Verification of Language Based Fault-Tolerance

Clara Benac Earle[1] and Lars-Åke Fredlund[2,3]

[1] Computing Laboratory, University of Kent, England
[2] LSIIS, Facultad de Informática, Universidad Politécnica de Madrid*
[3] Swedish Institute of Computer Science, Sweden

Abstract. In this paper we target the verification of fault tolerant aspects of distributed applications written in the Erlang programming language. Erlang programmers mostly work with ready-made language components. Our approach to verification of fault tolerance is to verify systems built using a central component of most Erlang software, a generic server component with fault tolerance handling.

To verify such Erlang programs we automatically translate them into processes of the μCRL process algebra, generate their state spaces, and use a model checker to determine whether they satisfy correctness properties specified in the μ-calculus.

The key observation of this paper is that, due to the usage of these higher-level design patterns, the state space generated from a Erlang program, even with failures occurring, is relatively small, and can be generated automatically.

1 Introduction

As software based critical systems are now becoming widely deployed, it is a crucial development task to analyse whether such systems will survive the inevitable faults (in hardware, in network links, in software components) that occur. However, this remains very difficult. It is hard to develop realistic failure models for simulation and testing (often the environment characteristics are not fully known), and to test and simulate for all possible faults would be very time consuming. Consequently, here is an area where there is a need for formal verification. But that too is hard. Most earlier work on verifying fault-tolerance target a single application only, are ad-hoc, and do not provide a reusable verification method. In this paper, instead, we propose a verification method based on model checking, that, since it addresses programs developed using higher-level design patterns which address fault-tolerance in a structured way, can be reused for a large set of such applications.

Erlang is a programming language developed at the Ericsson corporation for implementing telecommunication systems [1]. It provides a functional sub-language, enriched with constructs for dealing with side effects such as process creation and inter–process communication. Today many commercially available products offered by Ericsson are at least partly programmed in Erlang. The software of such products is typically organised into many, relatively small, source modules, which at runtime execute

* The second author was supported by an ERCIM grant and a Spanish Governmental Grant from the Ministerio de Educación Y Ciencia, with reference SB2004–0195.

R. Moreno Díaz et al. (Eds.): EUROCAST 2005, LNCS 3643, pp. 140–149, 2005.

as a dynamically varying number of processes operating in parallel and communicating through asynchronous message passing. The highly concurrent and dynamic nature of such software makes it particularly hard to debug and test. We therefore explore the alternative of software verification based on a formal proof system.

A key feature of the systems for which Erlang was primarily created is fault-tolerance. Switching systems should provide an acceptable level of service in the presence of faults. Erlang implements fault-tolerance in a simple way. Links between two processes A and B can be set up so that process B is notified of the termination of process A and vice versa. The default behaviour of a process that is informed of the abnormal termination (e. g., due to an exception) of another process is to terminate abnormally itself, although this behaviour can be modified. This process linking feature can be used to build hierarchical process structures where some processes are supervising other processes, for example restarting them if they terminate abnormally.

We start, in Section 2, by explaining the software components that are used to build quality Erlang software. The basic mechanisms for error handling in Erlang are described in Section 3. In Section 4, we describe how the generic server component of Erlang is extended with fault tolerance, and the actual translation from Erlang to μCRL is given in Section 5. The checking of correctness properties of fault-tolerant systems is discussed in Section 6, where as an example we analyse mutual exclusion and starvation properties of a server implementing a locking service for a number of client processes.

2 Erlang Components

A key aspect of the Erlang approach to development is the use of design patterns (provided by Erlang/OTP) which are encapsulated in terms of generic components. This approach simplifies the development cycle, as well as our verification of fault-tolerance.

Erlang/OTP provides a convenient component, the *generic server*, for programming server processes. A server is a process that waits for a message from another process, computes a response message and sends that back to the original process. Normally the server will have an internal state, which is initialised when starting the server and updated whenever a message has been received.

The behaviour module (`gen_server`) implements the common parts of a generic server process, providing a standard set of interface functions, for example, the function `gen_server:call` for synchronous communication with the server. The specific parts of the concrete client-server system are given in a call-back module.

We illustrate the functionality provided by the generic server component using a server in Figure 1 which also serves to introduce the concrete Erlang syntax. Informally the server implements a locking facility for a set of client processes. A client can acquire the lock by sending a `request` message, and release it using a `release` message.

