Int J Softw Tools Technol Transfer (2004) 5: 205–220 / Digital Object Identifier (DOI) 10.1007/s10009-003-0114-9

Development of a verified Erlang program for resource locking

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

[1] IT-University in Gothenburg, Box 8718, 402 75 Gothenburg, Sweden
e-mail: thomas.arts@ituniv.se
[2] University of Kent, Canterbury, Kent CT2 7NF, UK
e-mail: {cb47,jd1}@ukc.ac.uk

Published online: 19 December 2003 –

Abstract. In this paper, we describe a tool to verify Erlang programs and show, by means of an industrial case study, how this tool is used. The tool includes a number of components, including a translation component, a state space generation component and a model checking component.

To verify properties of the code, the tool first translates the Erlang code into a process algebraic specification. The outcome of the translation is made more efficient by taking advantage of the fact that software written in Erlang builds upon software design patterns such as client–server behaviours. A labelled transition system is constructed from the specification by use of the μCRL toolset. The resulting labelled transition system is model checked against a set of properties formulated in the μ-calculus using the CÆSAR/ALDÉBARAN toolset.

As a case study we focus on a simplified resource manager modelled on a real implementation in the control software of the AXD 301 ATM switch. Some of the key properties we verified for the program are mutual exclusion and non-starvation. Since the toolset supports only the regular alternation-free μ-calculus, some ingenuity is needed for checking the liveness property "non-starvation". The case study has been refined step by step to provide more functionality, with each step motivated by a corresponding formal verification using model checking.

Keywords: Formal methods – Software verification – Model checking – Functional programming – Erlang

1 Introduction

In this paper, we describe an approach to the verification of Erlang code which involves model checking an abstraction of the code by translating it into a process algebra.

The telecommunications company Ericsson is using the functional programming language Erlang [1, 13] for the development of concurrent/distributed software for telecommunications equipment. One of the larger examples of such a system is the AXD 301 high-capacity ATM switch [7], used to implement, for example, the backbone network in the UK. The software of this switch consists of about half a million lines of Erlang code.

This code is written in a development process which is rather similar to the Extreme Programming approach [23]: designers write and test itthemselves and in small iterations, and features are added to the code until a final release stage is reached.

At Ericsson, the software for large projects like the AXD 301 switch is written according to rather strict design principles. For the AXD software, a number of software components are used which have been specified for use in a number of Ericsson projects. 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 80% of the software implements code for this specific functionality of one of these components, the majority for the *generic server* component. The generic server is a component which implements a process with a simple state parameter and mechanism to handle messages in a *fifo* message queue.

Both the development process and the use of these library components ensure that the code is tested many times before the final implementation. For example, during development the software is often written during the day and tested overnight. The test cases are written by the designers in parallel with the code, and a test server automatically runs these test cases.

However, despite this extensive testing, for critical devices such as telecommunications switches, it is clearly preferable to have even higher levels of assurance that the code is correct. Our aim, therefore, is to build a formal verification tool that fits into this development process.

The tool supports (overnight) verification of properties, the purpose of which is to check aspects similar to the testing process. This paper describes the tool that we developed, and the use of the tool is illustrated by a case study taken from the AXD 301. The tool and approach are based on model checking a process algebraic representation of the Erlang code and therefore involve issues such as abstraction of code to specification, state space generation and model checking. The advantage of our approach and tool over testing should be clear: they cover a larger portion of the state space of the system; indeed, when the state space is finite, the whole state space can be verified.

The case study we describe in this paper is a distributed resource manager which was re-designed in the same way as real production code would be re-designed. In small iterations, the complexity of software was increased and properties were checked against these iterations in turn. Clearly our re-design is based on the same Ericsson design principles as the AXD 301 switch. Following these design principles and using real software components make the verification approach more realistic and easier; the message buffers of arbitrary Erlang processes are more complicated than the constrained message buffer in a generic server. Thus by using the semantics of the generic server, we obtain smaller state spaces.

Another requirement of our verification tool is that it should be accessible to Erlang programmers without forcing them to learn a specification language. Clearly, they will receive help when formulating the properties they want to prove, but in fact these need not change much during the iterations in the development. A special team provides proof scripts in which the properties are embedded, and these can be run against the Erlang code. The feedback of these scripts is in terms of traces in Erlang syntax, so that programmers can understand the counterexamples that the model checker has produced.

In one sense, this work is not new: 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 one uses a specification language in combination with a model checker to obtain a correct specification used to write an implementation in a programming language, or 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 is not proved correct by these approaches, but when an error is encountered, this may indicate an error in the implementation. As such, model checking can be seen as a very accurate testing method.

For the first approach, one of the most successful of the many examples is the combination of the specification language Promela and model checker SPIN [18]. The attractive merit of Promela is that this language is so close to the implementation language C that it becomes rather easy to derive the implementation from the specification in a direct, fault-free way. If one uses UML as the specification language and Java or C as the implementation language, one might require more effort (apart from the fact that model checking UML specifications is still an unresolved topic).

The work we describe here is part of the second approach, other examples of which include PathFinder [17] and Bandera [10], 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. We follow a similar approach to the translation of Java into Promela, checked by SPIN [17]; however, we translate Erlang into μCRL [16] and model check properties by using CÆSAR/ALDÉBARAN [15].

An earlier attempt for model checking Erlang code by Huch [19] differs in many ways from our approach. In contrast to Huch's approach, we consider data aspects which are crucial for the properties we wish to check in the Erlang code. In particular, Huch abstracts *case* statements by non-deterministic choices, which loses all reference to the data, whereas our model checking takes the data values into account.

This allows us to check for mutual exclusion and the absence of deadlock for the resource manager which will be the leading example of this paper. If one abstracts from the data in this program in such a way that *case* statements are translated into non-deterministic choices, then mutual exclusion is no longer guaranteed and can hence not be shown.

The paper is organised as follows: we start with a brief explanation of the AXD 301 switch in Sect. 2. Then we explain the software components we focused on, namely, 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 main part of our tool, the translation of Erlang code into a process algebra model, is presented in Sect. 5. This model is used to generate the labelled transition system in which the labels correspond to communication events between Erlang processes. We used additional external tools to generate the state space, reduce the state space with respect to bisimulation relations and model check several properties.

