Development of a Verified Erlang Program for Resource Locking

Thomas Arts and Clara Benac Earle

Ericsson, Computer Science Laboratory
Box 1505, 125 25 Älvsjö, Sweden
E-mail: {thomas,clara}@cslab.ericsson.se

Abstract. We have designed a tool to simplify model checking of Erlang programs by translating Erlang into a process algebra with data, called μCRL. As a case-study for this tool we focused on a simplified locker implementation after the locker that is present in the control software of the AXD 301 switch. The translation algorithm has been developed to handle this production-like code. We use the tools accompanying μCRL to generate the transition systems from the specification generated by our tool. With the CÆSAR/ALDÉBARAN tool set, we verified properties for our case-study.

1 Introduction

Within Ericsson the functional programming language Erlang [1] is used for the development of concurrent/distributed safety critical software. Faced with the task of creating support for the development of formally verified Erlang programs, as a subtask we have built a tool to enable the use of model checking for such programs. The tool is aimed to be accessible for Erlang programmers without forcing them to learn an extra language (specific for the model checking tool that is used).

Using model checking for the formal verification of software is by now a well known field of research. Basically there are two branches, either one uses a specification language in combination with a model checker to obtain a correct specification that is used to write an implementation in a programming language, or one takes the program code as a starting point and abstracts from 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, the use of model checking can be seen as a very accurate debugging 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 [14]. 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. In case one uses UML as specification language and Java or C as implementation language, one might need more effort (apart from the fact that model checking UML specifications is still an unsettled topic).

Also with respect to the second approach there are many examples, among which PathFinder [13] and Bandera [6] starting from Java code. There exists even an earlier attempt to use model checking on Erlang code by Huch [15]. Our approach could be added to this list, probably with the difference that we use the knowledge of the occurring design patterns used in the Erlang code to obtain smaller state spaces (cf. [2]). We follow a similar approach to the translation of Java into Promela, checked by SPIN [13]; however, we translate Erlang into μCRL [12] and model check by using CÆSAR/ALDÉBARAN [9]. Compared to Huch's approach we focus much more on the data part and do not abstract *case* statements by non-deterministic choices, but really check the data involved. For that reason we can check mutual exclusion and absence of deadlock for a small locker program that 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.

One of the main goals of our approach is to be able to deal with Erlang code that is written according to the design principles as advocated within Ericsson. Our starting point was a distributed locker algorithm as is running in Ericsson's AXD 301 ATM switch [4]. We started re-designing this locker in such a way that formal verification guides the development. In this paper we illustrate our ideas with one of the first locker prototypes in this development process.

In Section 2 we describe the locker algorithm that we consider in this paper. We show in Section 3 how this locker is implemented in Erlang, using the generic server and supervision tree design principles.

The Erlang modules can automatically be translated into a μCRL specification and in Section 4 we describe our contribution in the form of this translation tool. Verification of the μCRL specification for the classical properties: no deadlock, mutual exclusion and no starvation, is described in Section 5. In the conclusion in Section 6 we discuss the merits and shortcomings of our approach and put it in context with respect to other approaches.

2 Designing the algorithm

The case-study we have at hand in this paper is a classical locker algorithm. Several processes want access to one or more resources from a given, finite set. A locker process is playing arbiter, responding the requests for access to resources in such a way that all clients eventually get their demanded access, but no two clients get access to the same resource at the same time. The client sends one message containing all resources that are requested, waits until access is granted, accesses the resource, gives the resource free and starts asking for other resources again.

Several fault situations are easy to imagine and these should guide us towards solutions for the most rudimentary problems. We describe the analysis of these situations as a pre-study for the actual implementation. However, with the tools we discuss later, one could find these results in an experimenting fashion: implement an idea in Erlang and obtain all possible runs of the program automatically.

Here we discuss the fault situations, using a special notation for scenarios. A scenario is a sequence of states of the locker process. A state of the locker contains a fixed set of resources and for every resource we have three 'fields': the name of the resource, the client that has access to the resource, and the list of clients that want to access the resource. As an example of this notation, we sketch a possible starvation situation. There are two resources, A and B, and three clients, 1, 2 and 3. The algorithm is such that if a demanded set of resources is available for a certain client, then this client gets access to those resources. Here, 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. A continuous operation in this way causes client 3 to be waiting for ever to get access, i.e. client 3 is starving.

