Translating Erlang to μCRL

Thomas Arts* Clara Benac Earle† Juan José Sánchez Penas‡

Abstract

The language Erlang has been developed by Ericsson to implement large switching systems. Erlang is nowadays used by several companies for complex embedded systems.

The language μCRL is a process algebra with data. Several verification tools are available for μCRL and other process algebras, including a tool to create labelled transition systems from μCRL specifications. By having a translation from Erlang to μCRL we can apply the verification tools for process algebras and labelled transition systems to industrial code. The translation is aware of the major design component in the switching software. This knowledge is used to ensure that the size of the labelled transition system generated by the tools is smaller than with a naive translation.

1. Introduction

Driven by the need to verify a small, but critical, part of Ericsson's AXD 301 [4], we started to develop a verification tool. The industrial case-study [2] we considered had no up-to-date detailed specification, only a few hundred lines of code as a starting point. The code is written in the language Erlang. Erlang [1] is a functional programming language developed at Ericsson for the development of concurrent/distributed safety critical software. One of the larger examples of the use of Erlang is for the control software of the AXD 301 high capacity ATM switch used to implement, for example, the backbone network in the UK.

The fact that we are confronted with source code and an out-of-date specification is a rather general phenomena: verification of software is performed at a rather late stage, when engineers feel that what they have produced is more complex than they understand. They then want backup by some sophisticated tools that go beyond testing. Tools like model checkers exist, but are not directly applicable to the software they write. Our goal has been to bridge the gap and translate the source code to an input language for verification, so that a lot of tools are usable for the software. The translation should be effective and rather general, so that the back-end technology can easily be changed.

The code of our typical examples is written in Erlang, otherwise we could probably have used one of the comparable initiatives to translate real code into a specification framework, e.g. [5, 11]. However, the available tools to translate imperative and object-oriented code are too specific to be able to re-write for a functional language, like Erlang. Moreover, due to the functional nature of Erlang with light-weight concurrency, the target specification language typically differs from a target language for imperative or object oriented languages.

Erlang is sufficiently different from a process algebra to make it a challenge to come up with a translation. However, it is one of the best fitting specification formalisms around to use as a target. We choose to pick one specific process algebra with data, viz. μCRL [9], since data is crucial in our case. Besides, we had good experiences with the open-source tools [18] that support this language. In our experience, building the state space for a μCRL specification is more efficient that building the state space for a similar specification in LOTOS [14].

In this paper we describe the tool that we developed to verify two industrial case-studies: a resource manager application in the AXD 301 switch [2] and the scheduler of a video-on-demand server [3, 16]. By our focus on these case-studies we ensured that the translation is useful for a rather general class of verification problems. We are able to translate Erlang programs that respect the generic server design pattern; hence covering synchronous and asynchronous communication as well as most of the computational aspects of the language. Dynamic process creation is also supported in the translation, if performed by the supervison design pattern.

The verification approach is focussed on finding errors in the Erlang software and not on proving correctness of the software. Therefore, the approach is pragmatic. The traces to failure that we in the end obtain by model-checking tools should correspond to traces in the real software. The

*IT University of Gothenburg, Box 8718, 402 75 Gothenburg, Sweden, arts@ituniv.se

†University of Kent, Canterbury, Kent CT2 7NF, United Kingdom, cb47@kent.ac.uk

‡LFCIA, Computer Science Department, University of Corunha, C. Elvinha S/N. 15071, Corunha, Spain, juanjo@lfcia.org

translation ensures this criteria by its construction.

The paper is organised as follows: In Sect. 2 – 9 we describe how the difference between Erlang and process algebra is bridged. By means of code examples from Erlang we show the difficulties we encountered in the translation.

The examples in the paper are reduced to the most basic concepts. We refer to our website[1] for detailed examples with both Erlang source code and their translation to μCRL.

In Sect. 10 we describe the software architecture of the actual implementation and in Sect. 11 we present some results of using the tool and discuss its use in practice.

2. Processes and communication

Erlang is a language with light-weight processes and asynchronous message passing. The language supports both concurrency and distribution. The concurrent processes run in the same virtual machine (a *node* in Erlang terminology) and several virtual machines can be connected to obtain a distributed system. Syntactically there is no difference in communication between processes on the same node or on different nodes. Of course, distribution gives rise to true non-determinism and a slightly different fault behaviour. For our purpose, it suffices to model both distributed and concurrent Erlang processes as truly concurrent processes, like one has in a process algebra. The full non-determinism is covered in that way.

Every Erlang process has a unique identifier that is used to address the process. Every process also has a message queue in which the incoming messages are stored. The virtual machines guarantee that every message is delivered to the queue of the process that the message is sent to. If the receiving process does not (longer) exist, then the message is simply lost without warning. The receiving process actively reads the message buffer by a 'receive' statement. This receive statement is blocking as long as the expected message has not arrived.

