Verifying a Distributed Database Lookup Manager Written in Erlang

Thomas Arts[1] and Mads Dam[2]

[1] Computer Science Laboratory, Ericsson Utvecklings AB, 126 25 Stockholm, Sweden, +46 8 7199514, thomas@cslab.ericsson.se, http://www.ericsson.se/cslab/~thomas/

[2] Swedish Institute of Computer Science, Box 1263, S-164 28 Kista, Sweden, mfd@sics.se, http://www.sics.se/~mfd/home.html

Industrial Applications, experience report

Keywords: Telecommunication; Proof Checker; Distributed Algorithms; Distributed Databases; Formal Verification

Abstract. We describe a case-study in which formal methods were used to verify an important responsiveness property of a distributed database system which is used heavily at Ericsson in a number of recent products. One of the aims of the project was to verify the actual running code which is written in the distributed functional language Erlang. In a joint project between SICS and Ericsson we have over the past few years been developing a tableau-based verification tool for Erlang of considerable scope. In particular, we are capable of addressing — on the level of running program code — systems with unbounded behaviour along the many dimensions in which this happens in "real" programs, involving datatypes, recursive control structures, error handling and recovery, initialisation, and dynamic process creation. The database lookup manager considered here contains most of these features, giving rise to infinite state behaviour which is not very adequately handled using model checking or other approaches based purely on state space traversal. In the paper we introduce the case study, our approach to formalisation and verification, and discuss our experiences using the Erlang verification tool.

1 Introduction

Erlang is a functional programming language developed by Ericsson [AVWW96], which is used extensively for writing robust distributed telecommunication applications. Central in many of these applications is a distributed database, Mnesia [Mnesia], also written in Erlang. The Mnesia system is crucial to the robustness of almost all Erlang based product developed at Ericsson. It is, for instance, responsible for error recovery, the prompt and safe handling of which is essential in telecommunication applications. These features make the Mnesia system a rewarding object of study when trying out new verification techniques.

The case study at hand concerns only a small part of the Mnesia system, a protocol for the evaluation of a query which is distributed over several computers in a network. The starting point for this case study was the Erlang code

implementing the distributed database. The author of this code knew that the query lookup protocol was implemented in a tricky way and got interested in supporting his implementation with a clear and verified description.

We extracted, from the real implementation, the code for the distributed query evaluation protocol and added some code to provide a very simple simulated interface to parts of the system that were irrelevant for the problem at hand. The result was an Erlang program that could be seen as a very precise, and in some sense formal, description of the underlying algorithm. Isolation of the code responsible for the lookup mechanism and analysing the intended behaviour of the code resulted, as a side effect, in a clear and patentable picture of the underlying protocol [Nil99].

In Sect. 2 we present the distributed query evaluation in more detail. As input the protocol receives a database query divided into subqueries. These subqueries are distributed over the network in the form of processes on those computers where the specific data for a subquery is stored. By sending messages to the subquery processes, data is extracted from the database tables and sent along the network. One process is responsible for initialising the lookup process ring, and for collecting the resulting data. To avoid excessive delays and storage consumption, query answers are collected in segments, managed by the lookup manager. The task we set ourselves was to prove that the implementation provided a responsiveness property: that input queries are eventually being replied to.

The query lookup manager implements initialisation and query lookup phases in manners which are tightly interwoven. Both these phases are important for correct behaviour. Moreover, the code is evidently designed to cater for tables of arbitrary numbers and sizes, and for queries of arbitrary natures. Reflecting this, our aim was to prove correctness *uniformly* in these parameters, i.e. without fixing numbers and sizes of tables and queries in advance. This sort of problem is outside the scope of model checkers, symbolic or otherwise, or other techniques based purely on global state space traversal.

There are several reasons why we find this sort of verification exercise useful and interesting.

- First of all it is clearly relevant to verify the actual code rather than some abstraction of it, as this gives us more accurate and reliable information about the way the system is going to behave when it is eventually executed[1].
- Secondly, by analysing the code, and in particular, by analysing it in a compositional manner, as we do, we produce verication information which is reusable as the system grows. By contrast, most approximate analyses, such as ones based on abstract interpretation (c.f. [Cri95]), tend to be global ones, not readily reusable.
- Thirdly, and most significantly, the Erlang code itself is in fact already quite abstract, in the sense of providing designers and implementors with a concise set of primitives and language constructs which are efficiently implementable yet not at all far from a process calculus-like level of abstraction.

[1] Absolute accuracy, of course, is unattainable

– Fourthly we have the potential to maintain strong links between running and verified code. For instance, it will very often be possible to update proofs in a fully automatic way after minor code revisions, by reapplying proof tactics.
– As a longer term perspective, we are interested in developing object and component encapsulation techniques for which a code verification capability is essential.

To realize the verification we used a tool [ADFG98] which we are in the process of building, based on an approach to compositional verification which we have developed in some recent papers (c.f. [DFG98]). The approach uses a tightly integrated mix of state-space exploration and proof-editing techniques. System properties and specifications are given in a first-order temporal logic, a variant of Park's μ-calculus [Par76] tailored, in this case, specifically to Erlang. Proof goals are stated as general Gentzen-type sequents, proved in a goal-driven fashion by refinement and loop detection. The result is a very powerful proof system which supports model checking, compositional reasoning, and general coinductive or inductive reasoning, for instance about datatypes, in a uniform framework.