	A	B
access	1	
pending		
access	1	2
pending		
access	1	2
pending	3	3
access		2
pending	3	3
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 possibility for starvation. Clearly one does not want starvation in the program, but one still may accept it in the algorithm. If one has good evidence to believe that resources are not accessed very frequently, then the above situation might be very unlikely and one might choose to loose performance for client 3 in favor of a better over-all performance.

We, however, assume that the frequency of access to the resources can be rather high and that the different clients may have overlapping demands for resources[1]. Therefore, we need to decide upon a solution to this problem. We choose to use a 'first come, first serve' strategy. A resource is only available if there is no client

[1] If client 1, 2 and 3 would all ask for the same resources, this starvation problem would not occur.

waiting for it, i.e., both access field and list of pending processes for the demanded resource is empty.

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

Thus, in this solution, a client could have to wait for its resources, even if all demanded resources are unused at the moment. Some optimizations are possible, for example to time-stamp the pending processes and give access to the resource if the first of the pending processes has not yet waited a certain amount of time. This is more involved and we do not consider this or other optimizations in this version.

The action upon a client requesting for a list of resources will be:

- Look whether all demanded resources are available. A resource is available if no other process is accessing it, and there are no processes pending for this resource.
- If all demanded resources are available, then the client is notified and is given access to all resources.
- If any of the demanded resources is unavailable, then for every demanded resource, the client is placed at then end of the list of pending processes.

The client is assumed to release all the previously demanded resources by only one release message; upon a release, the client is removed from all resources and a calculation is performed to see whether one of the other clients can get access to its demanded resources. Similar to the reasoning above, we cannot give access to just any client for which all demanded resources are available. Even for the one resource case it is clear that we need to take a 'first come, first serve' policy. Thus, with only one resource and several clients, we would give the client at the head of the pending list access to the resource. However, one could wonder what happens if there are two resources and both have one or more clients in their pending list.

	A	B
access	1	1
pending	2	3, 2

Here we need an algorithm to decide whether client 2 or 3 gets access to the resource after that client 1 releases. The possibilities we could think of boil down to the construction of one list of the pending processes where the first client in this list for which all demanded resource are available gets access to these resources (i.e., is notified and removed from the pending lists and put into the access 'field'). Several ways of constructing this combined list are:

1. Merge all list and sort them on the client identifier. This means that the client with lowest identifier has highest priority. Hence, starvation is an obvious problem. When, in the above example, client 2 is given access and client 1 requests both resources again, then by the time client 2 releases, client 1 will be granted access. Repeatedly having 1 and 2 requesting access will cause 3 to starve.
2. Append all lists (and use a small optimization by a unique append, i.e. only appending the clients that are not yet present in the list). Clearly the same starvation problem as above occurs for this solution.
3. Construct a list that contains only those heads of the pending lists that do not occur in one of the tails of a pending list.
   The reason why this can work is that we have the clients always request all the resources at once. Hence, the clients are put in the pending list in a 'sorted' manner. A situation like

	A	B
access		
pending	$2, 3$	$3, 2$

   cannot occur in this setting, since either client 2 follows client 3 in all pending lists or vice versa. There might be clients in between, but the order cannot be reversed.
4. Add a time-stamp to any incoming request and save the client information with this time-stamp. The list is now obtained by appending and sorting the time-stamps.
   An equivalent approach is to separately store the list of requesting clients and use the order in which they requested as the priority order for giving access.

We have experimented with both version 3 and 4 and present version 3 here.

3 Locker Implementation in Erlang

The ideas sketched in the previous section are now to be implemented in Erlang. Clients and locker are implemented as Erlang processes that communicate with each other by message passing. The locker is implemented as a server, following one of the generic design patterns given in the Erlang distribution [8]. This *generic server* design pattern prescribes an implementation of the locker as a so called *callback module*. The actual loop that saves the state of the server and receives messages is implemented in a standard module and whenever a message arrives, the appropriate function in the callback module is executed. These callback functions return a new state and a possible reply message, which is by the standard module part send to the caller. In this way, the generic server principle implements synchronous communication on top of Erlang's asynchronous communication primitives. For a detailed operational semantics we refer to [2].