Names of functions and atoms begin with a lowercase letter, while variables begin with an uppercase letter. The usual data types are provided, e.g., lists, tuples (enclosed in curly braces) and numbers. Matching a value against a sequence of patterns, which happens in function applications and in the `case` expression, is sequential.

A programmer that uses the generic server component essentially has to provide two functions: `init` which is invoked when the generic server starts, and which should

```
init(A) -> {ok,[]}.

handle_call(request, Client, Pending) ->
    case Pending of
         [] -> {reply, ok, [Client]};
         _  -> {noreply, Pending ++ [Client]}
    end;
handle_call(release, Client, [_|Pending]) ->
    case Pending of
         [] -> {reply, done, []};
         _  -> gen_server:reply(hd(Pending), ok), {reply, done, Pending}
    end.
```

Fig. 1. The source code of an Erlang generic server

return the initial state of the server (the empty list in the example), and `handle_call` which is invoked when a call is made to the generic server, and a reply is expected by the caller. The `handle_call` function is invoked with three arguments, the message submitted in the call, a value that is used to reply to the message, and the current state of the generic server. It returns the new state of the server upon completion. The processing of calls by a generic server is sequential, i.e., there are never concurrent invocations of the callback functions; a generic server thus offers a convenient way of controlling the amount of concurrency in an application and of protecting the state of the server.

In the example the server may be called with a `request` or a `release` message. If the message is a request, and if `Pending` is the empty list, it replies to the caller with the atom `ok`, and the new state of the server is `[Client]`. If `Pending` is not empty, then the reply is postponed (until more messages arrive) and the new state of the server is obtained by adding `Client` to the end of `Pending`. In case of a `release`, the server may issue a reply to the waiting caller, using `gen_server:reply`.

Client processes use a uniform way of communicating with the server; when a reply is expected they issue a call `gen_server:call(Locker, Message)` where `Locker` is the process identifier of a generic server. The client process suspends until a value is returned by the server.

Note that the semantics of communication using the generic server component is less complex that the communication paradigm of the underlying Erlang language. Generic servers always receive messages sequentially, i.e., in FIFO (first-in first-out) order. Erlang processes in contrast can potentially receive messages sent to them in arbitrary order. Thus by focusing on the higher-level components, rather than the underlying language primitives, our verification task becomes easier (concretely, state spaces are reduced). We will see the same thing happening when considering fault tolerance.

3 Fault-Tolerance in Erlang

In Erlang, bidirectional links are created between processes by invoking the `link` function with the process identifier of the process to link to as argument. There is also a function `spawn_link` which atomically both spawns a new process, and creates a bidirectional link to it.

Terminating processes will emit exit signals to all linked processes. Erlang distinguishes between normal process termination (the toplevel function of the process returned a value) from abnormal process termination (e.g. a runtime error such as attempting to divide by zero). If a process terminates abnormally, linked process will by default terminate abnormally as well. However, a linked process can trap exit such exit signals, and thus escape termination, by calling `process_flag(trap_exit,true)`.

In this case, when an exit signal reaches the process it is transformed into an exit message and delivered to the process mailbox like any other message. Exit messages are of the form {`'EXIT',Pid,Reason`}, with `Pid` the process identifier of the process that terminated, and `Reason` the reason for termination. If a process terminates normally `Reason` is equal to `normal`.

This basic mechanism of Erlang for error handling is exploited by the Erlang generic server behaviour in order to build fault-tolerant client-server systems. The Erlang programmer that implements a server process using the generic server component has to take several possible types of faults into account. First, the server itself may be faulty and crash. Recovery should be implemented by designating a supervisor process that restarts the server process (or takes some other corrective action).

Another error condition occurs when the server may communicate with remote processes, or hardware devices, that can malfunction without crashing, and moreover without generating exit signals to linked processes. Such error conditions should be handled in a traditional manner using timeouts. We focus instead on the error condition when an explicit exit signal reaches the generic server process. For the Erlang programmer such signals are handled by providing a new callback function, `handle_info(Signal,State)` that gets passed the exit signal as argument, together with the current state of the server. The `handle_info` function should, similarly to the other callback functions, either return the new state of the server or stop. This function will be called only if no call to the server is being processed.