In Sect. 3 we briefly describe our approach to specification. In Sect. 4 the actual verification is described and an outline of the informal proof is presented. Then in Sect. 5 we describe in more detail our approach to formalisation of the proof, and its realisation in the verification tool. Large parts of the proof are easily automatable by tactics that perform model-checking like state exploration, or prove type adherence or termination of sequential functions. Since these tactics are often used within interactively developed proofs, our verification approach gives rise to proofs that easily become large enough (several thousand nodes) for tool support to be essential. We conclude, in Sect. 6, with some final remarks, reflecting on the approach followed and lessons learned from performing this case study.

2 A Process Verification Problem

In this section we explain the mechanism for query lookup and the property we have proved.

Whenever a query is formulated for the distributed database the query is analysed and divided into subqueries each addressing only one table, since the tables in which the requested information is stored are distributed over several computers. The subqueries are distributed over a network as processes located at the computer where the information is available. A request is sent to the first of the spawned processes, which reads data from a table. This results in several partially instantiated queries, which are sent to the next process. For every such partly instantiated query, the next process reads additional data from a table, resulting in further instantiations. The last process gathers all data and sends it to the requesting process. To avoid unnecessary delays in transmission, processing, and database lookup, and to avoid excessive storage consumption, query processing is split into segments.

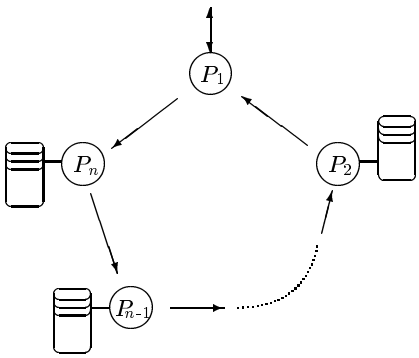

Fig. 1. Ring of processes attached to tables, with P_1 the initial process

We identify an initial process taking care of a query by partioning it into subqueries, represented by Erlang functions, whereafter for every such subquery a process is created on a computer where the subquery can find its information. All spawned processes execute the same function, which have one of the Erlang functions that represents the subquery as an argument. The processes are spawned in a ring configuration and the initial process may be seen as a distinguished member of this ring.

```
query_setup(Query,DBStructure) ->
  SubQueries = split_handle(Query,DBStructure),
  mk_ring(self(),SubQueries).

mk_ring(NextPid,SubQueries) ->
  case SubQueries of
    []->
      wait_for_request(NextPid);
    [Q|Qs] ->
      mk_ring(spawn(process_in_ring,[NextPid,Q,[]]),Qs)
  end.
```

In our approach we abstract from the actual computation of the subqueries and assume that this computation results in a list of functions with at least one element. For every such function a process is created on the appropriate machine, where the name of the machine is computed together with the subquery itself. For readability, we have chosen not to present the machine name and perform the spawning on only one machine. Spawning on several machines is done similarly, where the Erlang spawn primitive needs the machine name as an additional argument.

The function `process_in_ring` is spawned with three arguments, the process identifier (pid) of the next process in the ring, the function representing the subquery, and the empty list representing a local store for the process (see below for more details on this store).

After spawning the ring (Fig. 1), the initial process (P_1) executes the function[2]

```
wait_for_request(NextPid) ->
  receive
    {user_request,UserPid,NrSolutions} ->
      PacketSize = some_value_smaller(NrSolutions),
      NextPid!{[[]],PacketSize},
      counting(NextPid,UserPid,NrSolutions,[])
  end.
```

with as argument the next process in the ring (P_n). Now P_1 is ready to receive a message of the form {`user_request, UserPid, NrSolutions`} where the triple represents an atom `user_request` to identify the message type, the pid of the requesting process and the maximum number of solutions that the latter process wants to receive. Observe that, because of the asynchronous communication discipline of Erlang, a user request may arrive at the mailbox of the initial process long before it is actually processed.

Whenever this message arrives, a message is sent to the consecutive process in the ring (P_n), which is the first process able to perform a subquery lookup. The process P_1 subsequently calls the function `counting`, which collects all answers that the subqueries of the ring produce. The idea is that for all solutions that a process in the ring receives, it computes all new solutions using its subquery lookup function. This might result in an increase or decrease of the number of solutions. These new solutions are passed to the next process and so on, until P_1 receives the answers and can present them to the user.

