Extracting the processes structure of ERLANG applications

Jan Nyström and Bengt Jonsson

Department of Computer Systems
Department of Information Technology
Uppsala University

P.O. Box 337, S-751 01 Uppsala
Sweden
{jann,bengt}@docs.uu.se

Abstract

ERLANG is a concurrent functional language, especially tailored for distributed and fault-tolerant software. Its strength has been demonstrated by several successful commercial applications. An important part of ERLANG is its support for fault tolerance by implementing failure-recovery, through organising the processes of an ERLANG system into trees of processes, in which parent nodes monitor the failure status of their children and are responsible for their restart.

In this paper we present ongoing work on a tool that captures the static part of the global process structure of an application ERLANG. The tool extracts the process structure from the applications source code, and presents it as a process creation graph. The extracted process structure provides a starting point for understanding and analysis of aspects such as fault handling in ERLANG applications.

1 Introduction

ERLANG is a concurrent functional language, especially tailored for distributed and fault-tolerant software [Armstrong96, Dacker00]. The strength of the language is witnessed by several successful commercial applications [Blau99, Hinde00, Bluetail]. Distinguishing features of ERLANG include support for light-weight processes, asynchronous message passing together with processes creation, termination and communication as integral parts of the language. The Open Telecom Platform (OTP) [OTP] provides a number of libraries that support program design patterns which commonly occur in concurrent distributed software. Examples of such patterns, called "behaviors" in OTP are : event handlers, generic servers, and finite state machines.

An important part of ERLANG is its support for fault tolerance by implementing failure-recovery. OTP provides support for organising the processes of an ERLANG system into trees of processes, in which parent nodes monitor the failure status of their children and are responsible for their restart. The *supervisor behaviour* in OTP is used to program processes which monitor a set of children. Failures of the children processes are communicated to parent supervisor processes through links. A supervisor, which receives notification of a failure of a child may, then restarts a new copy of the child. The failure recovery mechanisms of ERLANG and OTP make it possible to write clear code, which is not obscured by defencive code. In order to use this style of programming, it is important to determine and understand the global process structure of the system. This structure includes the processes that an ERLANG system has, and which roles they have with respect to each other, in particular what processes are involved in the handling of process failures and which processes will be affected by a failure.

Unfortunately, the process structure of an ERLANG program is often obscured by good coding principles which employ abstractions to hide details; an ERLANG program is structured according to modules and functions, whereas process creation and communication may occur anywhere in the code. Common patterns of process creation, setup and communication will be hidden in library modules of the applications. Some of the process structure can be inferred when systemwide libraries, such as the OTP behaviours, are used. But currently, to obtain a global view of the system, with regards to the processes and their roles, we have to rely on external documentation or manual inspection of the source code.

The aim of our work is to capture the static part of the global process structure of an ERLANG application by analysis of the source code. By "the static part" we mean the processes started when the application is started and are to remain running (possibly restarted to handle faults) until the application terminates. This process structure will in essence be a truncated creation tree of the processes started by the application: the tree is truncated to omit processes that are created dynamically, to handle a certain situation during the application's execution.

As an analogy, traditional interprocedural program analysis extracts the call-graph of a program (e.g., [Agrawal00]); the call-graph is a graph of procedures, whereas our aim is to extract a graph of processes. Our aim is to some extent related to the area of fault analysis, where one is also interested in the potential effects of faults in a system component (e.g., [Sampath95]).

The process structure extracted should be possible to present to the designer for visual inspection, with the possibility to choose different views depending on the information sought. As an example, when trying to determine what parts of the application will be affected by an undue process termination a user may want to see which process are linked. An example of the type of information we have considered is found in Figure 2, which shows the creation tree of the OTP library application `os_mon` that monitors the underlying operating system.

The extracted process structure should also form a basis for further checks and analyses, since our long term goal is to add such to the tool. Below are some examples.

- With each OTP application is associated a *resource file* which, among other things, describes which module is responsible for starting the application, what other applications are required or included, and which process names will be registered by the application. OTPs *application behaviour* will perform certain checks on this information, such as ensuring that the registered process names are not already used, or implicit ones such as loading included applications. It would be desirable to check that the information in the resource file actually corresponds to the program, e.g., that the included applications are started and that the names listed as registered will be used by processes that are created.

