Verifying Generic Erlang Client–Server Implementations

Thomas Arts

Computer Science Lab, Ericsson Utvecklings AB
Box 1505, S–12525 Älvsjö, Sweden, thomas@cslab.ericsson.se

Thomas Noll

Lehrstuhl für Informatik II, Aachen University of Technology (RWTH)*
D–52056 Aachen, Germany, noll@informatik.rwth-aachen.de

Aachen University Computer Science Technical Report 00–08

Abstract

The Erlang Verification Tool is an interactive theorem prover tailored to verify properties of distributed systems implemented in Erlang. It is being developed by the Swedish Institute of Computer Science in collaboration with Ericsson.

In this paper we present an extension of this tool which allows to reason about the Erlang code on an architectural level. We demonstrate our approach by developing a verification method for client–server systems designed using the generic server implementation of the Open Telecom Platform. Our idea is to specify a set of transition rules which characterize the abstract behaviour of the generic server functions. In this way we can reason in a compositional way about any client–server application without having to consider the concrete implementation details of the generic part, which simplifies proofs dramatically.

We argue that, due to the generality of our approach, it is easily possible to put it into practice using other verification tools or to integrate further generic architectural elements such as finite–state machines or event handlers.

1 Introduction

The high quality demands on software for telecommunication applications may partly be ensured by the use of formal methods in the design and development. By the high degree of concurrency in those applications, *testing* is often not sufficient to guarantee correctness to a satisfactory degree. *Verification*, namely formally proving that a system has the desired properties, is therefore becoming a more and more widespread practice (see [CW96] for an overview).

*Most of the work has been done while the author was at the Department of Teleinformatics of the Royal Institute of Technology (KTH), Stockholm, Sweden.

Although a complete formal specification of a telecom application would probably be one of the best ways to ensure its correctness, in practice the descriptions are rather informal, written in natural language in combination with some fragments of for example the Standard Description Language SDL [SDL93]. Reasons for the absence of a complete formal specification can be found in the fact that the specification changes several times during development, triggered by experiments with a release or by changed requirements. It is felt too time consuming to modify both the formal specification and the code. Even the informal specification tends to run out of phase with the actual implementation and is often only updated after a release of the product.

Towards the end of a project, it is the running code that represents the best 'specification' of the actual application. Questions about the implementation are therefore best formulated in terms of this code: 'is there a possibility that this finite–state machine implementation deadlocks?', 'does this server implementation correctly respond to all possible requests?'.

In order to find answers for these questions, one might abstract from the code (having the informal specification helping in this) and check the questions in an obtained model [CGL94, Huc99]. If one takes this realistic, the verification is used for finding errors, more than proving correctness. If the model does *not* fulfill a certain property, then this might indicate an error in the code. It is common practice to analyze the given trace that leads to the found error and see whether this is also a valid trace in the actual code. The latter need not be, since a model (which is an abstraction) removes some, potentially essential, details.

In general the constructed model depends on the property one wants to prove. Often one does not directly construct the final model, but many models are built, which are refinements of an initially rough model. The model is refined until either a detected error can be identified as a real error or one has enough confidence in the detailedness of the model to believe that the code is error–free. In this analysis method, finding a trace to an error can efficiently and automatically be performed by a model checker. The construction of a model and its refinements, including checking the trace in the real code, can often only be done by hand or with some minor computer assistance.

Given that the code is the best formal description of the software, as an alternative to building the models and checking the traces, one can use a theorem prover to verify properties of the code directly. A kind of interactive theorem prover for the purpose of verifying properties of programs written in the functional programming language Erlang [AVWW96] has been developed by the Swedish Institute of Computer Science (SICS) in collaboration with Ericsson. This Erlang Verification Tool (EVT; [ADFG98]) can be regarded as a tableau–based prover with proof rules for first–order modal logic in Gentzen–style, extended by rules that reflect the semantics of Erlang, rules for decomposing proofs about compound systems to proofs about the components, and rules for induction and co–induction [DFG98].[1] The disadvantage that proofs should be provided by hand should be put against the advantages of obtaining certainty that a property holds for the code and of the possibility to reason about unbounded data structures, unbounded messages queues, and dynamic creation of processes. Moreover, bugs are detected by the fact that proofs cannot be provided, and the attempt to prove the property usually clearly indicates a trace in the code which can be used as a counterexample.

[1] A binary prerelease of EVT is available at `ftp://ftp.sics.se/pub/fdt/evt/index.html`.

In this paper we present an addition to this verification tool, which allows reasoning about the Erlang code on an architectural level. In this way we provide a general abstraction, automatically obtained from the code, that is detailed enough to prove many different properties of the program.

Many applications that we consider consist of several servers that communicate with their clients. These servers are all implemented in a predefined, generic way. The generic implementation takes care of starting the server, receiving synchronous and asynchronous messages, and providing debug and log information for maintenance purposes. We added the specification of the functions of this generic server as proof rules to the verification tool. More precise: based on the transition–system semantics of Erlang as presented in [Fre], we provide rules which describe the possible transitions that any Erlang process evaluating the respective function can take, restricted by the shape of the environment if necessary. Thus we abstract from the actual implementation of the server and concentrate on its specific behaviour instead, such that we can argue about any client–server application without having to consider the source code of the generic part. In this way we support a compositional style of reasoning which is relativized by the assumption that the concrete implementation of the generic module follows its specification. This abstraction is property–independent and can hence be used for all properties we are interested in, whereas we gain that we may skip many details, leading to much smaller proofs.

The remainder of this paper is organized as follows. Sect. 2 describes the class of systems addressed by our approach, namely, client–server systems implemented in Erlang using the `gen_server` module. In Sect. 3 we give the abstract representation of some of the `gen_server` functions; the entire specification can be found in Appendix A. Their implementation in EVT is discussed in Sect. 4, completed by Appendix B, which contains the corresponding logical formulae in EVT syntax. In Sect. 5 we address the correctness of our approach and conclude with some comparative remarks.

2 Generic Client–Server Implementations

Large software applications are built using a software architecture. Elements of such architectures are: databases, device drivers, finite–state machines, supervisors, monitors, servers, and many more. After putting the architecture together, the actual implementation of the components may start. Software engineering practice has taught that having all servers implemented in some general way is an advantage, both for development as well as for maintenance. Even better, when part of the server software is already written and used as basis for all specific servers, it serves the correctness of the whole application, since the generic part is well developed and tested. Therefore, the Open Telecom Platform (OTP; [OTP]), the set of libraries and design principles that comes along with Erlang, supports a standard, generic implementation of a server by providing the `gen_server` module. This module implements several interface functions providing synchronous and asynchronous communication, debugging support, error and timeout handling, and other administrative tasks. In order to obtain the required specific server functionality the programmer provides an instantiation for this generic server. This instantiation consists of a separate module, the so–called *callback module*, which contains (*callback*) functions that are invoked by the generic part of the server. Thanks to this software engineering practice we are able to easily abstract from the actual server implementation in the code.

When a client process wants to synchronously communicate with the server, it uses the standard `gen_server:call` function with a certain message as an argument. The generic part sends the message to the server process and blocks the client. In the server process, another function of the generic part receives the message and forwards it to the application–specific part by calling a function in the callback module. This callback function should return the response and the new server state. The new state is stored in the server process and the reply is returned to the client by the generic part of the server, completing therewith the synchronous event.

In greater detail, the typical flow of control in a `gen_server`–based client–server application is as follows.

