Procedia Computer Science
Volume 29, 2014, Pages 1525–1534
ICCS 2014. 14th International Conference on Computational Science

Detecting and Visualising Process Relationships in Erlang

Melinda Tóth and István Bozó

Eötvös Loránd University, Budapest, Hungary
{tothmelinda, bozoistvan}@elte.hu

Abstract
Static software analyser tools can help in program comprehension by detecting relations among program parts. Detecting relations among the concurrent program parts, e.g. relations between processes, is not straightforward. In case of dynamic languages only a (good) approximation of the real dependencies can be calculated. In this paper we present algorithms to build a process relation graph for Erlang programs. The graph contains direct relation through message passing and hidden relations represented by the ETS (Erlang Term Storage) tables.

Keywords: process relations, communication model, visualisation, Erlang

1 Introduction

Erlang [3] is a dynamically typed concurrent programming language. Erlang was designed to develop highly concurrent, distributed, fault tolerant systems with soft real-time characteristics. The dynamic and concurrent features of the language make static analysis hard, however the statically calculated information and the abstract representations built from the source code can help the developers in different phases of the software development lifecycle. The static analyses can help for debugging and maintenance, or for program comprehension.

The main goal of the RefactorErl project [9, 1] is to support program comprehension for Erlang developers in numerous ways. It provides a semantic query language for obtaining information about the source. The tool can generate call graphs with dynamic call information, and use them to perform side-effect analyses. RefactorErl provides a platform for module and function restructuring and clustering, it can generate function or module dependency graphs, and it offers several interfaces, *e.g,* it can be used from Emacs or the Erlang shell, and it provides web interfaces for multi-user usage.

Our current goal is to extend the functionality of the tool and implement *process relation* analysis. Roughly speaking, this means that we represent the processes as graph nodes and add the relation as edges to the graph. In this chapter we describe relations through message passing and through Erlang Term Storage (ETS) tables. The latter one define hidden relations among processes, and the algorithms presented in this paper can be adopted to other types of hidden relations (such as relation through files, and database usage).

Selection and peer-review under responsibility of the Scientific Programme Committee of ICCS 2014

doi:10.1016/j.procs.2014.05.138

The paper is structured as follows. In Section 2.1, we introduce the tool RefactorErl. In Section 2.2, we present the basic concurrent language construct of Erlang. In Section 3, we introduce a small client-server example. In Section 4, we describe a representation of process relations and give algorithms to detect them. Section 5 applies the presented algorithms to the motivating example. Section 6 discusses related work and Section 7 concludes the paper.

2 Background

2.1 RefactorErl

RefactorErl [6] is a static source code analyser and transformer tool for Erlang. The tool represents the source code in a Semantic Program Graph (SPG), which is a data structure designed to store lexical, syntactic and semantic information about the source code. RefactorErl provides an asynchronous semantic analyser framework and implements static semantic analyses based on this framework about variables, functions, modules, records, call graph and dynamic calls, side-effects, dependencies, dataflow etc.

Since RefactorErl provides a powerful platform for developing further analyses, we used it to built our process relation analysis. Both syntactic information (such as collecting message passing expressions) and semantic information (such as the possible values of an expression by dataflow reaching) can be gathered efficiently from the SPG using the query language. Therefore, the necessary collection of syntactic information can be transformed into queries during our analysis.

2.2 Examined Language Constructs

In this section we describe the subset of the language constructs that are relevant for describing our analyses. As described in the introduction, our analyses focus on the communication of parallel processes, and we cover here only the relevant language constructs. The reader can find a more detailed description in the documentation of the language [5].

2.2.1 Process Creation

Every process in Erlang is identified by a unique process identifier (`pid`). The function `self/0` returns the process identifier of the running process. Erlang processes can be created with any of the following functions: `spawn`, `spawn_link`, `spawn_monitor`, `spawn_opt`, etc. These spawning functions have similar behaviour, thus we describe here only the basic function`spawn/3`.