A straightforward attempt to map Erlang processes and communication to a process algebra is to create two process algebra processes: one buffer process and one process to implement the logic. The asynchronous communication is modelled by the synchronising actions of process algebra. One action pair to synchronise the sender with the buffer of the receiver and one action pair to synchronise the active receive in the logic part with the buffer. We have chosen to represent the unique identifier as data to a general communicating action instead of having unique communicating actions per pair of communicating Erlang processes.

In Erlang programs it is good practice to add your own process identifier to the messages that are sent. In that way, the receiving process is able to respond. The Erlang primitive `self()` returns the process identifier and by writing `Pid!{Msg,self()}` one sends a message containing both the term `Msg` and the process identifier of the sending process to the process with identifier `Pid`. With the receive statement one reads a message from the queue. Pattern matching is used to selectively read a certain message from the queue.

For the moment assume that we have embedded Erlang communication primitives in the functions `call` and `reply`[2]. The `call` is used to send a message, attached with the process identifier of the sender, and to wait for an answer. The function `reply` is used to return an answer to the caller. This way of embedding low level communication primitives in functions is common practise for industrial Erlang code and in the next section we will describe this in more detail.

The following, snapshot running in an Erlang process, describes a simple server that waits for a client request and replies with an acknowledgement. Variables start with an uppercase character in Erlang, constants and functions start with lowercase.

```
loop() ->
  receive
    {request,Client} ->
       reply(Client,acknowledge)
  end, loop().
```

The client to this server evaluates the function `call(Server,request)`, where `Server` is the process identifier of the server.

We translate the Erlang server process in a μCRL specification with two processes and actions to synchronise processes with this buffer and to synchronise the logic part with the buffer. The μCRL statement $sum(X\!:\!\mathcal{T}, p)$ is shorthand for a non-deterministic choice of all possible values of X of type $\mathcal{T}$.

```
proc server(Self: Pid) =
       sum(Client: Pid,
           receive(Self,request,Client).
           reply(Client,acknowledge,Self).
           server(Self))

proc buffer(Self: Pid, Messages: TermList) =
 b_receive(Self,data(Messages),
           pid(Messages)).
    buffer(Self,rmhead(Messages)) +
 sum(Msg: Term,
   sum(From: Pid,
       b_call(Self,Msg,From).
       buffer(Self,add(Msg,From,Messages))))
```

The process identifier is automatically added as a parameter to processes and communicating actions. The buffer and the

[1] See http://etomcrl.sourceforge.net

[2] For readers familiar with Erlang: the function embedding is part of the design patterns explained later.

process implementing the logic have the same process identifier. Communication is untyped in Erlang and we have to be ready to accept any term in a message queue. Communication in μCRL is specified by pairs of communicating actions; three pairs in our case.

```
receive | b_receive = ex_buffer
call | b_call = in_buffer
reply | replied = sync_reply
```

To simplify reading, we assumed a type Pid in the above example, but, in untyped Erlang, process identifiers are just terms. In the real translation we follow that concept and use type $Term$ instead of Pid.

The reader familiar with Erlang will have noticed that the fifo buffer above differs a lot from the semantics of an Erlang message queue. In an Erlang receive statement one can pattern match on the format of a message. In that way, one can leave certain messages in the queue and selectively take a message from the queue. This is rather difficult to model in μCRL (one easily ends up in creating a model that causes an infinite state space to be created).

A fifo queue is insufficient for the client process that communicates with the server above, since processes can freely send a message to the buffer of this client. If another message than the server reply arrives earlier in the message queue of the client, then the client will be blocked forever. The solution to overcome the problem of selective reading of the queue lays in carefully studying what happens in the real Erlang code. To our advantage, selectively reading a message from the message queue is only done in very restricted circumstances. Basically the mechanism to read a message other than the first message in the queue is only used for exactly this synchronisation. The client adds a special (unique) tag to the message and the server replies with the same tag added. The client is just waiting for any message with the right tag. All other messages that arrive to the queue in-between, are left untouched.

We can model this by having the `reply` action communicate directly to the `replied` action in the client, therewith circumventing the message queue of the client.

This solution only makes sense in a situation where we know which messages are of this special kind and if we know that other messages are dealt with in a fifo manner. But, that is exactly what we recognised when looking at a million lines project like the AXD 301 switching software. The code is written according to certain design patterns. About eighty percent of the communicating processes implements a server that uses the *generic server* pattern. This server restricts communication in a way that eases the transformation for us[3]. In the next section the server pattern and its translation to μCRL are explained in more detail. The supervision tree pattern is another frequently used pattern and is described in Sect. 6.

3. Design pattern: generic server

The *generic server* pattern is used to implement servers in Erlang. A server is a process that keeps state, waits for an incoming message, computes a response message depending on the incoming message and state, and replies to the message and updates the state. The gen_server module implements the generic parts of the server while the call-back module implements the specific functionality (the logic) of a particular instance of the server, i.e. the computation of the response message and the new state.

The above description is, on purpose, an over simplification of the generic server behaviour. The behaviour also takes care of a uniform way of error handling, of a uniform debugging facility, of monitoring nodes and observing whether clients are still alive, etc. In that way, the programmer really only needs to concentrate on the logic of the server. That, on its turn, allows us to easily abstract from a lot of details that the code would have if not implemented in the generic way. Our translation tool can, by means of this generic behaviour, concentrate on the logic and abstract from the implementation details of error handling, debugging, etc.

