Verifying Erlang Code: A Resource Locker Case-Study

Thomas Arts[1], Clara Benac Earle[2], and John Derrick[2]

[1] Ericsson, Computer Science Laboratory
Box 1505, 125 25 Älvsjö, Sweden
thomas@cslab.ericsson.se
[2] University of Kent, Canterbury
Kent CT2 7NF, United Kingdom
{cb47,jd1}@ukc.ac.uk

Abstract. In this paper we describe an industrial case-study on the development of formally verified code for Ericsson's AXD 301 switch. For the formal verification of Erlang software we have developed a tool to apply model checking to communicating Erlang processes. We make effective use of Erlang's design principles for large software systems to obtain relatively small models of specific Erlang programs. By assuming a correct implementation of the software components and embedding their semantics into our model, we can concentrate on the specific functionality of the components. We constructed a tool to automatically translate the Erlang code to a process algebra with data. Existing tools were used to generate the full state space and to formally verify properties stated in the modal μ-calculus.
As long as the specific functionality of the component has a finite state vector, we can generate a finite state space, even if the state space of the real Erlang system is infinite. In this paper we illustrate this by presenting a case-study based on a piece of software in Ericsson's AXD 301 switch, which implements a distributed resource locker algorithm. Some of the key properties we proved are mutual exclusion and non-starvation for the program.

Keywords: Model checking, formal verification, telecommunication, Erlang, process algebra

1 Introduction

Ericsson's AXD 301 is a high capacity ATM switch [5], used, for example, to implement the backbone network in the UK. The control software for this switch is written in the functional language Erlang [1]. The software consists of over five hundred thousands lines of Erlang code and complete formal verification of such large projects is too ambitious at the moment. However, for some critical parts of the software, it is worth spending some effort to increase trust in the chosen implementation.

L.-H. Eriksson and P. Lindsay (Eds.): FME 2002, LNCS 2391, pp. 184–203, 2002.

In Ericsson the software in such large projects is written according to rather strict design principles. For example in the AXD software, a few software components have been specified in the beginning of the project. These components can be seen as higher-order functions for which certain functions have to be given to determine the specific functionality of the component. About eighty percent of the software implements code for this specific functionality of one of these components, the majority of this for the *generic server* component. The generic server is a component that implements a process with a simple state parameter and mechanism to handle messages in a *fifo* message queue. The generic part of the component has been extensively tested and carefully thought through. In other words, if an error occurs in one of the several thousands of server processes, it is assumed to be an error in the specific functionality of that server causing the error.

To help increase trust in the particular implementation, we constructed a tool to translate server processes and their clients into a process algebraic model, such that we can generate all possible communication patterns that occur in the software. The model incorporates data as well, since messages sent between server and clients contain data that influences the behaviour of the protocol. The process architecture of a system is in general not derivable from the Erlang code; information about which process is communicating with a certain process is in principle only visible at runtime. However, the software component to ensure fault tolerance of the system, the so called *supervisor* contains static information about the relation between processes. Again, this component consists of a generic part and a few functions implementing the specific behaviour. By using the code of the specific functions of the supervisor processes, we are able to configure the process algebra models with a fixed (but flexible per translation) number of processes.

The case-study we had at hand implemented an algorithm for resource management. A server process, the so called *locker*, provides access to an arbitrary number of resources, for an arbitrary number of client processes. The clients may either ask for *shared* or *exclusive* access to the resources.

We used our tool, together with two external tools, on the key portions of the locker module as it appears in the AXD 301 switch. The external tools were used to generate the state space, reduce the state space with respect to bisimulation relations and to model check several properties. We successfully verified mutal exclusion and non-starvation for exclusive locks, priority of exclusive locks over shared locks and non-starvation of shared locks in the absence of exclusive requests. Proving the safety properties was rather straightforward, but the non-starvation is normally expressed by a formula with alternating fix points. Therefore, we used hiding and bisimulation to remove irrelevant cycles from the state space, and this allowed a simplified property to be checked.

The paper is organised as follows: we start with a brief explanation of the AXD 301 switch in Sect. 2. Thereafter we explain the software components we focussed on, viz. the *generic server* and *supervisors* in Sect. 3. The actual Erlang code, given in Sect. 4, is built using those components and along with the code

we describe the implemented algorithm. The key points of the translation of Erlang into a process algebra model are presented in Sect. 5. This model is used to generate the labeled transition system in which the labels correspond to communication events between Erlang processes. In Sect. 6 we summarize which properties have been proved for the code using model checking in combination with bisimulation reduction. We conclude with some remarks on performance and feasability, and a comparison to other approaches (Sect. 7).

2 Ericsson's AXD 301 Switch

Ericsson's AXD 301 is a high capacity ATM switch, scalable from 10 to 160 GBits/sec [5]. The switch is, for example, used in the core network to connect city telephone exchanges with each other.

From a hardware point of view, the switch consists of a switch core, which is connected on one side to several device processors (that in their turn are connected to devices), and on the other side to an even number of processing units (workstations). The actual number of these processing units depends on the configuration and demanded capacity and ranges from 2 till 32 (see Fig. 1).