However, in order not to overload the network, the processes in the ring are not sending all the answers they find, but just a fixed number *PacketSize*, which is dynamically determined by P_1 (via the function `some_value_smaller` where we abstract from the real computation) and depends on the number of requested solutions and the network load. Thus, the number *PacketSize* is sent along in the message from P_1 to the next process P_n in the ring. The latter process computes all answers it can find according to its subquery and sends at most *PacketSize* of these answers to the next process, whereas the remaining answers are kept in the store. All consecutive processes in the ring perform the same actions and eventually P_1 receives at most *PacketSize* answers. The process P_1 may now add these answers to its store and as long as the store is less than the demanded number of answers (`NrSolutions`) a message will be sent to the process P_n requesting to produce new answers.

```
counting(NextPid,UserPid,NrSolutions,Store) ->
  receive
     {Solutions,PacketSize} ->
        NewStore = Solutions ++ Store,
```

[2] In the real code this receive statement is incorporated in the function `mk_ring`, this has been modified for clarity of presentation.

```
        SolutionsToGet = NrSolutions - length(NewStore),
        case {Solutions,SolutionsToGet =< 0} of
          {_,true} ->            % enough solutions found
             UserPid!{user_response,NewStore}
          {[],_} ->              % no more solutions in DB
             UserPid!{user_response,NewStore}
          Otherwise ->
             NextPid!{[],PacketSize},
             counting(NextPid,UserPid,NrSolutions,NewStore)
        end
  end.
```

Except for the initial processes, all other processes in the ring, i.e. $P_2, \ldots, P_n$, are evaluating the function `process_in_ring`.

```
process_in_ring(NextPid,Filter,Store) ->
  receive
     {Solutions,PacketSize} ->
       case PacketSize =< length(Store) of
         true ->
            {ToSend,ToStore} = split(PacketSize,Store),
            NextPid!{ToSend,PacketSize},
            NewStore = ToStore ++ flatmap(Filter,Solutions),
            process_in_ring(NextPid,Filter,NewStore);
         false ->
            NewStore = Store ++ flatmap(Filter,Solutions),
            {ToSend,ToStore} = split(PacketSize,NewStore),
            NextPid!{ToSend,PacketSize},
            process_in_ring(NextPid,Filter,ToStore)
       end
  end.
```

These processes wait for a message containing at most *PacketSize* answers of the previous process and the value *PacketSize* itself. The number of stored answers is compared to the number *PacketSize* of demanded answers and if enough answers are already in the store, these are sent along to the next process and new answers are computed. In case not enough answers are stored, first all new answers are computed, whereafter at most *PacketSize* answers are sent to the next process and all other answers are stored for the next round. Answers are computed using the function `flatmap` which applies the function `Filter` to any partially instantiated query in the list `Solutions`. The function `Filter` has been generated from the original query and the database and was given as an argument of the spawned function. We abstract from this function and only assume that `Filter` is a terminating function that results in a (probably empty) list of arguments. The function `flatmap` results in the concatenation of all lists that result from applying `Filter` to all arguments of `Solutions`, which might

either be a longer or a shorter list than the `Solutions` itself. In this way, the store of the process may increase and decrease dynamically.

The function `split` divides a list in two sublist of which the length of the first list contains the first *PacketSize* elements of the list, provided that *PacketSize* is given as an argument to the function. Functions like `=<` and `++` have their usual meaning. In the verification process these functions are not considered as build-in functions, like they are in Erlang, but are specified separately.

The property that we want to verify is informally described as 'Is the retrieval of the information terminating?' In other words, given an arbitrary query and an arbitrary positive integer, whenever we build a ring corresponding to this query and send a message of the form {`user_request,MyPid,Number`} to the first process in the spawned ring, do we always eventually receive a message back with at most this `Number` of solutions in it?

3 The Specification Logic and its Proof System

It is not completely trivial to come up with a correct formal rendition of property outlined at the end of Sect. 2. A first step is to understand correctly the abstract execution mechanism of Erlang. We gave a core fragment of Erlang, involving, roughly, the features used in the present example, an SOS-style operational semantics. Among the more tricky features to model adequately is communication. In Erlang interprocess communication is asynchronous. Each process is equipped with one mailbox. Sending is non-blocking: The transmitted message is placed at the end of the mailbox belonging to the receiving process. Messages are subsequently read by retrieving the first message in the mailbox matching a given pattern. Since we need to analyse behaviour both at the level of processes and process communication and at the level of sequential function elaboration we are forcing a separation between the time at which a message packet crosses a process boundary (or: enters the schedulers domain, i.e. the process mailbox), and the time at which the packet is read from the mailbox by the receiving process.

A second step is to adequately account for the execution behaviour of processes in a formal property specification language. Our work has been based on a first-order fixed point calculus inspired by Park's μ-calculus [Par76,Koz83], extended with Erlang-specific features. In summary this logic is based on the first-order language of equality, extended with modalities reflecting state transition capabilities, least and greatest fixed points, along with a few additional primitives. Using μ-calculus correctly is by itself well known to be tricky. On the other hand we have found the μ-calculus recursive style of specification extremely natural and useful. We have used an equational style of specification, using the notation

$$prop(args) \Rightarrow body$$

for greatest fixed points (the body can be inferred from the head), and

$$prop(args) \Leftarrow body$$

for least fixed points (the head must be inferred from the body). Whereas this notation is fraught with danger (how are dependencies resolved?) a clear benefit of such a notation is that it encourages a programming language style of specification defining "larger", more complicated properties in terms of "smaller" ones.

The benefits of the equational style of specification becomes apparent, in particular, once properties are decomposed. To do this one typically need to express state, liveness, or safety properties embedded inside another invariant which needs to adequately capture all possible ways in which the processes can interact, and the consequences of these interactions. An example of the shape of property one obtains is (1) below.