- To start the server process the `gen_server:start` function is called. This function creates a new process, the *server process*, in which a function is started that implements the server. The first thing this function does is computing its initial state by calling the `init` function in the callback module. After that, the process waits for a request from a client process.

- The client uses the `gen_server:call` function to send a synchronous request to the server. The request is handled by the `handle_call` function in the callback module while the client is being suspended, waiting for the response. The current state of the server is passed as an argument to the callback function, which on its turn returns both a reply message and a new state. The reply message is sent to the suspended process.

- Alternatively, `gen_server:cast` can be used to send requests to the server where no response is expected. Here only the internal state of the server is changed according to the result of the `handle_cast` function in the callback module.

- Both `handle_call` and `handle_cast` can return a value indicating that the server should terminate. In this case, `gen_server` invokes the `terminate` function in the callback module to clean up before the process terminates.

Clearly the flow of control as described above may look different when error situations occur, such as a server that cannot be started or a call that cannot be handled. In those cases some standard error handling is performed by the generic server. In addition several options can be provided to the standard calls in order to have them behave slightly different.

The following example of a simple locker server implements a scheduler that arbiters the access to a single resource. It is usable by several clients at a time, communicating synchronously by executing function calls of the form `gen_server:call(Server, request)` and `gen_server:call(Server, release)` to request a lock and release it thereafter, respectively.

The example is classical and the properties of interest are likewise (formulated for a server with arbitrary many clients): no deadlock, no starvation, mutual exclusion, etc.

The client function is trivially implemented (for details on syntax and semantics of Erlang see [AVWW96]) in a module called `client` by a function with the same name that takes the process identifier of the server as a parameter (to establish communication).

```
-module(client).

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

The state of the server consists of a list of pending clients. More exactly, the client that currently has access to the resource is stored in the head, and all waiting clients are kept in the tail of this list. We start the server by evaluating `gen_server:start([])` to initialize it with an empty list of pending processes.

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

init(Requests) ->
  {ok, Requests}.

handle_call(request, From, Requests) ->
  case Requests of
       [] ->
          {reply, ok, [From]};
       _  ->
          {noreply, Requests++[From]}
  end;
handle_call(release, From, [_|Waiting]) ->
  case Waiting of
       [] ->
          {reply, done, Waiting};
       _ ->
          gen_server:reply(hd(Waiting), ok),
          {reply, done, Waiting}
  end;
handle_call(stop, From, Requests) ->
  {stop, normal, ok, Requests}.

terminate(Reason, Requests) ->
  ok.
```

The `gen_server:call` function in the client causes the `handle_call` function in the callback module to be executed. Note that the return value (either `ok` or `done`) is ignored by the client process, i.e., no check is performed to see if the value really is the expected one. The return of a value is a synchronization mechanism which in this case is independent of the actual value. In this particular example we could have restricted us to asychrounous communication for releasing by using `gen_server:cast` instead of `gen_server:call`, but for readability we aim to concentrate on only one communication primitive.

On the level of the actual execution, Erlang supports only one way of communicating messages, which is asynchronous. However, the `gen_server` implementation ensures a synchronized behaviour: the `gen_server:start` function will not return before the `init` function has returned a state, and the `gen_server:call` function only returns if the `handle_call` function returns a reply to the callee. In the `gen_server` module this synchronous communication is implemented by using the asynchronous primitives: a message is sent and a receive statement directly succeeds this send, waiting for the response. The main goal of the technique we present in this paper is to be able to abstract from the implementation of the synchronous communication. The fact that a unique identifier is provided for every message, that the message is read from the message queue in a certain way, that a timeout primitive is supported and all that, need not be of our concern.

In this abstract setting, the given server implementing the locker can be described as a server that stores a list of clients which claim access to the resource. The number of clients is arbitrary and in properties or proofs we want to make no assumptions about an upper bound. Access is granted to the first client in this list, the other clients are suspended. Only after a release of the client that accesses the resource, the client that is next in line gets access to it. The suspension of clients is implemented by not providing the reply immediately (returning {`noreply`, ...}), but sending it later (`gen_server:reply`(..., `ok`)).

3 The Verification Approach

Already without our addition, the Erlang Verification Tool can be used to verify the Erlang code of an application which makes use of the generic server implementation[2]. When establishing such a proof, one has to follow the simulation of the synchronous communication by the underlying asynchronous implementation. By the nature of the Erlang semantics this means that one should also prove some properties about the message queues of the client(s) and of the server, which seems irrelevant given the knowledge how the server works. In particular, when many clients are involved, a lot of non–determinism can be introduced in the proof by observationally equivalent traces. As such, the number of proof goals may be much larger than seems strictly necessary.

Another disadvantage in such a proof is that one gets confronted with details such as debug features implemented in the `gen_server` module. Although the verification is performed in a context where debug facilities are assumed to be disabled, one still generates an extra proof goal for testing the debug flag. This test is not atomic and since we work in a concurrent setting, even those few steps cause duplication of work, since another action may be chosen in another process at that same time.

In our approach we simplify the verification task by ignoring the concrete implementation of the `gen_server` module. We specify its abstract behaviour by making its syntactic constructs recognizable as keywords by EVT, and by adding appropriate transition rules to

[2]Although the EVT tool lacks support for modules at the moment, one can combine the callback, the `gen_server` and, if present, some client module into one bigger module by little effort.

the proof system. Since these transition rules are of a general nature, they can also be used for implementing our approach in other tools supporting Erlang. The actual implementation of these transition rules in the tool is described in Sect. 4.

3.1 Extending the Erlang Verification Tool

An Erlang system is specified by a composition of *processes*, each represented as $\langle e, pid, q\rangle$, where e is the Erlang *expression* being evaluated, pid is the uniquely defined *process identifier*, and q is the *mailbox queue* in which incoming messages are stored. In order to have the tool recognize a generic server function call, we add these as special syntactic constructs to the set of expressions:

$$\begin{aligned} e ::= \ldots \mid\ & \texttt{gen\_server:start}(e_1, e_2, e_3) \mid \texttt{gen\_server:call}(e_1, e_2) \mid \\ & \texttt{gen\_server:cast}(e_1, e_2) \mid \texttt{gen\_server:reply}(e_1, e_2) \mid \\ & \texttt{gen\_server:wait}(e) \mid \texttt{gen\_server:ready}(e) \mid \\ & \texttt{gen\_server:busy}(e_1, e_2) \mid \texttt{gen\_server:down}(e_1, e_2, e_3) \mid \\ & \texttt{gen\_server:down}(e) \end{aligned}$$

In this way, those function calls can be treated in a different way than the other function calls. The standard method is to search for a definition of the function, to substitute the arguments, and to continue with evaluating the body of the definition. The way the special function calls are treated is defined by an extension of the operational semantics which is defined by labeled transition rules (in the style of [Fre]).

A *reduction context* is an Erlang expression $r[\cdot]$ with a 'hole' $\cdot$ in it, which identifies the position of r where the next evaluation step takes place. In this way, the rules for the actual expression evaluation have to be given only for exceptional cases, namely, when all parameters of an expression construct are *values*, i.e., have been fully evaluated.

For example, process creation is formally described by the following rule[3]:

$$\langle r[\texttt{spawn}(f, [v_1, \ldots, v_n])], pid, q\rangle \xrightarrow{\tau} \langle r[pid'], pid, q\rangle \parallel \langle f(v_1, \ldots, v_n), pid', \varepsilon\rangle \qquad (1)$$

Here, f is a (function) atom, $v_1, \ldots, v_n$ are values, q is an arbitrary mailbox and ε denotes the empty mailbox. Thus, a process evaluating a `spawn` function call has a transition to a two–process system ($\parallel$ denotes parallel composition) which has to evaluate the expressions $r[pid']$ (pid' is the return value of `spawn`) and $f(v_1, \ldots, v_n)$, respectively. For the process identifiers pid and pid' we require $pid' \neq pid$.