The function call `spawn(`*`Mod`*`, `*`Fun`*`, `*`Args`*`)` creates a new process executing the given function *`Mod`*`:`*`Fun`*`(`Arg_1`, `Arg_2`, ..., `Arg_n`)` and returns the `pid` of the created process, where *Args* = `[`Arg_1`, `Arg_2`, ..., `Arg_n`]`. The newly created process is placed into scheduler queue of the virtual machine. If the given function does not exist, a `pid` is returned and a report is created about the error.

2.2.2 Processes Registration

Processes can be registered with a given name using the built in function `register(`*`Name`*`, `*`Pid`*`)`. After registration, the process can be addressed with its *Pid* or with its registered name. The process can be unregistered with the built in function `unregister(`*`Name`*`)`. The registered process is automatically unregistered when the process terminates.

2.2.3 Communication

Processes communicate by message sending and receiving.

Messages can be send with message sending operator (`!`) or with functions `erlang:send/2`, `erlang:send/3`, `erlang:send_after/3` etc. The expression $Expr_1$ `!` $Expr_2$ sends the value of $Expr_2$ to $Expr_1$ asynchronously, where $Expr_1$ must evaluate to a Pid or a registered process name.

Messages can be received with the `receive` construct. The receive expression suspends the execution of the executing process until a message is received. The received messages are matched sequentially against the given patterns. The return value of the receive expression will be the result of the firstly matching body. If none of the patterns match a new message is extracted from the message queue. The optional after branch is evaluated if none of the received messages matched in the given time interval `MilliSec` (milliseconds).

```
receive
    Pattern1 [when Guard1] -> Body1;
    ...
    PatternN [when GuardN] -> BodyN
after
    MilliSec -> AfterBody
end
```

2.2.4 Erlang Term Storage (ETS)

The ETS tables provide possibility to store large amount of data in the Erlang run-time system and constant access time to this data. The data is stored in dynamic tables as tuples. The table is linked to the creator process, when the process terminates the ETS table is deleted.

A new ETS table can be created by calling the function `ets:new(Name, Options)`. The function returns a table identifier, which can be used for accessing the table. The first argument is the name of the table, and the second argument is a list of the options.

New entities can be inserted into the table with the function `ets:insert(Tab, Data)`. `Tab` must be evaluated to a table identifier or an atom if the table is named. `Data` is either a tuple or a list of tuple expressions.

There are several ways to retrieve data from an ETS table. The most used ones are `ets:match(Table, Pattern)` and `ets:select(Table,MatchSpec)`.

3 Motivating Example – Job Server

We will use a simple client-server example to illustrate our model. The source code of the server and client can be found in Fig. 1 and 2.

The server module provide interface functions for starting (`start/0`) and stopping (`stop/0`) the server process.

The server module also provides interface functions for the client application:

- `connect/1` – connects the given client (`Cli`) to the server;
- `disconnect/1` – disconnects the given client (`Cli`) from the server;
- `do/3` – asks the server to execute the given function (`Fun`) from module (`Mod`) on the given table (`Tab`).

```erlang
-define(Name, job_server).
%%%%%% Client interface %%%%%%%%%%%%%%%%%%%%%%%%%%%%%
connect(Cli)        -> ?Name ! {connect, Cli}.
disconnect(Cli)     -> ?Name ! {disconnect, Cli}.
do(Mod, Fun, Tab) -> ?Name ! {do, Mod, Fun, Tab}.
%%%%%% Server interface %%%%%%%%%%%%%%%%%%%%%%%%%%%%%
start() ->    register(?Name, spawn_link(?MODULE, init, [])).
stop()  ->    ?Name ! stop.
%%%%%% Server implementation %%%%%%%%%%%%%%%%%%%%%%%%
init()->
    process_flag(trap_exit, true),
    ?MODULE:loop([]).
loop(State)->
    receive
        stop ->
            ok;
        {connect, Cli} ->
            ?MODULE:loop([Cli|State]);
        {disconnect, Cli} ->
            ?MODULE:loop(lists:filter(fun(A) ->
                                          A /= Cli
                                      end, State));
        {do, Mod, Fun, Tab} ->
            handle_job(Mod, Fun, Tab),
            ?MODULE:loop(State)
    end.
handle_job(Mod, Fun, Tab) ->
    Data = ets:select(Tab, [{{'$1','$2'},
                             [{'/=', '$1', result}],
                             ['$$']}]),
    Result = Mod:Fun(Data),
    ets:insert(Tab, {result, Result}).
```