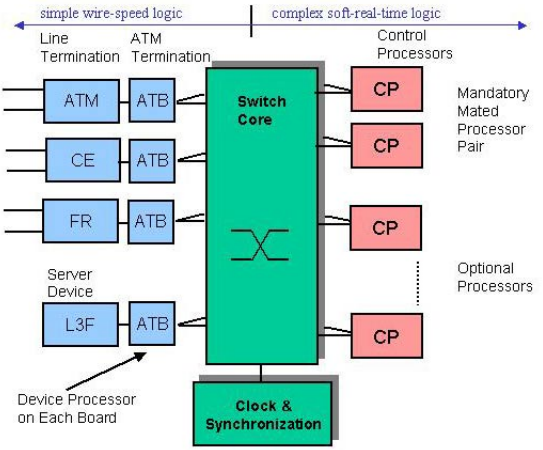

Fig. 1. AXD 301 hardware architecture

The workstations operate in pairs for reasons of fault tolerance; one workstation is assigned to be the *call control (cc)* node and the other the *operation and maintenance (o&m)* node. Simply put, call control deals with establishing connections, and operation and maintenance deals with configuration management, billing and such. Both the *cc* and *o&m* software consists of several applications, which on their turn implement many concurrently operating processes.

Every workstation runs one Erlang node, i.e., a program to execute Erlang byte code implementing several thousands of concurrent Erlang processes. The

critical data of these processes is replicated and present on at least two nodes in the system. In case a workstation breaks down, a new Erlang node is started on the pairing workstation and depending on the functionality of the broken node, either the *cc* or the *o&m* applications are started.

The distributed resource locker is necessary when the broken workstation is restarted (or replaced) and available again for operation. A new Erlang node is started at the workstation, and the pairing workstation can leave one of its tasks to the restarted workstation. Typically *o&m* will be moved, since that is easiest to move. Although easiest, this is not without consequences. Every *o&m* application may access several critical resources and while doing so, it might be hazardous to move the application. For that reason the designers of the switch have introduced a classical resource manager, here called a *locker*. Whenever any of the processes in any application needs to perform an operation during which that application cannot be moved, it will request a lock on the application. The lock can be shared by many processes, since they all indicate that the application is to remain at its node. The process that wants to move an application will also request a lock on that application, but an exclusive one. Using this lock, a process is guaranteed to know when it can safely move an application.

3 Erlang Software Components

In Ericsson's large software projects the architecture of the software is described by means of software components, i.e., the implementation is specified by means of communicating servers, finite state machines, supervisors and so. In the control software for the AXD about eighty percent of the software is specified in terms of such components, the majority of it as processes that behave like servers.

3.1 Generic Server Component

A server is a process that waits for a message from another process, computes a certain 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.

In Erlang one implements a server by creating a process that evaluates a (non-terminating) recursive function consisting of a receive statement in which every incoming message has a response as result.

```
serverloop(State) ->
  receive
    {call,Pid,Message} ->
       Pid ! compute_answer(Message,State),
       serverloop(compute_new_state(Message,State))
  end.
```

Erlang has an asynchronous communication mechanism where any process can send (using the `!` operator) a message to any other process of which it happens

to know the *process identifier* (the variable `Pid` in the example above). Sending is always possible and non-blocking; the message arrives in the unbounded mailbox of the specified process. The latter process can inspect its mailbox by the `receive` statement. A sequence of patterns can be specified to read specific messages from the mailbox. In the example above the first message in the mailbox which has the form of a tuple is read, where the first argument of the tuple should be the atom `call`, the variable `Pid` is then bound to the second argument of this tuple, and `Message` is bound to its last argument.

Of course, this simple server concept gets decorated with a lot of features in a real implementation. There is a mechanism to delay the response to a message, and some messages simply never expect a reply. Certain special messages for stopping the server, logging events, changing code in a running system and so on, are added as patterns in the receive loop. Debugging information is provided, used during development and testing. All together this makes a server a rather large piece of software and since all these servers have the same structure, it is a big advantage to provide a *generic server* implementation. This generic server has all features of the server, apart from the specific computation of reply message and new state. Put simply, by providing the above functions `compute_answer` and `compute_new_state` a fully functional server is specified with all necessary features for production code.

Reality is a bit more complicated, but not much: when starting a server one provides the name of a module in which the functions for initialisation and call handling are specified. One could see this as the generic server being a higher-order function which takes these specific functions, called *callback functions*, as arguments. The interface of these functions is determined by the generic server implementation. The initialisation function returns the initial state. The function `handle_call` is called with an incoming message, the client process identifier, and state of the server. It returns a tuple either of the form {`reply,Message,State`}, where the server takes care that this message is replied to the client and that the state is updated, or {`noreply,State`} where only a state update takes place. The locker algorithm that we present in this paper is implemented as a callback module of the generic server, thus the locker module implements the above mentioned functions for initialisation and call handling.

Client processes use a uniform way of communicating with the server, enforced by embedding the communication in a function call, viz. `gen_server:call`. This call causes the client to suspend as long as the server has not replied to the message. The generic server adds a unique tag to the message to ensure that clients stay suspended even if other processes send messages to their mailbox.

3.2 Supervisor Component

The assumption made when implementing the switch 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 vanished; 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 is implemented. This process evaluates a function that creates processes which it will monitor, which we refer to as its children. 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.

All the processes in the AXD 301 software are children in a big tree of supervisor processes. Thus, the locker and the clients of the locker also exist somewhere in this tree. In our case-study we implemented a small supervisor tree for only the locker and a number of clients (Fig. 2).