In Sect. 6, we focus on the mutual exclusion and non-starvation property which have been verified for the code using model checking in combination with bisimulation reduction. In Sect. 7, we show how we use scripts to automate the actual verification in the development process. We conclude in Sect. 8 with some remarks on performance and feasibility and a comparison with other approaches.

2 Ericsson's AXD 301 switch

Ericsson's AXD 301 is a high-capacity ATM switch, scalable from 10 to 160 Gbits/s [7]. The switch is used, for example, 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 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 control processors (workstations). The actual number of these control processors depends on the configuration and demanded capacity and ranges from 2 to 32 (Fig. 1).

The workstations (control processors) 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. Basically, call control deals with establishing connections, and operation and maintenance deals with configuration management, billing and such. Both the *cc* and *o&m* software consist of several applications which implement many concurrently operating processes.

Every workstation runs one Erlang node, i.e., a program to execute Erlang byte code implementing several thousand concurrent Erlang processes. The critical data of these processes are replicated and present on at least two nodes in the system. If a workstation breaks down, a new Erlang node is started on the workstation it is paired with and, depending on the functionality of the broken node, either the *cc* or the *o&m* applications are started.

A distributed resource locker is used 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 it is easiest to move, although this is not without consequences. Every *o&m* application may access several critical resources, and while this is done 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 need 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 this time an exclusive one. The purpose of this lock, therefore, is to enable guarantees to be given to processes about when they can safely move applications.

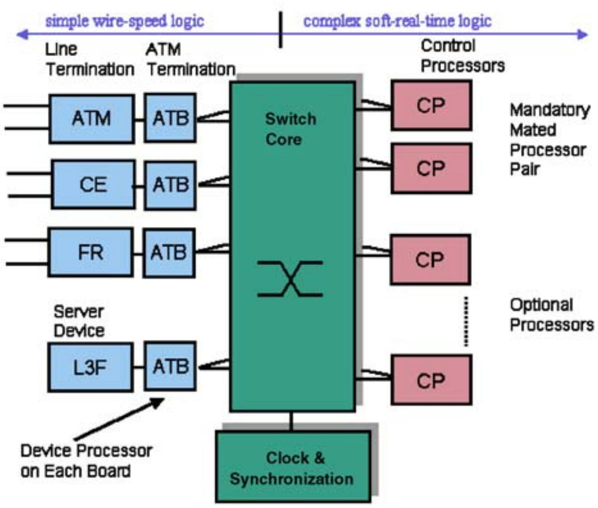

Fig. 1. AXD 301 hardware architecture

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 such. In the control software for the AXD, about 80% 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 message 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 the 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 which allows any process to 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 non-blocking and always possible; 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 many 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 and used during development and testing. Altogether this makes a server a rather large piece of software, and since all these servers have the same structure, there are considerable advantages in providing a *generic server* implementation. This generic server has all the features of the server, apart from the specific computation of reply message and new state. Simply put, by providing the above functions (`compute_answer` and `compute_new_state`), a fully functional server is specified with all the necessary features for production code.

Reality is a bit more complicated, but not much more: when starting a server one provides the name of the 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. A `handle_call` function 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 passed on 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 abovementioned functions for initialisation and call handling.

Client processes use a uniform way of communicating with the server, enforced by embedding the communication in a `gen_server:call` function call. This call causes the client to be suspended as long as the server has not replied to the message. The specific function call 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 due to 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 realised 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, referred to here 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.

4 The resource locker algorithm

The above sections have described the AXD 301 locker and the Erlang software components. We were interested in using this as a case study to validate our approach to verification of Erlang code. However, the actual implementation is overly complex for this purpose and therefore we re-implemented a small portion of the code, making appropriate simplifications where necessary.

Several prototypes have been developed and verified. In these prototypes, the resource locking process is implemented as a server process (called the “locker” in this paper). An arbitrary number of client processes can request and release resources by communicating with this server process.

In the first prototype, the locker provides access to one resource for an arbitrary number of clients. A second prototype [2] includes an arbitrary number of resources with exclusive access to them, i.e., two clients cannot get access to the same resource at the same time. In this section, we show code fragments of the third prototype [3], which supports exclusive and shared access to the resources. Some remarks about the first and second prototypes are made where they might be of interest.

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 supervision tree for only the locker and a number of clients (Fig. 2).

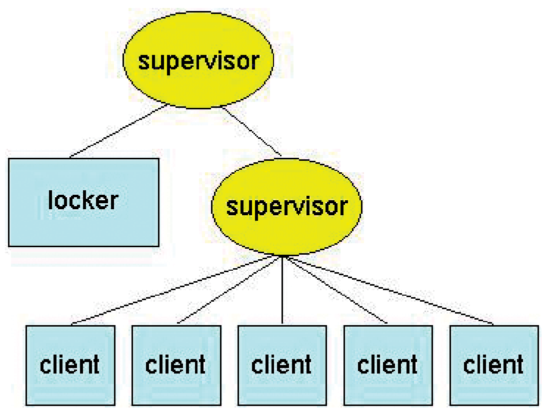

Fig. 2. Supervision tree for locker and clients

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 supervision tree and makes all processes run.

It is important to realise 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 supervision tree with a locker and two clients, one client repeatedly requesting shared access to resource `a`, the other repeatedly requesting exclusive access to resources `a` and `b`.