Figure 1: Job server skeleton code

The server implements the function `init/1` to initialize the server process and the iterating function (`loop/1`) that receives messages and performs the asked tasks. The function `loop(State)` stores the connected clients in the server state variable (`State`). If it receives the message `stop` then terminates. If it receives the {`connect, Cli`} message, then updates the server state with adding the new client to the list. If it receives the {`disconnect, Cli`} message, then updates the server state with removing the client from the list. If it receives the message {`do, Mod, Fun, Tab`}, it extracts the necessary data from the provided table, executes the given function and writes the result to the table and continues accepting jobs.

The client module provides an interface function (`start/1`) to start the client application. The function connects to the server process and creates a named, public `ets` table (`data`). In the next step it spawns a new input reader process and starts to execute the function `loop/2`. The function `input/1` reads commands iteratively from the input and sends these commands to the parent process. If it reads the atom `quit`, it stops reading input.

The function `loop/2` receives messages from the function `input/1` and forwards the jobs to the server. If it receives the atom `quit` from the input process it disconnects from the server

```
start(Client) ->
    server:connect(Client),
    ets:new(data, [named_table, public]),
    spawn(?MODULE, input, [self()]),
    loop(data, Client).

loop(Tab, Name) ->
    receive
        quit ->
            server:disconnect(Name),
            io:format("~p~n", [ets:match(Tab, {result, '$1'})]);
        {job, {Mod, Fun}} ->
            server:do(Mod, Fun, Tab),
            loop(Tab, Name)
    end.

input(Loop) ->
    case read_input() of
        quit ->
            Loop ! quit,
            ok;
        Job ->
            Loop ! {job, Job},
            input(Loop)
    end.

read_input() ->
    [ets:insert(data, Data) || Data <- init_data()],
    returns_the_job_to_be_executed().
```

Figure 2: Client skeleton code

and prints the results.

There are several processes in our example. There is a client process to communicate with the server, a client process for input reading, a server process, and processes to start and stop the server. We note here that the last two operations can be performed from the same process. Our goal is to present a model to represent these processes and the relations among them.

4 Representing Process Relationships

Based on our case study we will illustrate how we detect and build the communication model of Erlang programs to represent the relationships between processes. In this chapter we focus on two types of relationships: relationships through message passing and hidden dependencies through ETS tables.

We represent the process relationships in a labelled graph ($G = (V, E)$) that describes the communication of Erlang processes. The vertices ($v \in V$) of the graph are the processes. We use the `ModuleName:FunctionName/Arity` triple to identify the process p, and if p is registered we also use its name. The labelled edges of the graph ($e \in E$) represent:

- process creation ($\{spawn, spawn\_link\}$),
- process name registration ($register$),
- message passing ($\{send, Message\}$ - labelled with a tuple containing the sent message),
- ETS table creation ($create$),
- reading from an ETS table ($\{read, Pattern\}$ - labelled with a tuple containing the selection pattern),
- writing into an ETS table ($\{write, Data\}$ - labelled with a tuple containing the inserted data).