The flow of control between clients and locker should be as follows:

- a client *requests* the locker an exclusive lock on several resources,
- if all requested resources are available, the locker gives an *ok* to the client,
- when the client has performed the necessary operations on the resources, it notifies the locker by a *release* of the locks.

The locker schedules the clients on a first-come first-served basis as explained in the previous section. Note, however, that this scheduling is relative to the resource. A client that requests a resource that is taken, may be served later than the client requesting another, free resource, after it.

The client is programmed as a very simple process, just using the generic server call principle to communicate with the locker. The `gen_server:call` function hides synchronized communication with the server. The second argument of this function contains the message that is sent to the server, which calls the `handle_call` function in the callback module. The client is suspended until `handle_call` returns a reply value, which is passed by the server to be the return value of the `gen_server:call`. For this particular client we are not interested in the actual returned value and just use it for synchronization. The `spawn_link` function is used to create a new process, in this case running the `loop` function with the arguments `Locker` and `Resources`.

```
-module(client).

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

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

The atom `critical_section` between the two synchronous calls for request and release implements the so called critical section. In a real implementation some critical code should be placed in this critical section, but we abstract from that.

To implement the locks we use a record with the following fields:

- resource: the name of the resource,
- exclusive: the client which is using the resource,
- pending: a list of clients that want to access the resource.

The Erlang program for the locker process is given by a generic server callback module that accepts the messages {`request,Resources`} and `release`.

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

-record(lock,{resource, exclusive = none, pending = []}).

init(Resources) ->
```

```
    {ok,map(fun(Resource) ->
                #lock{resource = Resource}
             end, Resources))}.
```

The `init` function returns for every resource in a given list `Resources` a record of type `lock` where the first field contains the name of the resource and the other two fields are instantiated with the (default value) empty list.

```
handle_call({request,Resources}, Client, Locks) ->
    case check_availables(Resources,Locks) of
       true ->
          {reply, ok,
              map(fun(Lock) ->
                     update_exclusive(Lock,Resources,Client)
                  end, Locks)};
       false ->
          {noreply,
              map(fun(Lock) ->
                     add_pending(Lock,Resources,Client)
                  end, Locks)}
    end;

handle_call(release, Client, Locks) ->
    NewLocks =
      map(fun(Lock) ->
              release_lock(Lock,Client)
          end, Locks),
    Locks_updated =
      send_reply(NewLocks,all_pendings(NewLocks)),
    {reply, ok, Locks_updated}.
```

The generic server automatically supports every message in a `gen_server:call` with the process identifier of the sender and a tag (a kind of time stamp to distinguish different messages from the same client). When obtaining a request, the locker stores the combination of identifier and tag as a pair in the pending list (or exclusive field). When releasing, a new tag is used for the pair (since it is a new message) and removing the pair from the list should be done by only looking at the process identifier. Note that the locker cannot remove the tag already at the moment of receiving the request of a client, since the tag is necessary for a reply, as implemented by `send_reply`. This function checks for every pending client whether its resources are available. If so, the client is notified and the locks are updated.

```
send_reply(Locks,[]) ->
  Locks;
send_reply(Locks,[Pending|Pendings]) ->
  case obtainables(Locks,Pending) of
       true ->
```

```
      gen_server:reply(Pending,ok),
      send_reply(map(fun(Lock) ->
                         promote_pending(Lock,Pending)
                      end, Locks),
                  Pendings);
    false ->
      send_reply(Locks,Pendings)
end.
```

These are the only functions that contain side effects, viz. the sending and receiving of messages. All other functions are side-effect free and easy to implement.

In addition to client and locker code we also have implemented a so called *supervision tree*, a commonly used design principle to monitor the individual processes [8]. Basically the code for the supervision tree describes a process that is started, which monitors two processes, one is the locker, the other a new supervisor process, which monitors the clients. The code describes what should happen if one of the processes crashes and is instructed to restart clients and locker processes.