The locker is implemented as a callback module for the generic server. In the following subsections, we present parts of the actual implementation of the client and locker and 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 Implementing the client

The client process is implemented in a simple module. We can do this since we have abstracted away from all evaluations in clients which do not directly relate to requesting, obtaining and releasing the resources. The generic server *call* mechanism is used to communicate with the locker. It is a synchronous communication implemented by means of Erlang's asynchronous primitives.

```
-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 away from that. The variable `Type` represents the type of access a client is requesting on the list of resources. This can be either `shared` or `exclusive`, and `Resources` is bound to a list of resources which the client wants access to.

[1] The code is available at `http://www.cs.kent.ac.uk/~cb47/`

4.2 Implementing 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 having shared access to the resource, and `pending` containing a list of pending processes waiting for either shared or exclusive access.

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 which 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 optimise 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.

The server process continuously accesses its message queue, and whenever a call to the server has been made, the corresponding message will eventually arrive at the head of the queue. Then the `handle_call` function in the locker module is called. For a request message, 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 requesting exclusive access than for the clients requesting shared access. The algorithm prescribes that clients which requested shared access to a resource but are waiting for access should be by-passed by a client which 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 `send_reply` function checks whether a list of pending clients (either requesting exclusive or shared access) can be granted access. If so, the client receives the acknowledgement 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 straightforward to implement. Therefore, here we only illustrate the more interesting aspects.

The `check_availables` function is used to determine whether a new requesting client can immediately be served. 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 starvation. To illustrate this, imagine two resources and three clients such that client 1 requests resource A, client 2 requests resource B, and client 3 requests both resources. Client 1 releases and requests resource A again, client 2 releases and requests B again. If this continues repeatedly, client 3 will wait forever to get access, i.e., client 3 will starve.

	A	B	
access pending	1		client 1 requests and gets access to A
access pending	1	2	client 2 requests and gets access to B
access pending	1 3	2 3	client 3 requests access to A and B and is put in the pending list
access pending	 3	2 3	client 1 releases
access pending	1 3	2 3	client 1 requests and gets access to A again
access pending	1 3	 3	client 2 releases
access pending	1 3	2 3	client 2 requests and gets access to B again
	⋮		

This scenario indicates that in general one has to pay a price for optimal resource usage, namely, a possible starvation. Therefore, the implementation checks whether a client is waiting for a certain resource. Thus in our example, clients 1 and 2 are both appended to the list of pending processes (waiting for client 3). Like the exclusive case, for shared access the resource is available if no process holds the resource exclusively and no client is waiting for access to it. Therefore, the same conclusion holds, i.e., potential starvation is a consequence of optimal resource usage.

The `add_pending` function 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, since such a client should be at the head of the pending list.

The difference between the `check_available` and `all_obtainable` functions is that in the latter the clients have already been added to the pending lists of the requested resources and therefore it should be verified that they are at the head of these lists instead of verifying 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 a process algebra

To check that certain properties hold for all possible runs of a program, we automatically translate the Erlang modules into a process algebraic specification. This approach allows us to use tools developed for analysing process algebras rather than implementing tools that work directly on Erlang code ourselves. This has a number of benefits. For example, the use of a process algebra allows us to distinguish in a formal way communication actions and computation. It also means that complex issues such as efficient state space generation are dealt with by reusing existing toolsets which have been developed and refined over a number of years.

The process algebra we used to translate Erlang to is μCRL [16]. This process algebra is particularly suited to our requirements because we can express both communication and data in it.

Several tools have been developed to support verification of μCRL specifications [11, 24]. Our approach to verification uses a model checker from the CÆSAR/ALDÉBARAN toolset [15]. In order to input the μCRL specifications into the model checker, we need to convert the specification to an appropriate input format using the state space generation tool of the μCRL toolset. We have also experimented with static analysis tools to obtain specifications that resulted in smaller state spaces after generation, for example, the *confcheck* [22] tool from the μCRL toolset, which analyses particular (confluent) internal actions in order to eliminate them. However, this did not have as big an impact as we had hoped for. We have not yet investigated how to best fine-tune the translation for optimal use of these tools.

The translation from Erlang to μCRL is performed in two steps. First, we apply a source-to-source transformation on the level of Erlang, resulting in Erlang code that exhibits the same execution behaviour to an observer as the original code, but optimised for verification. Second, we translate the collection of Erlang modules into a μCRL specification. The advantage of having an intermediate Erlang format is that programmers can easily understand the more involved code transformations and therefore are better able to understand the smaller translation step to μCRL notation, and when some syntactic sugar is translated to more primitive operators, the step to μCRL is easier to implement. Moreover, the intermediate code can be input for other verification tools for Erlang (e.g., [4]).

5.1 Erlang-to-Erlang transformation

The source-to-source transformation of the Erlang modules contains many steps, and we mention only the more relevant ones, skipping trivial steps like removing the debug statements in the code.

Erlang supports higher-order functions, but μCRL does not. Luckily, most of the Erlang code in the switch uses only a few predefined higher-order functions, like `map`, `foldl`, `foldr`, etc. Thus we wrote a source-to-source translator to replace function call occurrences like

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

by a call to a new Erlang 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 can remove all calls to the `map` function from the Erlang code. Other higher-order functions are dealt with in a similar manner.

Because of a possible infinite state space, we avoid dynamic process creation in μCRL. Therefore, we generate the μCRL specification for a certain configuration in which the processes are fixed from the beginning. From the Erlang code in which the processes are generated dynamically, we obtain our specification by evaluating the supervision tree structure for the given configuration parameters.

In Sect. 4, we explained how to start the locker process. Evaluating the same expression, e.g., `locker_sup:start([{[a],shared},{[a,b],exclusive}])`, in our tool instead of in the Erlang runtime system results in one initial Erlang process in which all leaves in the supervision tree are started sequentially. This special process is later translated in the initialisation clause of the μCRL specification.

With some minor tricks the purely functional part of the Erlang code is rather easily translated into a term rewriting system on data, as is necessary in a μCRL model. Communication in Erlang is translated into communicating actions in μCRL, as described in the next section.

5.2 Erlang-to-μCRL transformation

Given the Erlang modules which are transformed as described above, the next step is to automatically generate a μCRL specification. Erlang is dynamically typed, whereas μCRL is strongly typed. Since we try to keep the specification in μCRL as close to the Erlang code as possible, we construct in μCRL an *ErlangTerm* data type in which all Erlang data types are embedded. All side-effect-free functions are added as a term rewriting system over this *ErlangTerm* data type. A standard transformation is used to translate Erlang statements into the term rewriting formalism. In addition, we have to define an equivalence relation on data types, which is rather involved. For instance, with only 14 different atoms and 7 data constructors, 440 equations are reserved for comparing data types, roughly two thirds of the whole specification.

With respect to the non-computation part, we benefit from the fact that the Erlang-to-Erlang transformation was generated for a specific configuration and contains all information on which processes are started. This allows us to define the initial configuration in the μCRL specification.

Manipulation of data in this process algebra is performed in a purely functional fashion, i.e., there are functions defined on the data that result in manipulated data but no communication can be incorporated in this computation part. Processes describe the communication pattern and the computations on the data; unlike with Erlang, these two parts are clearly separated, in the sense that no communication takes place in a computation. As a consequence, some code needs to be rewritten to be translated. To clarify the latter, in Erlang one can write a call to the `send_reply` function (see page 210 of this article), which results in a tuple. Part of that tuple is used in the next call to `send_reply`. Here we have to lift the communication to the same level of the `handle_call` function which is calling `send_reply`, i.e., not nested inside a computation. From an Erlang point of view, it would look like adding extra communication, where the last thing the `send_reply` function does is send a value to the process that has called this function.[2]

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

In our tool, this translation of function with nested communication is performed directly from Erlang to μCRL without the above intermediate Erlang code, which is only given to explain the translation. One could say that we implemented the well-known notion of a call stack by means of communication.

All functions which contain communication coincide with the notion of a process in μCRL. Certain restrictions with respect to these μCRL processes have to be taken into account; there is no pattern matching on data parameters of a process. Thus several clauses of the same Erlang function have to be translated in one μCRL process by explicit encoding of pattern matching by using the μCRL if-then-else construct (denoted by '`then <| if |> else`') and by calls to newly introduced processes.

The synchronous calls of the generic server can be translated directly in a synchronising 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 synchronisation between another process and the buffer of the server and synchronisation 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. Moreover, the synchronous calls of the generic server are implemented in Erlang by means of asynchronous primitives. Therefore, for every generic server process, we implement a buffer process in μCRL for both synchronous and asynchronous communication. We use the knowledge about the generic server component to implement this buffer: the generic server uses a *fifo* buffer structure. This is in contrast to an arbitrary Erlang process, where a message can be read from the buffer in any order. For illustration purposes, a simplified version of this buffer in μCRL is given below.