A complication which is more semantical than due to the recursive style of specification is Erlang's asynchronous communication. Since receivers are powerless to influence the delivery of packets into receivers mailbox, for the purpose of packet delivery events, and in the absence of a suitable fairness assumption (which we have not so far implemented), it is possible for packet delivery to continuously preempt progress by the local process. In this example we have been able to bypass this problem, as the ring structure enforces a synchrony property that ensures to a sufficient extent that mailboxes do not grow in unbounded manners.

3.1 The logic

Typical Erlang-related primitives are $\mathit{the\_term} = e$ to pick up the Erlang expression associated with the process under evaluation and compare this with the term e; *unevaluated* which is true if the Erlang expression under evaluation is not yet in normal form; and similar primitives for queues and process identifiers with are local or foreign to the system under consideration.

The modal operators $<\cdot>$ and $[\cdot]$ (not to be confused with the Erlang list constructors [] (the nil list) and $[hd|tl]$) are used to express transition capabilities. The formula $<>\phi$ holds if an internal transistion is enabled to a state satisfying ϕ. Similarly, we have a diamond operator for the non-internal transitions for sending and receiving, viz. $<P!V>\phi$ and $<P?V>\phi$. Observe that the receive modality is "appending to recipients mailbox". The box operator is the dual of the diamond operator, expressing that a formula should hold in all states reachable in one transition from the current state.

Using least and greatest fixed point temporal properties likely liveness and safety can easily be expressed. Furthermore simple data types, like lists and natural numbers, can be expressed using least fixed points:

$$\mathit{list}(L) \Leftarrow (L = []) \vee \exists H.\exists T.(\mathit{list}(T) \wedge (L = [H|T]))$$

Combinations of both greatest and least fixed points are used to express the complicated eventuality properties we deal with in this case-study. A representative example of the latter is the formula that expresses that the property *wait_for_input* holds for an arbitrary number of internal computation steps, until a certain shape of message is received and the property *continue* holds. The

properties *wait_for_input* and *continue* will typically be mutually recursive, so let us assume that *wait_for_input* is defined in the context of a definition

$$continue \Rightarrow \cdots (wait\_for\_input) \cdots .$$

Now *wait_for_input* is defined in the following way:

$$\begin{aligned} wait\_for\_input(RightForm) &\Rightarrow wait\_for\_input'(RightForm) \\ wait\_for\_input'(RightForm) &\Leftarrow \Box wait\_for\_input'(RightForm) \ \wedge \\ &\quad \forall P.\forall V.([P!V]false) \ \wedge \\ &\quad \forall P.\forall V.([P?V](RightForm(P,V) \ \wedge \\ &\quad continue)) \end{aligned} \tag{1}$$

The least fixed point ensures that the predicated process does not diverge (i.e. performs an infinite sequence of internal computation steps without ever writing an incoming message to its mailbox. The greatest fixed point on the other hand permits states satisfying *wait_for_input* infinitely often, as long as they are infinitely often separated by *continue* states.

4 Outline of the Proof

According to the informal property as stated in Sect. 2, we are dealing with two actions initiating the query lookup: first the ring is built and thereafter a request message is sent to the first process in this ring. For verification we are focusing on the outcome of the valuation of the Erlang expression:

```
Ring = spawn(query_setup,[Query,DBStructure]),
Ring!{user_request,self(),NrSolutions},
receive
     {user_response,Solutions}
end.
```

where we quantify over all possible values of $Query$, $DBStructure$, $NrSolutions$ and $Solutions$. We abstract from the first two variables by assuming the function `split_handle` to result in a list of functions, where the real interesting issue is the length of this list, which can be any positive integer determining the number of processes in the ring. The property we address in this paper is that evaluation of this Erlang expression is terminating. Similar properties of interest are:

- The number of received answers is equal to the number of demanded answers if that many answers exist in the database.
- The set of obtained answers is independent of the packet size, provided the latter is a positive number.

Given the experience of, e.g., the *wait_for_input* formula (1) formulating the responsiveness property is not too difficult. The specification will have the following

shape:

$$
\begin{aligned}
&\mathit{spec} \Rightarrow \mathit{spec'} \\
&\mathit{spec'} \Leftarrow \Box\mathit{spec'} \ \wedge \ \forall P.\forall V.[P!V]\mathit{false} \ \wedge \\
&\qquad \forall P.\forall V.[P?V]((P = \mathit{userpid}) \ \wedge \\
&\qquad \exists \mathit{From}.\exists N.(V = \{\texttt{user\_request}, \mathit{From}, N\}) \ \wedge \ \phi)
\end{aligned}
$$

where ϕ expresses responsiveness in a similar style, that eventually a user response is sent to the pid *From*, before returning to a state satisfying *spec*. Several details are omitted in this description: Information about process identifiers and the store have to be carried over to the property ϕ, and assumptions concerning the return address *From*, and the types of other arguments have to be made.

The basic style of specification is one of distinguishing abstract states in which (aggregate sets of) processes may find themselves. The abstract states will often correspond to infinitely many actual states of the process. For every process we define a few abstract states and formulate which properties should hold in these states and how one property depends on the other. The processes we consider are the initial process evaluating the given Erlang expression, a ring process (which is not the initial one), and, as part of an inductive argument, and a ring segment which includes the initial process.