The simplified version of the generic server is just a small extension of the server given in Sect. 2. We add state as a parameter of the loop and whenever a messages arrives, we need to call a function to evaluate a reply and to update the state. The generic server distinguishes three kind of messages: *call*, *cast* and *info* messages. A call is a synchronise event, where the client waits for a reply. The cast is the asynchronous version of the call, and the info messages serve the special purpose to deal with error events and such. In this paper we only consider the *call* messages, but the actual translation handles the full generic server with all three kind of messages.

```
loop(M,State) ->
    receive
       {call,Msg,Client} ->
          {reply,Reply,NewState} =
            M:handle_call(Msg,Client,State),
          reply(Client,Reply),
    end, loop(M,NewState).
```

The variable `M` contains the name of the module in which the function `handle_call` is implemented. This is the so called *call-back module*.

Remark that the programmer uses the standard generic server component and only provides the call-back module

[3] The restriction in the communication imposed by the design pattern allows a better understanding of complex systems. The same restriction that makes the system easier to understand for the engineers is making it easier for us to translate the system to the clean framework of process algebra.

when implementing a server. The generic part is static and stable over the years. Therefore, we can take the semantics of the generic part for granted and use it in our translation.

A typical example of a call-back module is given below. It implements a server that may receive either a *request* message or a *release* message. The state of the server is a list of clients that have requested. Whenever a client requests, it gets its position in the queue as reply and is added to the state. Whenever a client releases, it is replied an acknowledgement and is removed from the state.

```
handle_call(request,From,State) ->
  {reply,pos(From,State),add(From,State)};
handle_call(release,From,State) ->
  {reply,ack,remove(From,State)}.
```

The Erlang functions `pos`, `add` and `remove` should also be implemented in this call-back module, but are so straight-forward that we omit them here. The behaviour also provides functional embeddings of the communication primitives, similar to what was used in Sect. 2. A client would evaluate the `call` function. The implementation takes care of adding a unique tag and the process identifier of the client to the message. It also takes care of waiting for the arrival of a reply from the server with exactly the same unique tag in order to proceed.

Thus, a typical client that would request and release is implemented in Erlang by:

```
client(Server) ->
  Nr = gen_server:call(Server,request),
  gen_server:call(Server,release).
```

The translation of both server and client is similar to the translation given in Sect. 2. Instead of the receive action, we use a handle_call action and a non-deterministic choice to be able to either receive the request or the release. The buffer is basically the same, apart from the changed name of the receive action to `handle_call`. We provide the code for the client and server process in μCRL.

```
proc client(Self:Term, Server:Term) =
   call(Server, request, Self).
   sum(Nr: Term,
       replied(Self, Nr ,Server)).
   call(Server, release, Self).
   sum(Free: Term,
       replied(Self, Free, Server))

proc server(Self:Term,State:Term) =
   sum(From: Term,
       handle_call(Self,request,From).
       reply(Client,pos(From,State),Self).
       server(Self,add(From,State))) +
   sum(From: Term,
       handle_call(Self,release,From).
       reply(From,ack,Self).
       server(Self,remove(From,State)))
```

The μCRL functions for `pos`, `add` and `remove` are almost equal to the Erlang counterparts. In Sect. 8 the translation of such purely computational functions is explained in more detail.

The last sum in the client is generated automatically, because the `call` function is always returning a result. In Erlang one may choose to ignore the result, but in μCRL we have to explicitly bind it to a variable (`Free` in this case). This already indicates a subtle difference between a return value of a function in Erlang and a communication action in μCRL. This issue is explained in more detail in the next section as it is not specific for the generic server, but a more general phenomena.

In this section we have shown the basic principle of translating generic servers into μCRL. We left out the buffer, since that was presented in the previous section. Our tool handles real servers, which are more complicated than the example shown here, but the basic ideas are captured in this section. Our example call-back module is rather simplistic and in reality there are some issues that complicate matters. In the next few sections we focus on those complications.

4. Functions with side-effect

A significant difference between Erlang and a process algebra is that the latter forces to separate computation from communication. In Erlang, in contrast, a function that performs some calculations can also communicate. Thus, such a function has communication as side-effect of the computation. Of course, the function can call other functions that have side-effects and as such we can get deeply nested integration of computation and communication. Here, our first task is to identify Erlang functions with side-effect from the pure computations. These two classes of functions are translated differently.

Functions are classified as functions with side-effect when they make use of a communication function (like `call` or `reply`). By analysing the call graph of all involved modules, we can syntactically split the Erlang functions in the two demanded categories.

In the remainder of this section we focus on the part with side-effects, and how the computation and communication are separated in these functions. Issues related to the purely computational part are discussed in Sect. 8.