```
comm

gen_server_call | gscall = buffercall
gshcall | handle_call = call
gen_server_reply | gen_server_replied = reply

proc
    Server_Buffer(Self: ErlangTerm,
                  Messages: GSBuffer) =
     (bufferfull(Self).
      gshcall(Self,call_term(Messages),
              call_pid(Messages)).
```

[2] Assume a special tag for the Erlang receive to make sure the right message is read from the queue.

```
     Server_Buffer(Self,rmhead(Messages)))
    <| maxbuffer(Messages) |>
    (sum(Msg: ErlangTerm,
       sum(From: ErlangTerm,
           gscall(Self, Msg, From).
           Server_Buffer(Self,
          addcall(Msg,From,Messages))))+
     (gshcall(Self,call_term(Messages),
              call_pid(Messages)).
      Server_Buffer(Self,rmhead(Messages))))
```

The buffer associated with each process is parameterised 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.

The different clauses of the server's `handle_call` function are combined in one μCRL loop, using the state mentioned in the arguments of `handle_call` as the state of the loop. The unique process identifiers used in Erlang are integrated as an argument (`Self`) of all process calls and instantiated by the first call in the initial part.

For example, the Erlang code presented in Sect. 4.2 for the handling of a `request` message by the locker process is translated to μCRL as shown below.[3]

The `locker_serverloop` process synchronises with the buffer in the `handle_call` action, which has as arguments the identifier of the process, the message sent by the client process and the process identifier of the client. Then the availability of the resources is checked in the `locker_check_availables` function, which is the translation of `check_availables(Resources,Type,Locks)` in Erlang. Note that the pattern matching in Erlang is translated by means of selection functions that extract the `first`, `second`, etc. element of a tuple. If the resources are available, the client receives an `ok`, and the `locker_serverloop` is called with an update of the state which reflects that the resources are now being used by the client (function `locker_map_claim_lock`).

Otherwise, the `locker_serverloop` is called with a different update of the state to reflect that the client is waiting for the resources to be released. This is done slightly differently for shared access than for exclusive access, as explained in Sect. 4.2. Note that no message is sent to the client in this case.

```
locker_serverloop(Self: ErlangTerm,
                  State: ErlangTerm) =
sum(Client: ErlangTerm,
sum(Resources: ErlangTerm,
sum(Type: ErlangTerm,
  handle_call(Self,
    tuple(request,tuple(Resources,tuplenil(Type),
                        Client))).
  (gen_server_reply(Client,ok,Self).
   locker_serverloop(Self,
     tuple(locker_map_claim_lock(
         first(State),Resources,Client,Type),
       tuple(second(State),tuplenil(third(State)))))

  <| eq(equal(locker_check_availables(Resources,
                 Type,first(State)),true),true) |>

  (locker_serverloop(Self,
      tuple(locker_map_add_pending(first(State),
              Resources,Client,Type),
        tuple(append(second(State),cons(Client,nil),
           tuplenil(third(State))))))

   <| eq(equal(Type,exclusive),true) |>
   (locker_serverloop(Self,
      tuple(locker_map_add_pending(first(State),
              Resources,Client,Type),
        tuple(second(State),
          tuplenil(append(third(State),
                  cons(Client,nil))))))
         <| eq(equal(Type,shared),true) |>
         delta)