- An important aspect of fault-handling is that handling of the fault embraces all parts truly affected and does not unnecessarily inconvenience parts that are not affected by the fault. In the context of ERLANG applications where the faults mainly consists of undue process termination, the treatment is restart of the terminated process, possibly together with other processes still running. It is thus desirable to determine what parts of the system would be restarted on the termination of a particular process, if indeed any.

- An important parameter of the OTP *supervisor behaviour* is how often (with how high frequency) the supervisor will restart its children before terminating itself. If there is a hierarchy of supervisors it is desirable to know how often a process, or group of processes, would have to terminate before the application is terminated. The analysed restart frequency can be compared with constraints on desired maximal restart frequencies. For instance, it is often desirable that all supervisors in a middle management role, i.e., only supervise supervisors, has the restart frequency set to 0.

- Creating a process and linking it to its creator can be achieved in two different ways: either the process is spawned and then a call to link the process is made by the parent or child, or one can use the `spawn_link` built in function to ensure that the spawning and linking is indivisible. The latter way of creation and linking is important in supervision structures since otherwise one of the parties can terminates before the link has been established. It is thus desirable to ensure that all process creations within the supervision structure is performed using `spawn_link`.

Overview

In the remainder of the introduction section we will examine related work. The rest of the article is organised as follows: Section 2 introduces the parts of ERLANG and OTP which are germane to this article but not in the vocabulary of all ERLANG programmers. In Section 3 we will first present the structure we capture and rationale for the included parts, followed by a description of the construction of the structure. We end Section 3 giving the limitations of the work and how we handle incomplete or incorrect source. In Section 4 we present our conclusions and in the last Section 5 we discuss further work.

Related work

Erlang To analyse properties of a program one can use abstract interpretation; in the context of ERLANG this has been explored by Huch in [Huch99]. Huch views an ERLANG system as a set of expression evaluations in the context of the identity of the processes executing the expressions and their message queues. The abstraction consists of truncating the terms in the expression at a predefined depth. It is mentioned in the paper how one could tailor the interpretation so that for selected terms the terms are either kept as they are or truncated at a greater depth. The interpretation can only handle tail recursive programs and does not handle exceptions, links, nor process termination. The work of [Huch99] has been further developed in [Huch01] where he handles non tail recursive calls, through a technique of jumps which makes his approach much more realistic for real life programs although he does only cover a subset of ERLANG.

An ongoing effort for support in proving properties of ERLANG programs is the ERLANG Verification Tool(EVT) [Arts00] developed at the Swedish Institute of Computer Science in cooperation with Ericsson. EVT is a theorem prover with an embedding of the language in the proof rules. One of the results of this effort is a layered small step semantics of a large subset of the ERLANG language [Fredlund01]. This semantics covers a larger part of the language then the semantics used as base for the abstract interpretation in Huch papers. There is also a definition of the semantic of the sequential fragment of Core ERLANG [Carlsson01].

Finite state methods An interesting method of coming to grips with concurrent systems is to reduce them to finite state models by means of extracting a control skeleton from the code. Holzmann has extracted finite non-deterministic SPIN models for a C with threads, by abstracting statements and procedures not connected to the properties under inspection, in the tools FeaVer [Holzmann00a] and `AX` [Holzmann00b]. This method relies on user defined abstractions, i.e. deciding what procedures and variables are unimportant, and consequently if the user definitions are incorrect executions that violates the properties under investigation might not be found.

A similar approach, which does not rely on user definitions, is applied by Corbett and colleagues for JAVA in the Bandera project [Corbett00] where shape analysis is used to determine what variables are only accessible from one thread.

A more theoretically oriented approach is followed by Nielson et al [Nielson98], which derive finite-state control skeleton from programs in Concurrent ML where values abstract to their types.

Recently Arts and Earle has pursued similar ideas for ERLANG, translating ERLANG programs into μCRL models which can be model checked by CÆSAR/ALDÉBARAN tool set. In [Arts01] they investigate a simplified version of a resource locking mechanism in the AXD 301 ATM switch [Blau99].

2 Processes, behaviours and applications

ERLANG [Armstrong96, Barklund99] supports creation, management, communication, and failure handling, through a number of built in functions (BIF for short) and through the OTP libraries.