Assuming now that the set of reduction contexts has been extended accordingly to cope with the new syntactic constructs, we can formalize the intuitive meaning of the `gen_server` functions as described in the previous section as follows. Here we describe the starting of a server process and the handling of a server call; the complete specification is given in Appendix A.

Starting a server is similar to spawning a process, but the continuation of the process depends on the evaluation of the `init` callback function. This is the reason for adding the

[3] Actually, in the definition of the semantics a two–layer scheme is employed which separates expression–level from process–level steps. We will consider this distinction in Section 4 in greater detail.

special $\texttt{gen\_server:wait}$ construct:

$$\begin{array}{rl} & \langle r[\texttt{gen\_server:start}(mod, arg, opt)], pid, q\rangle \\ \xrightarrow{\tau} & \langle r[\texttt{gen\_server:wait}(spid)], pid, q\rangle \parallel \langle \texttt{init}(arg), spid, \varepsilon\rangle \end{array} \tag{2}$$

Here, *spid* denotes a fresh (server) pid, and the server is created with an empty mailbox. The term $\texttt{gen\_server:wait}(spid)$ should not be treated as a normal form, since then reductions in the context $r[\cdot]$ would be allowed, but rather as a construct from which currently no transitions are possible (similar to a `receive` statement with an empty mailbox).

According to the generic server description, the result of evaluating the `init` function should be a tuple with the state as its second argument. This state should be kept as part of the looping server (looping over: receiving a request, computing the answer and the next state, and responding).

$$\begin{array}{rl} & \langle r[\texttt{gen\_server:wait}(spid)], pid, q\rangle \parallel \langle \{\texttt{ok}, state\}, spid, sq\rangle \\ \xrightarrow{\tau} & \langle r[\{\texttt{ok}, spid\}], pid, q\rangle \parallel \langle \texttt{gen\_server:ready}(state), spid, sq\rangle \end{array} \tag{3}$$

Note that the identifier of the process that started the server is not known to the server in this specification. Since pids of newly created processes are unique, this causes no problem in our setting. The starting process 'remembers' which server it has started (by the obtained process identifier *spid*).

A call by a client can be handled by the server if it is in an idle state, denoted by the `gen_server:ready` construct. In this case, the server process invokes the `handle_call` callback function, and the client process is put into a waiting state until the request has been answered. Now, however, the server needs to store the pid of the calling client in order to be able to distinguish clients if several of them wait for the same server:

$$\begin{array}{rl} & \langle r[\texttt{gen\_server:call}(spid, req)], pid, q\rangle \parallel \langle \texttt{gen\_server:ready}(state), spid, sq\rangle \\ \xrightarrow{\tau} & \langle r[\texttt{gen\_server:wait}(spid)], pid, q\rangle \parallel \\ & \langle \texttt{gen\_server:busy}(\texttt{handle\_call}(req, pid, state), pid), spid, sq\rangle \end{array} \tag{4}$$

If the `handle_call` function yields a triple of the form $\{\texttt{reply}, answer, state\}$, then this *answer* is immediately returned to the waiting client, and the server changes into the idle state again:

$$\begin{array}{rl} & \langle r[\texttt{gen\_server:wait}(spid)], pid, q\rangle \parallel \\ & \langle \texttt{gen\_server:busy}(\{\texttt{reply}, answer, state\}, pid), spid, sq\rangle \\ \xrightarrow{\tau} & \langle r[answer], pid, q\rangle \parallel \langle \texttt{gen\_server:ready}(state), spid, sq\rangle \end{array} \tag{5}$$

The fact that the process identifier is stored in the second argument in the server expression, guarantees that the reply is received by the right waiting client.

As can be seen in our locker example, the `handle_call` may also return a tuple of the form $\{\texttt{noreply}, state\}$. In this case the client process stays in the waiting state (and does not have to be considered therefore), whereas the server becomes idle again:

$$\begin{array}{rl} & \langle \texttt{gen\_server:busy}(\{\texttt{noreply}, state\}, pid), spid, sq\rangle \\ \xrightarrow{\tau} & \langle \texttt{gen\_server:ready}(state), spid, sq\rangle \end{array} \tag{6}$$

Note that when we have two clients, one suspended and one calling the server, then both clients are in the wait state, but only one client can be activated by the return of the `handle_call`, viz. the process which called. The other process must be activated by explicitly using the `gen_server:reply` function:

$$\begin{aligned} & \langle r[\texttt{gen\_server:wait}(\mathit{spid})], \mathit{pid}, q\rangle \parallel \langle r'[\texttt{gen\_server:reply}(\mathit{pid}, \mathit{answer})], \mathit{spid}, \mathit{sq}\rangle \\ \xrightarrow{\tau} & \langle r[\mathit{answer}], \mathit{pid}, q\rangle \parallel \langle r'[\texttt{true}], \mathit{spid}, \mathit{sq}\rangle \end{aligned} \tag{7}$$

In this way semantical rules can be used to accurately describe the given example of the locker server. As can be seen, the asynchronous communication actions that are used in the `gen_server` module to implement synchronous message passing are abstractly represented by simple handshaking operations which do not consider the message queues of the client nor of the server.

Asynchronous communication, not on the level of Erlang, but on the level of `gen_server` is also supported. This is implemented via the `gen_server:cast` and `handle_cast` functions. The `gen_server:cast` mechanism is formalized similar to the `gen_server:call` mechanism. The only difference is that the client immediately proceeds without waiting for a server response. Having evaluated the `handle_cast` function (which is indicated by a `noreply` result tuple), the server just changes into the `gen_server:ready` state, modifying its local data according to the result.

Apart from the `reply` and `noreply` values, `handle_call` (and `handle_cast`) can instead return a result that indicates that the server has to terminate. If so, the `terminate` function in the callback module is invoked. In this situation, the response is stored on the server side until `terminate` has finished:

$$\begin{aligned} & \langle \texttt{gen\_server:busy}(\{\texttt{stop}, \mathit{reason}, \mathit{answer}, \mathit{state}\}, \mathit{pid}), \mathit{spid}, \mathit{sq}\rangle \\ \xrightarrow{\tau} & \langle \texttt{gen\_server:down}(\texttt{terminate}(\mathit{reason}, \mathit{state}), \mathit{pid}, \mathit{answer}), \mathit{spid}, \mathit{sq}\rangle \end{aligned} \tag{8}$$

The `terminate` function is supposed to return the value `ok` and after that, the client is released and the server process is removed:

$$\begin{aligned} & \langle r[\texttt{gen\_server:wait}(\mathit{spid})], \mathit{pid}, q\rangle \parallel \langle \texttt{gen\_server:down}(\texttt{ok}, \mathit{pid}, \mathit{answer}), \mathit{spid}, \mathit{sq}\rangle \\ \xrightarrow{\tau} & \langle r[\mathit{answer}], \mathit{pid}, q\rangle \end{aligned} \tag{9}$$

Note that the callback functions such as `init` and `handle_call` are specified in the callback module. We use the rules provided by the system to reason about their behaviour. Thus, abstraction by means of the semantical rules is only provided for the generic part of the server.

