Verifying Fault-Tolerant Erlang Programs

Clara Benac Earle
Departamento de Informtica, Universidad Carlos III de Madrid
cbenac@inf.uc3m.es

Lars-Åke Fredlund [*]
LSIIS, Facultad de Informática, Universidad Politécnica de Madrid
fred@babel.ls.fi.upm.es

John Derrick
Department of Computer Science, University of Sheffield
jd@dcs.shef.ac.uk

Abstract

In this paper we target the verification of fault tolerant aspects of distributed applications written in Erlang. Erlang is unusual in several respects. First, it is one of a few functional languages that is used in industry. Secondly the programming language contains support for concurrency and distribution as well as including constructs for handling fault-tolerance.

Erlang programmers, of course, mostly work with ready-made language components. Our approach to verification of fault tolerance is to verify systems built using two central components of most Erlang software, a generic server component with fault tolerance handling, and a supervisor component that restarts failed processes.

To verify Erlang programs built using these components 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 (supervisors and generic servers) that structure process communication and fault recovery, the state space generated from a Erlang program, even with failures occurring, is relatively small, and can be generated automatically. Moreover the method is independent from the actual Erlang program studied, and is thus reusable.

We demonstrate the approach in a case study where a server, built using the generic server component, implements a locking service for a number of client processes, and show that the server tolerates client failures.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification

General Terms Verification

Keywords Code Verification, Fault-Tolerance, Concurrency

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

1. Introduction

Through software industry reliability is more and more becoming a key objective. An integral part of many attempts to ensure reliability in computer science has been the application of techniques from mathematics and logic to the design and validation of programs and systems; these are often collectively known as 'Formal Methods' (FMs). As an example from industry, Microsoft are deploying FMs in the analysis of their Office (TM) suite of programs with the aim of eliminating whole categories of code errors [7].

The last decade has seen several high profile successes of formal methods based systems development, in many cases resulting from the application of *model checking*. In model checking key correctness properties of the system under analysis can be checked automatically. In the case of Erlang, the features that make the language attractive to programmers – simplicity, OTP library support – also make it particularly suitable for formal analysis through model checking.

Model checking is an automatic formal verification technique where a property is checked over a finite state system; it has been used successfully in the verification of numerous complex pieces of hardware and on specifications (e.g., automata, process algebras). The major advantages of model checking are that it is an automatic technique, and that when the model of the system fails to satisfy a desired property, the model checker always produces a counter example. These faulty traces provide a priceless insight to understanding the real reason for the failure as well as important clues for fixing the problem.

Using model checking for the formal verification of software is by now a well known field of research, it is in the details that we offer some novelty. There are essentially two approaches to the overall problem, either (i) one uses a specification language in combination with a model checker to obtain a correct specification that is used to write an implementation in a programming language, or (ii) one takes the program code as a starting point and abstracts that into a model, which can be checked by a model checker. Either way, the implementation itself is not proved correct by these approaches, but a model of it. Thus when an error is encountered in the model, this may indicate also an error in the implementation. As such, the use of model checking can be seen as a very accurate testing method.

The work we describe here follows the second approach, i.e., we build a model from actual program code. Related work pursuing the same approach include PathFinder [13] and Bandera [5] which consider the problem of verifying code written in Java. Our work has similar concerns and follows a similar approach except that we use the knowledge of the occurring design patterns used in the Erlang code to obtain smaller state spaces (cf. [3]). We follow a similar approach to the translation of Java into Promela, checked by SPIN [13]; however, we translate Erlang into the process algebra with data μCRL [12] and model check by using the CADP toolset [10].

The fact that we translate the Erlang code into a process algebraic specification and use tools developed for analysing process algebras rather than implementing tools that work directly on Erlang code has a number of benefits, primarily that we can reuse existing toolsets for efficient state space generation from process algebraic specifications.

Several tools have been developed to support verification of μCRL specifications [6, 23]. Our approach to verification uses a model checker from the CADP toolset. In order to input the μCRL specifications into the model checker, we need to convert the specification to an appropriate input format, in particular, we use the state space generation tool of the μCRL toolset. The logic used to express the properties we are interested in is the alternation-free μ-calculus [8]. Because we use standard tools, our contribution lies in the abstraction and its automation.