Processes and Failure Handling The basis for failure handling is that *links* can be created between cooperating processes. If an error occurs in a process, such as an uncaught exception, it will be abnormally terminated. By default, all processes linked to an abnormally terminated process will also be terminated. However, this default action can be overridden so that "supervisor" processes can monitor termination of "application" processes, and clean up and restart them. One of our aims is to extract this "supervision" structure from application code in order to analyse its properties for failure recovery. The extraction is performed by analysing the BIFs and statements in the program that affect this structure. These functions are explained below.

A process is created through a call to one of the family of `spawn/spawn_link` functions. All the `spawn` functions have as an argument what function the created process should execute. The full range of functions in the `spawn` family is given by optional arguments in different combinations, where the optional arguments are on which erlang-node the process should be created, and arguments to the created process start function. The `spawn_link` versions of the spawn functions behave in the same manner except they link the parent and child processes while creating the child. The `spawn` functions return the Process identifier (pid) of the created process.

Two processes can be linked, either by a call to `spawn_link` or to the BIF `link` with the other process pid as an argument. A process that is linked to another process will be informed when its linked peer terminates, in a manner which is determined by the boolean process flag `trap_exit`. In the default case, when `trap_exit` is false, the "informed" process is simply killed and will only be informed if the other process terminated abnormally; otherwise (if `trap_exit` is true) it will be informed through a special message, which is sent irrespective of whether the other process terminated normally or abnormally.

There exist a number of BIFs to examine and monitor processes, such as `is_process_alive`, `processes`, `process_info` and `monitor`.

Communication The communication is asynchronous via message passing with random access mailbox queues in which the order is guaranteed for messages from the same process but not for messages sent from different processes. The nonblocking send BIF `!` is called as "`Pid ! Message`" where the message may be any ERLANG term and the pid must be a correct process identifier. The mailbox is accessed through the receive statement which matches a number of clauses against the messages in the mailbox queue. The receive statement is blocking, but has an optional timeout clause that states how long it will wait for a matching message, after which time it will evaluate the timeout expression.

In general a process has to know the pid of another process in order to examine, terminate or communicate with it. But if a process is registered, that registered name can be used instead for the pid in many cases. For those BIFs that require a pid we can look up the pid of a registered process using the `whereis` BIF.

Behaviours Open Telecom Platform [OTP] defines a number of *behaviours* that are realisations of design patterns addressing common problems of distributed concurrent systems, such as telecommunication systems.

When using these behaviours you write a callback module that implements the actual behaviour, whereas the library module will take care of the parts shared by all processes with this behaviour. For example in the *supervisor behaviour* the callback module will contain the `init` function which computes what children should be started, and the supervisor library module will handle the actual starting, supervision, restarting and stopping of the child processes.

The behaviours defined in OTP are:

application is a packaging of system components, and has a number resources such as modules, registered names and processes. The processes can be loaded, started and stopped together and it can be checked that the needed resources are available when loading the application. Associated with an application is not only a callback module, but also a resource file which declares the resources needed by the application, such as the names that will be registered by the application, and what other applications have to be running before the application is loaded.

An application can be one of three different types: `permanent`, `transient` or `temporary`. The type decides what happens to the other applications when it terminates, in a similar way to the children of a supervisor.

gen_event is an event handler manager, i.e., manages a number of event handlers. The event handlers can be added and removed dynamically. The event manager will apply every present event handler, via a call to the callback module, to a received call or notification.

gen_fsm is used to write finite state machines, where the callback module has a function for each state describing the transitions made on events.

gen_server provides a simple way of writing the server part of server client applications, where the gen_server module handles debugging and termination of the parent.

supervisor is used to structure applications for fault-tolerance. The supervisor has a number of children which it monitors through links and restarts when needed. The callback module is only used to determine what children will be statically started by the supervisor, although children can also be added dynamically to the supervisor. It should be noted that if the supervisor is part of a supervision hierarchy and can be restarted on abnormal termination, only its statically started children will be restarted.

A supervisor has a restart strategy that determines what it should do when a child terminates, the strategies are:

`one_for_one` where only the abnormally terminated child is restared,

`one_for_all` where all children are shutdown and then restarted,

`one_for_rest` where all children started after the terminated child are shutdown and restarted.