    <| eq(equal(locker_check_availables(Resources,
                 Type,first(State)),false),true) |>
    delta))
```

The `delta` mentioned in the specification is a special symbol for deadlock. These possible deadlocks are introduced by the automatic translation due to the difference between Erlang and μCRL. If the Erlang program is type safe, i.e., no runtime-type error occurs, then these deltas will never cause a deadlock in the μCRL process. However, a runtime-type error, and hence a crash of the Erlang process, would result in a deadlock of the μCRL process.

Some matching constructs are part of a pure computation part in Erlang. In a translation of such a match, we cannot simply include a `delta`. In those cases, we add before the computation an assertion that evaluates to `assertion(true)` or `assertion(false)`. If the latter appears in the state space, this corresponds to a situation in which the Erlang program would have crashed on an impossible pattern matching, and we obtain for free a path from the initial state to the location where this happens. We also provide the possibility of adding 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.

In this way, the Erlang modules are translated into a μCRL specification. By using the state space generation tool for μCRL, we obtain the full state space, in the form of a labelled 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`. This makes debugging the Erlang program easy when a se-

[3] For completeness, one of these automatically generated μCRL specifications is available at `http://www.cs.kent.ac.uk/~cb47/`

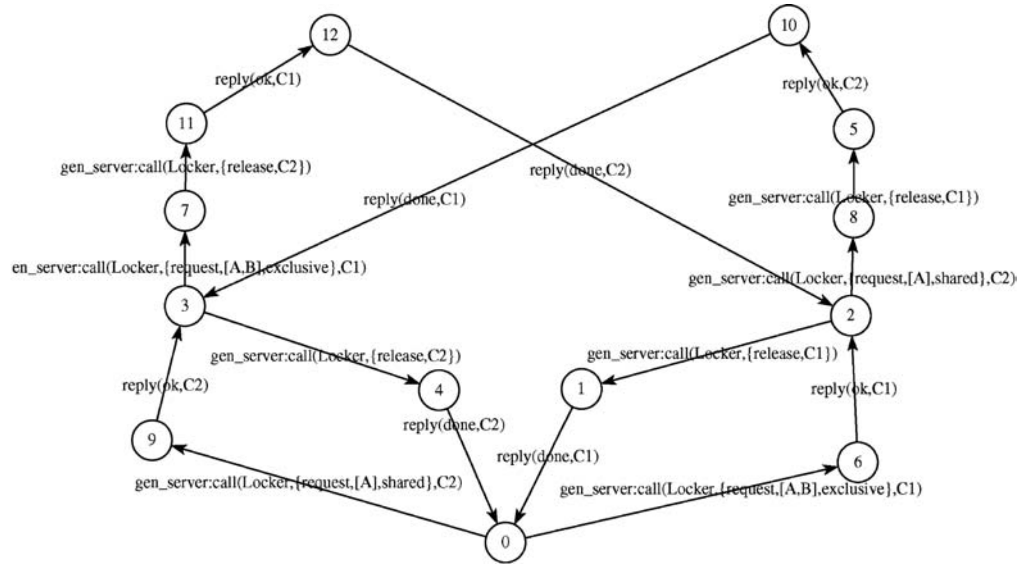

Fig. 3. Example of a small LTS

quence in the state space is presented as counter-example to a certain property. For that reason, the syntactically slightly different data structures in μCRL are translated back to Erlang data structures in the LTS.

Figure 3 displays an LTS for the scenario mentioned in Sect. 4, where a supervision tree was started 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`. In order to be able to show it here, all the communication with the buffer has been hidden and the LTS reduced by using suitable tools from the Cæsar/Aldébaran toolset.

Once we have obtained the state space, this toolset is used for verifying properties, as described in the next section.

6 Checking properties with a model checker

We have used the Cæsar/Aldébaran toolset to verify properties. This toolset provides a number of tools including an interactive graphical simulator, a tool for visualization of labelled transition systems (LTSs), several tools for computing bisimulations (minimisations and comparisons) and a model checker. 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 [9] is a formal verification technique which performs automatic checking of properties against finite state specifications. The major advantages of model checking are that it is an automatic technique and when the model of the system fails to satisfy a desired property, the model checker always produces a counter-example. These faulty traces provide priceless insight into understanding the real reason for the failure as well as important clues for fixing the problem.

The properties one wishes to check are formalised in an appropriate logic, and the specification is written here as an LTS. As mentioned previously, our specifications in μCRL are translated into LTSs, which are used as the model against which properties are checked.

The logic used to formalise properties is the regular alternation-free μ-calculus, which is a fragment of the modal μ-calculus [12, 20], 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.

Several safety and liveness properties have successfully been verified on the three prototypes of the locker. 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 to prove.

The use of regular alternation-free μ-calculus to express these properties allowed a sufficiently high level of abstraction, which meant we could reuse the expression of the properties in each of the different prototypes with minimal changes. As previously, we illustrate the process for the third prototype.

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. This property is crucial in the AXD 301 locker since otherwise when the process that wants to move an application has exclusive access to it, another process may get access to the application and perform critical operations at the same time.

To simplify checking this, we add two actions, `use` and `free`, to the Erlang code which are automatically translated into the μCRL specification.[4] As soon as a client process enters its critical section, the `use` action is applied to 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.[5] 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:

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

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 for the third prototype is formalised by

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

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. Note that the CÆSAR/ALDÉBARAN toolset allows us to use regular expressions over strings together with standard μ-calculus formulas.

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 30,000 states. Building an LTS for this example takes roughly 13 min, while checking the mutual exclusion property takes only 9 s. A bigger state space of 1 million states requires 1 h to be built and 4 min 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 are dealing with data, and a lot of computation is done in between the 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 `check_available` function is wrongly implemented such that when a client requests a resource, there is no check on whether 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 $\mathit{MUTEX}(\mathrm{A})$, the model checker returns **false** and the counter-example as shown in Fig. 4.