The problem with nested side-effects is best illustrated by an example. Assume an Erlang process that calls a function `p` in order to communicate its result. The function `p` itself contains a side-effect:

```
loop(X) -> Y = p(X), s(Y), loop(X).

p(X) -> Y = f(X), s(g(Y)), h(Y).
```

where f, g and h are pure computations and s is one of the side-effects, e.g., the reply function. A naive translation to μCRL of this process would define actions p and s and in order to handle the matching, it could bind Y and inline the code:

```
proc loop(X:Term) = s(p(X)).loop(X)
```

However, nest actions are not allowed in μCRL. Instead we therefore translate it to:

```
proc loop(X:Term) =
     s(g(f(X))). s(h(f(X))). loop(X)
```

We obtain this translation by a recursive source code transformation of the Erlang functions. In this transformation we lift all side-effect functions to the highest level and push pure computations down by duplicating them. Thus, we translate on the source code level all functions with side-effect to functions that look like:

```
p(X) -> s1(---),...,sN(---),---.
```

where --- stands for pure computations.

Whenever we encounter a statement Y=p(X) in the code, we could bind the variable Y to the last pure computation of the function p and substitute this in the μCRL code. Thus, we could inline the side-effect functions as actions in place of the call to p. However, the attentive reader has probably already noticed that this cannot work for recursive functions with side-effects; without knowing the number of recursive iterations, one is unable to unfold the definition and hence unable to inline the exact number of side-effects. Neither does this work for functions that perform a side-effect and cannot be de-composed. As an example, consider the simple program that performs a side-effect on every element of a list and returns ack as a result. All results are stored as a parameter of the function, i.e., the list of ack's is increasing.

```
loop(X,Rs) -> R = p(X), loop(X,[R|Rs]).

p([]) -> ack;
p([Head|Tail]) -> s(Head), p(Tail).
```

The standard solution to deal with recursive functions when writing a compiler is to implement a stack data structure to store the return values of the recursive calls. We adopt this idea, where the stack is implemented as a μCRL process and push and pop operations are communicating actions. With a source-to-source transformation we make sure that all functions with side-effects are in the previously mentioned format with either a pure computation as last expression or a call to another function with side-effect. We replace all pure computations by a push on a stack and pop this value in the code where we call the function.

The above Erlang example is translated to the μCRL code below (where the stack itself is omitted). The somewhat obscure notation for the if-then-else statement in μCRL is $then \lhd if \rhd else$.

```
proc loop(X:Term, R:Term) =
       p(X).
       sum(R:Term,
           pop(R).loop(X,cons(R,Rs)))

     p(X: Term) =
       push(ack)
       <| eq(X,nil) |>
       (s(head(X)).p(tail(X)))
```

By using the stack and pushing pure computation inside side-effect functions, we can deal with nested side-effects. The stack solution also provides a solution for the case where in the above match Y=p(X) the variable Y is replaced by a complicated pattern with several variables. The only thing we have to add to our translation is a sum construct for every occuring variable in the pattern. Note that the introduction of the sum construct is only used for the matches of patterns with functions that contain side-effects. A match with a pure function is translated differently, as explained in the next section.

Although the stack process solves our problem of translating nested side-effects, we also have to pay the price of more communication in the model and therefore an increased state space of the system. Moreover, the duplication of the pure functions gives rise to longer computation times in the model than in the real implementation.

5. Pattern matching in communication part

Both μCRL and Erlang allow pattern matching on data. In the previous section we have shown how one particular kind of pattern matching is elegantly translated by using communication via a stack process. In this section we focus on two of the possibilities, viz. pattern matching in function clauses and in communication primitives.

5.1. Function clauses

Process definitions in μCRL can only have variables as parameters, c.f. the definition of client and server in Sect. 3; and there is only one clause per process. Erlang functions that are translated to μCRL processes may have several clauses in which pattern matching decides which clause is evaluated.

Several Erlang function clauses can easily be combined in one case-statement, but that does not solve the problem. The pattern matching in the case is equivalent to pattern matching on function clause level. We treat those therefore similarly.

First, we compute the discriminating pattern to select a certain clause (c.f. [17]) and we use a nested if-then-else structure to determine which part of the function to evaluate. This if-then-else can later be directly mapped to μCRL.

Second, we replace the patterns in the arguments of the function to variables and replace bindings caused by these patterns to destructor functions. As an example, consider the following Erlang function:

```
loop(X,[]) -> s(done), loop(X,X);
loop(X,[Head|Tail]) -> s(Head),loop(X,Tail).
```

This code requires two destructors, viz. `hd` and `tl` to extract the head and tail of a list. With those two destructors the code is transformed to the Erlang code:

```
loop(X,Arg1) ->
  if
    nil == Arg1 ->
      s(done), loop(X,X);
    is_list(Arg1) ->
      s(hd(Arg1)), loop(X,tl(Arg1))
  end.
```

In general the conditions to check are more complicated than only checking whether an argument is a list or the empty list. We need to bind variables to terms in order to use them in the expressions and sometimes we even need destructor functions in the conditions, for example if we want to check whether the head of a list is equal to the integer one[4]. However, there are only finitely many possible patterns in Erlang. The simplified version of the computation of the conditions for given pattern P and expression E, where only lists, integers, and variables are considered is given below. The function returns a condition and a set of variable bindings [5].