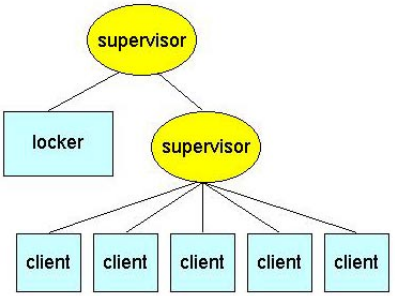

Fig. 2. Supervisor tree for locker and clients

The root of the tree has two children: the locker and another supervisor, which has as children all the client processes. As in the real software, the whole locker application is started by evaluating one expression, which starts building the supervisor tree and makes all processes run.

It is important to realize that we use this supervision tree to start the locker in different configurations. As an argument of the start function for the supervisor we provide the set of resources that the specific clients want to access. The expression `locker_sup:start([{[a],shared},{[a,b],exclusive}])`, for example, would start a supervisor tree with a locker and two clients, one client repeatedly requesting shared access to resource `a`, the other repeatedly requesting exclusive access to the resources `a` and `b`.

4 The Resource Locker Algorithm

In the previous section we described how the locker and client processes are placed in a supervision tree. We also mentioned that the locker is implemented as a callback module for the generic server. In this section we present the actual implementation of the client and locker and we explain the underlying algorithm.

We present a significant part of the actual Erlang code in order to stress that we verify Erlang code and to illustrate the complexity of the kind of code we can deal with. The full case-study contains about 250 lines of code in which many advanced features of Erlang are used[1].

4.1 Code of the Client

The client process is implemented in a simple module, since we have abstracted from all evaluations in clients that do not directly relate to entering and leaving the critical section. The generic server *call* mechanism is used to communicate with the locker.

```
-module(client).

start(Locker,Resources,Type) ->
    {ok,spawn_link(client,loop,[Locker,Resources,Type])}.

loop(Locker,Resources,Type) ->
    gen_server:call(Locker,{request,Resources,Type}),
    gen_server:call(Locker,release),
    loop(Locker,Resources,Type).
```

Between the two synchronous calls for request and release is the so called critical section. In the real implementation some critical code is placed in this critical section, but we have (manually) abstracted from that. The variable `Type` is either instantiated with `shared` or `exclusive` and `Resources` is bound to a list of resources that the client wants access to.

4.2 Code of the Locker

The code of the locker algorithm is given as a generic server callback module. The state of this server contains a record of type `lock` for every resource that the locker controls.

```
-module(locker).
-behaviour(gen_server).

-record(lock,{resource,exclusive,shared,pending}).
```

The `lock` record has four fields: `resource` for putting the identifier of the resource, `exclusive` containing the process that is having exclusive access to the resource (or `none` otherwise), `shared` containing a list of all processes that are having shared access to the resource, and `pending` containing a list of pending processes, either waiting for shared or for exclusive access.

[1] The code is available at `http://www.cs.ukc.ac.uk/people/rpg/cb47/`

The supervisor process constructs a list of all resources involved from the starting configuration and passes it to the initialisation of the locker. The locker initialisation function then initialises a `lock` record for every resource in that list. The state of the server is built by taking this list and constructing a tuple together with the lists for all exclusive requests and all shared requests that have not been handled so far.

```
init(Resources) ->
  {ok,{map(fun(Name) ->
              #lock{resource = Name,
                    exclusive = none, shared = [], pending = []}
           end,Resources),[],[]}}.
```

The latter two (initially empty) lists in the state of the server are used by the algorithm to optimize the computations performed when deciding which pending client is the next one that gets access. The first client in the pending list of the `lock` record is not necessarily granted permission to obtain the resource. It may be the case that the same client also waits for another resource, for which another client has higher priority. The priority could be reconstructed by building a graph of dependencies between the clients, but it is much easier to store the order in which the requests arrive.

Whenever a client requests a resource, the function `handle_call` in the locker module is called. This function first checks whether all requested resources are available. If so, it claims the resources by updating the `lock` records. The client receives an acknowledgement and the state of the server is updated accordingly. If the resources are not available, the `lock` records are updated by putting the client in the pending lists of the requested resources. The priority lists are changed, resulting in a new state for the server. No message is sent to the client, which causes the client to be suspended.

```
handle_call({request,Resources,Type},Client,{Locks,Excls,Shared}) ->
  case check_availables(Resources,Type,Locks) of
       true ->
         NewLocks =
           map(fun(Lock) ->
                  claim_lock(Lock,Resources,Type,Client)
               end,Locks),
         {reply, ok, {NewLocks,Excls,Shared}};
       false ->
         NewLocks =
           map(fun(Lock) ->
                  add_pending(Lock,Resources,Type,Client)
               end,Locks),
         case Type of
              exclusive ->
                {noreply, {NewLocks,Excls ++ [Client],Shared}};
              shared ->
```

```
                {noreply, {NewLocks,Excls,Shared ++ [Client]}}
        end
  end;
```

A client can release all its obtained resources by a simple `release` message, since the identity of the client is sufficient to find out which resources it requested. After removing the client from the fields in the `lock` record, it is checked whether pending processes now have the possibility to access the requested resources. This happens with higher priority for the clients that request exclusive access, than for the clients that request shared access. The algorithm prescribes that clients that requested shared access to a resource but are waiting for access, should be by-passed by a client that requests exclusive access.

```
handle_call(release, Client, {Locks,Exclusives,Shared}) ->
  Locks1 =
    map(fun(Lock) ->
           release_lock(Lock,Client)
        end,Locks),
  {Locks2,NewExclusives} =
    send_reply(exclusive,Locks1,Exclusives,[]),
  {Locks3,NewShared} =
    send_reply(shared,Locks2,Shared,[]),
  {reply,done, {Locks3,NewExclusives,NewShared}}.
```