The children of the supervisor can have tree types determining in which cases they should be restarted:

`permanent` in which case the process is always restarted if it terminates,

`transient` in which case it is restarted only if it terminates abnormally,

`temporary` in which case the process is not restarted.

There is a limit to the number of times a supervisor will restart its children, given by two numbers `maxR` and `maxT`. If more than `maxR` restarts are made within `maxT` seconds the supervisor terminates.

supervisor_bridge which enables a subsystem, not originally intended to be part of a supervision hierarchy, to be connected to a supervision hierarchy. The process having the *supervisor_bridge behavior* behaves as a bridge between the supervision tree and the subsystem.

3 The captured process structure

In this section we will describe the process structure as we can capture it today, and the rationale for the information included. Then we will explain how we extract the structure and what the current limitations are.

As an example we will use the result of applying the tool to the OTP application `os_mon`, the result an ERLANG term is shown in Figure 1. The `os_mon` application monitors the underlying operating system for disk, memory and CPU usage. We will also refer a particular view of the ERLANG term, presented in Figure 2 where the process creation tree of `os_mon` is shown.

What will the process structure contain and why

We will present, one by one, the types of information that we will collect for the statically created processes.

If the process has a registered name (a process can have at most one registered name) we note this since it gives processes that do not have its pid the ability to communicate with it through the registered name. As example of a registered process you can see in Figure 1 that the supervisor of the `os_mon` application is registered as `os_mon_sup`. In the process creation tree, presented in Figure 2, all processes are symbolised via a circle or a box and if the process has a registered name, it is written inside the circle or box.

Associated with each process are three process flags that can be set through the `process_flag/2` BIF: `trap_exit`, `error_handler` and `priority`. We will only note the value of the `trap_exit` boolean flag which determines how we are informed that a linked process terminates as mentioned above. An example of a process that sets the `trap_exit` process flag to true, is the process registered as `disksup`. It is shown in the process creation tree as a note beside the circle denoting `disksup` in the tree.

An important aspect in view of fault-tolerance is which processes are linked to each other. In the example we have no calls to `link` but all spawns are `spawn_link` and from that we can deduce that all the process are linked to their parent and no other links exist. In the process creation tree we show that a process is parent to another by a directed edge, and annotate the edge with *link* if it was created using a `spawn_link` call.

To keep track of the process creation structure we of course have to keep track of the children of a process, so each spawn is noted. In the example the processes `disksup`, `memsup` and `cpu_sup` are a children of `os_mon_sup`. And as we have mentioned in the process creation tree the parent child relation is shown trough directed edges.

It is also important to keep track of any communication that take place while the process structure is created since this can be vital to put processes in a correct start state. In the example the `disksup` processes sends itself a `timeout` message to ensure that it starts with handling a timeout and initiate

```
{application,
  os_mon,
   temporary,
  [os_mon_sup, os_mon_sysinfo, disksup, memsup, cpu_sup,
   os_sup_server],
  [],
  [{supervisor,
      <0.1.1>,
      one_for_one,
      4,
      3600,
      [{disksup,
          <0.1.2>,
          {disksup,start_link,[]},
          permanent,
          2000,
          worker,
          [{spawn_link,<0.1.2>,disksup,init},
           {behaviour,<0.1.2>,gen_server},
           {register,<0.1.2>,{local,disksup}},
           {exit, <0.1.2>, {{unknown_os_type,{unix,linux}},
                            {disk_sup,get_os,[]}}},
            {process_flag,<0.1.2>,trap_exit,true},
            {send,<0.1.2>,timeout}]},
       {memsup,
          <0.1.3>,
          {memsup,start_link,[]},
          permanent,
          2000,
          worker,
          [{spawn_link,<0.1.3>,memsup,init},
           {behaviour,<0.1.3>,gen_server},
           {register,<0.1.3>,{local,memsup}},
           {process_flag,<0.1.3>,trap_exit,true}]},
       {cpu_sup,
          <0.1.4>,
          {cpu_sup,start_link,[]},
          permanent,
          2000,
          worker,
          [{spawn_link,<0.1.4>,cpu_sup,init},
           {behaviour,<0.1.4>,gen_server},
           {register,<0.1.4>,{local,cpu_sup}}]}]},
    {register,<0.1.1>,{local,os_mon_sup}}]}
```

Figure 1: Static global process structure of the OTP application os_mon.