4 Implementation

As mentioned in the introduction, our `gen_server` verification approach has been implemented using the Erlang Verification Tool. Some minor additions had to be made to the tool itself, basically recognizing the special constructs and leaving their handling to us. Thus, whenever a special construct occurs, the tool is aware of the fact that this is not a normal form, but that transitions may arise from this term. It leaves it to the user to check which transitions are possible. We, as a user, provided tactics to analyze the term and evaluate the right transition. These tactics are combinations of proof rules that are applied to the proof goal. Hence, we specified the transition rules given in Sect. 3 as logical formulae.

This approach by specifying the transition rules as logical formulae has two advantages: first of all, the reasoning within the tool remains sound with respect to the abstract `gen_server` semantics. By using an abstract model of a server one introduces a potential unsoundness. If the property that one wants to prove depends on the actual implementation of the server, one might be able to falsely prove it for the abstraction, whereas it does not hold for the real program (this point is discussed a little further in the conclusions). By using logical rules for the transitions, one explicitly states in the assumptions how one expects the server to behave. Since the reasoning that involves these assumptions is based on the (sound) EVT proof system, soundness is guaranteed under the premise that the (low–level) implementation of the `gen_server` module behaves as described by the (high–level) specification.

Second, one obtains a greater flexibility for experimenting. It is easier to change a logical expression and a tactic than to modify the implementation of EVT itself. Moreover, several different specifications of the server may exist at the same time, all using the same tool, and one can also easily add support for other generic libraries without having to change the EVT source code.

In the following we give the logical representation of the transition rules which describe the starting of a server process. As mentioned in Sect. 3, the formal semantics of Erlang is given by a two–layer scheme, which is also used in the EVT implementation. First the Erlang expressions are provided with a semantics on the expression level. The actions here are a functional computation step, an output, a receiving of a message, and a call of a builtin function (like `spawn` for process creation) with side effects on the process–level state. Second, the transition behaviour of Erlang systems (that is, concurrent processes evaluating expressions in the context of a unique process identifier and a mailbox of incoming messages) is captured through a set of transition rules which lift the expression actions to the process level, and which describe the interleaving of concurrent actions. Here, possible process actions are computation steps, input, and output actions.

In this setting, (1) is decomposed into an expression–level rule which indicates the spawning action, and a process–level rule which models the actual process creation:

$$\mathtt{spawn}(f, [v_1, \ldots, v_n]) \xrightarrow{\mathtt{spawn}(f,[v_1,\ldots,v_n]) \to pid'} pid'$$

$$\frac{e \xrightarrow{\mathtt{spawn}(f,[v_1,\ldots,v_n]) \to pid'} e' \qquad pid' \neq pid}{\langle e, pid, q\rangle \xrightarrow{\tau} \langle e', pid, q\rangle \parallel \langle f(v_1, \ldots, v_n), pid', \varepsilon\rangle}$$

These transition rules in the tool automatically generate possible next states of an Erlang system when we want to prove that something holds in a certain next state or in all next states (diamond or box modality). Thus, given a `spawn` call in the program, we can reason about the state in which a new process is created, among the other possible next states that the tool computes for us. For the generic server behaviour we want to obtain a similar level of comfort. However, now we specify the transitions as logical formulae. For example, (2) gives rise to the following logical formula which describes the starting of a server.

$$\begin{array}{l}\forall mod : \mathtt{Atom}.\ \forall arg : \mathtt{Value}.\ \forall opt : \mathtt{List}.\\ \left(\begin{array}{l}\forall spid : \mathtt{Pid}.\\ \mathtt{gen\_server{:}start}(mod, arg, opt) \xrightarrow{\mathtt{spawn}(\mathtt{init},[arg])\to spid} \mathtt{gen\_server{:}wait}(spid)\end{array}\right) \wedge\\ \left(\begin{array}{l}\forall a : \mathtt{Action}.\ \forall e : \mathtt{Expr}.\ \mathtt{gen\_server{:}start}(mod, arg, opt) \xrightarrow{a} e \Rightarrow\\ \exists spid : \mathtt{Pid}.\ a = \mathtt{spawn}(\mathtt{init}, [arg]) \to spid \wedge e = \mathtt{gen\_server{:}wait}(spid)\end{array}\right)\end{array}$$

By associating a `spawn` action with the `gen_server:start` function call, we employ the process–level rule for `spawn` as given above for computing the next state. The server starts evaluating the `init` function[4], which may involve some standard reasoning with the tool. In Appendix B, the corresponding property is named `start_dia`.

However, since we now specify the transition as a logical rule, we also need to state that no other transition is possible from this point, as expressed by the second subformula of the conjunction (which is called `start_box` in Appendix B). The implementation of our generic server primitives in the tool is such that it presents all known actions at every point where one of these primitives may enable a transition. We manually have to prove that a certain transition is possible and that the others are not. The information needed for this proof is provided in the assumptions by means of a logical formula as presented above.

Since we try to achieve a high degree of automation in our proofs, we provided tactics for automatically proving that a certain action would take place and that the others are not possible. For example, when proving a diamond property for an expression where a server is started, the tactics scan in the assumptions of the goal for a property named `start_dia`, and this property is used to automatically prove that after the `gen_server:start` call a new process is created evaluating the `init` function. Similarly, `start_box` is used to establish a property to hold in all successors of the starting state.

Rule (3) expresses that, after the successful initialization, the server pid is returned to the spawning process. Since it makes a provision on the syntactic structure of the two expressions computed by the processes, we have to give four formulae in the EVT specification. The first two (`init_dia`/`init_box` and `wait_dia`/`wait_box`) specify the local effects of sending/receiving a message by/from the server, respectively.

$$\begin{array}{l}\forall state : \mathtt{Value}.\\ \left(\begin{array}{l}\forall spid : \mathtt{Pid}.\ \forall pid : \mathtt{Pid}.\\ \{\mathtt{ok}, state\} \xrightarrow{\mathtt{server}(spid,pid)\to\{\mathtt{ok},spid\}} \mathtt{gen\_server{:}ready}(state)\end{array}\right) \wedge\\ \left(\begin{array}{l}\forall a : \mathtt{Action}.\ \forall e : \mathtt{Expr}.\ \{\mathtt{ok}, state\} \xrightarrow{a} e \Rightarrow \exists spid : \mathtt{Pid}.\ \exists pid : \mathtt{Pid}.\\ a = \mathtt{server}(spid, pid) \to \{\mathtt{ok}, spid\} \wedge e = \mathtt{gen\_server{:}ready}(state)\end{array}\right)\end{array}$$

$$\begin{array}{l}\forall spid : \mathtt{Pid}.\\ \left(\forall v : \mathtt{Value}.\ \mathtt{gen\_server{:}wait}(spid) \xrightarrow{\mathtt{client}(spid)\to v} v\right) \wedge\\ \left(\begin{array}{l}\forall a : \mathtt{Action}.\ \forall e : \mathtt{Expr}.\ \mathtt{gen\_server{:}wait}(spid) \xrightarrow{a} e \Rightarrow\\ \exists v : \mathtt{Value}.\ a = \mathtt{client}(spid) \to v \wedge e = v\end{array}\right)\end{array}$$

[4] To be precise the function *mod*:`init` is called, but the tool, at the moment, lacks support for function calls in remote modules, such that we have to localize all calls.