We have built a tool, `etomcrl`, which automatically translates client-server systems, where the server uses the Erlang/OTP generic server behaviour, into a μCRL specification. We developed the `etomcrl` tool at the same time as we verified a small, but critical, part of a locker algorithm used in the software controlling Ericsson's AXD ATM switches. This case-study is described in [2]. The tool has also been used in the verification of another industrial case-study, the scheduler of a video-on-demand server [20, 4]. In this paper we extend the verification framework to fault-tolerant client-server systems, and by doing so we aim to come one step closer to the goal of being able to analyse industrial code.

2. Erlang

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. In the following, we give a brief review of the Erlang design patterns that are central to our model checking effort.

2.1 Generic server component

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.

Erlang/OTP provides a convenient component, the *generic server*, for programming server processes. 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 and the function `gen_server:cast` for asynchronous communication. 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 is also used as the leading example of this paper. 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.

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 the `gen_server:reply` function. In the example, the

```
-module(locker).
-behaviour(gen_server).
-export([start/0,init/1]).

start() -> gen_server:start_link({local, locker},locker1ver,[],[]).

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.
```

Figure 1. The source code of an Erlang generic server

generic server can be shut down by issuing a non-blocking call (a cast) to it; the server corresponds by shutting down (indicated by returning `stop`).

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.

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 unless no reply is expected, in which case the function `gen_server:cast` is called instead which doesn't cause the client to suspend. The generic server component, and its access functions, permits calls from remote (distributed) processes, with no change in syntax or semantics (except that further faults may occur).

Note that the semantics of communication using the generic server component is, in a sense, simpler 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 an 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 later on in this paper.

2.2 Supervisor component

A frequent assumption made when writing Erlang software is that any Erlang process may unexpectedly die, either because of a hardware failure, or a software error in the code evaluated in the process. The runtime system provides a mechanism to notify selected processes of the fact that a certain other process has terminated; this is realized by a special message that arrives in the mailbox of processes that are specified to monitor the vanished process.

On top of the Erlang primitives to ensure that processes are aware of the existence of other processes, a supervisor process component is available. This component evaluates a function that creates processes (known as childred) which it will monitor. The children may themselves be supervisor processes, supervising its children in turn. The result is a hierarchical, tree-like, process supervision structure. After creating these processes, it enters a receive loop and waits for a process to die. If that happens, it might either restart the child or use another predefined strategy to recover from the problem. A typical supervisor structure is shown in Figure 2.

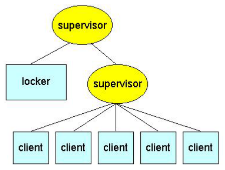

Figure 2. Supervisor tree for locker and clients

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. Links can be removed using `unlink(Pid)` which removes a link between the calling process and its argument.

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 signals and thus escape termination when an exit signal arrives, 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 both the Erlang generic server behaviour and the Erlang supervisor 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. Note that the library code in the `gen_server` module is fault-tolerant and is in this paper assumed to be correct, therefore, if there is an error, the error is assumed to be either in the call-back module of the generic server or in the code of the client.

First, the server itself may be faulty and crash. Recovery should be implemented by designating a supervisor process, using the supervisor component, that restarts the server process (or takes some other corrective action). In this paper we will not further discuss how to recover from server crashes by restarting.

Another error condition occurs when the server may communicate with remote processes (possibly servers), 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. The generic server component has support for setting timeouts, but we will not explore the proper handling of such “semantic” faults further in this paper. Note that the Erlang runtime system implements a heartbeat algorithm for detecting crashed or non-connected nodes, so that having a link to a remote process on an inaccessible node will eventually result in the reception of an exit signal.

In this paper 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. Note that this function will be called only if no call to the server is being processed, i.e., the restriction to sequential processing in the server is still kept.

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 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:

- the server expects to receive an exit message whenever a linked client crashes.
- the server establishes a process link to every client that issues a generic server call to it. No links should be established due to a client issuing a generic server cast.
- the server never removes a link between itself and a client process, i.e., it never calls the function `unlink`.