4.1 Identifying Processes

The dynamic nature of Erlang makes the static process detection hard, therefore we use data-flow reaching [12, 11] to calculate all possible values of expressions which contain the necessary information for process detection. A *Data-Flow Graph (DFG)* is built based on the syntax and semantics of the language containing the direct relations among the expressions of a given program. We can calculate the indirect data-flow relation based on the DFG as a transitive-reflexive closures of the direct relations. This closure calculation is called *data-flow reaching.*

We identify different types of process nodes for functions which take part in communication (*Identification Algorithm*):

1. A process node p_i is created in the graph for each `spawn*` call.
2. A process node is present in the graph for each function (f) which takes part in communication (when f sends or receives messages or spawns a new process). In this case we have to identify whether the function f already belongs to a process from the first group. Therefore, we calculate the backward call chain of the function f. If the backward call chain contains a spawned function, then the function f belongs to the process of the spawned function p_i. Thus, the communication edges generated by f are linked to p_i.
3. When a function g takes part in communication, but its backward call chain does not contain a spawned function we create a new process node p_j. This process is identified with the module, the name and the arity of g if there is no communicating function in the backward call chain. Otherwise we select the last communicating function h in the call chain and we identify the created p_j process with module, name and arity of h.
4. There is a "super process" (SP) in the graph which represent the runtime environment. It represents the fact that the communicating functions can be called from the currently running process, for example from the Erlang shell.

We create the listed process nodes in a predefined order:

1. At first we collect the spawn expressions from the source code and add them to the set S.
2. We create a process node p_s for each $s \in S$ spawned process and add it to the set P_s.
3. We collect the communicating functions C and create process nodes for them (using the second and third step of the identification algorithm).
4. We link every created p_s ($s \in S$) process to its parent process with a $spawn*$ edge.

5. We select each register expression from the source code and add the appropriate *register* link to the graph.

6. Each process node p_j that has no parent ($p_j \notin P_s$) is linked to the node SP.

To identify a spawned process we have to calculate the possible values of the actual parameters of the function call `spawn*(ModName,FunName,Args)`. We calculate the values based on the result of the data-flow analysis [12], while it is a static analysis it is always an approximation of the real dynamic information.

4.2 Message Passing

The next step is to add the message passing edges to the graph. We calculate the message passing edges based on the data-flow information presented in [11]. That analysis links the sent and received messages with a $flow$ edge in the Semantic Program Graph of RefactorErl (Section 2.1). We use the following algorithm to calculate the communication edges:

1. We select the message sending expressions from the source code and add it to the set M.

2. For each $m \in M$ we calculate the receive expression r_m which receives the sent message.

3. We calculate the containing process node p_m for each $m \in M$ expression and the containing process node p_r for each r_m, and add the $\{send, Message\}$ link from p_m to p_r (where $Message$ is the sent message from the expression m).

4.3 Hidden Communication – ETS tables

ETS tables can be considered as a form of shared memory in Erlang: one process can write some data in it and share it with other processes. Therefore, ETS tables represent hidden communication among processes, thus we add them as a special process relation to our model. Every created ETS table is added to our graph as a special process node, and read and write operations as special message passing edges:

1. The first step is to select the created ETS tables and add them to the set E.

2. For each $e \in E$ table we create a process node p_e and link it to the parent process. The parent process is the process of the function which calls the function `ets:new/2`.

3. The next step is to detect whether the found table can be referred using its name. We analyse the option list (the second parameter of the call `ets:new/2`) and calculate its possible values by data-flow reaching. If the `named_table` atom is one of them, then we have to calculate the possible names of ETS table by data-flow reaching, and add the name of the ETS table as an attribute to the process node.

4. Each ETS table manipulation (e.g. `insert*, delete*`) is added as write operation to the graph between the ETS table node and the process of the expression calling the `ets` functions.

5. Each query operation (e.g. `match*, select*`) is added as read operation to the graph between the ETS table node and the process of the expression calling the `ets` functions.

