A Tool for Verifying Software Written in Erlang

Thomas Arts[1], Gennady Chugunov[2], Mads Dam[2], Lars-åke Fredlund[2], Dilian Gurov[2], and Thomas Noll[3]

[1] Ericsson Computer Science Laboratory, Ericsson Utvecklings AB,
thomas@cslab.ericsson.se
[2] Swedish Institute of Computer Science,
gena@sics.se,mfd@sics.se,fred@sics.se,dilian@sics.se
[3] Department of Teleinformatics, Royal Institute of Technology, Sweden,
noll@it.kth.se

Abstract. The present paper presents an overview of the main results of the ASTEC project Verification of Erlang Programs, focusing in particular on the Erlang verification tool. This is a theorem-proving tool which assists in obtaining proofs that Erlang applications satisfy their correctness requirements formulated in a specification logic. We give a summary of the verification framework as supported by the tool, discuss reasoning principles essential for successful verification such as inductive and compositional reasoning, and an efficient treatment of side-effect-free code. The experiences of applying the verification tool in an industrial case study are summarised, and an approach for supporting verification in the presence of program libraries is outlined.
The verification tool is essentially a classical proof assistant, or theorem-proving tool, requiring users to intervene in the proof process at crucial steps such as stating program invariants. However, the tool offers considerable support for automatic proof discovery through higher-level tactics tailored to the particular task of verification of Erlang programs. In addition, a graphical interface permits easy navigation through proof tableaux, proof reuse, and meaningful feedback about the current proof state, to assist users in taking informed proof decisions.

1 Introduction

Software written for telecommunication applications has to meet high quality demands. Such software is often highly concurrent, and *testing* is not by itself sufficient to ensure reliability requirements are met. At the Ericsson corporation the functional language Erlang [1] has been developed for programming telecommunication applications. Today many commercially available products offered by Ericsson are partly programmed in Erlang. The software of such products is typically organised into many, relatively small source modules, and at runtime consists of a dynamically varying number of processes operating in parallel and communicating through asynchronous message passing.

Our approach to verification of software programmed in Erlang is code verification: we prove that Erlang code satisfies a set of properties formalized in a suitable logic: a specification language based on Park's μ–calculus [24, 18], extended with Erlang–specific features. This is a quite powerful logic, due to the presence of least and greatest fixed point recursion, allowing the formalization of a wide range of behavioural properties. The verification problem in this context is not decidable, but can be automated to a considerable extent, requiring human intervention in a few, but critical points.

An alternative approach to the verification of Erlang programs is to use abstract interpretation to create a finite-state abstraction checkable with standard model-checking techniques. This approach is taken by Huch [17], where a concrete abstract interpretation is suggested essentially reducing infinite data domains to finite ones. However, the infinite state spaces arising from unbounded message queues or unbounded process spawning, which are characteristic for open distributed systems, are not handled there.

The effort on the verification of Erlang programs has taken place within a collaborative project between the Formal Design Techniques group at the Swedish Institute of Computer Science and the Ericsson Computer Science Laboratory, and funded by the ASTEC (Advanced Software Technology) competence center. So far, the project effort has been directed towards establishing the mathematical machinery [6, 7, 9, 8], providing basic tool support [3], performing case studies [2], and motivating the chosen verification framework [12]. The present paper present an overview of the main results of the Erlang verification effort, and focuses in particular on the Erlang verification tool, a theorem proving tool that assists in obtaining proofs that Erlang applications satisfy their correctness requirements formulated in the specification logic.

In Section 2 we summarise the verification framework supported by the tool: the Erlang programming language, its formal semantics, the property specification language, and the proof system. In Section 3 we describe the implementation of the tool. Particular emphasis is placed on aspects of the tool which are less often found in comparable tools like [10, 23, 25, 5], notably the discharge mechanism which implements a well-founded induction scheme to handle program recursion. Section 4 discusses the principles of inductive and compositional reasoning applied to the verification of Erlang programs. Since large fragments of Erlang applications are purely functional, i.e., do not rely on side effects like process communications, an efficient and compositional approach to dealing with such classical functional code is absolutely essential, and is also discussed in Section 4.

Section 5 summarises our experience with using the tool in a typical case study: the verification of a distributed database query evaluation protocol. Section 6 presents an approach to dealing with modularity in an elegant fashion. Typical Erlang applications make extensive use of standard libraries, implementing everything from basic operations on lists to complex distributed data bases. The paper proposes a semantic approach to extending the capabilities of the tool for handling library modules without considering the actual implementation of these modules. Finally the paper is concluded with a discussion of the merits and shortcomings of the tool.