The function `send_reply` checks for a list of pending clients (either requesting exclusive or shared access) whether they can be granted access. If so, the client receives the acknowledgement that it was waiting for, and the state of the server is updated.

```
send_reply(Type,Locks,[],NewPendings) ->
  {Locks,NewPendings};
send_reply(Type,Locks,[Pending|Pendings],NewPendings) ->
  case all_obtainable(Locks,Type,Pending) of
       true ->
         gen_server:reply(Pending,ok),
         send_reply(Type,
                    map(fun(Lock) ->
                           promote_pending(Lock,Type,Pending)
                        end,Locks),Pendings,NewPendings);
       false ->
         send_reply(Type,Locks,Pendings,NewPendings ++ [Pending])
  end.
```

The above mentioned Erlang functions in the locker combine message passing and computation. The rest of the function is purely computational and rather straight forward to implement. Here we only show the more interesting aspects.

The function `check_availables` is used to determine whether a new requesting client can immediately be helped. A resource is available for exclusive access

if no client holds the resource and no other client is waiting for exclusive access to it. Note that it is not sufficient to only check whether no client accesses the resource at the time, since this could cause a starvation situation. Imagine two resources and three clients, such that client 1 requests resource A, client 2 requests resource B, and thereafter client 3 requests both resources. Client 1 releases and requests resource A again, client 2 releases and requests B again. If this repeatedly continues, client 3 will wait for ever to get access, i.e., client 3 will starve.

	A	B
access	1	
pending		
access	1	2
pending		
access	1	2
pending	3	3
access		2
pending	3	3

	A	B
access	1	2
pending	3	3
access	1	
pending	3	3
access	1	2
pending	3	3
⋮		

This scenario indicates that in general one has to pay a price for optimal resource usage: viz. a possible starvation. Therefore, in the implementation it is checked whether a client is waiting for a certain resource. Similar to the exclusive case, for shared access the resource is available if no process holds the resource exclusively, neither is a client waiting for access to it.

The function `add_pending` simply inserts the client in the pending lists of the resources it is requesting. An optimisation is applied when inserting clients in the pending list: clients requesting exclusive access are mentioned before the ones requesting shared access. This allows a quick check to see if there is a client exclusively waiting for a resource, such a client should then be at the head of the pending list.

The difference between the functions `check_available` and `all_obtainable` is that in the latter the clients have already been added to the pending lists of the requested resources and therefore it should be checked that they are at the head of these lists instead of checking that these lists are empty. Moreover, there might be several clients able to get access to their resources after only one release, e.g. resources that were taken exclusively can be shared by several clients and a client that occupied several resources can free those resources for a number of different clients.

5 Translating Erlang into Process Algebra

In order to check that certain properties hold for all possible runs of the program, we automatically translate the Erlang modules into a process algebraic

specification. The translation approach means that we do not have to make an efficient state space generation tool ourselves, it also allows us to distinghuish in a formal way communication actions and computation, and allows us to use tools developed for analyzing process algebra's.

The process algebra we used to translate to is μCRL [15], where we in particular used the fact that we can express data in this algebra. Several tools have been developed to support verification of μCRL specifications [9,24]. We mainly used the state space generation tool and experimented with static analysis tools to obtain specifications that resulted in smaller state spaces after generation.

We have experimented with translating the synchronous communication imposed by the *call* primitive of the generic server component directly in a synchronizing pair of actions in μCRL. This results in comfortably small state spaces, much smaller than when we implement a buffer for a server and use both synchronization between client and buffer of the server and synchronization between buffer and server. The latter is, however, necessary if we use the more extended functionality of the generic server, where we also have an asynchronous way of calling the server.

The buffer associated with each process is parameterized by its size and by default unbounded; during the verification process the buffer is bound to a certain size to allow the verifier to experiment with the size. The latter is important, since some errors cause a buffer overflow, which induces a non-terminating generation of the state space. However, if the message queue is bound to a low enough value, the buffer overflow is visible as an action in the state space. We use the knowledge about the generic server component to implement a restricted buffer in μCRL: the generic server uses a *fifo* buffer structure. This is in contrast with a classic Erlang buffer where an arbitrary Erlang process can read messages from the buffer in any order.

Moreover, we add several assertions that, if not fullfilled, cause the Erlang program to crash. These assertions mainly originate from pattern matching in Erlang, which is not as easily expressed in μCRL. As soon as the action `assertion(false)` occurs in the state space, the corresponding Erlang process would have crashed and we obtain for free a path from the initial state to the location where this happens. We provided the possibility to add user defined actions. By annotating the code with dummy function calls, we may add extra actions to the model to allow us to explicitly visualize a certain event. This feature was used, for example, when proving mutual exclusion, as is described in the next section.

Erlang supports higher-order functions, but μCRL does not. Luckily, in practice only a few higher-order alternatives are used, like `map`, `foldl`, `foldr`, *etc.* For the purpose of this locker version we wrote a source to source translation on the Erlang level to replace function occurrences like

```
map(fun(X) -> f(X,Y1,...,Yn) end, Xs)
```

by a call to a new function `map_f(Xs,Y1,...,Yn)` which is defined and added to the code as

```
map_f([],Y1,...,Yn)     -> [];
map_f([X|Xs],Y1,...,Yn) -> [f(X,Y1,...,Yn)| map_f(Xs,Y1,...,Yn)].
```

By using this transformation we automatically get rid of all `map` functions in the Erlang code.