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 in Figure 3. 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).
```

Figure 3. A client accessing the server

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

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

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 describe the translation to μCRL of Erlang fault-tolerant client-server systems. First we briefly explain the translation of Erlang systems without taking fault tolerance into account, further details are provided in [1].

For the purpose of verification Erlang programs are translated into the μCRL process algebra [11] by an automatic translator tool [1]. The μCRL toolset can then generate labelled transition systems corresponding to the μCRL specifications, which are used in model checking.

5.1 Introduction to μCRL

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. Data is separated into distinct types, which are characterised by their sets of data constructors. Moreover functions can be defined over data using rewrite rules. We illustrate the intuitive semantics of the language using a simple producer/consumer example in Figure 4.

The example illustrates both the data and process part of μCRL. A sort `Natural` is defined having constructors `s` and `0`, and a function `leq` defines the "less-than-or-equal" operation using a rewrite system (rules found under the section `rew`).

The initial process configuration of a μCRL specification is given in the `init` section. In the example the two processes `producer` and `consumer` are specified to run in parallel (using the parallel operator "`||`").

The `producer` process offers to synchronise on a `snd` action, using a natural number as parameter, and then recurses, until the counter `N` reaches the `Limit` argument. In μCRL the syntax for an "if-then-else" choice is *`thenproc`* `<| ifexpr |>` *`elseproc`*. The `consumer` process repeatedly offers to synchronise with another process (communication in μCRL is always synchronous) on a `rcv` action, regardless of the natural number parameter (the `sum` operator can be understood as universal quantification).

Finally the `comm` section describes which actions synchronise, e.g., in the example `snd` and `rcv` are specified to synchronise (communication in μCRL is normally binary).

```
%%% Data part
sort
    Natural
func
    0:  -> Natural
    s: Natural -> Natural
map
    leq: Natural # Natural -> Bool
var
    N,N1: Natural
rew
    leq(0,N) = T
    leq(s(N),0) = F
    leq(s(N),s(N1)) = leq(N,N1)

%%% Process part
act
    snd, rcv: Natural

comm
    snd | rcv = comm

proc producer(N:Natural,Limit:Natural) =
  snd(N).producer(s(N),Limit)
  <| leq(N,Limit) |>
  delta

proc consumer =
  sum(Message:Natural, rcv(Message)).consumer

init producer(0,s(s(s(s(0))))) || consumer
```

Figure 4. μCRL example

5.2 Translation

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. Henceforth we will focus on the translation of Erlang side-effect functions.

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 supervisor description for setting up the system, which contains a listing of its processes and which function (and module) a process should start to execute. The generic server processes are found by analysing which processes initially execute a function in a module with the generic server behaviour attribute.

We illustrate the translation using an example, where the `loop` function in Figure 3 has been translated into μCRL.

```
proc loop(MCRLSelf:Term,Locker:Term) =
  gen_server_call(Locker,request,MCRLSelf).
  sum(MCRLFree0:Term,
      gen_server_replied(MCRLSelf,MCRLFree0,Locker).
      gen_server_call(Locker,release,MCRLSelf).
      sum(MCRLFree1:Term,
          gen_server_replied(MCRLSelf,MCRLFree1,Locker).
          loop(MCRLSelf,Locker)))
```

The `MCRLSelf` parameter of the `loop` process represents the process identifier of the process. The communication with the generic server is split into a call part where the `gen_server_call` synchronises with the message buffer, and a return part where the action `gen_server_replied` synchronises directly with the generic server. The summation operator (`sum`) can be understood as an infinite summation where the variable `MCRLFree1` is replaced by any possible Erlang value.

The translation of the `handle_call` server-side function, for a `request`, given in Figure 1 is shown below:

```
proc serverloop(MCRLSelf:Term,Pending:Term) =
  sum(Client:Term,
      handle_call(MCRLSelf,request,Client).
      (gen_server_reply(Client, ok, MCRLSelf).
       serverloop(MCRLSelf,cons(Client,nil))
                    <| eq(equal(Pending,nil),true) |>
       serverloop(MCRLSelf,list_append(Pending,cons(Client,nil))))
         + ...
```

Informally, the action `handle_call` synchronises with the buffer on a synchronous call, and the server replies to a call using the action `gen_server_reply`. If the state of the server (`Pending`) is the empty list, an immediate reply is sent, otherwise the new client is entered at the tail of the request queue (`Pending`) of the server. Note that the description of which action synchronize is given in a separate declaration in μCRL, we omit the declaration in this paper.

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 translation of the generic server primitives of Erlang to μCRL uses a buffer. The communication mechanism in case a client process crashes is similar to the one used in the other generic server communication primitives, with the difference that the exit message sent by a client that crashed is handled by the generic server primitive `handle_info`. We choose to represent the sending of this exit message in μCRL by the action `crash` which synchronises with the buffer of the server. The extended μCRL buffer that communicates with the `crash` action is shown in Figure 5.

As stated before, a server expects to receive an exit message in its mailbox whenever a linked client crashes. Because the server links to a client process in the `handle_call` function, the exit messages from a crashing client can only be received after the client issues a `gen_server:call`. Thus, in the Erlang code of a client, the client can crash:

- between issuing a `gen_server:call` and receiving the a reply from the server.
- after receiving the reply to the `gen_server:call` from the server.
- after issuing a `gen_server:cast` if there was at least one `gen_server:call` to the same server before.

Note here especially that the number of crash points are reduced considerably, because of the use of a higher-level component, than if the lower language fault tolerance mechanism was used. The benefit is, again, both reduced state space (with regards to verification) and for the programmer an easier task to write a fault tolerant program.

We insert the μCRL code corresponding to a client crashing automatically. The crashing of a client process is modelled in μCRL by the action `crash`. This action synchronises with the buffer of the server in the same way as the `gen_server_cast` action. The `crash` action appears in the μCRL specification of a client as an alternative to every μCRL communication action that corresponds to a communication function in Erlang. The possibility to crash is represented in μCRL as a non-deterministic choice between the `crash` action and the μCRL action corresponding to the Erlang communication function.

Note that in Erlang, if a client crashes between issuing a `gen_server:call` and receiving a reply from the server, the server will finish the evaluation of the `handle_call` before reading the exit message from the mailbox. This is not problematic, since, in Erlang, sending a message never fails, even if the message is sent to a non-existing process. However, communication between processes in μCRL is handshaking, therefore, the action `gen_server_reply` used to send the reply from the server to the client process tries to communicate with the action `gen_server_replied` in the client, even if the process client has crashed. To simplify the translation, we let the client process receive the reply message, even after crashing. This means that after every `crash` action that occurs after a `gen_server:call` action, the `gen_server_replied` action is added. Similarly, a client could crash after it makes a call to the server, and before the server attempts to link to it using the `link` function. However, the semantics of the `link` function in Erlang guarantees that if a process attempts to link to a terminated process, an exit signal is sent, thus our treatement is correct.

As an example, the Erlang code in Figure 3 is translated to a μCRL process, with a possibility to crash, below.