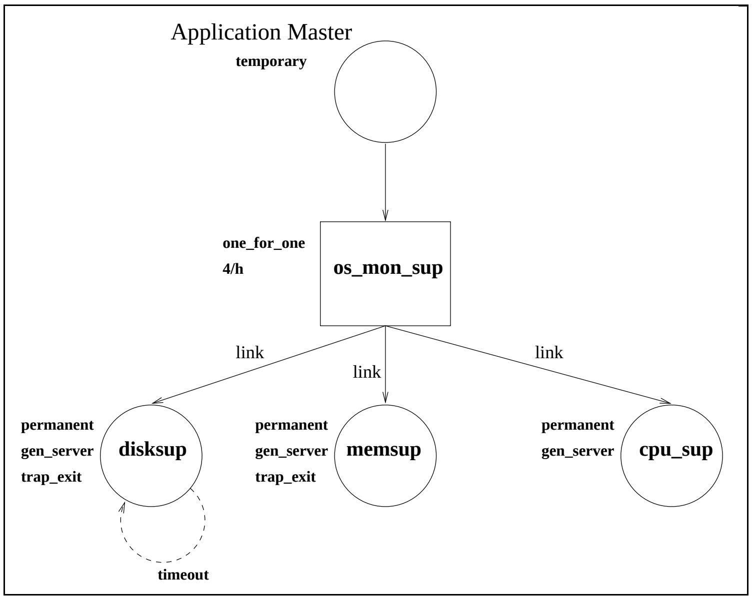

Figure 2: Creation tree for the OTP application `os_mon`.

itself. The messages are presented as directed dashed edges in the processes creation tree.

What behaviour a process has provides a good succinct information on how the process will behave. In the example the *application*, *supervisor* and *gen_server behaviours* are represented, which only leaves out the *gen_event*, *gen_fsm* and *supervisor_bridge behaviours* of the OTP defined behaviours. In the process creation tree the behaviours are represented differently from each other, the application is annotated by the application_master process which will be added behind the scenes by the application library call `start`. The processes that have the *supervisor* and *supervisor_bridge behaviours* will be annotated by the box representing the process as opposed to the other processes circles. The remaining behaviours will be annotated next to the process circle.

Each behaviour has a set of parameters, but we are only interested in a few of these and will present below the ones we will add to the extracted structure. For the *gen_event*, *gen_fsm*, *gen_server* and *supervisor_bridge* we do not add any of the parameters.

In the case of the *application behaviour* we will add the name, the type of application, registered processes and the included applications. In the example the name is `os_mon`, the type is `temporary`, the registered names are `os_mon_sup`, `os_mon_sysinfo`, `disksup`, `memsup`, `cpu_sup`, `os_sup_server` and the there are no included applications. It can be noted that no processes will be registered as `os_mon_sysinfo` or `os_sup_server`. In the process creation tree we will only note the type `temporary`.

For the *supervisor behaviour* there are several interesting parameters: the restart strategy, the maximum restart frequency and the types of the children. For the children we will also keep track of the (module, function) pair that is used to start the child as well as what argument were given. In the example the restart strategy is `one_for_one`, the maximum restart frequency is 4/hour and all the children are `permanent`. In the process creation tree the restart strategy and maximum restart frequency is annotated next to the supervisor box whereas the per child parameters are annotated by the child process.

All of the supervisor parameters that we add to the extracted process structure would be needed to perform the analysis sketched in introductory section. One could well imagine that we would like to keep track of the callback module of each behaviour; we have not done so but it can easily be added.

The observant reader will have noted in the extracted structure of `os_mon` that there is an exit of one of the processes. This is because the `os_mon` application does not support linux. Apart from the exit and that the `cpu_sup` process does not set `trap_exit` to true there is nothing surprising in the process graph.

How the process structure is constructed

Our process structure tree is constructed using an instrumented evaluator for Core ERLANG. Core ERLANG [Carlsson00] is an intermediate format used in the OTP ERLANG compiler where syntactic sugar has been removed and a restricted set of constructs and formats are used. There are also constructs in Core ERLANG that are not present in ERLANG, such as, "let" and "letrec" which are used to replace explicit matching and local functions generated by list expressions.

