime. In comparison with our results Percept2 can detect the hierarchy of the supervisors correctly. However it does not know the supervisor specific information, for example restart strategy, child specifications, etc. In some cases Percept2 can provide more accurate results, because it can detect new children added to supervisors dynamically. Our tool is a static code analyser, and can detect only those children which specification is present in the source code. However our tool can detect child specifications that are not necessarily added to the supervisors in a certain runtime configurations. In conclusion, the two tools have slightly different focus, as Precept2 is a dynamic and RefactorErl is a static analyser tool. The tools should be used as complementary tools.

Akka

Akka [10] is a library for Scala and Java programs. It supports the implementation of distributed concurrent and fault tolerant software using the Actor model. The Actor model takes care of the thread handling and resource locking, thus the developer can focus on the business logic of the software. Supervisors are present in Akka and it is very similar to the supervisors in Erlang. The processes are represented as actors and the supervisor can monitor these actors. Supervisors can start, stop and restart the monitored processes, and also halt them. Akka supports two restart strategies only. Currently there is no static code analyser support for supervisors in aid of Akka. Several IDE-s can provide dynamic information, like call stacks and class diagrams which can be used to get some information about the supervisors, but it is not nearly as accurate as our results for Erlang supervisors.

Cloud Haskell

Cloud Haskell [11] is a library for Haskell programs. It supports the implementation of distributed and concurrent software mainly for clusters. In Cloud Haskell there is a package called *distributed-process-supervisor* [12] which supports supervisors very similar to the supervisors used in Erlang. There is no static code analyser tool that support supervisor analysis in Cloud Haskell. The SourceGraph [13] can be used to make function call graphs and detect module import hierarchy. From these graphs there are some information available about the supervisors, but our results for Erlang supervisors are more accurate and user friendly.

7 Conclusions

In this paper we discussed a new method for extracting the relevant information from the source code written in Erlang implementing the `supervisor` behaviour. For the analysis we used the static analysis tool RefactorErl and

the information available in its *Semantic Program Graph*. RefactorErl performs lexical, syntactic and different semantic analyses on the source code and stores the extracted results in the SPG. We presented the algorithm of the new analysis with its pseudocode and explained the steps in detail. We introduced the formal extension of the SPG with the new nodes and the way the new semantic information is embedded in the SPG. To make the supervisor hierarchy more explicit we defined a supervisor graph that is a view of the SPG that includes only information about supervisors. We defined two levels of granularity for supervisor graphs, that is a full and a compact view. We discussed the steps and rules how these graphs are generated from the SPG. Finally we demonstrated our approach on an example and showed the generated graphs, both a full and a compact view.

The presented method was examined on different open source Erlang applications, and the results seemed to be useful in software comprehension.

References

[1] RefactorErl - Refactoring Erlang Programs. `http://plc.inf.elte.hu/erlang`, 2015. Accessed: 2016-06-30.

[2] István Bozó, Dániel Horpácsi, Zoltán Horváth, Róbert Kitlei, Judit Kőszegi, Máté Tejfel, and Melinda Tóth. RefactorErl, Source Code Analysis and Refactoring in Erlang. In *Proceeding of the 12th Symposium on Programming Languages and Software Tools*, Tallin, Estonia, 2011.

[3] Erlang Programming Language. `http://www.erlang.org`. Accessed: 2016-06-30.

[4] Francesco Cesarini and Simon Thompson. *Erlang Programming*. O'Reilly Media, 2009.

[5] Melinda Tóth and István Bozó. Static Analysis of Complex Software Systems Implemented in Erlang. In *Central European Functional Programming School*, volume 7241 of *Lecture Notes in Computer Science*, pages 440–498. Springer, 2012.

[6] Zoltán Horváth, László Lövei, Tamás Kozsik, Róbert Kitlei, István Bozó, Melinda Tóth, and Roland Király. Modeling Semantic Knowledge in Erlang for Refactoring. In *International Conference on Knowledge Engineering, Principles and Techniques, KEPT 2009, Selected papers*, pages 38–53, 2009.

[7] M. Tóth, I. Bozó, and Z. Horváth. Applying the Query Language to Support Program Comprehension. In *Proceeding of International Scientific Conference on Computer Science and Engineering, ISBN 978-80-8086-164-3*, pages 52–59, Stara Lubovna, Slovakia, September 2010.

[8] Martin Logan, Eric Merritt, and Richard Carlsson. *Erlang and OTP in Action*. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2010.

[9] Huiqing Li and Simon Thompson. Multicore profiling for erlang programs using percept2. In *Proceedings of the Twelfth ACM SIGPLAN Workshop on Erlang*, Erlang '13, pages 33–42, New York, NY, USA, 2013. ACM.

[10] Akka. `http://akka.io`. Accessed: 2016-06-30.

[11] Jeff Epstein, Andrew P. Black, and Simon Peyton-Jones. Towards haskell in the cloud. In *Proceedings of the 4th ACM Symposium on Haskell*, Haskell '11, pages 118–129, New York, NY, USA, 2011. ACM.

[12] Supervisors for The Cloud Haskell Application Platform. `https://hackage.haskell.org/package/distributed-process-supervisor`. Accessed: 2016-06-30.

[13] SourceGraph: Static code analysis using graph-theoretic techniques. `https://hackage.haskell.org/package/SourceGraph`. Accessed: 2016-06-30.