Furthermore the expression–level synchronization actions have to be lifted to process level. We only give the formula for the server side (server_dia/server_box); the dual one (client_dia/client_box) defining complementary actions of the form $spid!\texttt{sync}(pid, v)$) is of a similar shape.

$$\forall e : \texttt{Expr}.\ \forall e' : \texttt{Expr}.\ \forall spid : \texttt{Pid}.\ \forall pid : \texttt{Pid}.\ \forall v : \texttt{Value}.\ \forall sq : \texttt{Queue}.$$
$$\left(e \xrightarrow{\texttt{server}(spid,pid)\rightarrow v} e' \Rightarrow \langle e, spid, sq\rangle \xrightarrow{spid?\texttt{sync}(pid,v)} \langle e', spid, sq\rangle\right) \wedge$$
$$\left(\langle e, spid, sq\rangle \xrightarrow{spid?\texttt{sync}(pid,v)} \langle e', spid, sq\rangle \Rightarrow e \xrightarrow{\texttt{server}(spid,pid)\rightarrow v} e'\right)$$

Now the standard synchronization mechanism implemented in EVT is used to model the actual communication between the two sync events. Note that both the client and the server pid is used to match the synchronization actions.

Similar formulae are obtained for (4) to (9) describing the synchronous gen_server:call mechanism and the termination of a server process, and for the remaining gen_server constructs. All single properties are collected in a big conjunction named gen_server_spec, which completely specifies the abstract behaviour of gen_server systems, and which can be found in Appendix B.

By implementing the transition rules as logical formulae we gain a flexible and easily adaptable extension of the tool. However, by the fact that we also need the 'negative' information that a certain transition is the only possible, we have some overhead in disproving extra generated subgoals for all other types of transitions. These subgoals are, however, relatively easy to reject and we implemented tactics to deal with them automatically.

For example, when proving a property of the locker server with two clients, we may derive the following goal (where the first client has already been granted a request and is now ready to release and the second client is requesting the resource):

$$\begin{aligned}\texttt{gen\_server\_spec} \vdash\ & \langle \texttt{gen\_server:ready}([pid_1]), spid, sq\rangle \parallel \\ & \langle \texttt{gen\_server:call}(spid, \texttt{release}), pid_1, q_1\rangle \parallel \\ & \langle \texttt{gen\_server:call}(spid, \texttt{request}), pid_2, q_2\rangle : \Box\phi\end{aligned}$$

The formula $\Box\phi$ has to be shown, which means that we should prove ϕ in all possible next states. By using the tactics, we automatically derive the two next states of this system, viz. either the server accepts the release or the request.

We have experimented with our implementation of the generic server specifications applying it to some small examples. Indeed the proofs are easier and shorter than without using this extension. On larger examples this benefit can only become more significant.

5 Conclusions and Future Work

In this article we presented an addition to the Erlang Verification Tool that enables us to reason on a higher level about the code implementing a client–server architecture using the generic server paradigm. The addition drastically reduces the amount of details one needs to consider when proving a property of an Erlang application that uses this client–server architecture, therefore resulting in shorter proofs.

We formalized the operational semantics of the generic server behaviour similar to the way in which the operational semantics of more primitive Erlang functions is specified. For implementing this formalization we defined it in terms of logical formulae such that the only change in the tool that we required was the extension of the list of recognizable keywords by the `gen_server`–specific function names. It turned out that, in comparison to proofs based on the concrete implementation, the logical formulae support a more effective reasoning about client–server systems. The efficiency of this reasoning has been increased by adding tactics that automatically prove subgoals about impossible alternative transitions.

By specifying only the 'essential' behaviour of any reasonable `gen_server` implementation, we introduce a certain unsoundness in the proofs. In other words, if we succeed with our approach to prove a certain property of an Erlang program that contains servers implemented by the `gen_server` module, then it depends on the property whether it really holds for the actual code. This, however, is a consequence of abstraction and not at all a problem in practice. First of all we are not so much interested in proving correctness but rather in finding errors in the program. If we find an error with respect to this abstraction, it is most likely an error in the real code as well. Second, if the property is independent of the specification of the generic server, then the property should hold for the actual code. Since we abstract several server steps into one handshake operation in our semantics, the property should be at least τ*–insensitive*, i.e., its validity should not depend on the number of internal actions the system evaluates. The τ–insensitivity is a minimal requirement of the property, but insufficient, since there are several other issues involved as well that make it very hard to formalize the exact independence criteria. For example, the property should be independent of: the way the messages are provided by a unique tag, the number of messages in the mailbox of a server, the priority used to read messages from a mailbox, the debug and fault tolerance additions not specified by us, etc.

Pragmatically, our concern is to provide a framework in which we can prove properties of the code in an abstract setting, where we use one abstraction for all possible properties. This abstraction is very close to the real implementation, but there will always exist properties for which it turns out to be too general. However, if we can prove a certain property about the abstraction, then we increased the level of confidence in the code; if we find that a certain property does not hold by reasoning in this abstracted setting, then, most likely, this corresponds to an error in the real program. Our technique is therefore rather close to the model–checking approach. However, we only need one abstraction for arbitrary properties and we do not need to abstract over unbounded data structures, dynamic process spawning, or dynamic network creation. Our approach obtains the abstraction automatically, but needs human assistance in non–trivial proofs, whereas the latter can often automatically be handled when using a model checker.

In order to compare efforts, we experiment with using the same formalization of the behaviour of the generic server and its callback module with model–checking tools (such as Truth/SLC [LLNT99]). Since one possible source of infinite state spaces, the unbounded message queue of an Erlang process, is abstracted away in our model, this approach should potentially be more successful. However, we can still apply those tools only to examples where the state space is finite; in particular, the number of processes must be bounded.

The Open Telecom Platform contains several additional generic architectures, such as generic finite–state machines, generic event handlers, generic supervision trees, etc. Those concepts can be formalized and added to EVT along the same lines. A major part of the tactics that we have already written would directly be useful for those other generic concepts. In this way, with only little extra effort, the verification of even more realistic large applications can be simplified.

Acknowledgement

We thank Lars–åke Fredlund for his support in making the necessary changes in the tool and explaining and helping us with the implementation of the tactics.

References

[ADFG98] T. Arts, M. Dam, L.–å. Fredlund, and D. Gurov. System description: Verification of distributed Erlang programs. In *Proc. CADE'98*, volume 1421 of *Lecture Notes in Artificial Intelligence*, pages 38–41. Springer–Verlag, 1998.

[AVWW96] J.L. Armstrong, S.R. Virding, M.C. Williams, and C. Wikström. *Concurrent Programming in Erlang.* Prentice Hall International, 2nd edition, 1996.

[CGL94] E.M. Clarke, O. Grumberg, and D.E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, 1994.

[CW96] E.M. Clarke and J.M. Wing. Formal methods: State of the art and future directions. Technical Report CMU–CS–96–178, Carnegie Mellon University, Pittsburg, USA, 1996.

[DFG98] M. Dam, L.-å. Fredlund, and D. Gurov. Toward parametric verification of open distributed systems. In *Compositionality: the Significant Difference*, volume 1536 of *Lecture Notes in Computer Science*, pages 150–185. Springer–Verlag, 1998.

[Fre] L.-å. Fredlund. Towards a semantics for Erlang. Unpublished manuscript. Swedish Institute of Computer Science.

[Huc99] F. Huch. Verification of Erlang programs using abstract interpretation and model checking. *ACM SIGPLAN Notices*, 34(9):261–272, 1999. Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '99).

[LLNT99] M. Lange, M. Leucker, T. Noll, and S. Tobies. Truth – a verification platform for concurrent systems. In *Tool Support for System Specification, Development, and Verification*, Advances in Computing Science. Springer–Verlag Wien New York, 1999.

[OTP] Open Telecom Platform (OTP). Ericsson Utvecklings AB, `http://www.erlang.org/documentation/doc/index.html`.