2 Foundation

In this section we briefly highlight the foundations of the tool: the Erlang programming language, a specification logic for capturing correctness requirements of Erlang programs, and a proof system for proof derivation.

2.1 The Erlang Programming Language

Erlang/OTP is a *programming platform* providing the necessary functionality for programming of open distributed (telecom) systems: a language Erlang with support for concurrency, and middleware OTP (Open Telecom Platform) providing ready-to-use components (libraries) and services such as e.g. a distributed data base manager, support for "hot code replacement", and design guidelines for using the components.

In the following we consider a core fragment of the Erlang programming language with dynamic networks of processes operating on data types such as integers, lists, tuples, or process identifiers (pid's), using asynchronous, call–by–value communication via unbounded ordered message queues called mailboxes. Real Erlang has several additional features such as modules, distribution of processes (onto nodes), and support for interfacing with non–Erlang code written in, e.g., C or Java.

Besides Erlang *expressions* e we operate with the syntactical categories of *matches* m, *patterns* p, *guards* g, and *values* v. The abstract syntax of Core Erlang expressions is summarised as follows:

$$
\begin{aligned}
e ::=\ & \mathrm{V} \\
\mid\ & op(e_1, \ldots, e_n) \\
\mid\ & \texttt{begin}\ e_1, \ldots, e_n\ \texttt{end} \\
\mid\ & e(e_1, \ldots, e_n) \\
\mid\ & \texttt{case}\ e\ \texttt{of}\ m\ \texttt{end} \\
\mid\ & \texttt{catch}\ e \\
\mid\ & \texttt{receive}\ m\ \texttt{end} \\
\mid\ & e_1 ! e_2
\end{aligned}
$$

$$
\begin{aligned}
v &::= op(v_1, \ldots, v_n) \\
p &::= op(p_1, \ldots, p_n) \mid V \\
m &::= p_1\ [\texttt{when}\ g_1]\ \texttt{->}\ e_{11}, \ldots, e_{1k}; \\
&\quad\ \vdots \\
&\quad\ p_n\ [\texttt{when}\ g_n]\ \texttt{->}\ e_{n1}, \ldots, e_{nk} \\
g &::= e_1, \ldots, e_n
\end{aligned}
$$

Here V ranges over Erlang variables, and *op* ranges over a set of primitive constants and operations including the integers ranged over by i, tupling $\{e_1, e_2\}$, the empty list $[\,]$, list prefix $[e_1|e_2]$, pid constants ranged over by *pid*, and atom constants ranged over by a and f.

An Erlang *process*, here written $\langle e, pid, q\rangle$, is a container for the evaluation of an expression e. A process has a unique process identifier (pid) which is used to identify the recipient process in communications. Communication is always binary, with one (anonymous) party sending a message (a value) to a second party identified by its process identifier. Messages sent to a process are put in its mailbox q, queued in arriving order. To express the concurrent execution of two sets of processes s_1 and s_2, the syntax $s_1 \parallel s_2$ is used.

The functional sublanguage of Erlang is rather standard: atoms, integers, lists and tuples are value constructors; $e(e_1, \ldots, e_n)$ is a function call; `begin` $e_1, \ldots, e_n$ `end` is sequential composition;

$$
\begin{array}{l}
\texttt{case}\ e\ \texttt{of} \\
\quad p_1\ [\texttt{when}\ g_1]\ \texttt{->} e_{11}, \ldots, e_{1k}; \\
\quad \ldots \\
\quad p_n\ [\texttt{when}\ g_n]\ \texttt{->} e_{n1}, \ldots, e_{nk} \\
\texttt{end}
\end{array}
$$

is matching: the value that e evaluates to is matched sequentially against patterns (values that may contain unbound variables) p_i, respecting the optional guard expressions e_{ig}. The expression $e_1!e_2$ represents sending (the value of e_2 is sent to the process with process identifier e_1) whereas `receive` m `end` inspects the process mailbox q and retrieves (and removes) the first element in q that matches any pattern in m. Once such an element v has been found, evaluation proceeds analogously to `case` v `of` m. Expressions are interpreted relative to an environment of "user defined" function definitions of the shape:

$$
\begin{array}{l}
f(p_{11}, \ldots, p_{1k})\ \texttt{->} \tilde{e_1}; \\
\vdots \\
f(p_{n1}, \ldots, p_{nk})\ \texttt{->} \tilde{e_k}.
\end{array}
$$

The constructs involving side effects (non functional behaviour) are: `receive` for reading from the mailbox which is associated with the process evaluating the expression and "!" for sending a value to a process identified by its pid, In addition side effects are possible through built-in functions like `self`(), evaluating to the pid of the process evaluating this expression, `throw`(v) for raising an exception v (that can be handled by a `catch` expression). and `spawn`($f, [v_1, \ldots, v_n]$), resulting in a new process being generated executing the function call $f(v_1, \ldots, v_n)$.