Calculating *write* edges. The relation $\overset{\mathbf{1f}}{\leadsto}$ denotes the first order data flow reaching [12]. $n_1 \overset{\mathbf{1f}}{\leadsto} n_2$ means that the value of node n_2 can be a copy of the value of node n_1. We use this relation to calculate the ETS *write* edges in the following steps:

1. Collect the function calls which refer to an ETS table and change it. Add it to the set W. For example, `ets:insert(Tab, Data)`.

2. For each $w \in W$ call calculate the referred ETS table with data-flow reaching. At first the possible values of w_1 (denoted with E_w) have to be calculated: $e \in E_w$ and $e \overset{\mathbf{1f}}{\leadsto} w_1$ (where w_1 is the first parameter of the call expression w, e is an expression which value can flow to w_1). If there is an expression $e \in E_w$ which is an atom and its value is *some_name*, then we select the process node (p_e) referring to the named ETS table *some_name*. Otherwise we should find a table reference in E_w which creates the ETS table (a call to `ets:new/2`), and select the process node p_e of the created ETS table.

3. Determine the process node where the call `ets:insert/2` belongs to: p_w. To identify this process we use a similar algorithm that was presented in the second and third step of the *Identification algorithm*. We determine the function f which contains w, and calculate the process of f.

4. Connect the process node p_w and the found ETS table node p_e.

Calculating the *read* edges works similarly to *write* edges, but we have to analyse the query functions of the `ets` module.

The function `ets:rename/2` also has to be considered, because it changes the name of the ETS table, and only the new name can be used. To handle this we will refine our analysis using the Control-Flow Graphs of Erlang programs [11].

5 Motivating Example – Resulting Model

We will illustrate the presented algorithms step by step using the client-server example from Section 3. We highlight only some element of the algorithm here.

Process identification.

- There are two spawn expressions in our example at line 7 in the client module, and in line 21 in the server module. We add their expression nodes to the set S.

- We collect all functions that take part in communication, e.g : We create the process node `client:start/1`, because there is no spawned process in its backward call chain, and the last function in its backward call chain is the function `client:start/1` itself; but we do not create a process node for `client:loop/1`, because it contains `client:start/1` in its backward call chain and it already has a process node.

Message passing information.

- The `pid` of the recipient process in the message sending expressions of the server code comes from the macro `?Name`, thus the `receive` expressions for the server code must be in the spawned and registered process `job_server`. This is the receive expression from the body of the function `server:loop/1`. The `pid` in the code of the client is a variable,

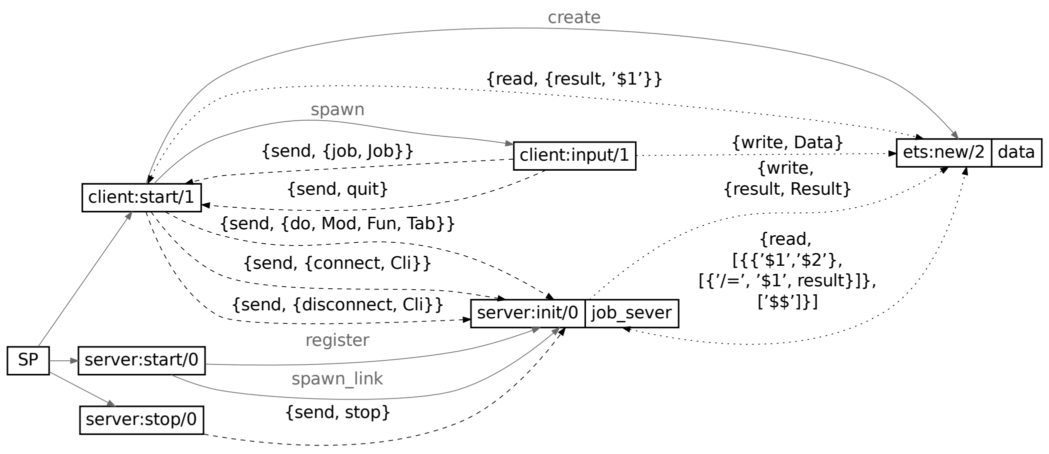

Figure 3: Communication Model (extended with ETS usage)

so we use data-flow reaching to calculate its value, which is identified as the result of the function `self()` in the body of the function `client:start/1`. Therefore, the receive expression must be in the process of `client:start/1`: it is the receive expression from `client:loop/2`.