[SDL93] CCITT Specification and Description Language (SDL). Technical Report 03/93, International Telecommunication Union, 1993. `http://www.itu.int/`.

A Complete Transition Rule Set for `gen_server`

server start (*spid* fresh):

$$\langle r[\texttt{gen\_server:start}(mod, arg, opt)], pid, q\rangle$$
$$\longrightarrow \langle r[\texttt{gen\_server:wait}(spid)], pid, q\rangle \parallel \langle \texttt{init}(arg), spid, \varepsilon\rangle$$

server started:

$$\langle r[\texttt{gen\_server:wait}(spid)], pid, q\rangle \parallel \langle \{\texttt{ok}, state\}, spid, sq\rangle$$
$$\longrightarrow \langle r[\{\texttt{ok}, spid\}], pid, q\rangle \parallel \langle \texttt{gen\_server:ready}(state), spid, sq\rangle$$

initialization failed:

$$\langle r[\texttt{gen\_server:wait}(spid)], pid, q\rangle \parallel \langle \{\texttt{stop}, reason\}, spid, sq\rangle$$
$$\longrightarrow \langle r[\{\texttt{error}, reason\}], pid, q\rangle$$

server call:

$$\langle r[\texttt{gen\_server:call}(spid, req)], pid, q\rangle \parallel \langle \texttt{gen\_server:ready}(state), spid, sq\rangle$$
$$\longrightarrow \begin{array}{l}\langle r[\texttt{gen\_server:wait}(spid)], pid, q\rangle \parallel \\ \langle \texttt{gen\_server:busy}(\texttt{handle\_call}(req, pid, state), pid), spid, sq\rangle\end{array}$$

implicit server reply:

$$\langle r[\texttt{gen\_server:wait}(spid)], pid, q\rangle \parallel$$
$$\langle \texttt{gen\_server:busy}(\{\texttt{reply}, answer, state\}, pid), spid, sq\rangle$$
$$\longrightarrow \langle r[answer], pid, q\rangle \parallel \langle \texttt{gen\_server:ready}(state), spid, sq\rangle$$

reply delayed or `gen_server:cast/handle_info` handled:

$$\langle \texttt{gen\_server:busy}(\{\texttt{noreply}, state\}, pid), spid, sq\rangle$$
$$\longrightarrow \langle \texttt{gen\_server:ready}(state), spid, sq\rangle$$

explicit server reply:

$$\langle r[\texttt{gen\_server:wait}(spid)], pid, q\rangle \parallel \langle r'[\texttt{gen\_server:reply}(pid, answer)], spid, sq\rangle$$
$$\longrightarrow \langle r[answer], pid, q\rangle \parallel \langle r'[\texttt{true}], spid, sq\rangle$$

server termination (upon call):

$$\langle \texttt{gen\_server:busy}(\{\texttt{stop}, reason, answer, state\}, pid), spid, sq\rangle$$
$$\longrightarrow \langle \texttt{gen\_server:down}(\texttt{terminate}(reason, state), pid, answer), spid, sq\rangle$$

server terminated:

$$\langle r[\texttt{gen\_server:wait}(spid)], pid, q\rangle \parallel \langle \texttt{gen\_server:down}(\texttt{ok}, pid, answer), spid, sq\rangle$$
$$\longrightarrow \langle r[answer], pid, q\rangle$$

server cast:

$$\langle r[\texttt{gen\_server:cast}(spid, req)], pid, q\rangle \parallel \langle \texttt{gen\_server:ready}(state), spid, sq\rangle$$
$$\longrightarrow \langle r[\texttt{ok}], pid, q\rangle \parallel \langle \texttt{gen\_server:busy}(\texttt{handle\_cast}(req, state), pid), spid, sq\rangle$$

server termination (upon `gen_server:cast/handle_info`):

$$\langle \texttt{gen\_server:busy}(\{\texttt{stop}, \mathit{reason}, \mathit{state}\}, \mathit{pid}), \mathit{spid}, \mathit{sq}\rangle$$
$$\longrightarrow \langle \texttt{gen\_server:down}(\texttt{terminate}(\mathit{reason}, \mathit{state})), \mathit{spid}, \mathit{sq}\rangle$$

message handling:

$$\langle \texttt{gen\_server:ready}(\mathit{state}), \mathit{spid}, \mathit{msg} \cdot \mathit{sq}\rangle$$
$$\longrightarrow \langle \texttt{gen\_server:busy}(\texttt{handle\_info}(\mathit{msg}, \mathit{state}), \mathit{pid}), \mathit{spid}, \mathit{sq}\rangle$$