4.1 The ring invariant

The basic difficulties in proving the specification to hold are the unbounded number of ring processes which can be created, and the unbounded number of query replies which can be requested. To address these difficulties we resort to induction. We identify two invariants:

1. An invariant to hold of each of the ring processes $P_2, \ldots, P_n$ (c.f. Fig. 2).
2. A sort of structural and temporal invariant for a ring segment of the shape $P_1, P_n, \ldots, P_i$.

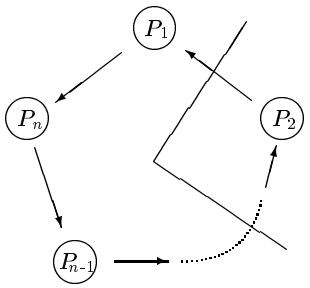

Fig. 2. Induction on number of processes in ring

Let us call the first invariant *proc_wait_for_input* and the second invariant for *rootspec*. We first need to show that *rootspec* is strong enough to derive the end specification we wish to establish, i.e. a sequent of the shape

$$x : \mathit{rootspec}(\cdots) \vdash x : \mathit{spec}(\cdots). \tag{2}$$

The task is thus to prove that *rootspec* holds of the process initially evaluating `query_setup`:

$$\text{some assumptions} \vdash \mathit{proc}(\texttt{query\_setup}(\cdots,\cdots) : \mathit{rootspec}(\cdots) \tag{3}$$

Using straightforward, and fully automatable, state exploration techniques which we return to in the following section we can reduce (3) first to a subgoal of the shape

$$\text{some assumptions} \vdash \mathit{proc}(\texttt{mk\_ring}(\cdots,\cdots) : \mathit{rootspec}(\cdots) \tag{4}$$

and then, by continuing state exploration, to a subgoal of the shape

$$\begin{aligned}\text{some assumptions} \vdash \mathit{proc}(\texttt{mk\_ring}(\cdots,\cdots) \parallel \\ \mathit{proc}(\texttt{process\_in\_ring}(\cdots)) : \mathit{rootspec}(\cdots)\end{aligned} \tag{5}$$

The idea is to prove two lemmas, one stating the correctness of `process_in_ring`,

$$\text{some assumptions} \vdash \mathit{proc}(\texttt{process\_in\_ring}(\cdots)) : \mathit{proc\_wait\_for\_input}(\cdots) \tag{6}$$

and one concerning the composability of *rootspec* with *proc_wait_for_input*,

$$C1 : \mathit{rootspec}(\cdots), C2 : \mathit{proc\_wait\_for\_input}(\cdots) \vdash C1 \parallel C2 : \mathit{rootspec}(\cdots) \tag{7}$$

Subgoal (7) states a compositional property of root and ring processes: putting together a (possibly aggregate) process (P_1) acting as a root with a (possibly aggregate) process acting as a ring element results in an aggregate process which again acts as a root. Obviously the correctness of this statement is crucially dependent on input and outputs being properly connected, which are matters we will not be concerned with here.

By themselves, (6) and (7) are not sufficient to conclude (5). However, using (6) and (7) it is possible to reduce to a goal which is actually an instance of the goal (4), and the remarkable fact is that, in principle, an inductive argument can be set up such that at this point the proof can be completed (c.f. [DFG98]). In realizing this proof, however, a number of complications must be attended to which we return to in Sect. 5.

4.2 Properties of the separate processes

We are thus left with two main subgoals, one of the shape (6), and one of the shape (7). We do not comment further on (7) other than observe that the ring process property *proc_wait_for_input* we are looking for must be strong enough to permit (7) to be proved. Instead we turn to *proc_wait_for_input*.

We start by observing the role of a special token that is initially sent by the first process (P_1) in the ring and implies termination as soon as it is also received by this process, i.e. when the token has gone through the entire ring. This special token ({[],PacketSize}), which we call the *end_token* for convenience, is repeatedly send by P_1 to P_n after initially sending {[[]],PacketSize} once. In case the number of demanded solutions is larger than the number of solutions present in the database, the process P_1 can only respond to the user when this *end_token* is received from the process P_2.

The first process in the ring P_1 plays a special role and the abstract states we distinguish for this process are

1. the process is waiting for a user_request,
2. a non-end_token is sent to the next process (P_n) and the process is waiting for a message,
3. an end_token is sent to the next process and the process is waiting for a message,
4. a non-end_token is received and not enough answers are collected,
5. the end_token is received or another token is received and enough answers are collected.

Our choice to follow the real code and not to abstract from the actual counting of the number of answers, causes the state space of this first process in the ring to be unbounded. For this reason, modelchecking is infeasible for this part of the proof as well, but with our verification tool such a proof can be handled.

States that we distinguish for the processes in the ring are characterized by whether or not they receive an end_token and whether or not they send an end_token. Crucial is the observation that after receiving an end_token once, only end_tokens can be received successively. The latter is a property of the ring and not of the process itself, but when proved for the ring, we use it in our formalization to disallow the state transition from receiving an end_token to receiving a non-end_token.

For a process in the ring $(P_2, \ldots, P_n)$ we define four abstract states, depending again on the end_token:

1. the process awaits the reception of an arbitrary message,
2. the process receives an *end*-message and sends a message to the next process,
3. the process receives a non-*end*-message and sends a message to the next process,
4. the process waits for receiving a successive *end*-message.