Example 1 (RPC). Below we sketch a (very) simplistic RPC service in Erlang.

```
server() ->
  receive
    {F, Pid} -> spawn(exec, [F,Pid]), server()
  end.

exec(F,Pid) -> Pid!F().
```

Note above that Erlang variables are always upper-case (`F` and `Pid`) while atoms are lower-case (`server`, `exec` and `spawn`). The `server` function is continuously prepared to receive tuples containing the name of a function `F` and a process identifier `Pid`. It then spawns off a new process evaluating the `exec` function, which simply invokes the received function `F` and sends any result to the process addressed by the process identifier `Pid`.

Since Erlang is an untyped language a possible outcome of sending a wrongly typed message to the server process is that the newly spawned process will terminate due to a runtime error.

A Semantics for Erlang The formal semantics of Erlang is given as an operational semantics in the form of a set of rules for deriving labelled transitions between structured states [26]. Our semantics for Erlang is a small–step operational one [11], which is motivated by the free intermixing of functional and side-effect concerns found in Erlang.

A natural approach to handling the different conceptual layers of entities in the language (i.e., functional expressions and concurrent processes) supporting modular (i.e., compositional) reasoning, is to organise the semantics hierarchically, in layers, using different sets of transition labels at each layer, and extending at each layer the structure of the state with new components as needed. Thus first the Erlang expressions are provided with a semantics that does not require any notion of processes. The actions here are a computation step τ, an output $pid!v$, $read(q, v)$ which represents the reading of a value v from the queue of the process in whose context the expression executes, and $f(v_1, \ldots, v_n)$ which represents the calling of a builtin function (like `spawn` for process spawning) with side effects on the process level state.

Then the transition behaviour of Erlang systems (concurrent processes executing expressions in the context of a unique process identifier and a mailbox of incoming messages) is captured through a set of transition rules separated into two cases: (i) a single process constraining the behaviours of an Erlang expression and (ii) the (parallel) composition of two Erlang systems into a single one expressed by the parallel composition construct "||". The system actions are computation steps τ, output $pid!v$ and input $pid?v$.

Example 2 (Erlang Semantics).

We will illustrate the operational semantics by considering the case of a built-in function with side-effects, like for instance `spawn`. On the level of Erlang expression the evaluation of a built-in function like `spawn` is covered by the transition rule:

$$\frac{\mathit{isProcFun}(\mathrm{f})}{f(v_1, \ldots, v_n) \xrightarrow{f(v_1, \ldots, v_n)} v}$$

where $\mathit{isProcFun}(f)$ tests whether the function f names a built-in function, and v represents any Erlang value (akin to an input parameter). As seen in the above rule the operational semantics is infinitely branching, due to occurrence of the

v placeholder. Any complications caused by this are naturally handled on the level of the proof system, via proper introduction of quantifiers and such. On the process level spawning is handled more directly in the following rule

$$\frac{e \xrightarrow{\texttt{spawn}(f,[v_1,\ldots,v_n])} pid' \quad pid' \neq pid}{\begin{array}{c}\langle e, pid, q\rangle \\ \xrightarrow{\tau} \\ \langle e', pid, q\rangle \parallel \langle f(v_1,\ldots,v_n), pid', \epsilon\rangle\end{array}}$$

where ϵ denotes the empty queue. One of the interleaving rules also makes a provision for process spawning:

$$\frac{s_1 \xrightarrow{\tau} s_1{}' \quad pids(s_1{}') \cap pids(s_2) = \emptyset}{s_1 \parallel s_2 \xrightarrow{\tau} s_1{}' \parallel s_2}$$

The condition $pid' \neq pid$ ensures that the process identifier of the newly spawned process is locally unique, and the condition $pids(s_1{}') \cap pids(s_2) = \emptyset$ ensures the same under parallel composition.

2.2 The Property Specification Language

Reasoning about applications programmed in Erlang generally requires *compositional reasoning*, i.e., the capability to reduce arguments about the behaviour of compound entities to arguments about the behaviours of its parts. To support compositional reasoning, a specification language for Erlang has to capture the labelled transitions at each layer of the transitional semantics (expressions and processes). Poly–modal logic is particularly suitable for the task, employing box and diamond *modalities* labelled by the transition labels: a structured state s satisfies formula $\langle\alpha\rangle\,\phi$ if there is an α–derivative of s (i.e., a state s' such that $s \xrightarrow{\alpha} s'$ is a valid labelled transition) satisfying ϕ, while s satisfies $[\alpha]\,\phi$ if all α–derivatives of s satisfy ϕ.