B Complete Logical Specification of gen_server

```
/*
  gen_server.pde: EVT specification of gen_server behavior
  Author: Th. Noll
  Load with 'read_formula_defs("gen_server.pde");'
*/


/* 1) "positive" behaviour (for existential/diamond properties) */

/* 1)a) client-side actions */

/* start server */
start_dia: prop =
  forall Mod: erlangAtom. /* callback module */
  forall Arg: erlangValue. /* initialization parameters */
  forall Opt: erlangList. /* list of options */
  forall SPid: erlangPid.  /* server pid */
  ( start(Mod, Arg, Opt) --spawn(init, [Arg])~>{result, SPid}-->
    wait(SPid)
  );

/* server call */
call_dia: prop =
  forall SPid: erlangPid. /* server pid */
  forall Req: erlangValue. /* request */
  forall CPid: erlangPid. /* client pid */
  ( call(SPid, Req) --client(SPid)~>{call, Req, CPid}--> wait(SPid) );

/* server cast */
cast_dia: prop =
  forall SPid: erlangPid. /* server pid */
  forall Req: erlangValue. /* request */
  ( cast(SPid, Req) --client(SPid)~>{cast, Req}--> ok );

/* wait for server return value */
wait_dia: prop =
```

```
  forall SPid: erlangPid. /* server pid */
  forall V: erlangValue. /* server return value */
  ( wait(SPid) --client(SPid)~>V--> V );

/* raise client-side synchronization actions to system level */
client_dia: prop =
  forall E: erlangExpression. /* initial expression */
  forall F: erlangExpression. /* successor expression */
  forall CPid: erlangPid. /* client pid */
  forall SPid: erlangPid. /* server pid */
  forall Q: erlang_queue. /* message queue */
  forall V: erlangValue. /* transmitted value */
  ( E --client(SPid)~>V--> F =>
    proc<E, CPid, Q> --SPid!synch(CPid, V)--> proc<F, CPid, Q>
  );


/* 1)b) server-side actions */

/* server initialized */
init_dia: prop =
  forall State: erlangValue. /* server state */
  forall SPid: erlangPid. /* server pid */
  forall CPid: erlangPid. /* client pid */
  ( {ok, State} --server(SPid, CPid)~>{ok, SPid}--> ready(State) );

/* initialization failed */
stopinit_dia: prop =
  forall Reason: erlangValue. /* stop reason */
  forall SPid: erlangPid. /* server pid */
  forall CPid: erlangPid. /* client pid */
  ( {stop, Reason} --server(SPid, CPid)~>{error, Reason}--> true );

/* process call request */
readycall_dia: prop =
  forall State: erlangValue. /* server state */
  forall SPid: erlangPid. /* server pid */
  forall CPid: erlangPid. /* client pid */
  forall Req: erlangValue. /* request */
  ( ready(State) --server(SPid, CPid)~>{call, Req, CPid}-->
    busy(handle_call(Req, CPid, State), CPid)
  );

/* process cast request */
```

```
readycast_dia: prop =
  forall State: erlangValue. /* server state */
  forall SPid: erlangPid. /* server pid */
  forall CPid: erlangPid. /* client pid */
  forall Req: erlangValue. /* request */
  ( ready(State) --server(SPid, CPid)~>{cast, Req}-->
    busy(handle_cast(Req, State), CPid)
  );

/* process other message */
readyinfo_dia: prop =
  forall State: erlangValue. /* server state */
  forall Msg: erlangValue. /* message */
  forall CPid: erlangPid. /* client pid */
  ( ready(State) --readq(eps, Msg)-->
    busy(handle_info(Msg, State), CPid)
  );

/* implicit reply after call */
answer_dia: prop =
  forall Answer: erlangValue. /* response */
  forall State: erlangValue. /* server state */
  forall SPid: erlangPid.  /* server pid */
  forall CPid: erlangPid. /* client pid */
  ( busy({reply, Answer, State}, CPid)
    --server(SPid, CPid)~>Answer--> ready(State)
  );

/* stop after call */
stopcall_dia: prop =
  forall Reason: erlangValue. /* stop reason */
  forall Answer: erlangValue. /* response */
  forall State: erlangValue. /* server state */
  forall CPid: erlangPid. /* client pid */
  ( busy({stop, Reason, Answer, State}, CPid) --istep-->
    down(terminate(Reason, State), CPid, Answer)
  );

/* terminated after call */
term_dia: prop =
  forall Answer: erlangValue. /* response */
  forall SPid: erlangPid.  /* server pid */
  forall CPid: erlangPid. /* client pid */
  ( down(ok, CPid, Answer) --server(SPid, CPid)~>Answer--> true );
```

```
/* no reply */
noanswer_dia: prop =
  forall State: erlangValue. /* server state */
  ( busy({noreply, State}) --istep--> ready(State) );

/* explicit reply */
reply_dia: prop =
  forall CPid: erlangPid. /* client pid */
  forall Answer: erlangValue. /* response */
  forall SPid: erlangPid.  /* server pid */
  ( reply(CPid, Answer) --server(SPid, CPid)~>Answer--> true );

/* stop after cast */
stopcast_dia: prop =
  forall Reason: erlangValue. /* stop reason */
  forall State: erlangValue. /* server state */
  ( {stop, Reason, State} --istep--> down(terminate(Reason, State)) );

/* raise server-side synchronization actions to system level */
server_dia: prop =
  forall E: erlangExpression. /* initial expression */
  forall F: erlangExpression. /* successor expression */
  forall SPid: erlangPid. /* server pid */
  forall CPid: erlangPid. /* client pid */
  forall V: erlangValue. /* transmitted value */
  forall Q: erlang_queue. /* message queue */
  ( E --server(SPid, CPid)~>V--> F =>
    proc<E, SPid, Q> --SPid?synch(CPid, V)--> proc<F, SPid, Q>
  );


/* 2) "negative" behaviour (for universal/box properties) */

/* 2)a) client-side actions */

/* start server */
start_box: prop =
  forall Mod: erlangAtom. /* callback module */
  forall Arg: erlangValue. /* initialization parameters */
  forall Opt: erlangList. /* list of options */
  forall A: erlang_intaction.
  forall E: erlangExpression.
  ( start(Mod, Arg, Opt) --A--> E =>
```

```
    ( exists SPid: erlangPid.
      A = spawn(init, [Arg])~>{result, SPid} /\ E = wait(SPid)
    )
  );

/* server call */
call_box: prop =
  forall SPid: erlangPid. /* server pid */
  forall Req: erlangValue. /* request */
  forall A: erlang_intaction.
  forall E: erlangExpression.
  ( call(SPid, Req) --A--> E =>
    ( exists CPid: erlangPid.
      A = client(SPid)~>{call, Req, CPid} /\ E = wait(SPid)
    )
  );

/* server cast */
cast_box: prop =
  forall SPid: erlangPid. /* server pid */
  forall Req: erlangValue. /* request */
  forall A: erlang_intaction.
  forall E: erlangExpression.
  ( cast(SPid, Req) --A--> E => A = client(SPid)~>{cast, Req} /\ E = ok );

/* wait for server return value */
wait_box: prop =
  forall SPid: erlangPid. /* server pid */
  forall A: erlang_intaction.
  forall E: erlangExpression.
  ( wait(SPid) --A--> E =>
    ( exists V: erlangValue. A = client(SPid)~>V /\ E = V )
  );

/* raise client-side synchronization actions to system level */
client_box: prop =
  forall E: erlangExpression. /* initial expression */
  forall F: erlangExpression. /* successor expression */
  forall CPid: erlangPid. /* client pid */
  forall SPid: erlangPid. /* server pid */
  forall Q: erlang_queue. /* message queue */
  forall V: erlangValue. /* transmitted value */
  ( proc<E, CPid, Q> --SPid!synch(CPid, V)--> proc<F, CPid, Q> =>
    E --client(SPid)~>V--> F
```

```
  );


/* 2)b) server-side actions */

/* server initialized */
init_box: prop =
  forall State: erlangValue. /* server state */
  forall A: erlang_intaction.
  forall E: erlangExpression.
  ( {ok, State} --A--> E =>
    ( exists SPid: erlangPid. exists CPid: erlangPid.
      A = server(SPid, CPid)~>{ok, SPid} /\ E = ready(State)
    )
  );

/* initialization failed */
stopinit_box: prop =
  forall Reason: erlangValue. /* stop reason */
  forall A: erlang_intaction.
  forall E: erlangExpression.
  ( {stop, Reason} --A--> E =>
    ( exists SPid: erlangPid. exists CPid: erlangPid.
      A = server(SPid, CPid)~>{error, Reason} /\ E = true
    )
  );

/* process request */
ready_box: prop =
  forall State: erlangValue. /* server state */
  forall A: erlang_intaction.
  forall E: erlangExpression.
  ( ready(State) --A--> E =>
    ( /* handle call */
      exists SPid: erlangPid.  /* server pid */
      exists CPid: erlangPid.  /* client pid */
      exists Req: erlangValue.  /* request */
      ( A = server(SPid, CPid)~>{call, Req, CPid} /\
        E = busy(handle_call(Req, CPid, State), CPid)
      )
    ) \/
    ( /* handle cast */
      exists SPid: erlangPid.  /* server pid */
      exists CPid: erlangPid.  /* client pid */
```

```
      exists Req: erlangValue.  /* request */
      ( A = server(SPid, CPid)~>{cast, Req} /\
        E = busy(handle_cast(Req, State), CPid)
      )
    ) \/
    ( /* handle info */
      exists Msg: erlangValue. /* message */
      exists CPid: erlangPid.  /* client pid */
      ( A = readq(eps, Msg) /\
        E = busy(handle_info(Msg, State), CPid)
      )
    )
  );

/* implicit reply after call */
answer_box: prop =
  forall Answer: erlangValue. /* request */
  forall State: erlangValue. /* server state */
  forall CPid: erlangPid.  /* client pid */
  forall A: erlang_intaction.
  forall E: erlangExpression.
  ( busy({reply, Answer, State}, CPid) --A--> E =>
    ( exists SPid: erlangPid. A = server(SPid, CPid)~>Answer /\
      E = ready(State)
    )
  );

/* stop after call */
stopcall_box: prop =
  forall Reason: erlangValue. /* stop reason */
  forall Answer: erlangValue. /* response */
  forall State: erlangValue. /* server state */
  forall CPid: erlangPid.  /* client pid */
  forall A: erlang_intaction.
  forall E: erlangExpression.
  ( busy({stop, Reason, Answer, State}, CPid) --A--> E =>
    A = istep /\ E = down(terminate(Reason, State), CPid, Answer)
  );

/* terminated after call */
term_box: prop =
  forall Answer: erlangValue. /* response */
  forall CPid: erlangPid.  /* client pid */
  forall A: erlang_intaction.
```

```
  forall E: erlangExpression.
  ( down(ok, CPid, Answer) --A--> E =>
    ( exists SPid: erlangPid.
      A = server(SPid, CPid)~>Answer /\ E = true
    )
  );

/* no reply */
noanswer_box: prop =
  forall State: erlangValue. /* server state */
  forall CPid: erlangPid.  /* client pid */
  forall A: erlang_intaction.
  forall E: erlangExpression.
  ( busy({noreply, State}, CPid) --A--> E =>
    A = istep /\ E = ready(State)
  );

/* explicit reply */
reply_box: prop =
  forall CPid: erlangPid. /* client pid */
  forall Answer: erlangValue. /* response */
  forall A: erlang_intaction.
  forall E: erlangExpression.
  ( reply(CPid, Answer) --A--> E =>
    ( exists SPid: erlangPid.
      A = server(SPid, CPid)~>Answer /\ E = true
    )
  );

/* stop after cast */
stopcast_box: prop =
  forall Reason: erlangValue. /* stop reason */
  forall State: erlangValue. /* server state */
  forall CPid: erlangPid.  /* client pid */
  forall A: erlang_intaction.
  forall E: erlangExpression.
  ( busy({stop, Reason, State}, CPid) --A--> E =>
    ( A = istep /\ E = down(terminate(Reason, State)) )
  );

/* raise server-side synchronization actions to system level */
server_box: prop =
  forall E: erlangExpression. /* initial expression */
  forall F: erlangExpression. /* successor expression */
```

```
  forall SPid: erlangPid. /* server pid */
  forall CPid: erlangPid.  /* client pid */
  forall Q: erlang_queue. /* message queue */
  forall V: erlangValue. /* transmitted value */
  ( proc<E, SPid, Q> --SPid?synch(CPid, V)--> proc<F, SPid, Q> =>
    E --server(SPid, CPid)~>V--> F
  );


/* complete specification of gen_server implementation */
gen_server: prop =
  start_dia /\ call_dia /\ cast_dia /\ wait_dia /\ client_dia /\
  init_dia /\ stopinit_dia /\ readycall_dia /\ readycast_dia /\
  readyinfo_dia /\ answer_dia /\ stopcall_dia /\ term_dia /\
  noanswer_dia /\ reply_dia /\ stopcast_dia /\ server_dia
  /\
  start_box /\ call_box /\ cast_box /\ wait_box /\ client_box /\
  init_box /\ stopinit_box /\ ready_box /\ answer_box /\ stopcall_box /\
  term_box /\ noanswer_box /\ reply_box /\ stopcast_box /\ server_box;
```