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 use only a small number of clients and resources, this already illustrates the substantive behaviour. As 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. In our case study, there are at most 32 Erlang nodes and at most 16 lockers, all of which have only a small number of resources (applications) to manage. We have checked the properties for scenarios with at most five clients to develop our methodology. Later we plan to scale up this approach once we have determined the optimal strategies.

6.2 Non-starvation

Starvation occurs when a client that has requested access to resources never receives permission from the locker to

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

[5] This free action is non-synchronizing and therefore can assume the role of the *release* message, but it also contains the resources which are released.

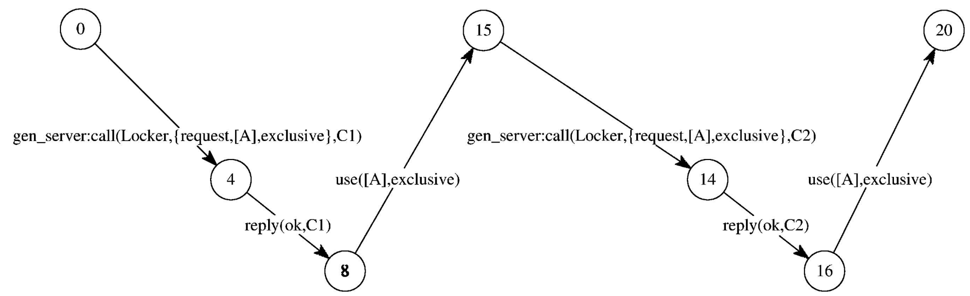

Fig. 4. Mutex counter-example

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 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 takeover process). In that setting, the starvation of clients requesting shared access cannot occur, as we prove below. The reason is the synchronised 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 then is an acknowledgement sent to the releasing client.

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

Because the algorithm contains a certain form of starvation, the property one wants to check for non-starvation must 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.

6.2.1 Non-starvation for exclusive access

Proving that there is no starvation for the clients requesting exclusive access to the resources turns out to be tricky. This is because there are traces in the LTS which do not correspond to a fair run of the Erlang program.

The Erlang runtime 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. Then client 3 requests B, obtains access to it, releases the resource and requests it again. Figure 5 shows a part of the LTS where there is a clear starvation situation for client 2, viz., infinitely often traversing the cycle that client 3 is responsible for $(4 \to 23 \to 10 \to 24 \to 4 \to \dots)$.

The above scenario, however, does not reflect the real execution of the program since the Erlang runtime system will eventually 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 evident that client 2 has the possibility of accessing 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 were shown with resource A instead of B mentioned, then this would indicate a real starvation.

One could think of a number of solutions to the problem of cycles in the LTS which do not correspond to fair infinite computations in the Erlang program. For example, one could explicitly model the Erlang runtime scheduler. However, modelling the scheduler is a rather complex solution which would significantly increase the size of the LTS. 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 occurred 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. To do that, we need to characterise the unrealistic cycles. An unrealistic cycle corresponds to unfair execution of a number of clients which are *independent* of the client one wants to prove non-starvation for.

In our specific case, a client depends on another client when the intersection of the sets of resources they request is non-empty. Given that one is interested in proving non-starvation of a certain client, computing the clients which 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 guaran-

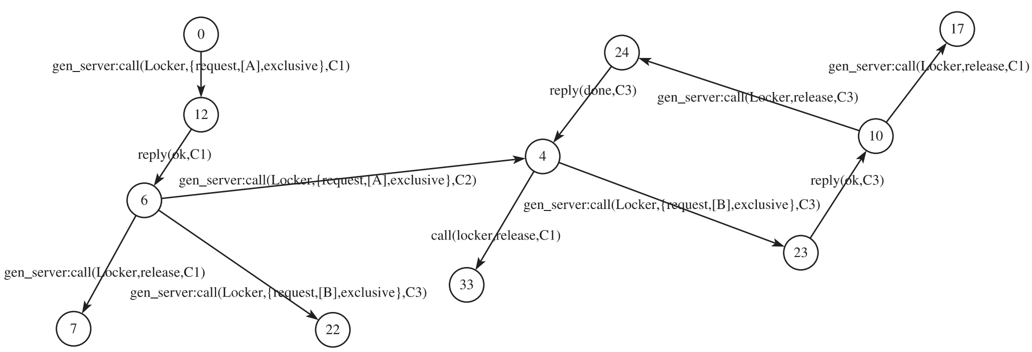

Fig. 5. Unreal starvation of client 2

teed occurrence 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 which 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 of 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 [21], 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 and obtained a model in which pure τ cycles no longer occur. Thus we removed all unfair cycles.

With respect to 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 which 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 request 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.

6.2.2 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 $NON-STARVATION(C)$ for every client C in the configuration. As such, the verification is performed successfully.

7 Automation of verification

In the previous sections, we described the automatic translation of Erlang to μCRL and showed how the properties of mutual exclusion and non-starvation are verified. In this section, we explain how the verification of properties can be automated.

Automation is achieved by using the Script Verification Language (SVL) from the CÆSAR/ALDÉBARAN toolset. SVL allows us to simplify and automate the verification by means of high-level operators on the LTSs, for instance, minimisation, label-hiding, label-renaming and model-checking operators, and several methods of verification. Moreover, Bourne shell commands can be invoked within an SVL description; thus the tool to translate Erlang to μCRL, *etomcrl*, and the μCRL tools to build the LTS can be called within the script.

This script is called with the Erlang term which should start the application in a certain configuration. The supervision design contains all information for the Erlang loading system to automatically locate the necessary