Every state is captured in a property, but also the relation to the other abstract states is reflected in this same property using the fixed point operators. The proof boils down to the observation that if the end_token is repeatedly received the process is forced to pass on at least one element of its store. Thus the store becomes smaller and smaller and when empty, the process sends the end_token as well. Note again that a property outside the view of the process in the ring should ensure that after receiving an end_token we cannot receive a non-end_token anymore. This property is hidden in the relation between the properties of consecutive states, but is proved in the more general setting.

5 Proof Search and Automation

The success of our interactive theorem-proving based approach in large-scale applications is heavily dependent on three factors:

1. Robust tactics that help solve and reduce subproblems of clearly identifiable natures.
2. Use of such tactics to the maximal extent possible, to eliminate user intervention whenever possible.
3. A user interface that helps users navigate and assist the theorem proving process in a meaningful way, when such assistance is really required.

To minimize user intervention we adopt as lazy an approach to proof search as we have found possible, using existential variables to delay commitments to existential witnesses, proof goals stated as general Gentzen-type sequents to delay commitments to disjunctive choices, and a lazy approach to induction using loop detection which we have introduced in some recent papers (c.f. [DFG98]).

5.1 Induction and Discharge

As we outlined in the previous section we use a very tightly integrated mix of state-space exploration and proof-editing techniques. As in most proof editors the proof construction process is a goal-driven one: Proof goals in the form of Gentzen-type sequents are refined in steps by the application of one of a number of primitive proof rules.

Most proof goals call for induction (or coinduction) for their proofs. Many types of induction are involved in an example such as the one we consider here:

- Induction on number of evaluation steps.
- Induction on size of data values, such as numbers or lengths of lists.
- Induction on the structure of function expressions.

Induction on the number of execution steps from some initial configuration is typically used if we prove that computing the length of a list results in a natural number, or that comparing two numbers results in a boolean. Coinduction is used, typically, for invariants, by showing that the invariant remains unbroken after any number of computation steps. General programs involve data type operations, communication, and, maybe, dynamic creation of new processes, in manners which are interwoven to considerable extents, as happens in our database lookup manager. To handle these complications, most parts of the proof will involve induction and coinduction at many levels simultaneously, in manners which, when properly formalized, may be exceedingly complicated. Our proof theoretic approach, using loop detection, or discharge, allows very substantial parts of this formalisation to be almost completely hidden from the user. The discharge mechanism implemented in the tool follows the principles laid out in [DFG98]. In effect the discharge mechanism attempts to cast the proof as so far constructed as a proof by simultaneous induction, by seeking an ordering that

makes the dependency relation between induction and coinduction variables a well-founded one. Maintaining the constraints on this dependency ordering is done by the proof editor. Thus there is no need for users to specify the sequence, nesting, or mutual dependencies of simultaneous inductive arguments, or even to state that induction is being used. All this is managed by the tool. However, the user will need to have a basic understanding of the general principles of simultaneous induction for the operation of the discharge rule to be understandable. And, most importantly, the tool has *no* built-in support for finding inductive assertions. Such support can be programmed (as tactics), or must alternatively — as in our case — be provided explicitly.

5.2 Proof Construction

Our proof approach, and the size of problems which we address, gives rise to complications concerning proof sharing and proof construction which we have had to address.

A naive implementation of a proof editor for Erlang quickly runs out of space, because of the large number of independent transitions. Observe that independence is a feature not only in-between processes, but also within a single process independent choices can be viewed as arising between writing an incoming message to the local mailbox, or letting local computation progress. As a consequence, state spaces for even small, single processes grow very significantly. To handle this we implemented an inference rule, *copy_discharge* for subproof sharing to close proof branches in case they are seen to have already been dealt with elsewhere.

Example 1. The Erlang semantics is such that one can always receive a message in the mailbox. Thus, in many properties we state that either an internal action is possible, or the process may receive something in the mailbox. Here the proofnode has two branches, performing the action or receiving the message. After performing the action, one normally should be able to receive the message anyway and after receiving the message, one can still perform the action. Instead of searching for, and constructing, the proof twice, we use copy_discharge to join the nodes. Since this is done recursively, one easily sees that the prooftree would grow exponentialy when we lack this copy_discharge.

Observe that a correct implementation of the copy_discharge feature is computationally quite expensive: to check for circularity, to support "undo", and to interact correctly with existential variables.

To support discharge and, in particular, subproof sharing it seems essential to maintain a "current" proof tree, and have rules of proof elaborate this proof tree through side effects. Observe that this makes the proof construction process very different from that of other proof editing tools (such as PVS [ROS92,Sha96], Coq [DFH$^+$], Lego [Lego], Isabelle [Pau94],...) which maintain only the leaves, but not the internal structure of proof trees. Thus, in these tools one shares subproofs by having the user formulate lemma's which are used for several leaves. We

overcome this user intervention and in case a subproof need not be performed, this is detected automatically.