With some minor tricks the side-effect free part of the Erlang code is rather easily translated into a term rewriting system on data, as necessary in a μCRL model. With respect to the part with side-effects, we are faced with two problems:

1. in μCRL we have to specify exactly which processes start and with which arguments they are started,
2. in μCRL a process is described with all side effects as actions on the top level. Thus, a function with side-effect cannot return a result.

The first problem is tackled by using the supervision tree that describes the Erlang processes that should be started. Using this structure, a translation to μCRL's initial processes is performed automatically. The second problem is solved by analyzing the call graph of functions that contain side-effects, i.e., functions that call the server or handle this call. We implement a stack process comparable to a call-stack when writing a compiler. Given the call graph, we can replace the nested functions with side-effect by functions that send as their last action a message to the stack process to push a value, since returning it is not possible. The stack process implements a simple stack with push and pop operations. A *pop* message is sent to the stack process directly after the point where the nested function has been called. This solution works fine, but, clearly, increases the state space.

With our translation tool for Erlang we can automatically generate μCRL models for Erlang programs like the one presented in Sect. 4. We build such models for a certain configuration in the same way as we start the code in a certain configuration. In Sect. 3.2 we explained how to start the locker process, for example, by evaluating `locker_sup:start([{[a],shared},{[a,b],exclusive}])`. Evaluating the same expression in our tool instead of in the Erlang runtime system, results in a μCRL model[2] for this configuration.

By using the state space generation tool for μCRL, we obtain the full state space, in the form of a labeled transition system (LTS), for the possible runs of the Erlang program. The labels in this state space are syntactically equal to function calls in Erlang that accomplish communication, e.g. `gen_server:call` and `handle_call`. This makes debugging the Erlang program easy when a sequence in the state space is presented as counter example to a certain property.

Once we have obtained the state space, the CÆSAR/ALDÉBARAN toolset [14] is used for verifying properties, as is described in the next section.

[2] For completeness one of these automatically generated μCRL models is available at `http://www.cs.ukc.ac.uk/people/rpg/cb47/`

6 Checking Properties with a Model Checker

The verification of safety and liveness properties are crucial in this application and were the key requirements that the AXD 301 development team were interested in. Safety properties include mutual exclusion for exclusive locks and priority of exclusive locks over shared locks. These and other properties have succesfully been verified and here we explain in detail how mutual exclusion (Sect. 6.1) and non-starvation (Sect. 6.2) are proved. The liveness property, non-starvation, is the more difficult of the two.

In order to verify the properties we have used the CÆSAR/ALDÉBARAN toolset which provides a number of tools including an interactive graphical simulator, a tool for visualization of labeled transition systems (LTSs), several tools for computing bisimulations (minimizations and comparisons), and a model checker [14]. Many aspects of the toolset were found useful for exploring the behaviour of the algorithm, but here we concentrate on the model checker.

Model checking (e.g. [7]) is a formal verification technique where a property is checked over a finite state concurrent system. 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 propery, 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.

The logic used to formalize properties is the regular alternation-free μ-calculus which is a fragment of the modal μ-calculus [20,11], a first-order logic with modalities and least and greatest fixed point operators. Logics like *CTL* or *ACTL* allow a direct encoding in the alternation free μ-calculus.

6.1 Mutual Exclusion

To prove mutual exclusion we formulate a property expressing that when a client gets exclusive access to a resource, then no other client can access it before this client releases the resource. In order to simplify checking this we add two actions, `use` and `free`, to the Erlang code which are automatically translated into the μCRL specification[3]. As soon as a client process enters its critical section, the `use` action is applied with the list of resources the client is requesting as an argument.

Before the client sends a release message to the locker process, it performs a `free` action. In the logic we specify the action in plain text or with regular expressions. However, the formalism does not permit binding a regular expression in one action and using it in another. Therefore, we have to specify mutual exclusion for every resource in our system. We defined a macro to help us improve readability:

$$BETWEEN(\mathsf{a}_1,\ \mathsf{a}_2,\ \mathsf{a}_3) = [\mathsf{-}^* \,.\, \mathsf{a}_1 \,.\, (\neg \mathsf{a}_2)^* \,.\, \mathsf{a}_3]\mathit{false}$$

[3] The tools allow renaming of labels in the LTS, which could have been used as well.

stating that 'on all possible paths, after an (a_1) action, any (a_3) action must be preceded by an (a_2) action'.

The mutual exclusion property depends on the number of resources. For a system with two resources, A and B, the mutual exclusion property is formalized by

$$\begin{aligned} &MUTEX(\mathsf{A},\ \mathsf{B}) = \\ &\quad BETWEEN('use(.*\mathsf{A}.*, exclusive)',\ 'free(.*\mathsf{A}.*)',\ 'use(.*\mathsf{A}.*,.*)') \ \wedge \\ &\quad BETWEEN('use(.*\mathsf{B}.*, exclusive)',\ 'free(.*\mathsf{B}.*)',\ 'use(.*\mathsf{B}.*,.*)') \end{aligned}$$

Informally the property states that when a client obtains exclusive access to resource A no other client can access it until the first client frees the resource, and the same for resource B.

The mutual exclusion property has been successfully checked for various configurations up to three resources and five clients requesting exclusive or shared access to the resources.