In the client-server applications that we want to verify using the fault-tolerant extension, the state of the server contains information about the state in which its clients are in, for example, in the locker in Figure 1, the state of the locker reflects whether a client is accessing a resource or whether is waiting to get access to it. If a client terminates abnormally, the system should be able to recover gracefully without a complete restart, i.e., the state of the server process should be cleaned up accordingly.

4 Fault-Tolerance in Generic Servers

Our goal is to check the correctness of generic servers in the presence of crashing clients. The class of servers that we can analyse for fault tolerance have the following characteristics: (i) the server expects to receive an exit message whenever a linked client crashes, and (ii) the server establishes a process link to every client that issues a generic server call to it.

Although the above conditions may appear arbitrary, they are in fact indicative of a class of servers that safely implement a stateful protocol between itself and its clients, through call and reply exchanges. Thus, in a sense, these conditions give rise to a new Erlang high-level component which refines the basic Erlang generic server component.

As an example of a fault-tolerant server let us reconsider the simple server in Figure 1. The main loop of a client that accesses the locker is given below. Every client process sends a `request` message followed by a `release` message.

```
loop(Locker) ->
    gen_server:call(Locker, request),
    gen_server:call(Locker, release),
    loop(Locker).
```

We implement a locker which recovers from the abnormal termination of a client process by first adding the functions `process_flag` and `link` to the call-back module of the locker given in Figure 1 as shown below.

```
init(A) -> process_flag(trap_exit,true), {ok,[]}.

handle_call(request, {ClientPid,Tag}, Pending) ->
    link(ClientPid),
    case Pending of
         [] -> {reply, ok, [Client]};
         _  -> {noreply, Pending ++ [Client]}
    end;
```

The locker process now gets linked to the clients when they request a resource. If a client crashes, the locker will receive an exit message. As previously mentioned, exit messages are handled by the generic server function `handle_info` provided by the Erlang generic server behaviour. A trivial implementation of this function just returns the state of the server.

```
handle_info({'EXIT',ClientPid,Reason},Pending) -> {noreply, Pending}.
```

Now, if a client process crashes immediately after sending the `request` message to the locker, then the locker will process the `request` message before the exit signal. If there the resource is available, then the locker will send an `ok` message to the client that crashed and will put the client in the pending list. Since this client has crashed, it cannot release the resource, therefore, all other clients requesting the resource are put in the pending list and will eventually starve. If the resource is not available, the client will be put in the pending list, and when the resource is available, we have the same starving situation described before. Starvation also occurs if the client crashes while accessing the resource and before releasing it. However, if the client crashes after releasing, then the program behaves correctly. Of course, more than one client process may crash, therefore, we need to consider all the combinations of clients crashing at different points in the program execution. Already we can see that testing fault-tolerant code for a simple protocol like the one presented here is quite complex. Our goal is to use a high-level language, a process algebra, and use tools to automatically generate all these combinations and to check that key properties, deadlock-freedom, mutual exclusion, and non-starvation, are fulfilled.

The implementation of the `handle_info` function for the locker is given below.

```
handle_info({'EXIT',ClientPid,Reason},Pending) ->
    NewPending = remove(ClientPid,Pending),
    case available(ClientPid,Pending) of
         true -> gen_server:reply(hd(NewPending), ok),
                 {noreply, NewPending};
```

```
        _    -> {noreply, NewPending}
    end.

remove(ClientPid,[]) -> [];
remove(ClientPid,[{ClientPend,TagPending}|Rest]) ->
      case ClientPid == ClientPend of
        true  -> Rest;
        false -> [{ClientPend,TagPending}|remove(ClientPid,Rest)]
      end.

available(ClientPid,[]) -> false;
available(ClientPid,[{ClientPend,TagPending}]) -> false;
available(ClientPid,[{ClientPend,TagPending}|Rest]) -> ClientPid == ClientPend.
```

When the locker receives an exit message, i.e., a client process has terminated abnormally, then if the client is in the pending list, then it is removed from it. Moreover, if the client was accessing the resource (i.e., it was in the head of the pending list), then, the resource is available and therefore the locker gives access to the resource to a client which was waiting for it. This is similar to when a client sends a `release` message.