All processes together can now be started with only one function call, viz. `supervisor:start`, with in the arguments the number of clients one wants to start it with and the list of resources one considers.

4 A μCRL specification

The Erlang modules described in the previous section are automatically translated into one μCRL specification. The data is directly translated from Erlang to μCRL without any abstraction. The specification is used to generate the transition system, which is used for model checking.

The translation is performed in two steps. First we apply a source-to-source transformation on the level of Erlang, resulting in Erlang code that should be executable in the same way as the original, but is optimized for verification. Second we translate the collection of Erlang modules into one μCRL specification. The advantage of having an intermediate Erlang format is that programmers can easily understand the more severe manipulations of the code and therefore are better able to understand the smaller step to μCRL notation. Moreover, the intermediate code can be input for other verification tools.

4.1 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.

We use the supervision tree structure to obtain a finite set of initial processes. We start the translator with the same arguments as that we would need to build and start the supervision tree. This allows us to bind the number of clients and resources to a certain value. For every different number we need to run a different transformation. The supervisor processes are taken away and the new initialization function only creates the processes of locker and clients. The handling of a process that crashes is left to be detected in the transition system.

We replace (a predefined set of) higher order functions like `map` by a first-order alternative, since the target specification language does not support higher order functions. Thus, a call `map(fun(X) -> f(X,Y1,...,Yn) end, Xs)` is replaced 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)].
```

In the next phase we determine all functions with side-effects, i.e., those functions that do send or receive a message or call a function doing so. This is a call-graph problem where we keep a list of side-effect free functions in the library modules. The `gen_server:call` function and `handle_call` function are typically added to the functions that contain side-effects.

The most involved operation is now to get rid of the use of return values of functions with side-effects. In μCRL a process may have side-effects, but has no return value; on the other hand, a function in μCRL has a return value, but may not contain a side-effect. In case an Erlang function (in)directly causes a side-effect, its computation part and side-effect part have to be split. For the source-to-source transformation, it suffices to make sure that all return values are matched in a variable and to provide decomposition of the data structure of this return value by means of side-effect free functions. Currently we can deal with basic data types and the compound data types lists, tuples, records and mixtures of these.

4.2 Erlang to μCRL transformation

Given the Erlang modules that are transformed as described above, we generate one μCRL specification from these modules. Erlang is dynamically typed whereas μCRL is strongly typed. Therefore, we construct in μCRL a data type *ErlangTerm* in which all Erlang data types are embedded. All side-effect free functions are added as a term rewriting system with 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. In this particular case with only 14 different atoms and 7 data constructors, 440 equations are reserved for comparing data types, roughly two third of the whole specification.

With respect to the part with side-effects, 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. The Erlang processes coincide with the μCRL processes, where a non-terminating Erlang function describes the main loop of the process in the Erlang case. However, when translating this loop, we cannot translate recursive calls to Erlang functions with side-effects in a direct way to μCRL. In μCRL computation and side-effects cannot be intermingled. The solution is found in the definition of a separate μCRL process implementing a call stack. Communication with this call stack is used to return the values of the computation.

Certain restrictions with respect to the μCRL functions have to be taken into account; there is only one function clause possible, with only sequential composition, non-deterministic choice, and an if-then-else statement for control. We translate case statements and pattern matching by using the if-then-else construct and calls to newly introduced process functions. The `handle_call` and `gen_server:call` are translated into communicating actions in μCRL. The different clauses of the `handle_call` function are combined in one μCRL loop, using the state mentioned in the arguments of `handle_call` as 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.