Aachener Informatik-Berichte

This list shows all technical reports published since 1997. To obtain copies of technical reports please write to: Informatik-Bibliothek, RWTH Aachen, Ahornstr. 55, 52056 Aachen

97-01 *	Jahresbericht 1996
97-02 *	J. Faassen: Using full parallel Boltzmann Machines for Optimization
97-03 *	A. Winter / A. Schürr: Modules and Updatable Graph Views for PROgrammed Graph REwriting Systems
97-04 *	M. Mohnen / S. Tobies: Implementing Context Patterns in the Glasgow Haskell Compiler
97-05	S. Gruner: Schemakorrespondenzaxiome unterstützen die paargrammatische Spezifikation inkrementeller Integrationswerkzeuge
97-06 *	M. Nicola / M. Jarke: Design and Evaluation of Wireless Health Care Information Systems in Developing Countries
97-07 *	P. Hofstedt: Taskparallele Skelette für irregulär strukturierte Probleme in deklarativen Sprachen
97-08 *	D. Blostein / A. Schürr: Computing with Graphs and Graph Rewriting
97-09 *	C.-A. Krapp / B. Westfechtel: Feedback Handling in Dynamic Task Nets

97-10 * M. Nicola / M. Jarke: Integrating Replication and Communication in Performance Models of Distributed Databases

97-13 * M. Mohnen: Optimising the Memory Management of Higher-Order Functional Programs

97-14 * R. Baumann: Client/Server Distribution in a Structure-Oriented Database Management System

97-15 * G. H. Botorog: High-Level Parallel Programming and the Efficient Implementation of Numerical Algorithms

98-01 * Jahresbericht 1997

98-02 * S. Gruner/ M. Nagel / A. Schürr: Fine-grained and Structure-oriented Integration Tools are Needed for Product Development Processes

98-03 * S. Gruner: Einige Anmerkungen zur graphgrammatischen Spezifikation von Integrationswerkzeugen nach Westfechtel, Janning, Lefering und Schürr

98-04 O. Kubitz: Mobile Robots in Dynamic Environments

98-05 * M. Leucker / St. Tobies: Truth — A Verification Platform for Distributed Systems

98-07 * M. Arnold / M. Erdmann / M. Glinz / P. Haumer / R. Knoll / B. Paech / K. Pohl / J. Ryser / R. Studer / K. Weidenhaupt: Survey on the Scenario Use in Twelve Selected Industrial Projects

98-08 H. Aust: Sprachverstehen und Dialogmodellierung in natürlichsprachlichen Informationssystemen

98-09 Th. Lehmann: Geometrische Ausrichtung medizinischer Bilder am Beispiel intraoraler Radiographien

98-10 M. Nicola / M. Jarke: Performance Modeling of Distributed and Replicated Databases

98-11 A. Schleicher / B. Westfechtel / D. Jäger: Modeling Dynamic Software Processes in UML

98-12 W. Appelt / M. Jarke: Interoperable Tools for Cooperation Support using the World Wide Web

98-13 * K. Indermark: Semantik rekursiver Funktionsdefinitionen mit Striktheitsinformation

99-01 Jahresbericht 1998

99-02 F. Huch: Verifcation of Erlang Programs using Abstract Interpretation and Model Checking — Extended Version

99-03 R. Gallersdörfer / M. Jarke / M. Nicola: The ADR Replication Manager

99-04 * M. Alpuente / M. Hanus / S. Lucas / G. Vidal: Specialization of Functional Logic Programs Based on Needed Narrowing

99-07 * Th. Wilke: CTL+ is exponentially more succinct than CTL

99-08 * O. Matz: Dot-Depth and Monadic Quantifier Alternation over Pictures

2000-01 Jahresbericht 1999

2000-02 * Jens Vöge / Marcin Jurdziński: A Discrete Strategy Improvement Algorithm for Solving Parity Games

2000-04 * Andreas Becks, Stefan Sklorz, Matthias Jarke: Exploring the Semantic Structure of Technical Document Collections: A Cooperative Systems Approach

2000-05 Mareike Schoop: Cooperative Document Management

2000-06 Mareike Schoop (Ed.): Proceedings of the Fifth International Workshop on the Language-Action Perspective on Communication Modelling

2000-07 Markus Mohnen / Pieter Koopman (Eds.): Proceedings of the 12th International Workshop of Functional Languages

* These reports are also available electronically as PostScript files
via anonymous FTP on `ftp.informatik.rwth-aachen.de`
or via WWW at URL `ftp://ftp.informatik.rwth-aachen.de/pub/reports/index.html`