5 Translating Fault-Tolerant Systems to μCRL

In this section we briefly review the translation to μCRL of Erlang fault-tolerant client-server systems, full details are provided in [3].

For the purpose of verification Erlang programs are translated into the μCRL process algebra [5] by an automatic translator tool [2]. In μCRL behaviour is described on two levels, as traditional process behaviour using the process algebra operators of μCRL (sequencing, parallel composition, recursion, communication using synchronisation, etc), and data kept by processes and exchanged in communications. Functions can be defined over data types using rewrite rules.

The translation of Erlang mimics the separation between process behaviour and functional behaviour present in μCRL. A pre-analysis step partitions Erlang functions into two categories: the ones with pure functional computation, and the ones with side effects (e.g., communication to/from a generic server). The side-effect free Erlang functions are translated into μCRL functions, which are defined using a set of rewrite rules. Thus such Erlang functions do not generate any state. In contrast the side-effect Erlang functions are translated into μCRL processes, using the process operators.

The translation of communications with a generic server uses an intermediate buffer process implemented in μCRL, which stores sent messages until the translated generic server process is ready to receive them. Thus the asynchronous nature of communication in Erlang is kept in the translated code. The translation of non-tail recursive side-effect functions uses an explicit call-stack to keep track of recursive calls.

Which processes (e.g., generic servers and clients) to translate is computed by analysing the code for setting up the system. The generic server processes are found by analysing which processes initially execute a function in a module with the generic server behaviour attribute.

The fault-tolerant extension of Erlang only affects the process part of Erlang, hence, the translation of the functional part of fault-tolerant Erlang remains the same. For the process part, the fault-tolerant extension of Erlang assumes that a server expects to

receive an exit message in its mailbox whenever a linked client crashes, and that this exit message is received and handled by the generic server primitive `handle_info`. The translation to μCRL therefore needs to take into account this implicit communication between the client and the server, and the translation of the `handle_info` function.

The μCRL toolset [4] is used to generate a state space from the μCRL translation. Obviously, the state space generated for a client-server system with this client process is larger than the one where the client cannot crash. For example, the state space generated in a scenario with two client processes which cannot crash contains 33 states and 48 transitions, while the state space for the same scenario with crashing clients consists of 326 states and 584 transitions.

6 Model Checking Properties in Fault-Tolerant Systems

Once the labelled transition system has been generated by the μCRL toolset from the μCRL specification (the result of translating the Erlang program), the CADP toolset is used to check whether safety and liveness properties hold. Such correctness properties are formulated in the regular alternation free μ-calculus [8, 7]. Informally, the modalities in the logic are relaxed to sequences of actions characterised by regular expressions.

Action label are enclosed in quotes (e.g., $'$`crash`$'$) and can contain wildcards (e.g., $'.*$`crash`$.*'$ matches any action that has the text string `crash` somewhere in its name), $\neg regaction$ matches any action that does not match the action regular expression $regaction$, $regaction_1 \vee regaction_2$ is disjunction. Actions can be composed using the normal regular expression operators, i.e., $|$ denotes alternative, $*$ zero or more occurencies, . is sequencing, and $-$ matches any action. Comments can be enclosed in formulas using the `(* comment *)` notation.

6.1 Deadlock Freedom

Since we model crashing of client processes, actually we are introducing deadlock states. To verify that a client-server system is deadlock-free except for the states where all clients have crashed, we formulate a fault-tolerant version of the classical deadlock-freedom property. The property we are interested in states that no deadlocks occurs as long as not all the processes in the system have crashed. This property can be expressed by explicitly stating the crash actions in the formula.

For instance, supposing there are three processes in the system. Then we define a action sequence macro denoting the sequences containing 0, 1, or 2 crashes:

$$
\begin{array}{l}
BETWEEN\_0\_AND\_2\_CRASHES() = \\
\quad ((\neg'.*\texttt{info}.*')^* \ \texttt{(* 0 crashes *)} \quad | \\
\quad (\neg'.*\texttt{info}.*'*'.*\texttt{info}.*'.(\neg'.*\texttt{info}.*')^* \ \texttt{(* 1 crash *)} \quad | \\
\quad (\neg'.*\texttt{info}.*'*'.'.*\texttt{info}.*'.(\neg'.*\texttt{info}.*')^*.'.*\texttt{info}.*'.(\neg'.*\texttt{info}.*')^*))
\end{array}
$$

Using the macro, the deadlock freedom property becomes:

$$[BETWEEN\_0\_AND\_2\_CRASHES()]\langle - \rangle true$$

This formula will spot the deadlocks unrelated to complete crashes of the system. In general, for N processes in the system, one must write N-1 lines of the form $('.*\texttt{info}.*'.(\neg'.*\texttt{info}.*')^*)$ in the macro above.