For example, a scenario with five clients requesting exclusive access to three resources where client 1 requests A, client 2 requests B, client 3 requests A, B and C, client 4 requests A and B, and client 5 requests C, contains about 30 thousand states. Building an LTS for this example takes roughly thirteen minutes, while checking the mutual exclusion property takes only nine seconds. A bigger state space of one million states needs one hour to be built and four minutes to be checked for mutual exclusion. Part of the reason that building the LTS takes much more time than checking a property is that we deal with data and that a lot of computation is done inbetween two visible actions (only visible actions correspond to states in the LTS).

As stated in the previous section, model checking is a powerful debugging tool. Imagine that the code of the locker contains the following error: the function `check_available` is wrongly implemented such that when a client requests a resource there is no check that the resource is being used by another client. Now consider a scenario with two clients, client 1 and client 2, requesting the same resource A. Given the LTS for this scenario and the property $MUTEX(\mathsf{A})$, the model checker returns **false** and the counter example as shown in Fig. 3.

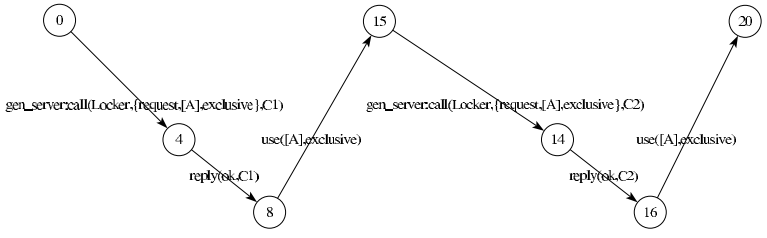

Fig. 3. mutex counterexample

The counter example generated depicts an execution trace of client 1 requesting and obtaining resource A and client 2 requesting and obtaining resource A,

that is, both processes enter the critical section and, therefore, mutual exclusion is not preserved. The numbers that appear inside the circles correspond to the numbers of the states as they appear in the complete LTS. By keeping the Erlang code and our μCRL specification as close as possible, this trace helps us easily identify the run in the Erlang program.

Although we only use a small number of clients and resources, this already illustrates the substantive behaviour. Like with testing software, we choose our configurations in such a way that we cover many unique situations, however, in contrast to testing, we explore all possible runs of a certain configuration. Faults that occur when ten clients request sets out of eight resources are most likely found as well in configurations with five clients and four resources.

6.2 Non-starvation

Starvation is the situation where a client that has requested access to resources never receives permission from the locker to access them. Because exclusive access has priority over shared access, the algorithm contains potential starvation for clients requesting shared access to resources that are also exclusively requested. More precisely, the clients requesting exclusive access have priority over all clients that are waiting for shared access, therefore the ones requesting shared access can be withheld from their resources.

Within the use of the software in the AXD at most one client is requesting exclusive access to the resources (the take-over process). In that setting, the starvation of clients requesting shared access cannot occur, as we prove below. The reason is the synchronized communication for the release. As soon as the client requesting exclusive access sends a release to the locker, all waiting shared clients get access to the resources they requested (they share them). Only after this is an acknowledgement sent on the release.

Here we look at more general cases where more than one client is requesting exclusive access to the resources (since this type of scenarios may occur in a more general setting).

Because of the fact that the algorithm contains a certain form of starvation, the property one wants to check for non-starvation has to be specified with care. The following cases have been verified: non-starvation of clients requesting exclusive access and non-starvation of clients requesting shared access in the presence of at most one exclusive request.

Non-starvation for exclusive access. Proving that there is no starvation for the clients requesting exclusive access to the resources turned out to be tricky. This is caused by the fact that there are traces in the LTS that do not correspond to a fair run of the Erlang program.

The Erlang run-time system guarantees that each process obtains a slot of time to execute its code. However, in the LTS there are traces where certain processes do not get any execution time, even though they are enabled along the path. To clarify this, let us consider a scenario with two resources and three clients.

Client 1 requests resource A and obtains access to it, client 2 request resource A and has to wait. Thereafter client 3 requests B, obtains access to it, releases the resource and requests it again. In the LTS there is a clear starvation situation for client 2, viz. infinitely often traversing the cycle that client 3 is responsible for ($4 \to 23 \to 10 \to 24 \to 4 \to \ldots$ in Fig. 4). The above scenario, however, does

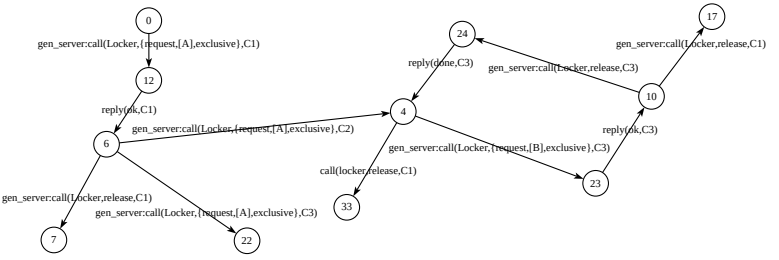

Fig. 4. Unreal starvation of client 2

not reflect the real execution of the program since the Erlang run-time system will schedule client 1 to execute its code. Client 1 will sooner or later release resource A, which causes client 2 to get access to the resource. In the LTS, it is visible that client 2 has the possibility to access resource A, but the unfair cycle of client 3 hides the fact that this will happen. Note, though, that we cannot simply forget about every cycle. If the cycle would be shown with resource A instead of B mentioned, then this would indicate a real starvation.