$$cond(P,E) = \begin{cases} \langle true, \{P \mapsto E\}\rangle & var(P) \\ \langle is\_list(E) \wedge \phi \wedge \psi, \sigma \cup \tau\rangle & P = [H|T] \\ & \langle \phi, \sigma\rangle = cond(H, hd(E)) \\ & \langle \psi, \tau\rangle = cond(T, tl(E)) \\ \langle equal(P,E), \emptyset\rangle & \text{otherwise} \end{cases}$$

The more complicated version of the above function is successfully used in our source-to-source transformation to map different patterns in function clauses to variables in the arguments of the clauses and nested conditions in the body of the clause.

5.2. Communication primitives

The above described function clauses are translated to μCRL process definitions. For function clauses that are communication primitives and that are translated to μCRL communicating actions, a similar pattern matching transformation is necessary. In this case, however, one cannot introduce the if-then-else construct in the same way.

As an example, consider the previous `handle_call` function clauses for a server, where the client can now request and release either resource a or resource b. The server administers whether the resource is free by keeping a tuple as state variable, containing a boolean value per resource.

```
handle_call({request,a},Client,{A,B}) ->
  {reply,A,{false,B}};
handle_call({request,b},Client,{A,B}) ->
  {reply,B,{A,false}};
handle_call({release,R},Client,State) ->
  {reply,ack,update(R,State)}.
```

Here we have two matches that need to be translated differently. The `handle_call` function is translated in a non-deterministic choice between the alternatives and embedded in a server loop. That loop has `State` as a parameter and the tuple {`A,B`} should be decomposed as described in the previous section. The message (and similarly the client) should be treated differently. For those parameters, the variables are isolated and put in a sum construct. The matching is done by the pattern matching mechanism of μCRL.

```
server(Self:Term,State:Term) =
 sum(Client: Term,
  handle_call(Self,tuple(request,a),Client).
  reply(Client,element(1,State),Self).
  server(Self,tuple(false,element(2,State))))
 +
 sum(Client: Term,
  handle_call(Self,tuple(request,b),Client).
  reply(Client,element(2,State),Self).
  server(Self,tuple(element(1,State),false)))
 +
 sum(R: Term,
  sum(Client: Term,
   handle_call(Self,tuple(release,R),Client).
   reply(Client,ack,Self).
   server(Self,update(R,State))))
```

Of course, we use the knowledge we have on our communication primitives to decide which parameters need to be transformed to match on the process level and which are to be transformed in a sum construct. Typically the matching on the process level is translated source-to-source, whereas the introduction of non-determinism and sum construct is left to a later stage.

Lacking in the above translation is the introduction of the conditions that we compute for the pattern match. A programmer could easily handle the same message in two different clauses of the `handle_call` function by differentiating the state in which the message arrives. This way of programming is de-recommended in the style guides, but occurs now and then in code fragments. We therefore have to put conditions in the loop that correspond with the possible patterns of the state and only then non-deterministically match the possible messages.

[4] The if-statements in Erlang do not allow destructors in the conditions, therefore, we use nested case-statements instead of the if-statement, but explaining it by means of an if-statement is clearer.

[5] The set of variable bindings is a list in the real implementation, where variables that have been bound before need to be matched against a value if they occur more than once in the pattern.

6. Design pattern: supervision tree

One of the key features of most distributed systems, in particular those for which Erlang is used, is fault tolerance. Erlang supports fault-tolerance by means of the supervision tree, a structure where the processes in the internal nodes (supervisors) monitor the processes in the external nodes (workers).

The creation of the processes architecture of the system is encoded inside the supervision tree initialisation. This fact can be used in order to extract the processes of the system from the source code and the input (configuration) provided by the user. Process algebras allow the creation of new processes, but the set of tools developed for μCRL does not support this feature. We partially evaluate the supervision tree, using the fact that we know the semantics of that design pattern, in order to obtain its structure and a list of all created worker processes.

The use of the supervision design pattern is so common that using it to find the created processes is no severe limitation. We cannot handle Erlang applications in which processes are spawned outside the scope of the supervision tree, but these are not commonly encountered in production code. A more important limitation of our approach is that we do not encode the fault tolerance of the supervision tree in our model. Thus, we only look at successful executions. The fault-tolerant behaviour is currently being the object of further research.

7. Higher-order functions

Erlang is a functional language that supports higher-order functions, something which most specification language avoid for the inherent complexity of the analysis. The expressiveness of a higher-order function is as useful for a good program as design patterns. Therefore, it is a pity that μCRL is a first order language.