This example highlights the need to reconsider the properties used to verify nonfault-tolerant systems in order to verify fault-tolerant systems. In the following two subsections we discussed how mutual exclusion and non-starvation can be verified.

6.2 Mutual Exclusion

The formulation of the mutual exclusion property for the non-fault-tolerant locker is given below. To make verification easier two actions are introduced in the Erlang code of the client to signal the entering (`use`) and the exiting (`free`) of the critical section.

$$BETWEEN(\mathtt{a}_1,\ \mathtt{a}_2,\ \mathtt{a}_3) = [-^* \cdot \mathtt{a}_1 \cdot (\neg \mathtt{a}_2)^* \cdot \mathtt{a}_3]false$$
$$MUTEX() = BETWEEN('\texttt{use}(.^*)','\texttt{free}(.^*)','\texttt{use}(.^*)')$$

The formula states that 'on all possible paths, after an `use` action, any further `use` action must be preceded by an `free` action'. Intuitively, the formula means that if a client process is accessing the resource, then no other client process can access it until the resource has been freed. This formula does not hold in the state space generated for the a scenario with two crashing clients. The CADP model checker gives the following counter-example.

```
"call(locker,request,C1)"
"reply(C1,ok,locker)"
"action_use(C1)"
"info(locker,{EXIT,C1,EXIT})"
"call(locker,request,C2)"
"reply(C2,ok,locker)"
"action_use(C2)"
```

The counter-example shows that the mutual exclusion property is violated, since the resource is accessed by two process clients, client 1 and client 2, without being freed. However, the counter-example is also showing that, client 2 is accessing the resource after client 1 has crashed, therefore, strictly speaking, client 1 is not accessing the resource because it is dead.

In order to show that the mutual exclusion property is verified in the fault-tolerant first version of the locker case-study, we need to take the client crashes into account, as is done in the property below.

$$FT - BETWEEN(\mathtt{a}_1,\ \mathtt{a}_2,\ \mathtt{a}_3,\ \mathtt{a}_4) = [-^* \cdot \mathtt{a}_1 \cdot (\neg \mathtt{a}_2 \vee \mathtt{a}_3)^* \cdot \mathtt{a}_4]false$$
$$FT - MUTEX() = FT - BETWEEN('\texttt{use}(.^*)','\texttt{free}(.^*)',,'\texttt{use}(.^*)')$$

To illustrate the power of model checking as a debugging tool, consider the following erroneous implementation of the `handle_info` function of the locker. After a client crashes, access to the resource is given to the client that was waiting to get access in the head of the pending list.

```
handle_info({'EXIT',ClientPid,Reason},Pending) ->
    NewPending = remove(ClientPid,Pending),
    case NewPending == [] of
         false -> gen_server:reply(hd(NewPending), ok),
                  {noreply, NewPending};
         _ ->     {noreply, []}
    end.
```

This code is correct for the case where a client crashes after obtaining access to the resource, but it is wrong if the client crashes after releasing the resource. Testing concurrent code is tricky, in particular, in this example, only the right combination of more than three clients, a client crashing after releasing the resource and the other two or more clients waiting in the pending list triggers the error in the fault-tolerant code.

6.3 Non-Starvation

As with the verification of mutual exclusion, the fault-tolerant behaviour of the system we want to verify needs to be taken into account in order to prove the non-starvation property. Thus, instead of the following property that checks for non-starvation of the client C, which because of crashes is not satisfied,

$$\begin{array}{l} NONSTARVATION(C) = \\ \quad [-^* \,.\, {}'\texttt{gen\_server:call}(.^*\texttt{request}.^*, C)'] \\ \qquad \mu X.(\langle - \rangle true \ \wedge \ [\neg {}'\texttt{reply(ok},C)']X) \end{array}$$

we use the following "fault-tolerant" one, which is satisfied by the locker:

$$\begin{array}{l} NONSTARVATION(C) = \\ \quad [-^* \,.\, {}'\texttt{gen\_server:call}(.^*\texttt{request}.^*, C)'] \\ \qquad \mu X.(\langle - \rangle true \ \wedge \ [\neg {}'\texttt{reply(ok},C)' \ \vee \ {}'\texttt{info}(.^*, \texttt{C}, .^*)']X) \end{array}$$