Additionally to support reasoning about data, the usual logical connectives are brought in from (many-sorted) first order logic, including term equality, quantifiers, lambda abstraction and applications. In the following we let t range over general terms, and T_t over the sorts (types), although these sorts will usually not be written out in formulas.

The presence of recursion on different layers requires also the specification language to be recursive. Adding recursion in the form of least and greatest fixed–points to the modalities described above results in a powerful specification language, broadly known as the *μ–calculus* [24, 18]. Roughly speaking, least fixed–point formulas $\mu X.\phi$ express eventuality (liveness) properties, while greatest fixed–point formulas $\nu X.\phi$ express invariant (safety) properties. Nesting of fixed points allows complicated reactivity and fairness properties to be expressed. Note that as usual references to fixed points can be made only under an even number of negations, to ensure that the corresponding fixed points exists (due to monotonicity).

The syntax of the logic can then be summarised:

$$\begin{array}{rl} \phi ::= & t_1 = t_2 \mid \texttt{true} \mid \texttt{false} \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \\ \mid & \exists T : T_t.\phi \mid \forall T : T_t.\phi \mid \lambda T : T_t.\phi \mid \phi\, t \text{ (application)} \\ \mid & \langle\alpha\rangle\,\phi \mid [\alpha]\,\phi \\ \mid & \nu X.\phi \mid \mu X.\phi \mid X \text{ (fixed point reference)} \end{array}$$

This powerful logic is capable of expressing a wide range of important system properties, ranging from type–like assertions to complex reactivity properties of the interaction behaviour of a telecommunication system. As a syntactic convention fixed point formulas can be named, e.g., $name \Leftarrow \phi$ abbreviates the least fixed point $\mu X.\phi[X/name]$ and $name \Rightarrow \phi$ abbreviates the greatest fixed point $\nu X.\phi[X/name]$ (X is assumed fresh in ϕ).

The semantics of a formula in the logic is defined in the usual (denotational) fashion, as the set of Erlang systems that satisfy the formula (see [7] for details).

Example 3. The type of natural numbers is the least set containing zero and closed under successor. The property of being a natural number can hence be defined recursively as a least fixed–point, assuming the term constructors 0 and +1:

$$nat \;\Leftarrow\; \lambda N.\,(N = 0 \;\vee\; \exists V.\,(nat\; V \wedge N = V + 1))$$

2.3 The Proof System

Reasoning about open distributed systems written in Erlang requires reasoning about their interface behaviour relativised by assumptions about certain system parameters. Technically, this can be achieved by using Gentzen–style proof systems, allowing free parameters to occur within the *proof judgments* of the proof system. The judgments are of the form $\Gamma \vdash \Delta$, where Γ and Δ are sets of assertions. A judgment is deemed *valid* if, for any interpretation of the free variables, some assertion in Δ is valid whenever all assertions in Γ are valid. Parameters are simply variables ranging over specific types of entities, such as messages, functions, or processes. For example, the proof judgment $\psi\; x \vdash \phi\; P(x)$ states that object P has property ϕ provided the parameter x of P satisfies property ψ.

The proof rules of the proof system are mostly standard from accounts of first-order logic in Gentzen–style proof systems, with rules like $\forall_R$ and $\forall_L$ shown below:

$$(\forall_L)\; \frac{\Gamma, \phi\{v/V\} \;\vdash\; \Delta}{\Gamma, \forall V : T_t.\phi \;\vdash\; \Delta}\, v \in T_t$$

$$(\forall_R)\; \frac{\Gamma \;\vdash\; \phi, \Delta}{\Gamma \;\vdash\; \forall V : T_t.\phi, \Delta}\, V \textit{ fresh}$$

To this regular proof system two rules were added: the first a cut–like rule, here called term–cut, for decomposing proofs about a compound system to proofs about the components, the second a discharge rule based on detecting loops in the proof. Roughly, the goal is to identify situations where a latter proof node is an instance of an earlier one on the same proof branch, and where appropriate fixed points have been unfolded. The discharge rule thus takes into account the history of assertions in the proof tree. In terms of the implementation this requires the preservation of the proof tree during proof construction. Combined, the term–cut rule (cf. Section 4.1) and the discharge rule (cf. Section 4.2) allow general and powerful induction and co–induction principles to be applied, ranging from induction on the dynamically evolving architecture of a system, to induction on finitary and co–induction on infinitary datatypes.