Since higher-order functions are a real extension to a language, there is no simple way of translating these functions to first-order variants. Luckily, most of the Erlang code on our case-studies only uses a few predefined higher-order functions, like `map`. We therefore designed the translation to handle only those special cases that we encountered, like we only handle a few design patterns. We defined a source-to-source transformation on the selected functions to flatten them to first-order alternatives. Any occurrence of the function `map`

```
  map(fun(P) -> f(P,E1,...,En) end, Xs)
```

where `P` is a pattern, `Xs` an expression returning a list and `E1`,...,`En` arbitrary expressions, is replaced by a call to a unique function `map_f(Xs,E1,...,En)`. The unique function is added to the code and defined as:

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

Although for many functions a similar transformation pattern can be used, there is no general way of translating higher-order concepts into μCRL.

8. Data and pure functions

Although Erlang has a fixed and small set of constructors, the translation of the data part is more complicated than one would wish. Basically it is a syntactic conversion of constructors, destructors and selectors. The latter two implemented as μCRL functions that directly correspond to the Erlang functions. However, an obstacle in this is that not all Erlang data structures are inductively defined. The integers, which most programming languages support, are probably the best example of that. In μCRL all data structures need to be defined inductively and the advised way of defining integers is by means of naturals, which are represented as zero and its successors. This might be a theoretically rather clean approach, in practise it means unreadable specification for larger numbers, slow computations and tools that complain about a too deep term depth when numbers get large.

Another obstacle is that syntactic equality is not a predefined relation, but that this relation has to be specified. In particular for rich sets of data structures (which we use), this results in a large amount of defining rules.

Erlang is dynamically typed and has very flexible typing rules; μCRL is strongly typed with a simple and restricted type system. Since we try to keep the specification in μCRL as close to the Erlang code as possible we construct in μCRL a data type *Term* in which all Erlang data types are embedded. The tool supports most Erlang data types: lists, integers, atoms, tuples, and records. However, the recently added Erlang bit-syntax implementing the data structure of bit sequences, is not considered by our tool.

8.1. Pattern matching

In Sect. 5 we have discussed the matching of function clauses and expressions for the functions that have side-effects. For the pure functions, the translation is much simpler. The header of function clauses can directly be copied, since for the term matching on that level, matching in Erlang and μCRL are the same. We have to rewrite the body of the function clauses, where all statements have a fixed translation to μCRL rewrite rules. For example, an Erlang function with `case` statement

```
functionName (P1,P2,...,Pn) ->
           case E of
              Q1 -> E1;
```

```
        ...
        Qm -> Em
      end.
```

where P1, ..., Pn, Q1, ..., Qm are patterns and E, E1, ..., Em are expressions is translated in several rewrite rules (where recursively the expressions are translated). The notation var(P1,...,Pn) stands for all variables in the patterns P1, ..., Pn.

```
functionName(P1,P2,...,Pn) ->
   case1(var(P1,...,Pn), E).

case1(var(P1,...,Pn), Q1) -> E1;
...
case1(var(P1,...,Pn), Qm) -> Em.
```

In this translation case1 stands for a function symbol uniquely chosen for the translation of every case statement. We can perform this transformation source-to-source and only in the last phase translate the Erlang code to μCRL.

Another statement with a similar translation is the Erlang match P = E. The way to deal with this statement is again to call a function and lift the match to the rewrite level. Functions with a match

```
functionName(P1,...,Pn) -> P = E, Expr.
```

are source-to-source translated to

```
functionName(P1,...,Pn) ->
   match1(var(P1,...,Pn), E).

match1(var(P1,...,Pn), P) -> Expr.
```

Note that, although the translation of these statements looks rather straightforward and easy, we slightly change the semantics in the translation. As long as we stay on the source-to-source level there is no danger, but a direct translation to μCRL would affect the behaviour of the program.

For example, Erlang uses priority rewriting, i.e. patterns are tried from top to bottom and if an expression matches a pattern, the other alternatives are not visited. In μCRL any matching rule could be taken. At the moment we therefore check that there are no overlapping patterns in the definition, but in fact, one should rewrite the patterns to a non-overlapping set.

9. Module system

Erlang code is divided into modules, each module consisting of a sequence of attributes and function declarations. Process algebras on the contrary, do not have module systems, although some tools (e.g., the CADP tool set [8] for LOTOS [14]) support a module system.

To prepare the conversion of the given collection of Erlang modules into one μCRL specification, we perform a source-to-source transformation. Every call to a function *f* is replaced by the Erlang qualified call *modulename:f*, where *modulename* is the name of the module where the function *f* is implemented.

Some modules in the standard library are translated once and for all to μCRL and the code of those functions is simply linked in at translation time. For the other functions, we assume all necessary modules given and change the name of the function definition and function call to the same name, viz. *modulename_f*, in the μCRL translation.

10. Overview of the tool

In the Sect. 2 – 9, we have described the highlights of translating Erlang to μCRL. In this section we describe the architecture of the translation tool, called etomcrl, in which we clarify the order in which steps described before are taken.

We used Erlang as implementation language for etomcrl. Remember that our tool takes several Erlang source code modules and its initialisation parameters as input, and generates a specification in μCRL as output. The module etomcrl is the main module of the tool implementation. The function supervisor starts the compilation process, that takes as arguments the module, function and arguments of the supervision tree behaviour implementing the system.