7 Conclusions and Related Work

One of the aspects that makes the programming language Erlang popular among developers of business-critical systems is the inclusion of constructs to handle fault-tolerance. Our approach to verification of such fault-tolerant systems has several components. First, Erlang systems are translated into μCRL specifications. Next, the μCRL toolset generates the state space from the algebraic specification, and finally, the CADP toolset is used to check whether the system satisfies correctness properties specified in a the alternation-free μ-calculus.

To enable analysis of fault behaviour we introduce during the translation phase to μCRL explicit failure points in the algebraic specification, in a systematic way, where the system processes may fail. The key observation is that, due to the usage of higher-level design pattern that structure process communication and fault recovery, the number of such failure points that needs to be inserted can be relatively few, and can be inserted in an application independent manner. In other words, the state spaces generated from a failure model can be generated automatically, are relatively small, and are thus amenable to model checking.

We have demonstrated the approach in a case study where a server, built using the generic server design pattern, implements a locking service for the client processes accessing it. The server necessarily contains code to handle the situation where clients can fail; if it did not the server would quickly deadlock. In the study we verify, using the automated translation and model checking tool, systems composed of a server and a set of clients with regards to crucial correctness properties such as deadlock freedom, mutual exclusion and liveness.

The formal verification of fault-tolerant systems has been studied in several case-studies such as e.g. [9, 10]. In contrast to our approach, they target a single application only, are ad-hoc, and often do not provide a reusable verification method.

General models for the verification of fault-tolerant algorithms are also present in the literature, for example [6]. The main difference with our approach is that our models (similar to the software) are on a higher-abstraction level than those works; there is more intelligence built-in the Erlang component programming model than in general model, and it is interesting to see, that using such a model actually makes it easier to verify the correctness of the solution.

References

[1] J.L. Armstrong, S.R. Virding, M.C. Williams, and C. Wikström. *Concurrent Programming in Erlang*. Prentice Hall International, 2nd edition, 1996.

[2] T. Arts, C. Benac Earle and J. J. Sánchez-Penas. Translating Erlang to μCRL. Application of Concurrency to System Design, 2004. ACSD 2004. Proceedings. Fourth International Conference on, Vol., Iss., 16-18, pp. 135-144, June 2004.

[3] C. Benac Earle. Model Checking the Interaction of Erlang Components. PhD thesis, University of Kent, UK. February 2005.

[4] J.-C. Fernandez, H. Garavel, A. Kerbrat, R. Mateescu, L. Mounier, and M. Sighireau. CADP (CÆSAR/ALDÉBARAN development package): A protocol validation and verification toolbox. In *Proc. of CAV*, LNCS 1102, p. 437–440, Springer-Verlag, Berlin, 1996.

[5] J. F. Groote. The syntax and semantics of timed mCRL. Technical report SEN-R9709, CWI, Amsterdam, 1997.

[6] T. Janowski and M. Joseph. Dynamic Scheduling and Fault-tolerance: Specification and Verification. Real-Time Systems. Vol. 20, Issue 1, Kluwer Academic Publishers. 2001.

[7] D. Kozen. Results on the propositional μ-calculus. *TCS*, **27**:333-354, 1983.

[8] R. Mateescu. Local Model-Checking of an Alternation-free Value-Based Modal Mu-Calculus. *Proceedings of the International Workshop on Software Tools for Technology Transfer STTT'98*, Aalborg, Denmark, July 1998.

[9] J. Rushby. Systematic Formal Verification for Fault-Tolerant Time-Triggered Algorithms. IEEE Transactions on Software Engineering, volume 25, number 5, 1999.

[10] F. Schneider, S. M. Easterbrook, J. R. Callahan and G. H. Holzmann, Validating Requirements for Fault Tolerant Systems using Model Checking. *Proceedings, 3rd International Conference on Requirements Engineering*, 4-13, Colorado, Springs, Colorado, April 1998.