5.3 Tactics

The construction of proof trees by side effects has drastic impact on the programming of tactics, for instance. The benefit, besides the support of discharge and (in particular) copy-discharge, is that the entire proof tree becomes available for inspection and navigation. In fact, to help keep the information manageable we implemented a facility for suppressing the creation on new nodes. The cost of maintaining the complete proof tree, on the other hand, is that tactics programming becomes much more difficult, and that the attractive, and very tight, connections between term and proof structure evident from e.g. type theory, get lost. So far we have implemented a rather "dirty" solution, giving users access to the basic proof rules themselves, to a set of basic rules for accessing and traversing proof trees, to a small set of tactic constructors, like sequential composition, conditional, etc, and to a higher-order tactic definition facility.

```
/* resolvable: Proof branch can be closed */
rule resolvable =
  eq_r() /* Node is provable equality */
  orelse id() /* Node is instance of id rule */
  orelse ...
  orelse copy_discharge() ;

/* rightexpandable: Goal can be reduced but not closed */
rule rightexpandable =
  or_r() orelse and_r() orelse ... orelse all_r
  orelse box_sem() ; /* Chase transition */

rule rightreduce =
  block
    if isleaf() /* Node is not yet reduced */
    then if resolvable ()
      then skip
      else if rightexpandable()
        then block next_above() ; rightreduce end
        else fail("rightreduce")
    else fail("rightreduce")
  end ;
```

Fig. 3. Tactic for simple "model checking"

Another example is outlined on Fig. 3 which is shown less for its details than to give a general impression of the shape of tactics we used for the example. In our case study tactics were indispensable. They permitted us ot produce very

large parts of the proofs entirely automatically. We implemented tactics for a wide range of purposes, and of very different generality. For instance it is quite easy to implement simple proof strategies for boolean formulas as tactics.

Example 2. A coarse approximation of the Erlang function `process_in_ring` as presented in Sect. 2, just receiving an integer, incrementing it by one and passing it on:

```
fun process_in_ring =
  {NextProcess} ->
    receive {N} ->
      begin NextProcess!(N+1), process_in_ringNextProcess end
    end
  end
```

The following "wait_for_input" property expresses the behaviour of such ring processes in state transition terms:

$$\begin{aligned} \mathit{wait\_for\_input}(\mathit{pid}_1, \mathit{pid}_2) &\Rightarrow \mathit{wait\_for\_input}'(\mathit{pid}_1, \mathit{pid}_2) \\ \mathit{wait\_for\_input}'(\mathit{pid}_1, \mathit{pid}_2) &\Leftarrow [] \mathit{wait\_for\_input}'(\mathit{pid}_1, \mathit{pid}_2) \\ &\quad \wedge\ \forall P. \forall V. [P!V] \mathit{false} \\ &\quad \wedge\ \forall P. \forall N. [P?N]((P = \mathit{pid}_1) \wedge \\ &\qquad\qquad \mathit{nat}(N) \rightarrow \mathit{respond}(\mathit{pid}_1, \mathit{pid}_2)) \end{aligned}$$

$$\begin{aligned} \mathit{respond}(\mathit{pid}_1, \mathit{pid}_2) &\Rightarrow \mathit{respond}'(\mathit{pid}_1, \mathit{pid}_2) \\ \mathit{respond}'(\mathit{pid}_1, \mathit{pid}_2) &\Leftarrow [] \mathit{respond}'(\mathit{pid}_1, \mathit{pid}_2) \\ &\quad \wedge\ \forall P. \forall V. [P!V]((P = \mathit{pid}_2) \wedge \\ &\qquad\qquad \mathit{nat}(V) \wedge \mathit{wait\_for\_input}(\mathit{pid}_1, \mathit{pid}_2)) \end{aligned}$$

Using a tactic based on `right_reduce` above the proof goal (6) was proved automatically, with subproof sharing, using 212 nodes, 1 application of discharge, and 7 applications of copy-discharge. Turning subproof sharing off the same tactic required 530 nodes and 12 applications of discharge. The size increase is due to one subproof being duplicated thrice.

For larger sequential functions than the one considered in Ex. 2 the issue of subproof sharing becomes very urgent, and it is not hard to realize that an exponential growth in proof size will be the rule rather than the exception.

Also for sequential function evaluation we found tactics very helpful. The `counting` function, for instance, appeals to a number of small auxillary functions like `length`, `split`, `flatmap`, or comparison operators like $\geq$ which are implemented as functions as well. Frequently small lemmas are needed to show termination, or to show type preservation properties which are not guaranteed in general, as Erlang is an untyped language.

Tactics in a style similar to that of Fig. 3 were developed to prove type adherence of Erlang expressions. With these tactics we could automatically prove, for instance, that

– if `Store` represents a list, then `length(Store)` results in a number,
– if `Store` represents a list and `PacketSize` a number, then `PacketSize =< length(Store)` results in a boolean, and
– appending two lists results in a list.

These sorts of tactics were used to bring down the complexity of the proof by reducing large proof goals to smaller ones which could eventually be completed using one of these tactics.