Fig. 1 shows the three main steps in the etomcrl tool. First, a source-to-source transformation is performed on the level of Erlang, resulting in Erlang code that exhibits the same behaviour to an observer as the original code, but is optimised for verification. Second, the side-effect-free part of the code is separated from the part with side-effects, since the translation is different for each of the two parts. Third, the translated files are combined into a single μCRL specification.

- etoe: the first phase of the transformation can be seen as a preprocessor, that performs some Erlang to Erlang source code transformations. The main transformations are: the supervision tree is evaluated in order to extract the processes of the system, as introduced in Sect. 6. The lower module is used to remove higher order functions as explained in Sect. 7. The noio module is removing the calls to the i/o module; we verify embedded systems and are not interested in output to the console. Finally, the code is analysed and split in two different parts: the side-effect-free part with only pure computations and the side-effect part. These parts are going to be processed in different ways in the next stage, as explained in Sect. 4.

- etopa: the second phase of the translation is from Erlang source code to an internal representation very close to the process algebra syntax and semantics.

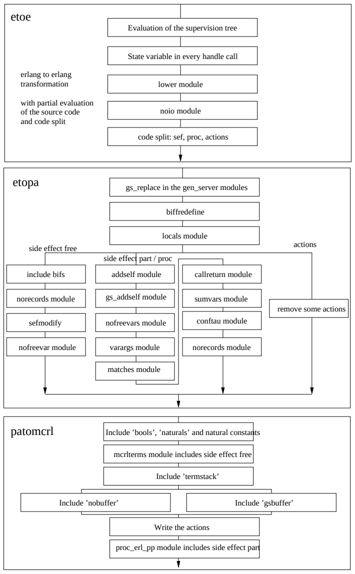

Figure 1. Architecture of the tool

The main transformations are: `gs_replace` changes Erlang gen_server related code as it is explained in Sect. 3 in all the call-back modules implementing the generic server behaviour. The `locals` module takes care of encoding the modules into the function names, as explained in Sect. 9.

For the side-effect-free part of the code, the following translations are performed: Some Erlang library functions are included for translation. Records are translated to a data structure that can be defined inductively. The module `sefmodify` changes the function clauses related with the matching problem explained in Sect. 8.1. The module `nofreevar` replaces the underscores in the Erlang source code to uniquely chosen free variables.

For the side-effect part of the code, the following translations are performed: The `addself` and `gs_addself` modules change the code such that the process identifiers can be used as arguments, as explained in Sect. 2. The `nofreevars` module is also applied to this part of the code for the same reason as above. The modules `varargs` and `matches` are performing the transformations explained in Sect. 5. The `callreturn` module introduces the stack explained in Sect. 4. The module `sumvars` is introducing the sum construct, as explained in Sect. 4.

- `patomcrl`: the third phase is a backend for translating the internal, process algebra, representation to μCRL. The main steps in this phase are: the code for the standard inductive definitions for the μCRL sorts bools and naturals are introduced. The module `mcrlterms` translates the side-effect-free part to μCRL syntax. The buffer and stack have standard μCRL implementations that are inserted. Actions are inserted as communication actions in the μCRL specification, and finally, `proc_erl_pp` translates the side-effect part from the internal notation to μCRL syntax.

The tool could be reused for other kind of transformation, e.g. if we want to extract a LOTOS specification [14], we only need to write a new back-end for translating from the internal representation to the LOTOS syntax. Therefore, even though the tool has been built for a quite concrete purpose, its main ideas can be reused for similar approaches.

11. Conclusions

In this paper we described how a functional language with support for concurrency and distribution can be translated to a process algebra. The ingenuity of the translation shows in the choices we made for mapping concepts of one language to concepts of the other. For example, we make strong use of the design patterns in Erlang to enable a smooth translation. By translating Erlang to μCRL we can use formal verification tools developed for μCRL and labelled transition systems.

Other approaches to the formal verification of software include the specification language Promela and model checker SPIN [12], PathFinder [11], and Bandera [10]. In the first case Promela is very close to C while the targeted language for the latter two is Java. Relevant tools developed for Erlang include a theorem prover with the Erlang semantics built into it [6, 7] and the model-checker of Huch [13] which works on code directly. The theorem prover can in an inefficient way be used to symbolically explore part of the state space. Its power is though in interactive proofs of a different nature, whereas the model checking approach is efficient and automatic. Huch's approach differs from ours in the way he abstracts data aspects which we consider crucial. In particular, he abstracts *case* statements by nondeterministic choices, losing all reference to the data.

The tool that we constructed to perform the transformation has been evaluated by two major case-studies of which the results are reported elsewhere [2, 3]. The tool allows us and others to apply formal verification tools on real industrial code. The tool has been written in such a way that it is not μCRL specific, but can easily be ported to other process algebras or similar approaches.

Before and during the development of the tool, we have repeatedly asked ourselves whether it would be better to build a verification tool directly on the level of Erlang instead of translating Erlang to a process algebra. However, for a small group like ours, it is much easier to build a translation tool and use all the research done over the years by other groups, than to concentrate on doing the research ourselves and get only part of all theory implemented. In this way we benefit from years of experience with building verification tools and optimising those tools and pay the minimal price of having to write a kind of compiler ourselves.