```
comm
     gen_server_call | handle_call = call
     gen_server_reply | returned = return

proc locker(Self: Term,Locks: Term) =
    sum(Client: Term,
        sum(Resources: Term,
            handle_call(Self,tuple(request,Resources),Client).
            (gen_server_reply(Client,ok,Self).
             locker(Self,
                    map_update_exclusive(Locks,Resources,Client))
                  <| eq(check_availables(Resources,Locks),true) |>
             locker(Self,
                    map_add_pending(Locks,Resources,Client))))) +
    sum(Client: Term,
        handle_call(Self,release,Client).
        send_reply(Self,map_release_lock(Locks,Client),
                   all_pendings(map_release_lock(Locks,Client))).
        sum(Locks2: Term,
            rcallresult(Self,Locks2).
            gen_server_reply(Client,ok,Self).
            locker(Self,Locks2)))

send_reply(Self:Term,Locks:Term,MCRLArg1:Term) =
     (wcallresult(Self,Locks)
       <| eq(equal(MCRLArg1,nil),true) |>
      (gen_server_reply(hd(MCRLArg1),ok,Self).
       send_reply(Self,
                  map_promote_pending(Locks,hd(MCRLArg1)),
                  tl(MCRLArg1))
         <| eq(obtainables(Locks,hd(MCRLArg1)),true) |>
       send_reply(Self,Locks,tl(MCRLArg1))))
```

After this automatic transformation, we can verify a specific configuration, in which the clients repeatedly request all available resources. In order to perform several verifications at once, in particular to verify all situations in which the clients repeatedly request an arbitrary (varying) subset of the resources, we modified the μCRL specification by hand. We used μCRL's possibility to express non-determinism for this. The μCRL specification is used to generate a transition system. The number of states for the generated systems depends on the configuration. We tried several configurations, up to three clients and four resources, the largest resulting in about a million states. Creating such large state spaces takes a few hours on a single processor workstation. Even though this is time consuming, improving this has not highest priority; we plan to focus on small examples in the development phase of the software. Larger examples take more time, but so does testing. The development of on-the-fly model checking and parallelization of the model checker might increase performance dramatically in a later stage.

5 Verifying the model

The three properties we want to verify for this locker are: absence of deadlock, mutual exclusion and no starvation. All are classical properties that are well studied in literature. The first is trivially shown, the second and third need the right formulation and the support of a model checker. Mutual exclusion is a safety property, whereas no starvation is a liveness property. The safety properties are easier to check than the liveness properties, as is explained later and depends on the fact that some infinite traces in the specification are excluded in a real Erlang execution because of the underlying Erlang scheduler.

5.1 Mutual Exclusion

The property for mutual exclusion should express that a resource can only be accessed by one client at the same time. In order to show this, we added two actions to the μCRL specification `use` and `free` with a resource as an argument. As soon as we enter the critical section, the `use` action is applied for all resources that the client requested. Before leaving the critical section, the resources are given free again. We use the macro

$$\textsc{Until}(a_1, a_2) = [\text{-}^*.a_1.(\neg a_2)^*.a_1]\mathit{false}$$

stating that 'on all possible paths, after an a_1 action, any other a_1 action must be preceded by an a_2 action'. The mutual exclusion property depends on the number of resources. In fact we need a different formula for any number of resources. For a system with two resources, r_1 and r_2, the mutual exclusion property is formalized by

$$\begin{aligned}\mathrm{MUTEX}(r_1, r_2) = {} & \textsc{Until}(\mathtt{use}(r_1), \mathtt{free}(r_1)) \wedge \\ & \textsc{Until}(\mathtt{use}(r_2), \mathtt{free}(r_2))\end{aligned}$$

A new version of the model checking tool within the CÆSAR/ALDÉBARAN toolset [9] is under construction and with this new release, we should be able to formulate one property for an arbitrary number of resources.

The mutual exclusion property has been shown for configurations with 2 resources and 2 and 3 clients where the clients repeatedly request an arbitrary (none empty) subset of the resources as well as for the situation with 4 resources and 3 clients. The latter consisted of a model with a million states and it took a few hours to verify the mutual exclusion property. A recently developed parallel model checker has been used to check our largest transition system. The few hours have been reduced to nine minutes on about fifty processors [5]; a promissing development for scaling this approach.

5.2 Starvation

Proving that there is no starvation for the processes turned out to be a problem. This is caused by the fact that there are traces in the transition system that do not correspond to a fair run of the Erlang program. The Erlang processes are scheduled by the use of a certain scheduler and in the model we have (on purpose) abstracted from scheduling and consider all possible sequences of actions, even those in which one single processes gets all execution time.