```
%echo no starvation process 0
verify "properties/non_starvation.mcl" in
    observational reduction with aldebaran of
    rename ".*ok.*" -> "ok",
           ".*request.*" -> "request",
           ".*release.*" -> "release" in
    hide ".*pid(2).*", ".*pid(1).*" in
    "locker.bcg";

%echo no starvation process 1
verify "properties/non_starvation.mcl" in
    observational reduction with aldebaran of
    rename ".*ok.*" -> "ok",
           ".*request.*" -> "request",
           ".*release.*" -> "release" in
    rename ".*pid(2).*" -> "other" in
    hide ".*pid(0).*"  in
    "locker.bcg";

%echo no starvation process 2
verify "properties/non_starvation.mcl" in
    observational reduction with aldebaran of
    rename ".*ok.*" -> "ok",
           ".*request.*" -> "request",
           ".*release.*" -> "release" in
    rename ".*pid(1).*" -> "other" in
    hide ".*pid(0).*"  in
    "locker.bcg";
```

Fig. 6. SVL script for verification of non-starvation for certain configurations

modules. Thus this one Erlang term suffices for automatically generating a μCRL specification. The script contains further instructions to use the state space generation tool in order to build the LTS from the μCRL specification. The same script is used to verify the properties for this LTS with the model-checking tool. The outcome of the model checker is either *true* or *false*, and in the latter case a counter-example is saved. The script ensures that this counter-example is stored for later inspection. In this way, given the simple scripts, our tool automatically verifies properties of real Erlang applications.

However, when verifying the non-starvation property, we perform several manipulations of the LTS, reduce the LTS with respect to observational bi-simulation and only then verify the property. This is expressible in a script, but at present our tool cannot generate such a script automatically (see Fig. 6).

We accept that a certain ingenuity is necessary to create both property and script, but given the fact that we want to use our tool in an iterative development process, we want to minimise the number of times that these properties have to be updated because of a small change in the configuration or application.

In the following two subsections, we show that verification is parametric with respect to a given configuration and with respect to the application with certain restrictions.

7.1 Independence of configuration

The properties given in Sect. 6 depend on the actual names of the resources.

For example, for the mutual exclusive property a *BETWEEN* clause is added for every resource available in the system. We solved this by using only one property, specifically,

$$\begin{aligned}&\mathit{MUTEX} = \\ &\quad [-^* \,.\, '\mathit{use}(\mathit{exclusive})' \,.\, (\neg' \mathit{free}(\mathit{exclusive})')^* \,. \\ &\quad\; '\mathit{use}(.^*)']\mathit{false}\end{aligned}$$

and rename the appropriate actions in the state space. Thus, given that r is a resource in the system, we rename the labels "$\mathit{use}(r, \mathit{exclusive})$" to "$\mathit{use}(\mathit{exclusive})$", etc. After this renaming, it suffices to check the above property to prove mutual exclusion for resource r. This is repeated for all resources. The script that performs the renaming and checking is generated from the configuration. Of course, renaming and model checking several times are in general more expensive than only performing the model checking with a more complicated property. We could also automatically generate this more complicated property from the configuration, but we would then need to describe the property as a kind of template with an unbounded number of these *BETWEEN* clauses. The motivation for our approach is that the properties should be easy to read and understand and that we want to stick to a standard logic. We sacrifice efficiency of verification for understandability of the property.

For verification of non-starvation, we go one step further. Here the property depends on the client's process identifier. We limit ourselves to one property here as well, namely,

$$\begin{aligned}&\mathit{NONSTARVATION} = \\ &\quad [-^* \,.\, '\mathit{request}']\mu X.(\langle - \rangle \mathit{true} \wedge [\neg' \mathit{ok}']X)\end{aligned}$$

We have to build a graph of processes that depend on a common resource. From the configuration we obtain information on which resources a client needs. By storing the process identifier of the client together with the resources this client requests in the vertex of a graph and by adding an edge whenever two nodes have a resource in common, it is easy to obtain all processes that depend on a certain client, i.e., all those in the same closely connected component. This is straightforward to implement, but realising that this algorithm is what we need for verification of non-starvation is not part of the automation.

For every client we repeat the same steps in a script. We hide all processes (i.e., rename to τ) that are not dependent on this process and rename all actions of dependent processes to a constant *other*. In this way, only the *request*, *ok*, and *release* messages of the process for which we want to verify non-starvation remain as labels in the LTS. The above property can therefore be used for our verification purposes. In Fig. 6, such a script is presented for a configuration with three clients of which two depend on each other, in particular, the client process with process identifier 0 requests exclusive access to resource A, the client process with process identifier 1 requests exclusive access to resource B, and the client process with process identifier 2 also requests exclusive access to resource B.

Since non-starvation has to be checked for all clients separately, there is not the same decrease in performance as for the mutual exclusion property. In the most optimal case, one renaming per group of dependent processes could suffice. Again we motivate our choice by claiming that the property in combination with the script is easier to understand than the more involved properties that we would get if we considered a whole group at once.

7.2 Independence of development iteration

Using the technique of creating scripts as described above, we obtain a situation in which the mechanical steps of performing a verification are independent of the configuration. This is useful when an application has reached a certain point in its development and is verified for a number of configurations.

However, the application will be modified and features added. As long as those modifications do not influence the syntax of the messages that are communicated, the verification approach is not affected. Changes in the communication, though, normally require investigation into whether the properties and scripts have to be changed.

In our case study, the syntax of the messages is only slightly modified through the iterations on the code. Let us consider the mutual exclusion property. In the first iteration of the locker algorithm, there is only one resource in the system; therefore, the mutual exclusion property in this case was defined as:

$$MUTEX = [-^* \,.\, 'use' \,.\, (\neg 'free')^* \,.\, 'use']false$$

The same property holds for the second iteration of the code where there are several resources but with only exclusive access to them. For every resource r, the actions $use(r)$ are renamed to use.

However, in the case of the third iteration, where resources may also be shared by different clients, the above property is not sufficient. Here we only want to prove mutual exclusion for exclusive access, but we need to take into account that the resources may also be obtained for shared access. Thus the mutual exclusion property is the one shown in the previous subsection. Note that the property is a slight modification of the property presented here, where instead of the first use we write $use(exclusive)$, instead of $free$ we write $free(exclusive)$ and instead of the second use we write $use(.*)$, which stands for both exclusive and shared access to the resource. In other words, not much need be changed in the properties to employ the automatic verification from one iteration of the code to the next.

8 Conclusions

In this paper, we discuss an approach to developing verified Erlang code. This paper is an extended version of the contribution to FMICS [2], where an earlier iteration of the resource manager is described. In this paper, we focus on the iteration of the resource manager as described by us for FME [3], with two types of access to resources. Compared to the FME contribution, we describe the construction of the tool in greater detail and focus on the support for the development process.