One could think of a number of solutions to solve the problem of cycles in the LTS that do not correspond to fair infinite computations in the Erlang program. For example, one could explicity model the Erlang run-time scheduler. However, modelling the scheduler is a rather complex solution that would increase the size of the LTS notably. Besides, we would be scheduling the actions in the μCRL code, not in the real Erlang code. Thus we would not be sure that starvation really occurs in the Erlang implementation.

Another possible solution is to encode the unrealistic cycles, i.e., the ones that the real scheduler would exclude, in the property so that they are ignored. In order to do that we need to characterize the unrealistic cycles. An unrealistic cycle corresponds to unfair execution of a number of clients that is *independent* of the client one wants to prove non-starvation for.

In our specific case a client depends on another client when the disjunction of the sets of resources they request is non-empty. Given that one is interested in proving non-starvation of a certain client, then computing the clients that are independent of this client is done by taking the complement of the reflexive, transitive closure of this dependency relation. If we now consider all actions of independent clients to be internal actions (τ actions in process algebra terminology), then non-starvation of the client C we are interested in, could be

expressed by the guaranteed occurence of $'reply(ok,C)'$ in any path starting from $'gen\_server : call(.^*request.^*, C)'$, modulo possible cycles with only τ steps. This can be expressed by the following formula in the μ-calculus, where we allow only finite cycles of actions that are neither τ, nor $'reply(ok,C)'$ actions. Infinite sequences of only τ actions are, however, permitted:

$$[-^* \,.\, 'gen\_server : call(.^*request.^*, C)'] \\ \mu X.(\nu Y.(\langle - \rangle true \ \wedge [\neg\tau \ \wedge \ \neg' reply(ok,C)']X \ \wedge \ [\tau]Y)).$$

The disadvantage with the above formula is that it has alternating fixed point operators and hence the model checker cannot verify this property.

The solution is to reduce the state space by use of observational equivalence [23] and a facility to do this is provided by the CÆSAR/ALDÉBARAN toolset. By applying this reduction we replaced actions of independent processes by internal actions, we obtain a model in which pure τ cycles no longer occur. Thus, we removed all unfair cycles.

Modulo observational equivalence, the formula to prove non-starvation becomes much simpler and in particular is alternation-free:

$$NONSTARVATION(C) = \\ [-^* \,.\, 'gen\_server : call(.^*request.^*, C)']\mu X.(\langle - \rangle true \ \wedge \ [\neg' reply(ok,C)']X)$$

Verification of non-starvation for a configuration of clients and resources is now performed by consecutively selecting a process that requests exclusive access to a set of resources. We manually determine the set of processes that is independent of this process, and then hide the labels of the independent processes The LTS obtained is reduced modulo observational bisimulation, and we can then verify the above given property on the reduced LTS.

In this way we successfully verified non-starvation of the clients requesting exclusive access to resources in several configurations. We also found a counter example, by checking this property for a process that requests shared access to resources in a configuration where two clients ask exclusive access to resource A and a third requests shared access to A. In this case we see that the third client is starving. This is exactly as we expect, since clients demanding exclusive access have priority over clients asking for shared access.

Non-starvation for shared access. Even though clients that request shared access to a resource may potentially starve, as explained above, we can still prove non-starvation of all the clients in the system, provided that at most one client demands exclusive access.

In analogy to the procedure described above, we hide the actions of independent processes and verify $NONSTARVATION(C)$ for every client C in the configuration. As such, the verification is performed successfully.

7 Conclusions

In this paper we describe an approach to verify properties of Erlang code. The approach consists of the following steps. First, the Erlang code is automatically translated to a μCRL specification. Second, a labeled transition system (LTS) is generated from this μCRL specification by using tools from the μCRL toolset. We then code up the property of interest in the alternation-free μ-calculus, and the LTS is checked against this property using the CÆSAR/ALDÉBARAN toolset. For some properties we transform the LTS (e.g., using hiding for non-starvation) so that we can model check with a simple formulation of the property of interest (e.g., one without alternating fixed points).

The case-study we have at hand, a critical part of the AXD 301 software consisting of about 250 lines of Erlang code, implements a resource locking problem for which we prove the obvious properties, viz. mutual exclusion and non-starvation. Mutual exclusion algorithms have been studied before (e.g. [10, 19,21,22]) and these algorithms have been proved correct. Automatically proving the same properties on a slightly different algorithm implemented in a real programming language, however, lifts formal methods from an academic exercise to industrial practice.

Similar projects for different programming languages exist, such as the verification on Java code [8,16] using the specification language Promela and LTL model checker SPIN [17]. The difference with those approaches is that we make extensive use of components on top of the language primitives, therewith obtaining smaller state spaces for similar problems. Moreover, the underlying logics for the model checkers differ, which makes different properties expressible in both approaches.

For Erlang there are also other relevant verification tools developed, e.g., a theorem prover with Erlang semantics build into it [3,13] and the model checker of Huch [18]. Huch's model checker works on Erlang code directly and provides the possibility to verify LTL properties.

The main difference between Huch's approach and the approach we sketch in this paper, is that Huch uses abstract interpretation to guarantee small (finite) state spaces. In Huch's approach all data is abstracted to a small, fixed set and tests on the data are often translated to non-deterministic choices. This approach is not suitable for our situation, since a non-deterministic choice whether a resource is available or not will result in error messages that do not reflect reality. That is, the properties we wish to verify are very data dependent and thus this particular approach to abstract interpretation will not work here.