3 The Erlang Verification Tool

The proof system introduced in the previous section has been implemented in a proof assistant (or proof checker) named the "Erlang verification tool" here, abbreviated EVT. This tool has been tailored to the underlying proof system; rather than working with a set of open goals, the underlying data structure is an acyclic proof graph, to account for the checking of the side conditions of the discharge rule. The main reason for developing a new proof assistant tool prototype, rather than adapting existing mature proof checkers like Isabelle [25], COQ [10], PVS [23] or NuPrl [5] is precisely our desire to experiment with the rule of discharge and the underlying proof graph, in order to potentially enable more efficient checking of these conditions than a coding of the discharge rule in a general purpose proof checker would permit.

Two notable releases of the verification tool exists. The first release was reported in [3] and was an experimental prototype tailored especially to the verification of Erlang code. The second and current tool release is more general, permitting the embedding of theories for other languages. Apart from the support for Erlang, so far only an experimental embedding of a variant of the value-passing Calculus of Communicating Systems [19] (CCS for short) has been added to the proof checker.

The current tool is, like the theorem provers HOL [14] and Isabelle [25], implemented in Standard ML [20], and the standard ML toplevel is used for interaction with tool users.

3.1 Terms, Variables, Formulas and Proofs

The tool has as foundation a simple variant of many-sorted first order logic. Accordingly terms are typed (based on their unique term constructors), but there is also a notion of subtyping to permit a hierarchy of types. Types can be equipped with type specific parsers and unparsers, to enable reading and printing of terms, and formulas, in native formats such as e.g. the Erlang constructs. Likewise derived formula constructs, with language specific semantics, can be defined. The introduction of subtyping in the underlying theory, can, as usual, introduce typing proof obligations during parsing of terms and formulas.

For types considered to be freely-generated (intuitively the types where "semantic equality" coincides with the syntactic notion of equality) such as e.g. the natural numbers, recursive predicates can be automatically generated that permit structural induction style arguments about elements of the type.

Sequents $\Gamma \vdash \Delta$ are pairs of ordered sequences of formulas (assertions) $\Gamma = \phi_1, \ldots, \phi_n$ and $\Delta = \psi_1, \ldots, \psi_k$. These formulas may contain free variables, which are of two kinds: *parameters* which are generated by rules such as $\forall_R$ above, and *meta-variables*, the result of postponing the choice of a witness in a proof rule such as $\forall_L$. To ensure that assignments to meta-variables are sound a simple scheme based on associating indices to variables, from [27], is used. Bound variables are represented using de Bruijn indices, to permit checking equality of formulas quickly up to α-conversion, which is important for obtaining efficient implementations of the discharge rule.

From a user's point of view, proving a property of an Erlang program using the verification tool involves "backward" (i.e., goal-directed) construction of a proof graph (tableau). A proof graph is, here, an acyclic directed graph of proof nodes containing sequents and rooted in an initial proof node. Each proof node in the graph is either a leaf node, meaning that it either represents an open goal, or that the sequent was solved by the application of an axiom proof rule without premises, or it is a parent node that has been reduced by applying the proof rule and such that its children nodes correspond to the premises of the rule. An application of the discharge rule is represented in the proof graph by a directed arc from the discharged node to its companion node. Arcs in the proof tree are labelled by the proof rule that caused the arc to appear, to permit flexible display of proofs and portable proofs (to allow for, as an example, proof carrying code schemes [21]).

Open proof goals may also be (copy)discharged (or subsumed in more standard terminology) when instances of the goal can be found elsewhere in the proof graph. In practice application of the copydischarge rule is absolutely essential to, for example, combat the state explosion caused by the interleaving semantics of Erlang. However, there are two restrictions to its use. First, no open proof goal can be copydischarged against an ancestor proof node. Second an acyclicity condition is enforced to prevent cyclic copydischarges, for example such that node N is copydischarged against node M which is in turn copydischarged against node N. A finished proof graph is a proof graph that contains no open goals.

The application of a proof rule can be canceled (undone), resulting potentially in non-local cancellation effects on the proof tree when e.g. the companion node of a copydischarge node is canceled, naturally also causing the copydischarge to fail. Another such problematic case is when a meta-variable is assigned, and canceled in a proof branch, but where the meta-variable is also present in another proof branch. In such a situation both the assignment, and cancellation, may also affect the proof steps in the second proof branch. To permit a sound cancellation scheme in spite of these difficulties a global ordering of proof sequents is introduced, based on the absolute order in which proof nodes were introduced by applications of proof rules.