As noted earlier, there are a number of approaches to verifying code. For example, a formal development process might start with a formal specification and use verified refinement steps to produce code compliant with the original specification. The development process our work fits into is different on a number of fronts. First, we are working in a context of an established process which makes full use of software libraries which have been extensively tested. Second, we wish our verification to sit alongside the standard coding and testing of the Erlang components and to use verification to check key properties of the code.

To this extent, our approach consists of the following steps. The Erlang code for a component is automatically translated to a process algebraic specification written in μCRL. We then generate a labelled transition system (LTS) from this μCRL specification by using components of the μCRL toolset. The properties of interest are then written in the logic of the model checker we use; here we use the regular alternation-free μ-calculus to express non-starvation and mutual exclusion. The labelled transition system is then checked against this property using the CÆSAR/ALDÉBARAN toolset. For some properties it is necessary to transform the LTS (e.g., using hiding for non-starvation) so that we can model check with a simpler formulation of the property of interest (e.g., one without alternating fixed points).

The case study we discussed in this paper was drawn from a critical part of the AXD 301 software consisting of about 250 lines of Erlang code, which implements a resource-locking problem for which we prove properties such as mutual exclusion and non-starvation. Although we re-implemented the software to substantially simplify the code, the principles underlying the code we used are exactly the same as those in the actual switch code. In the code of the resource manager in the AXD 301, both the resource manager and a leader election protocol are combined. We separated these two concepts and concentrated on a clean implementation of both. In this paper, we have described the resource locker code. We used the same design principles, coding style and libraries as were used in the production code. Even the names of variables and functions are the same in our implementation as in the original software.

Our approach has advantages and disadvantages. The ability to automate many aspects of the process is one of the key advantages; however, we currently have to fix the number of clients and resources per verification. Tackling this issue, and determining how to verify properties for arbitrary numbers of clients and resources without a crippling performance overload, is part of our ongoing work. Relevant to this might be the use of theorem proving since the Erlang theorem prover [14] 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 [5]. However, such a proof has to be provided manually, and this contrasts with the ability to automate, which is an advantage of model checking. However, with a theorem prover one can reason about sets of configurations at once and not fix the number of clients and resources per attempt. Integrating the two approaches might offer some combined benefits.

The translation of Erlang into μCRL which we discussed above is performed automatically and is sufficiently robust to deal with a large enough part of the language to make it applicable to serious examples. Although the tool which calculates the state spaces for μCRL models [11] is advanced and stable, it still takes on the order of a few minutes up to several hours to generate a state space. Once the model is generated, model checking is relatively quick: with the CÆSAR/ALDÉBARAN toolset it takes only a few seconds up to a few minutes. This comparative difference is partly due to the computation on the complex data structures we have in our algorithm.

Some further optimisations could be envisaged. In some cases, it is unnecessary to generate the whole state space, for example when the property of interest does not hold. 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 a collaborative effort between CWI and Aachen University [8]. With such a tool, a cluster of machines can be used to quickly analyse rather large state spaces. Experiments have shown that an LTS with 20 million states would be generated in a few hours.

All of this work points to a situation where the formal verification of Erlang programs is slowly becoming practically possible, particularly for the development of new programs [2]. An experiment involving using the tool to develop the scheduler software for a video-on-demand server underwrites this [6]. Our ongoing work includes the extension of our translation tool to cover 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 of 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. Especially helpful were the tool development teams at INRIA Rhône-Alpes and CWI with their support and advice, and Lars-Åke Fredlund and Dilian Gurov from SICS with their contribution to the discussions. We thank Howard Bowman from the University of Kent for useful explanations.

References

1. Armstrong JL, Virding SR, Williams MC, Wikström C (1996) Concurrent programming in Erlang, 2nd edn. Prentice-Hall, Upper Saddle River, NJ
2. Arts T, Benac Earle C (2001) Development of a verified distributed resource locker. In: Proceedings of FMICS, Paris, July 2001
3. Arts T, Benac Earle C, Derrick J (2002) Verifying Erlang code: a resource locker case-study. In: Proceedings of the Formal Methods Europe symposium (FME02), Copenhagen, July 2002
4. Arts T, Dam M, Fredlund LÅ, Gurov D (1998) System description: verification of distributed Erlang programs. In: Proceedings of CADE'98, Lindau, July 1998. Lecture notes in artificial intelligence, vol 1421, Springer, Berlin Heidelberg New York, pp 38–42
5. Arts T, Noll T (2000) Verifying generic Erlang client-server implementations. In: Proceedings of IFL2000, Aachen, Germany, September 2000. Lecture notes in computer science, vol 2011, Springer, Berlin Heidelberg New York, pp 37–53
6. Arts T, Sánchez Penas JJ (2002) Global schedular properties derived from local restrictions. In: Proceedings of the ACM SIGPLAN Erlang workshop, Pittsburgh, October 2002
7. Blau S, Rooth J (1998) AXD 301 – a new generation ATM switching system. Ericsson Rev 1:10–17
8. Bollig B, Leucker M, Weber M (2001) Local parallel model checking for the alternation free μ–Calculus. In: Proceedings of the 9th international SPIN workshop on model checking software, Grenoble, France, April 2002. Lecture notes in computer science, vol 2318, Springer, Berlin Heidelberg New York, pp 128–147
9. Clarke EM, Grumberg O, Peled D (1999) Model checking. MIT Press, Cambridge, MA
10. Corbett J, Dwyer M, Hatcliff L (2000) Bandera: a source-level interface for model checking Java programs. In: Proceedings of the teaching and research demos at ICSE'00, Limerick, Ireland, 4–11 June 2000
11. CWI (1999) A language and tool set to study communicating processes with data. Available at: `http://www.cwi.nl/~mcrl`
12. Emerson EA, Lei CL (1986) Efficient model checking in fragments of the propositional Mu-Calculus, In: Proceedings of the 1st LICS, Cambridge, MA, June 1986, pp 267–278
13. Open Source Erlang (1999) `http://www.erlang.org`
14. Fredlund L-Å et al (2003) A tool for verifying software written in Erlang, Int J Software Tools Technol Trans (in press) (`http://link.springer.de/`)
15. Fernandez JC, Garavel H, Kerbrat A, Mateescu R, Mounier L, Sighireau M (1996) CADP (CÆSAR/ALDÉBARAN development package): a protocol validation and verification toolbox. In: Proceedings of CAV, New Brunswick, NJ, July 1996. Lecture notes in computer science, vol 1102, Springer, Berlin Heidelberg New York, pp 437–440
16. Groote JF (1997) The syntax and semantics of timed μCRL. Technical report SEN-R9709, CWI, June 1997. Available at: `http://www.cwi.nl`
17. Havelund K, Pressburger T (2000) Model checking JAVA programs using JAVA PathFinder. Int J Software Tools Technol Trans 2(4):366–381
18. Holzmann G (1991) The design and validation of computer protocols. Prentice-Hall, Upper Saddle River, NJ
19. Huch F (1999) Verification of Erlang programs using abstract interpretation and model checking. In: Proceedings of ICFP'99, Paris, September 1999
20. Kozen D (1983) Results on the propositional μ-calculus. J Theor Comput Sci 27:333–354
21. Milner R (1980) A calculus of communicating systems. Springer, Berlin Heidelberg New York
22. Van de Pol JC (2001) A prover for the μCRL toolset with applications, version 0.1. Technical report SEN-R0106, CWI, Amsterdam. Available at: `http://www.cwi.nl`
23. Wells D (1999) Extreme programming: a gentle introduction. Available at: `http://www. extremeprogramming.org`
24. Wouters AG (2001) Manual for the μCRL tool set, version 2.8.2. Technical report SEN-R0130, CWI, Amsterdam