We identified three main restrictions in verification formalisms that we considered. First, specification languages lack the support in the development tools that modern programming languages have. A simple thing like a debugger or a way to write code in modules instead of one big specification are often missing. Second, programming languages have powerful constructs both in statements and in data structures, e.g. higher-order functions, list comprehensions, records, inheritance. These constructs are seldomly supported by specification languages, which most of the time remind of languages from the early eigthies. Third, specification languages have poor and inefficient support for arithmetics. Hence, a tool to create a rather small state space can still spend an amazing amount of time in just performing simple arithmetic.

As a small comment, we can underwrite the conclusion of Lamport and Paulson [15]: specification languages should not be typed. At least, if one translates a programming language to a specification language, a type system is often in the way. The programming language has certainly a type system and hence the types need not be checked on a specification language level. Moreover, the type system of a modern language is easily incompatible with the type system of the specification languages around. Hence, the types get in the way when translating.

Despite some limitations in the process algebra languages, the tools developed for them (e.g. [8, 18]) make a translation very rewarding. The time it takes to create a state space of a reasonably complicated system or the time necessary for model checking some properties has never been a restriction in our case-studies. We have been able to verify several properties of real code with a reasonable complexity. Without counting the about 2000 lines of code that are given as design patterns and library code, our case studies consisted of a few hundred lines of code. From the experiment we can conclude that this verification approach scales to larger size examples. We hope to be able to improve the integration of several tools in order to make source code verification even simpler in the future.

Acknowledgements

We thank the developers of the μCRL and CADP tools for their suggestions and help during our verification attempts; Ulf Wiger for his generous help in providing us with part of the source code of the AXD 301 switch; Victor Gulias and the rest of VoDKA developers for their help with the source code of the video-on-demand server; and John Derrick for his support and for proof-reading the paper. The work described in this paper was partially supported by MCyT, Spain, Project TIC 2002-02859.

References

[1] J. Armstrong, M. Williams, C. Wikstrom, and R. Virding. *Concurrent Programming in Erlang*. Prentice-Hall, Englewood Cliffs, New Jersey, USA, second edition, 1996.

[2] T. Arts, C. Benac Earle, and J. Derrick. Development of a verified Erlang program for resource locking. *Int. J. on Software Tools for Technology Transfer*, 2004. to appear.

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

[4] S. Blau and J. Rooth. Axd 301 – a new generation atm switching system. *Ericsson Review*, 1, 1998.

[5] J. C. Corbett, M. B. Dwyer, J. Hatcliff, and Roby. Bandera: a source-level interface for model checking Java programs. In *Int. Conf. on Software Engineering*, p. 762–765, 2000.

[6] L.-Å. Fredlund, D. Gurov, T. Noll, M. Dam, T. Arts, and G. Chugunov. A verification tool for Erlang. *Int. J. on Software Tools for Technology Transfer*, 4:405–420, 2003.

[7] L.-Å. Fredlund. A framework for reasoning about Erlang code. PhD thesis, Dept. of Microelectronics and Information Technology, Royal Institute of Technology, Stockholm, 2001.

[8] H. Garavel, F. Lang, and R. Mateescu. An overview of CADP 2001. *European Association for Software Science and Technology (EASST) Newsletter*, 4:13–14, 2002.

[9] J. F. Groote and M. A. Reniers. Algebraic process verification. In *Handbook of Process Algebra*, p. 1151–1208. Elsevier, 2001.

[10] J. Hatcliff and M. Dwyer. Using the Bandera tool set to model-check properties of concurrent Java software. *Lecture Notes in Computer Science*, 2154, 2001.

[11] K. Havelund and T. Pressburger. Model checking Java programs using Java pathfinder. *Int. J. on Software Tools for Technology Transfer*, 2, April 2000.

[12] G. J. Holzmann. *Design and validation of computer protocols*. Prentice-Hall, Inc., 1991.

[13] F. Huch. Verification of Erlang programs using abstract interpretation and model checking. *ACM SIGPLAN Notices*, 34(9):261–272, Sept. 1999.

[14] ISO/IEC. Lotos, a formal description technique based on the temporal ordering of observational behaviour. *IS 8807*, February 1989.

[15] Lamport and Paulson. Should your specification language be typed? *ACM Transactions on Programming Languages and Systems*, 21, 1999.

[16] J. J. Sánchez Penas and C. Abalde Ramiro. Extending the VoDKa architecture to improve resource modeling. In *2nd ACM SIGPLAN Erlang Workshop (PLI'03)*, Uppsala, Sweden, August 2003.

[17] P. Wadler. Efficient compilation of pattern matching. In S. Peyton-Jones, editor, *The implementation of Functional Programming Languages*, p. 78–103. Prentice Hall, 1987.

[18] A. G. Wouters. SEN-R0130, manual for the μCRL tool set (version 2.8.2). Technical report, CWI, Amsterdam, 2001.