```
proc loop(MCRLSelf:Term,Locker:Term) =
  gen_server_call(Locker,request,MCRLSelf).
  (crash(Locker,tuple(EXIT,tuple(MCRLSelf,tuplenil(EXIT)))).
   sum(MCRLFree0: Term,
   gen_server_replied(MCRLSelf,MCRLFree0,Locker)).
   delta
    +
    sum(MCRLFree0: Term,
    gen_server_replied(MCRLSelf,MCRLFree0,Locker).
    (crash(Locker,tuple(EXIT,tuple(MCRLSelf,tuplenil(EXIT)))).
     delta
       +
       (gen_server_call(Locker,release,MCRLSelf).
(crash(Locker,tuple(EXIT,tuple(MCRLSelf,tuplenil(EXIT)))).
 sum(MCRLFree1: Term,
     gen_server_replied(MCRLSelf,MCRLFree1,Locker)).
     delta
+
sum(MCRLFree1: Term,
  gen_server_replied(MCRLSelf,MCRLFree1,Locker).
     (crash(Locker,tuple(EXIT,tuple(MCRLSelf,tuplenil(EXIT)))).
      delta
     +
     loop(MCRLSelf,Locker)))))))
```

The `handle_info` function has the same syntax as the `handle_cast`, and its semantics are basically the same, with the difference that a `handle_cast` function implements the handling of a `cast` message while the `handle_info` function implements the handling of an `exit` message.

The μCRL toolset 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 specifications which is automatically translated from the source Erlang program, the CADP toolset [9] is used to verify that safety and liveness properties are fulfilled. Such correctness properties are formulated in the regular alternation free μ-calculus [17, 16]. Informally, the modalities in the logic are relaxed to sequences of actions characterised by regular expressions.

Action label are enclosed in quotes (e.g., $'\texttt{crash}'$) and can contain wildcards (e.g., $'.*\texttt{crash}.*'$ matches any action that has the text string `crash` somewhere in its name), $\neg \mathit{regaction}$ matches any action that does not match the action regular expression $\mathit{regaction}$, $\mathit{regaction}_1 \vee \mathit{regaction}_2$ matches any action that matches either $\mathit{regaction}_1$ or $\mathit{regaction}_2$.

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.

In addition simple formula macros with parameters are used, e.g., $\mathit{AlwaysPossible}(\mathsf{a}) = [\texttt{-}^*]\langle \mathsf{a} \rangle \mathit{true}$ defines a macro expressing which expresses that after any sequence of actions, it must be possible to perform the action parameter `a`. Thus, we can check that the action `ok` is always enabled using $\mathit{AlwaysPossible}(\texttt{ok})$.

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:

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

Using the macro, the deadlock freedom property becomes:

$$[\mathit{BETWEEN\_0\_AND\_2\_CRASHES}()]\langle \texttt{-} \rangle \mathit{true}$$

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.

$$
\begin{array}{r}
\mathit{BETWEEN}(\mathsf{a}_1,\ \mathsf{a}_2,\ \mathsf{a}_3) = [\texttt{-}^* \,.\, \mathsf{a}_1 \,.\, (\neg \mathsf{a}_2)^* \,.\, \mathsf{a}_3]\mathit{false} \\
\mathit{MUTEX}() = \mathit{BETWEEN}('\texttt{use}(.^*)','\texttt{free}(.^*)','\texttt{use}(.^*)')
\end{array}
$$

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.

$$
\begin{array}{l}
\mathit{FT} - \mathit{BETWEEN}(\mathsf{a}_1,\ \mathsf{a}_2,\ \mathsf{a}_3,\ \mathsf{a}_4) = \\
\qquad [\texttt{-}^* \,.\, \mathsf{a}_1 \,.\, (\neg \mathsf{a}_2 \vee \mathsf{a}_3)^* \,.\, \mathsf{a}_4]\mathit{false}
\end{array}
$$

$$
\begin{array}{l}
\mathit{FT} - \mathit{MUTEX}() = \\
\qquad \mathit{FT} - \mathit{BETWEEN}('\texttt{use}(.^*)','\texttt{free}(.^*)','\texttt{info}(.^*)','\texttt{use}(.^*)')
\end{array}
$$

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.
```

```
comm
  gen_server_call | gscall = buffercall
  gen_server_cast | gscast = buffercast
  crash | gsinfo = bufferinfo
  gshcall | handle_call = call
  gshcast | handle_cast = cast
  gshinfo | handle_info = info
  gen_server_reply | gen_server_replied = reply

proc
    Server_Buffer(MCRLSelf: Term, Messages: GSBuffer) =
     (bufferfull(MCRLSelf).
      (gshcast(MCRLSelf,cast_term(Messages)).
         Server_Buffer(MCRLSelf,rmhead(Messages))
       <| is_cast(Messages) |>
       (gshinfo(MCRLSelf,info_term(Messages)).
          Server_Buffer(MCRLSelf,rmhead(Messages))
        <| is_info(Messages) |>
        (gshcall(MCRLSelf,call_term(Messages),call_pid(Messages)).
           Server_Buffer(MCRLSelf,rmhead(Messages))
         <| is_call(Messages) |>
         delta))))
     <| maxbuffer(Messages) |>
     (sum(Msg: Term,
          sum(From: Term,
              gscall(MCRLSelf, Msg, From).
              Server_Buffer(MCRLSelf, addcall(Msg,From,Messages)))) +
      sum(Msg: Term,
          gscast(MCRLSelf, Msg).
          Server_Buffer(MCRLSelf, addcast(Msg,Messages))) +
      sum(Msg: Term,
          gsinfo(MCRLSelf, Msg).
          Server_Buffer(MCRLSelf, addinfo(Msg,Messages))) +
      (gshcast(MCRLSelf,cast_term(Messages)).
         Server_Buffer(MCRLSelf,rmhead(Messages))
       <| is_cast(Messages) |>
       (gshinfo(MCRLSelf,info_term(Messages)).
          Server_Buffer(MCRLSelf,rmhead(Messages))
        <| is_info(Messages) |>
        (gshcall(MCRLSelf,call_term(Messages),call_pid(Messages)).
           Server_Buffer(MCRLSelf,rmhead(Messages))
         <| is_call(Messages) |>
         delta))))
```