The reason, as to why we instrumented an evaluator, rather then trying a static program analysis, is that when starting an application and generating the static processes the programmer has the use of, and in many cases also need for, the full expressive power of ERLANG. Hence we would have to deal with the full language in order to determine what processes where generated by an application.

Our reason for writing an evaluator of Core ERLANG as opposed to ERLANG itself (for which there already exist an evaluator in the distribution of ERLANG) is that it was much simpler to deal with code where matters such as macros, files inclusion and records had already been taken care of and it has fewer constructs.

The analysis does so far consist of evaluating a call to `application:start/1`, where the result will contain the value computed, collected "side-effects" and the resulting environment. The collection of "side-effects", or rather processes creation and management, is on the form presented in Figure 1. The resulting environment will contain the modules and applications loaded during the evaluation.

The evaluation will handle all constructs in the usual manner except `receive` and function calls which are divided into five categories:

Calls to local functions in the current module. In this case we simply evaluate the call.

Calls to functions in library modules which do not influence the processes structure. In this case we make concrete ERLANG terms of the arguments and return the abstraction of the value resulting from applying the function to the concrete arguments. An example of such a module is `lists`.

Calls to functions in library modules which we do not support. In this case we write a warning informing the user that the function call was not performed and return $\perp$, which is a value not present in the program. This we will call a nop-behaviour. An example of such a module is `proc_lib`.

Calls to functions in library modules which influence the processes structure. In this case we replace the calls with calls to internal evaluator functions that will modify the behaviour and make note of the "side-effects". How these functions are handled will be described in more detail below. An example of such a module is `supervisor`.

Calls to functions that are not in the local module or a library module. In this case we evaluate as normal, possibly loading the module.

Receive The `receive` statement is handled specially since our evaluator does not spawn any processes or allow the creation of ports, and consequently there is in many cases no one who could send the message we are waiting for. The `receive` statement always return the special value $\perp$.

Even if we should create processes we would have to set timeouts to guarantee termination, since we might not get a message due to an error somewhere else in the system. Another reason to set timeouts would be that ports are external systems which we can not start since we have no control over them.

Function calls Below, we roughly describe for each library module, which is treated in a special manner, how the functions in that module are treated.

erlang The `erlang` module consists of the built in functions of ERLANG, many of them dealing with processes that must be dealt with in a special manner. First there are a number of unsupported functions which we replace with a nop-behaviour, e.g. `cancel_timer, disconnect_node, get, load_module, monitor, open_port, process_info, registered` and `trace`. Some of these nullified functions, e.g., `registered` and `get`, we want to handle properly but we have not, as of now, had the time to implement.

The functions below have interesting effects visa vi the processes and we handle them as follow:

`apply` This is a meta application function which calls the argument function, the calls are handled in the same manner as if the call was made directly.

`exit, fault, halt, link, process_flag, register, unlink, unregister` These functions are treated in the same manner, nothing is executed and $\perp$ is returned. We will however include the action among the "side-effects" we collect. Note that `exit, fault` and `halt` would have terminated the execution and in the case of `halt` have halted the erlang-node.

`self` This function returns the pid we use internally for the current process, it is not a pid connected with any actual process.

`spawn, spawn_link` These functions create a new unique pid which is not connected to any actual process. It is a syntactically correct pid in the sense that a pid type check of the pid will be true. This pid is returned but no process is spawned. We make a note that a spawn has taken place and the pid of the "spawned" process. In case where it was a `spawn_link` we will also note that a link was setup.

`throw` In this case we will throw the concrete value of the argument to throw.

The remaining functions in the `erlang` module are treated as functions belonging to a module without influence on the process structure.

application The functions in the `application` module will, with the exception of the three described below, be treated as functions in an unsupported library module.

`start` The application to be started is loaded into the evaluator, if it has not been loaded already, i.e., the *application resource file* is read, parsed and stored in the evaluator environment. The *application behaviour* and parameters of the application are noted, and a call to the `init` function of the application's callback module is evaluated.

`get_key, get_env` The `get_key` and `get_env` functions look up information in the *application resource file* and the key, value list called `env` in the *application resource file*, respectively. Since we have loaded the *application resource file* into the evaluator we will have to looks these values up in the evaluator environment.

gen_event, gen_fsm, gen_server, supervisor_bridge The functions in these modules will, with the exception of the two described below, be treated as functions in an unsupported library module.