A proof graph can also contain discharges with respect to nodes not actually in the same proof tree but in another proof tree. Such non-local copydischarges are referred to as applications of lemmas, or lemmadischarges. Again an acyclicity test is performed, to prohibit mutual dependencies between lemmas.

A (finished) proof is then a collection of finished proof graphs such that all non-local discharges are made within the collection of proof graphs.

3.2 Rules, Tactics and Tacticals

The basic proof rules of the proof assistant are implemented in the tool as *tactics*, which are functions (in the Standard ML sense) from a sequent (the current goal, or the conclusion) to a tuple consisting of a list of sequents (the premises of the rule) and a list of assignments to meta-variables caused by the tactic. Thus, if the (SML) type of sequents is `sequent` and meta-variables are of type `var` and terms are represented by the type `term` then the type of a tactic is

```
type tactic =
  sequent -> sequent list * (var * term) list
```

Most rules are implemented as triggering on a particular assertion position in a sequent, and thus require a natural number argument to determine where in the sequent the rule is applied. Thus, for instance, the tactics implementing the proof rule $\forall_R$ has the signature:

```
forall_r: int -> tactic
```

Assertions, on both sides of a sequent, are numbered starting from one.

As most other proof assistants do, EVT provides tactic combinators (tacticals) to offer a facility to derive new sound tactics from basic tactics. Examples of such tacticals offered in EVT are, for example,

```
t_skip:    tactic
t_compose: tactic -> (tactic list)
           -> tactic
t_orelse:  tactic -> tactic
            -> tactic
t_bool:    (seq -> bool) -> tactic -> tactic
           -> tactic
t_fix:     'a -> ('a -> ('a -> tactic) -> tactic)
           -> tactic
```

The tactical `t_skip` succeeds returning simply its sequent argument. The tactical `t_compose t tl` applies the tactic `t` to the current sequent and then applies the tactics in `tl` to the matching resulting goals. The combined tactic fails if the number of goal sequents does not match the number of tactics in the list `tl`. In the evaluation of `t1 t_orelse t2`, the tactic `t2` is applied only if `t1` fail. The `t_bool` tactical performs a test on the current sequent and if successful continues with tactic t1, otherwise tactic t2 is chosen. Finally t_fix can be used to write recursive tactics, the first argument being an arbitrary initialization value, the second a function of an arbitrary parameter and a "continuation", and returning a tactic.

As an example of the usage of such tactical combinators a derived tactic forall_all_r is given below, that tries to apply the rule $\forall_R$ to all right-hand assertions. We assume that the number of assertions to the right is returned by the function length_assertions_right:

```
val forall_all_r =
  t_fix 1
  (fn position => fn continuation =>
    t_bool
      (fn seq =>
       position <= length_assertions_right seq)
      (t_orelse
        (t_compose
          (forall_r position)
          [continuation (position+1)])
        (continuation (position+1)))
      t_skip) : tactic;
```

3.3 User Interfaces and Commands

The standard user interface to the proof assistant is the conventional command line interface of Standard ML (of New Jersey) to which a number of commands to interact with the proof assistant has been added. Conceptually the user interface defines notions such as "which is the current proof graph" and "which is the current proof node". The commands of the proof assistant operate on proof graphs, possibly with side effects. For instance, there are commands to start a new proof, to define a lemma, to navigate proof graphs (to modify the notion of the current proof node), to navigate the hierarchy of proof graphs, to grow (or complete) a proof graph by applying a tactic to its current sequent resulting possibly in new proof branches, and to cancel a previous proof step. As another example the discharge and copydischarge proof rules are implemented as commands rather than tactics, since they cause global effects to the graph structure.

A clear alternative to combining tactics using tacticals is to directly use the Standard ML programming language facilities to define functions executing proof commands. This works reasonably well, but has the disadvantage that all intermediate proof nodes are kept. In contrast, using tactical combinators, no intermediate proof nodes are ever kept.

A second, graphical, user interface is under development. This user interface consists of two parts: the first is programmed in Java and provides additional user assistance through the implementation of modern theorem prover features [4] such as "proof-by-pointing" (to suggest, based on the proof context, the next proof rule to apply), a more structured database of lemmata, proof recording and playback, etc. The second component of the graphical user interface is used to visualize and navigate through the proof graph, and is implemented by interfacing with the daVinci [13] graphical visualisation system.