ETS usage information.

- We create the process node `ets:new/2` in the graph and link it to its parent process `client:start/1`.
- We select the write operations from the source code: there is a call `ets:insert` in the 32st line of the server and in the 29st line of the client. We link the process node of the manipulated ETS table to the process node of the caller function. For example in the server code we create a link from the process `server:init/0` to `ets:new/2, data`.

The graph, extended with the ETS usage, is shown in Figure 3.

6 Related work

Different static analysis tools have been designed for Erlang, but none of them can create the communication graph at compile time.

Analysing parallel and distributed Erlang software are key research topics. For example, one goal of the ParaPhrase project [7, 8] is to analyse Erlang programs, statically detect parallel patterns and transform the source code to adjust the advantages of manycore architectures.

The RELEASE project [10] aims to help in developing well designed scalable distributed Erlang software by defining the SD Erlang (Scalable Distributed Erlang). They define language primitives to create process groups. The frequently communicating processes should be placed to the same group on the same Erlang node. However, the explicit process placement based on the communication flow, is not straightforward for the programmers. Therefore SD Erlang aims to design automatic process placement based on connectivity distance metrics. Our analysis could be extended to handle and visualise SG Erlang as well.

The goal of the static analyser tool, Dialyzer [4], is to identify software discrepancies and defects, such as type mismatches, race condition defects, etc. The tool detects message passing by analysing the Core Erlang code [2], and can report concurrent programming defects.

7 Summary

We presented a model to represent the communication between Erlang processes. The model contains the most important relations between processes, i.e. process hierarchy, communication through message passing, and hidden relations generated by ETS usage.

We store the result of our analysis in the graph of RefactorErl. These results can help in code comprehension tasks of the Erlang developers. One possible way is the visualisation of the relations in a graph, but we also want to integrate the result of the analysis into the semantic query language of RefactorErl.

We want to make the presented algorithms more efficient and more precise based on the result of the control-flow analysis and by adding background knowledge about the Erlang/OTP.

References

[1] 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.

[2] Richard Carlsson. An introduction to core erlang. In *Proceedings of the PLI Erlang Workshop*, 2001.

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

[4] *The DIALYZER*, 2014. `http://www.it.uu.se/research/group/hipe/dialyzer`.

[5] Ericsson AB. *Erlang Reference Manual.* `http://www.erlang.org/doc/reference_manual/part_frame.html`.

[6] Zoltán Horváth, László Lövei, Tamás Kozsik, Róbert Kitlei, Melinda Tóth, István Bozó, and Roland Király. Modeling semantic knowledge in Erlang for refactoring. In *International Conference on Knowledge Engineering, Principles and Techniques (KEPT)*, pages 38–53, Cluj-Napoca, Romania, July 2009.

[7] ParaPhrase project: Parallel Patterns for Adaptive Heterogeneous Multicore Systems. `http://paraphrase-ict.eu/`, 2014.

[8] ParaPhrase Enlarged project. `http://paraphrase-enlarged.elte.hu`, 2014.

[9] RefactorErl Home Page. `http://plc.inf.elte.hu/erlang/`, 2014.

[10] RELEASE project: A High-Level Paradigm for Reliable Large-Scale Server Software. `http://www.release-project.eu/`, 2014.

[11] 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.

[12] Melinda Tóth, István Bozó, Zoltán Horváth, and Máté Tejfel. First order flow analysis for Erlang. In *Proceedings of the 8th Joint Conference on Mathematics and Computer Science (MACS)*, 2010.