Figure 5. The Generic Server Buffer Process

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.

However, the model checker fed with the μCRL specification automatically obtained from the erroneous Erlang code and the FT-MUTEX property gives, in few seconds, the following counter-example:

```
"call(locker,request,C1)"                    (1)
"reply(C1,ok,locker)"                        (2)
"call(locker,request,C3)"                    (3)
"info(locker,{EXIT,C3,EXIT})"                (4)
"action_use(C1)"                             (5)
"action_free(C1)"                            (6)
"reply(C1,ok,locker)"                        (7)
"call(locker,request,C2)"                    (8)
"call(locker,release,C1)"                    (9)
"reply(C2,ok,locker)"                        (10)
"action_use(C2)"                             (11)
"reply(C1,done,locker)                       (12)
"action_use(C1)"                             (13)
```

Here we have a trace where client 1 requests and is granted permission to access to the resource and is put in the pending list (lines (1) and (2)), thereafter client 3 requests access to the resource and is put in the pending list after client 1 (line (3)). Then client 3 crashes and because the pending list is not empty, an ok message saying that the resource is available is sent to the first element in the pending list, which is client 1 (line (7)). Client 2 requests access to the resource and is put in the pending list after client 1 (line (8)). In the meanwhile, client 1 has used the resource and sends a message to the locker to release it (line (9))[1]. The locker checks if there is any client waiting in the pending list to get access to the resource and gives access to the resource

[1] Note that client 1 has not yet received the ok message sent to it after client 3 crashed

to client 2 (line `(11)`) which access the critical section. But then client 1 receives the message `done` that the server sent after client 1 released, and then reads the `ok` message with which it was given permission to access the resource, thus entering the critical section as well (line `(13)`). At this point, the mutual exclusion property fails because both client 2 and client 1 are accessing the resource at the same time.

The counter-example is useful because it gives precise information on where the error occur and makes possible to reproduce it. This trace can also be used to generate tests for the system.

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 in a systematic way explicit failure points in the algebraic specification, 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 before, both using theorem provers, for example PVS [19], and model checkers like SPIN [14] and CADP [9]. These tools have been applied to several case-studies, some examples can be found in [18, 21]. 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 [15]. 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. However, in the case of [15], they consider time, which might not fit the Erlang model we propose. Further research is therefore needed.

Related to our work is the formal verification method proposed in [22] for distributed JavaSpaces, a distributed programming model where agents share information via a common space, and where the space handles the details of concurrent access to the data. The method is illustrated by verifying a fault-tolerant algorithm where some identical processes (workers) compute a value independently and one special process (master) publishes the result of the computation. Fault-tolerance is imposed on to the system as follows: any worker can suddenly stop, in which case the system recovers gracefully and the process is never restarted. The master can suddenly stop as well but it is restarted only once. The JavaSpaces architecture and the fault-tolerant algorithm are manually written in μCRL and verified by the combination of the μCRL toolset and the CADP toolset, in a similar way as we do. The key difference is that we obtain the μCRL specifications automatically from the Erlang code, while they manually translate the Java code into μCRL. This is a significant advantage since, as the authors themselves admit, "We think that the automatic translation of the [Java] code is very important from a methodological point of view and for the "industrial" application of the verification technique [...]".