3.4 Checking Discharge Conditions

Of particular interest is the implementation of the discharge proof rule. Given a proof node N_d representing an open proof goal (a sequent $\Gamma_{dn} \vdash \Delta_{dn}$) in the proof tree, henceforth called the *discharge node* and an ancestor node in the proof tree, henceforth called the *companion node* N_c (a sequent $\Gamma_{cn} \vdash \Delta_{cn}$), the discharge proof rule checks whether the companion node, and some auxiliary conditions formulated on the global proof graph, motivates the discharging of proof node N_d. An obvious generalisation is to search for suitable companion nodes among all the ancestor nodes of N_d. A characterisation of the conditions regulating when such a discharge step is sound is exposed in [7, 9], here only a sketch of the implementation is given.

The overall idea is to keep track of unfoldings of fixed point definitions by annotating fixed points with ordinal variables, corresponding to the number of unfoldings. Unfolding a least fixed point definition to the left of the turnstile (in Γ), or a greatest fixed point definition to the right of the turnstile (in Δ), replaces an ordinal variable κ associated with the fixed point with a new ordinal variable κ', and introduces an ordinal inequation $\kappa' < \kappa$ as an additional assumption in Γ. The discharge condition then amounts to proving, using a well-founded induction scheme on ordinals, that some ordinal in a formula in Γ_{cn} (or Δ_{cn}) occurs decreased in Γ_{dn} (or Δ_{dn}) and such that otherwise proof node N_d is an instance of N_c (the bases cases of the induction schema are trivial, due to the semantics of fixed point approximations and sequent semantics). This simple scheme is complicated by the fact that multiple fixed point definitions may be unfolded, necessitating to consider decreasing measures computed on vectors of ordinals among several pairs of companion and discharge nodes rather than on individual ordinals and a single companion and discharge node pair.

More concretely, first all candidate substitutions ρ, which map variables (parameters) in N_c (Γ_{cn}, Δ_{cn}) to terms are calculated, such that for any formula $\phi \in \Gamma_{cn}$ then $\phi\rho \in \Gamma_{dn}$ (and correspondingly if $\phi \in \Delta_{cn}$ then $\phi\rho \in \Delta_{dn}$). If no such substitution exists, the discharge rule fails. Then all ordinal variables κ occurring in Γ_{cn} or Γ_{dn} are enumerated, and it is checked whether either (i) $\kappa\rho = \kappa$ (the ordinal is *preserved*) or (ii) it is inferable from Γ_{dn} that $\kappa\rho < \kappa$ (the ordinal is *progressing*). If the set of progressing ordinals is empty, then clearly the induction scheme is not applicable and the discharge rule fails.

Next the set of all discharge nodes N'_d related to the discharge node N_d is computed. These are the discharge nodes whose corresponding companion nodes N'_c occur on the path between N_c and N_d (or are related, by transitivity, to some other discharge node N''_d that is in turn related to N_d). For these related discharge nodes the sets of progressing and preserved ordinals have been prerecorded, for all possible substitutions, and the remaining task is now to find a decreasing measure such that: given any combinations of substitutions, and by selecting one progressing ordinal variable per discharge node, a total ordering $<_l$ on these ordinal variables can be found such that if $k <_l k'$ then at the discharge node where k' progresses, k is preserved. This corresponds to a lexicographical ordering on the vector of related ordinal variables.

3.5 Embedding of Erlang

The Erlang program constructs are encoded as terms of the many-sorted first-order logic. The current tool generation contains a definition of the transition relations (on the expression and system levels) as recursive predicates in the underlying logic. In addition, and to improve the speed with which new transitions are computed, a set of low-level rules was implemented directly, for inferring transitions $e \xrightarrow{\alpha} e'$ that trigger on the syntactic shape of the Erlang construct e. An example of such a rule is shown below, for the case of input under parallel composition to the left in a sequent (S is assumed fresh in Γ, Δ):

$$\| ?_L \; \frac{\begin{array}{c}\Gamma, s_1 \xrightarrow{pid?v} S, s' = S \parallel s_2 \;\vdash\; \Delta \\ \Gamma, s_2 \xrightarrow{pid?v} S, s' = s_1 \parallel S \;\vdash\; \Delta\end{array}}{\Gamma, s_1 \parallel s_2 \xrightarrow{pid?v} s' \;\vdash\; \Delta}$$

In general the handling of operational semantics in EVT is split into two parts: one language dependent part where tactics corresponding to the operational semantics of the language in question are introduced and a second, largely language-independent part, for deriving valid transitions from such sets of operational semantics tactics.