5.4 Using the Tool in Practice

Mixing automated and interactive verification in the manner we propose puts very considerable demands on the user interface, to aid users control of possibly very large proofs. The tactic programming language gives a lot of help, providing facilities for naming and retrieving nodes, and for defining search and navigation procedures. The simple tactics we developed for "model checking", type check, and termination, turned out to be surprisingly robust, requiring little adaptation even for quite substantial modifications to the functions and properties being checked. In our case study so far we have proved a number of properties for the ring process, and for various approximations of it in the style of Ex. 2. The most sophisticated of those proofs contains about 2000 proof nodes, of which two-third has been generated automatically. We also proved a version of the composition property as stated in (7). This proof uses in the order of 700 nodes, and so far we have not mechanized this. It is representable of a kind of proof which we expect to be able to mechanize almost completely in the future. To help visualisation we interfaced our tool to the daVinci graph display facility [FW94]. Small graphs, less than 1000 nodes, are easily displayed by daVinci, and it provides good help, for instance in debugging proof tactics. For larger proofs graphs really need to be displayed incrementally (not very well supported currently) or in segments, to avoid excessive delays.

6 Conclusions

Our report is a tentative one, reporting more on qualitative than quantitative experiences with the use of a novel approach to code verification for distributed systems. The report must be a tentative one, since there really are not many tools or proof approaches around with a similar scope of addressing dynamic process networks on the level of actual running code without resorting to approximate techniques. The database lookup manager which we addressed was about 200 lines of code and explored most "core" features of the Erlang language including list and number processing, communication, and dynamic process creation. Experience with Erlang at Ericsson has indicated that — as a rule of thumb — one line of Erlang code corresponds to six lines of C.

A central issue on which we have as yet little to say is scalability. Since our proof system is highly compositional it is actually realistic to hope to reuse

proofs together with their associated code modules. As yet, however, we have little practical experience with this.

The proof approach which we follow requires user intervention. We have developed tactics which are quite robust and manage to produce large parts of proofs without any user intervention at all. Moreover it is quite realistic in many cases to hope to automate almost the entire proof search process, even in cases when model checking-like techniques fail. The critical point at which user intervention is really essential is, of course, in the identification of inductive assertions. In the example studied here this was not at all easy. A particular source of headache was the handling of process identifiers which in Erlang play a role not unlike names in the π-calculus. Even though our handling of process identifiers (pids) and pid creation in Erlang is as yet imperfect, the tool was able to assist the identification of inductive assertions quite substantially, by having tactics which were sufficiently robust to often accomodate smaller formula modifications completely automatically.

Acknowledgements

Hans Nilsson deserves our special thank for bringing forward the verification problem we considered in this paper and for the time he spent in explaining us the details. We should like to thank Lars-Åke Fredlund for his helpful hints and his constant support for the proof system. We thank Gena Chugunov for for digging into some nasty details of the proof. The second author was supported by the Swedish National Board for Technical and Industrial Development (NUTEK) through the ASTEC competence centre.

References

[AVWW96] J. Armstrong, R. Verding, C. Wikström and M. Wiliams, *Concurrent Programming in Erlang*. 2:nd edition, Prentice-Hall, 1996.

[ADFG98] T. Arts, M. Dam, L-Å. Fredlund, and D. Gurov, System Description: Verification of Distributed Erlang Programs. In *Proceedings 15th Conference on Automated Deduction*, LNAI 1421, p. 38–42, July 1998.

[Cri95] R. Cridlig, Semantic Analysis of Shared-Memory Concurrent Languages Using Abstract Model Checking. *In* Proc. PEPM'95.

[Dam95] M. Dam, Compositional proof systems for model checking infinite state processes. In *Proceedings CONCUR'95*, LNCS 962, p. 12–26, 1995.

[DFG98] M. Dam, L.-Å. Fredlund and D. Gurov, Toward Parametric Verification of Open Distributed Systems. To appear in: H. Langmaack, A. Pnueli, W.-P. De Roever (eds.), *Compositionality: The Significant Difference*, Springer Verlag, 1998.

[DFH$^+$] G. Dowek, A. Felty, H. Herbelin, G. Huet, C. Murthy, C. Parent, C. Paulin-Mohring, and B. Werner. *The Coq proof assistant user guide*, Technical report, INRIA-Rocquencourt, May 1993.

[FW94] M. Fröhlich and M. Werner. The graph visualization system daVinci – a user interface for applications. Technical Report 5/94, Department of Computer Science, Bremen University, 1994.

[Koz83] D. Kozen, Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[Lego] `http://www.dcs.ed.ac.uk/home/lego/`

[Mnesia] C. Wikström, Hans Nilsson and Håkan Mattson, Mnesia Database Management System, In *Open Telecom Platform users manual*, Open Systems, Ericsson Utvecklings AB, Stockholm, Sweden, 1997.

[Nil99] H. Nilsson, Patent application, 1999.

[Par76] D. Park, Finiteness is mu-ineffable. *Theoretical Computer Science*, 3:173–181, 1976.

[Pau94] L. C. Paulson. *Isabelle: A Generic Theorem Prover*, LNCS 828, 1994

[ROS92] J. Rushby, S. Owre and N. Shankar. PVS: A prototype verification system. In *Proceedings 11th Conference on Automated Deduction*, LNAI 607, pp. 748–752, 1992.

[Sha96] N. Shankar. PVS: Combining specification, proof checking, and model checking. In *Proceedings of Formal Methods in Computer-Aided Design*, LNCS 1166, pp. 257–264, November 1996.