References

[1] T. Arts, C. B. Earle, and J. J. Sánchez-Penas. Translating erlang to μcrl. In *Fourth International Conference on Application of Concurrency to System Design*, June 2004.

[2] T. Arts, C. Benac Earle and J. Derrick. Verifying Erlang code: a resource locker case-study, In *Proc. of Int. Symposium on Formal Methods Europe*, LNCS 2391, p. 183-202, Springer-Verlag, Copenhagen, Denmark, July 2002.

[3] T. Arts and T. Noll. Verifying generic Erlang client-server implementations. In Proc. of IFL2000*, LNCS 2011, p. 37–53, Springer Verlag, Berlin*, 2000.

[4] T. Arts and J. J. Sánchez-Penas. Global scheduler properties derived from local restrictions. In Proc. ACM SIGPLAN Erlang workshop, *Pittsburg, USA*, October 2002.

[5] J. Corbett, M. Dwyer, and L. Hatcliff. Bandera: A source-level interface for model checking java programs. In Teaching and Research Demos at ICSE'00, *Limerick, Ireland*, June 2000.

[6] CWI. μmcrl: A *language and tool set to study communicating processes with data*, February 1999.

[7] Building a better bug-trap. *Economist Technology Quarterly*, June 2003.

[8] E. Emerson and C.-L. Lei. Efficient model checking in fragments of the propositional mu-calculus. In Proc. LICS, pages 267–278, 1986.

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

[10] H. Garavel, F. Lang, and R. Mateescu. An overview of cadp 2001. *European Association for Software Science and Technology (EASST) Newsletter*, 4:13–24, August 2002.

[11] J. F. Groote. The syntax and semantics of timed mcrl. Technical Report SEN-R9709, CWI, Amsterdam, 1997.

[12] J. F. Groote, W. Fokking, and M. Reiniers. Modelling concurrent systems: Protocol verification in μCRL, April 2000.

[13] K. Havelund and T. Pressburger. Model checking JAVA programs using JAVA pathfinder. *STTT*, 2(4):366–381, March 2000.

[14] G. Holzmann. *The Design and Validation of Computer Protocols*. Pretence Hall, 1991.

[15] T. Janowski and M. Joseph. Dynamic scheduling and fault-tolerance: Specification and verification. *Real-Time Systems*, 20:51–81, January 2001.

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

[17] R. Mateescu. Local model-checking of an alternation-free value-based modal mu-calculus. In *Proceedings of the International Workshop on Software Tools for Technology Transfer STTT'98, Aalborg, Denmark*, July 1998.

[18] J. Rushby. Systematic formal verification for fault-tolerant time-triggered algorithms. *IEEE Transactions on Software Engineering*, 25(5):651–660, September 1999.

[19] S. O. J. Rushby and N. Shankar. Pvs: A prototype verification system. In *11th International Conference on Automated Deduction (CADE), Deepak Kapur eds. Lecture Notes in Artificial Intelligence, volume 607, pages 748-752, Springer Verlag, Saratoga, NY*, June 1992.

[20] J. J. Sánchez Penas and C. Abalde Ramiro. Extending the VoDKa

architecture to improve resource modeling. In *proceedings of the 2nd ACM SIGPLAN Erlang Workshop (PLI'03), Uppsala, Sweden*, August 2003.

[21] F. Schneider, S. M. Easterbrook, J. R. Callahan, and G. H. Holzmann. Validating requirements for fault tolerant systems using model checking. In *Proceedings, 3rd International Conference on Requirements Engineering, 4-13, Colorado Springs, Colorado*, April 1998.

[22] J. van de Pol and M. V. Espada. Formal specification of javaspaces architecture using μcrl. In *Proc. of 5th int. conf. on Coordination Models and Languages, York, UK, April 2002, COORDINATION LNCS 2315, pp. 274-290, Springer Verlag*, 2002.

[23] A. G. Wouters. Manual for the μcrl tool set (version 2.8.2). Technical Report SEN-R0130, CWI, Amsterdam, 2001.