We want to base our *no starvation* property on the notion of *an action is eventually followed by another action*. In particular, the request of a resource is eventually followed by using that resource. One way of formulating this property is:

$$\textsc{EvtFollow}(a_1, a_2) = [\text{-}^*.a_1].\mu X.(\langle\text{-}\rangle \mathit{true} \wedge [\neg a_2]X)$$

We used this in a context where we instantiated the actions a_1 and a_2 by the request for a resource and the entering of the critical section, respectively. For the latter, we use the confirmation by the locker, i.e., the returned `ok` message. The actual property, like in the mutual exclusion case, depends on the number of clients and resources. For three clients and two resources we have:

$$\begin{aligned}&\text{NO\_STARVATION}(c_1, c_2, c_3, i_1, i_2, i_3) = \\ &\qquad \textsc{EvtFollow}(c_1, i_1) \wedge \textsc{EvtFollow}(c_2, i_2) \wedge \textsc{EvtFollow}(c_3, i_3)\end{aligned} \tag{1}$$

Unfortunately, this property does not hold, even for simple scenario's where definitely no starvation occurs. As an example consider the following simple scenario with three clients and two resources. The clients repeatedly request only one resource, where client 1 and 2 request A, and client 3 requests B. In such a scenario there is no starvation, since both clients may access their resource, release it and request it again. In the μCRL specification we have the possibility of a loop in which client 3 continuously requests and releases resource B. The clients requesting resource A simply do not get any scheduling time in this sequence. However, in the Erlang program this loop is not present, because of the scheduler. Thus, the problem is to disregard unrealistic loops in the transition system. Removing such loops from the transition system, if we at all could find a way to do so, is incorrect. In a realistic setting, such a loop could be executed a few times before the scheduler enables the other processes. What in a realistic setting is excluded, is the infinite traversal of only this loop.

We would like to weaken the EvtFollow property, such that non-fair paths, which exist in the model, but not in the implementation due to the scheduling by the Erlang run-time system, are ignored. Because of limitations in the model checking tool (evaluator 3.0) we need to express this property in alternation free μ-calculus. External advice was required to come up with the following reformulation of EvtFollow, describing that even if a loop exists before reaching a_2, it is still possible (from every state of the loop) to reach a_2 after a finite number of steps (modality $\langle -^{*}.a_2\rangle true$).

$$\textsc{EvtFollow}(a_1, a_2) = [-^{*}.a_1.(\neg a_2)^{*}]\langle -^{*}.a_2\rangle true$$

This property is weaker and in combination with Property (1) it holds for the above mentioned scenario's. Unfortunately, it is too weak, i.e., ignores loops that should be considered. Property (1) with this weaker EvtFollow holds for the first scenario mentioned in Section 2 in which we have starvation in the Erlang context. Recall that for that scenario, client 1 and 2 on their turn take priority over client 3. Thus, there is an ignored loop with only actions of client 1 and 2, although it causes client 3 to starve.

We need to be more precise in the kind of actions that we ignore in a loop and which not. Thinking a little longer about this, it turns out that all actions may appear in the loop. Neither a request nor a release of any other client should be ignored. No matter with action one would like to ignore, there is always a plausible scenario possible from which it is clear that one cannot ignore that action. Even a whole loop should in principle be allowed, as long as it does not occur infinitely often if other actions along the path are also enabled. In our opinion this goes beyond the expressiveness of the logic we use.

Currently we investigate several possibilities to work around this problem, viz. adding explicit scheduling to the μCRL specification, having the model checker changed, or using a different logic (and model checker) that enables reasoning with fairness.

One might wonder whether starvation is an important property at all, since even if a theoretical starvation problem occurs, it might happen that in reality the process always gets served. In regular implementations a timer is set after sending a message and the starvation as such shows as a time out on the client site. This time out is normally followed by a retry and as such the process might get served after a few attempts. We experimented with that by adding such time outs and removing the check for the pending list in the function `check_available` (which leads to starvation for the scenario which was discussed in Section 2). Running this program does not show a starvation at first sight. The client does get access to the resource, occasionally. If we implement the clients with an access time of say, 500 ms, in the critical section, then starvation will show up in the form of a time out of some of the clients. The total number of served requests gets lower, in particular for the clients for which we know that they theoretically starve.