`start, start_link` For all these modules the behaviour is noted and the callback module's `init` function is evaluated. There is for both `start` and `start_link` function a version that has an extra `Name` argument under which the process would be registered. In the case where this extra argument is present the process is noted as registered.

supervisor The functions in the `supervisor` module will, with the exception of the one described below, be treated as functions in an unsupported library module.

`start_link` We note the *supervisor behavior* and the callback modules `init` function is evaluated. The result of the call will contain the behavior parameters and description of the children with the per child specified parameters. The parameters are noted and then a call is made to the module, function pair given as start function, for the each child. There is a version of the `start_link` function that has an extra `Name` argument under which the process would be registered in the case where this extra argument is present the process is noted as registered.

Limitations

The most important limitation is that we do not spawn processes, which will have two effects: first we can only follow the creation of the static process structure along application and supervision trees since we only get one spawn further, secondly since we do not spawn processes any communication that would take place between two processes during setup can not be handled.

Another severe limitation we have, is that termination is not guaranteed, since the limiting factor of the execution is spawning. This would normally not be a problem in correct code. However since we in many cases return the extra value $\perp$ as the result of a computation the behavior of the setup may change. A simple example of a nontermination is found in the small function below in Figure 3, where the function `is_process_alive` will return $\perp$.

```
wait_for_termination(Pid) ->
  case is_process_alive(Pid) of
    false -> ok;
    _ -> wait_for_termination(Pid)
  end.
```

Figure 3: Function for which the analysis will loop.

A technically not very challenging limitation is that we handle but few of the library modules that will influence the process structure. To correct this is just a matter of devoting time to implement them.

Partial or partly incorrect source

It has been a decision right from the start that we should try to handle partial and incorrect code to the greatest extent possible. Having a good understanding of the process structure is valuable from the beginning, even when large portions of the application are still to be realised, and may actually help in structuring the remaining realisation.

We can handle those modules that are syntactically correct, i.e., those that pass through the initial phases of the compiler and from which the compiler manages to produce Core ERLANG.

The handling of incorrect code, that is code that generates an exception, is to catch the exception and replace it with a "special" value denoting unknown result($\perp$), we have chosen to use an atom that would not normally occur in the code. A problem with this solution is that incomplete and incorrect code will generate warnings, not only where they initially occur, but also as a result of our handling of the faults which may change the behavior of the code.

Dealing with incomplete code can be handled much in the same manner as incorrect code, the difference is that when we either try to load a module that does not exist, or find a function not exported from the module, we notify the user of the error and let the call result in the special value $\perp$.

4 Conclusions

We have shown, using the simple means of an instrumented evaluator, that we can extract a prefix of the static parts of the global process structure of an ERLANG application. The extracted process structure, although incomplete, provides a starting point for understanding and analysis of such aspects such as fault handling in ERLANG applications.

The evaluation can however only handle the static parts, or rather the parts known to be static, and other methods have to be employed in order to extract and describe the processes created dynamically.

5 Further work

There are two important aspects that have to be addressed: spawning and determining the static/dynamic border of process creation. The spawning can either be achieved by spawning a new evaluator or to let the evaluator contain several processes and explicitly manage scheduling and communication. The spawning avoids the management of processes and communication but we would have to add a framework for collecting the information gathered by the different evaluators, but that seems easier than management of the processes.

Determining the border between static and dynamic process creation can not be achieved in the general case, but some reasonable approximation must be made that is less conservative then the one we currently use. One may use a combined limit of maximal time and number of nested function calls used, the accuracy of this approach is of course highly dependent on the application analysed. An advantage of using a limit approach to the evaluation is that we will ensure termination. Another and interesting approach would be to combine evaluation of Core ERLANG terms with finite state methods, where the dynamic parts are approximated by finite models extracted from the code.

References

[7] [Agrawal00] Agrawal, G., 2000: *Demand-Driven Construction of Call Graphs,* In Proceedings of 9th International Conference in Compiler Construction, ed. D.A. Watt , LNCS Vol.1781, pp.125–140, Springer-Verlag, 2000.

[1] [Armstrong96] Armstrong, J., Virding, R., Wikström, C., Williams, M., 1993: *Concurrent Programming in Erlang,* 2nd ed., Prentice Hall, 1996.