The Erlang theorem prover can be used to prove similar properties, in particular if one uses the extra layer of semantics for software components added to the proof rules [4]. However, such a proof has to be provided manually, in contrast to more automatic approach we have explained here [4]. However, an advantage of the theorem prover is that one can reason about sets of configurations at once, and not fix the number of clients and resources per attempt.

[4] However, tactics can increase the degree of automation for the theorem prover

The translation of Erlang into μCRL is performed automatically. Our tool can deal with a large enough part of the language to make it applicable for serious examples. The tool computing the state spaces for μCRL models [9] is very well developed and stable. However, despite the many optimisations, it takes a few minutes up to hours to generate a state space. Whenever the model is obtained, model checking with the CÆSAR/ALDÉBARAN toolset [14] takes a few seconds up to a few minutes. Thus, the generation of the state space is rather slow compared to verifying it, which is partly due to the computation on the complex data structures we have in our algorithm. In particular, in the case when the property does not hold, creating the whole state space is often unnecessary: a counter-example could be provided without having all states available. A collaboration between both providers of the external tools recently resulted in an on-the-fly model checker to overcome this inconvenience. At the same time a distributed state space generation and model checking tool are being built as cooperation between CWI and Aachen University [6]. With such a tool, a cluster of machines can be used to quickly analyse rather large state spaces. Experiments showed the generation of an LTS with 20 million states in a few hours. We have not found serious performance problems and by these new developments we expect to push them forward even more.

Formal verification of Erlang programs is slowly becomming practically possible, particularly the development of new programs [2]. We plan to extend our translation tool to cover a few more components and to deal with fault tolerance. At the moment, crashing and restarting of processes is not considered inside the μCRL model, so that properties about the fault tolerance behaviour cannot be expressed. In the near future we plan to verify more software and construct a library of verified Erlang programs that can be used within Ericsson products.

Acknowledgements. We thank Ulf Wiger from Ericsson for providing us with the case-study and clarifying the use of this code. Specially helpful were the tool development teams at INRIA Rhône-Alpes and CWI with their support and advices, and Lars-Åke Fredlund and Dilian Gurov from SICS with their contribution in the discussions. We thank Howard Bowman from the University of Kent for useful explanations.

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 and C. Benac Earle. Development of a verified distributed resource locker, In Proc. of FMICS, Paris, July 2001.

[3] T. Arts, M. Dam, L-Å. Fredlund, and D. Gurov. System Description: Verification of Distributed Erlang Programs. In *Proc. of CADE'98*, LNAI 1421, p. 38–42, Springer-Verlag, Berlin, 1998.

[4] T. Arts and T. Noll. Verifying Generic Erlang Client-Server Implementations. In *Proc. of IFL2000*, LNCS 2011, p. 37–53, Springer Verlag, Berlin, 2000.

[5] S. Blau and J. Rooth. AXD 301 – A new Generation ATM Switching System. *Ericsson Review*, no 1, 1998.

[6] B. Bollig, M. Leucker, and M. Weber. Local Parallel Model Checking for the Alternation Free μ–Calculus. tech. rep. AIB-04-2001, RWTH Aachen, 2001.

[7] E.M. Clarke, O. Grumberg, D. Peled. Model Checking, MIT Press, December 1999.

[8] J. Corbett, M. Dwyer, L. Hatcliff. Bandera: A Source-level Interface for Model Checking Java Programs. In *Teaching and Research Demos at ICSE'00*, Limerick, Ireland, 4-11 June, 2000.

[9] CWI. `http://www.cwi.nl/~mcrl`. A *Language and Tool Set to Study Communicating Processes with Data*, February 1999.

[10] E. W. Dijkstra. Solution of a Problem in Concurrent Programming Control. In *Comm. ACM*, 8/9, 1965.

[11] E.A. Emerson and C-L. Lei. Efficient Model Checking in Fragments of the Propositional Mu-Calculus, In *Proc. of the 1st LICS*, p. 267-278, 1986.

[12] Open Source Erlang. `http://www.erlang.org`, 1999.

[13] L-Å. Fredlund, et. al. A Tool for Verifying Software Written in Erlang, To appear in: *STTT*, 2002.

[14] 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.

[15] J. F. Groote, The syntax and semantics of timed μCRL. Technical Report SEN-R9709, CWI, June 1997. Available from `http://www.cwi.nl`.

[16] K. Havelund and T. Pressburger, Model checking JAVA programs using JAVA PathFinder. *STTT*, Vol 2, Nr 4, pp. 366–381, March 2000.

[17] G. Holzmann, *The Design and Validation of Computer Protocols*. Edgewood Cliffs, MA: Pretence Hall, 1991.

[18] F. Huch, Verification of Erlang Programs using Abstract Interpretation and Model Checking. In *Proc. of ICFP'99*, Sept. 1999.

[19] D. E. Knuth. Additional Comments on a Problem in Concurrent Programming Control. In *Comm. ACM*, 9/5, 1966.

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

[21] L. Lamport. The Mutual Exclusion Problem Part II - Statement and Solutions. In *Journal of the ACM*, 33/2, 1986.

[22] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, Inc. San Francisco, California, 1996.

[23] R. Milner. A Calculus of Communicating Systems, Springer 1980.

[24] A. G. Wouters. Manual for the μCRL tool set (version 2.8.2). Tech. Rep. SEN-R0130, CWI, Amsterdam, 2001.