Interesting in this context is that we only detected the performance problem after sufficiently increasing the time spent in the critical section. Here one can argue that testing would not have been sufficient and that the error could show unexpectedly after having the software in use for a long time. Hence, we find starvation an important property to verify.

6 Conclusions

The main contribution of this work lays in the development of an automatic translation of a class of Erlang programs into μCRL. This enables a development of Erlang programs that goes hand in hand with formal verification; leading to formally verified programs. We do not expect smart abstractions or clever tricks performed by the users of this tool, assuming from them a limited knowledge on verification issues. We provide 'push-button verification' that fits in the existing development cycle. As a leading example for developing our tool we used an implementation of a locker algorithm. Verification of this locker algorithm has only partly been successful. Absence of deadlock and mutual exclusion could be proved, but it could not be shown effectively that the algorithm is starvation free. It is subject to further research to find a way around this problem.

The number of states in our models was not much more than a million, such that real performance problems were not encountered. It takes a while for a complete verification, but a few hours is still considered acceptable in this stage. The majority of the work is put in getting the specification right and formulating the right properties. In this case, in particular for 'no starvation', we have spent much time in the formulation of the, still not satisfactory, property.

We use an approach similar to PathFinder or the Bandera project [13, 6]. It would be interesting to see if a Java version of the same case study could easily be handled by using those tools, but we have not found the opportunity to do so. Running the model checking approach of Huch [15] directly on this example is impossible, since that version does not support the generic server design principle. We could change the program by removing this generic server implementation and use a direct implementation in Erlang instead. However, the approach of Huch would translate the choice whether to return an `ok` message to the client or to store the client in the pending list, to be a non-deterministic choice. By abstracting away the data in that way, mutual exclusion does not hold for the obtained transition system.

Another approach to verification of Erlang programs which differs from model checking is the use of a theorem prover for checking properties. The Swedish Institute of Computer Science have in cooperation with Ericsson developed a kind of theorem prover specially focussed on Erlang programs [3]. Advantage of using this tool compared to the model checking approach are the possibility of using the full μ-calculus (instead of alternation free), the possibility to reason over an unbounded number of clients and resources, and the completeness of the approach, i.e., if a proof is given, it holds for the program and not only for the specification. Since model checking allows an easier automation, we aim on using this technique for prototyping and use the theorem prover approach for the version we are satisfied with.

With this verification of the locker case-study we posted several questions for further research and we solved several practical issues on the way. We continue with adding features to the locker, such as shared locks and fault-tolerance, therewith increasing the need for an even better translation tools.

Acknowledgements

We would like to thank Radu Mateescu and Hubert Garavel from INRIA Rhone-Alpes, Izak van Langevelde, Jaco van de Pol and Wan Fokkink from CWI, and Lars-Åke Fredlund and Dilian Gurov from SICS for taking part in the discussions on this case study and supporting us with their advises.

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 T. Noll, Verifying Generic Erlang Client-Server Implementations. In *Proceedings IFL2000*, LNCS 2011, p. 37–53, Springer Verlag, Berlin, 2000.

[3] T. Arts, G. Chugunov, M. Dam, L-Å. Fredlund, D. Gurov, and T. Noll A Tool for Verifying Software Written in Erlang To appear in: *Int. J. Software Tools for Technology Transfer*, 2001.

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

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

[6] 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.

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

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

[9] 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 the 8th Conf. on Computer-Aided Verification*, LNCS 1102, p. 437–440, Springer Verlag, Berlin, 1996.

[10] W. Fokkink, *Introduction to Process Algebra*, Texts in Theoretical Computer Science, Springer Verlag, Heidelberg, 2000.

[11] J. F. Groote, W. Fokkink, M. Reiniers, *Modelling Concurrent Systems: Protocol Verification in μCRL.* course lecture notes, April 2000.

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

[13] K. Havelund and T. Pressburger, Model checking JAVA programs using JAVA PathFinder. *Int. J. on Software Tools for Technology Transfer*, Vol 2, Nr 4, pp. 366–381, March 2000.

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

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