[11] [Arts00] Arts, T., Chugunov, G., Dam, M., Fredlund, L.-å., Gurov, D., Noll, T., 2000: *A Tool for Verifying Software Written in Erlang,* submitted to Software Tools for Technology Transfer.

[18] [Arts01] Arts, T., Earle, C.B, 2001: *Development of a Verified Erlang Program for Resource Locking,* Proceedings of the sixth International Workshop on Formal Methods for Industrial Critical Systems, 2001.

[19] [Barklund99] Barklund, J., Virding, R., 1999: *Specification of the Standard Erlang programming language,* draft version 0.7, `http://www.erlang.org/download/erl_spec47.ps.gz`, June 1999.

[3] [Blau99] Blau, S., Rooth, J., Axell, J., Hellstrand, F., Buhrgard, M., Westin, T., Wicklund, G., 1999: *AXD 301: A new generation ATM switching system,* Computer Networks, No.31, pp.559–582, Elsevier Science B.V., 1999.

[5] [Bluetail] Product information on Bluetail Mail Robustifier, `http://www.bluetail.com/products/bmr/`.

[20] [Carlsson00] Carlsson, R, Gustavsson, B., Johansson, E., Lindgren, T., Nyström, S.-O., Pettersson, M., Virding, R., 2000: *Core Erlang 1.0 language specification,* Technical Report 2000-030, Department of Information Technology, Uppsala University, 2000.

[13] [Carlsson01] Carlsson, R., 2001: *Core Erlang introduction and formal sematics,* Manuscript, Private communication, 2001.

[16] [Corbett00] Corbett, J.C., 2000: *Using Shape Analysis to Reduce Finite-State Models of Concurrent Java Programs,* ACM Transactions on Software Engineering and Methodology, Vol.9, No.1, pp.51–93, January 2000.

[2] [Dacker00] Däcker, B., 2000: *Concurrent Functional Programming for Telecommunications: A Case Study of Technology Introduction,* Licentiate thesis, Computer Communication System Laboratory, Department Of Teleinformatics, Royal Institute of Technology, Sweden, 2000.

[12] [Fredlund01] Fredlund, L.-å., 2000: *A Framework for Reasoning About Erlang Code*, Part of forthcoming Ph.D. thesis in the fall of 2001, Private communication, 2001.

[4] [Hinde00] Hinde, S., 2000: *Use of Erlang/OTP as a Service Creation Tool for IN Services,* Proceedings of Sixth International Erlang/OTP Users Conference, `http://www.erlang.se/euc/00/one2one.pdf`, 2000.

[14] [Holzmann00a] Holzmann, G.J., Smith, M.H., 2000: *Automating software feature verification,* Bell Labs Technical Journal, April-June 2000, Special Issue on Software Complexity, 2000.

[15] [Holzmann00b] Holzmann, G.J., 2000: *Logic Verification of ANSI-C Code with SPIN,* In Proceedings of 7th International SPIN Workshop, eds. K. Havelund, J. Penix and W. Visser, LNCS Vol.1885, pp.131–148, Springer-Verlag, 2000.

[9] [Huch99] Huch, F., 1999: *Verification of Erlang Programs using Abstract Interpretation and Model Checking,* Proceedings of ICFP'99, ACM SIGPLAN Notices, Vol.34, No.9, pp.261-272, ACM Press, 1999.

[10] [Huch01] Huch, F., 2001: *Model Checking Erlang Programs – Abstracting the Context-Free Structure,* Manuscript, Private communication, 2001.

[17] [Nielson98] Nielson, H.R., Amtoft, T., Nielson, F., 1998: *Behaviour Analysis and Safety Conditions: a Case Study in CML,* LNCS Vol.1382, pp.255-269, Springer-Verlag, 1998.

[6] [OTP] OTP, 2000: *OTP Documentation,* `http://www.erlang.org/doc/current/doc/index.html`.

[8] [Sampath95] Sampath, M., Sengupta, R., Lafortune, S., Sinnamohideen, K., Tekenekekzis, D., 1995: *Diagnosability of Discrete-Event Systems,* IEEE Transactions on Automatic Controls, Vol.40, No.9, pp.1555-1575, 1995.