3.6 Tactics for Deriving Transitions

The present tool implements four high-level tactics, `diasem_l`, `diasem_r`, `boxsem_l` and `boxsem_r`, for reasoning about combinations of program terms and modalities. For example, the `diasem_r` and `boxsem_r` tactics tries to achieve the result of the rules $\langle\,\rangle_r$ and $[\,]_r$ below, with intuitive definitions (assuming $t \xrightarrow{\alpha} t_1, \ldots, t \xrightarrow{\alpha} t_n$ and no other such continuation state t_x exists):

$$\langle\,\rangle_r \; \frac{\Gamma \vdash t_1 : \phi, \ldots, t_n : \phi, \Delta}{\Gamma \vdash s : \langle\alpha\rangle\phi, \Delta}$$

$$[\,]_r \; \frac{\Gamma \vdash t_1 : \phi, \Delta \quad \ldots \quad \Gamma \vdash t_n : \phi, \Delta}{\Gamma \vdash s : [\alpha]\phi, \Delta}$$

Above, the syntax $t : \phi$ represents the statement t has the property ϕ, a synonyme of $\phi\, t$.

The means of realising tactics achieving the effect of these rules is by repeatedly applying language specific operational semantic tactics such as, e.g., $\| ?_L$ shown before, together with simple general simplification steps like splitting conjunctions and reasoning about term equalities. In addition language dependent tactics for handling data are applied.

4 Inductive and Compositional Reasoning

Code verification is an inherently complex activity in which the complexity of the program behaviour is essentially multiplied with the complexity of the property being analyzed. Using the proof system without any planning requires a large amount of low–level inference steps and decisions to be made which makes verification of industrial–scale software impractical. To make our verification method more practical, we have to lift the level of reasoning to a suitably high level of abstraction. *High–level reasoning* means reasoning "in chunks", i.e., *decomposing* proof obligations about compound objects to proof obligations about the components, and dealing with these using tactics designed to automate the lower–level reasoning steps. At the same time, reasoning about ongoing behaviour involves *inductive* and *co–inductive* arguments which have sometimes to be combined with compositional reasoning. Below we discuss (co–)inductive and compositional reasoning and their rôle in structuring proofs and higher–level reasoning.

There is clearly no general method for verification of arbitrary Erlang programs which is effective and at the same time leads to economic proofs. However, one can do much better in specialised cases which are well–understood. A main direction of research is the identification of fragments of Erlang and the property specification language for which efficient verification methods exist. One such fragment is the side–effect–free one, in which an Erlang expression is evaluated purely for its value, and is not affecting the environment in which it is evaluated in terms of sending messages, reading from the message queue, or process spawning. Below we give a high–level treatment of side–effect–free function calls based on compositional reasoning.

4.1 Induction and Discharge

Automating the verification of components usually faces the difficulty of handling recursively defined behaviour. This requires inductive and co–inductive reasoning, depending on whether one investigates properties of terminating or ongoing behaviour. Many types of induction are involved in examples such as the case study considered in the next section:

- 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. Co–induction 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. To handle these complications, most parts of the proof will involve induction and co–induction 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. In effect the discharge mechanism, as described in the previous section, 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 co–induction 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* builtin support for finding inductive assertions. Such support can be programmed (as tactics), or must alternatively be provided explicitly.

4.2 Compositional Reasoning

The essence of compositional verification is the reduction of an argument about the behaviour of a compound system to arguments about the behaviour of its components. A system P containing component Q can be represented through term substitution as $P[Q/X]$, where X is a variable ranging over entities of the type of Q. We can relativise an assertion $P[Q/X] : \phi$ about the compound object $P[Q/X]$ to a certain property ψ of its component Q by considering Q as a parameter for which property ψ is assumed, provided we can show that Q indeed satisfies the assumed property ψ. Technically, we achieve this through a *term–cut* proof rule of the shape:

$$(\mathsf{TermCut})\ \frac{\Gamma \vdash Q : \psi, \Delta \qquad \Gamma, X : \psi \vdash P : \phi, \Delta}{\Gamma \vdash P[Q/X] : \phi, \Delta}$$

Very often, constructors occurring within the scope of recursion give rise to unbounded state spaces. An example is a process spawning statement, giving rise to the formation of an unbounded process set. In such cases we have to combine (co–)inductive with compositional reasoning. For example, after a new process P has been spawned off by a recursive process Q one can apply the above term–cut rule to relativise the proof on the specification of Q rather than on its implementation, thus avoiding new processes from being generated by Q explicitly in the process term, and thus allowing the (co–)induction through loop detection and discharge to go through.

The above term–cut rule provides the basic low–level facility for compositional reasoning. Applying the rule requires a suitable choice of the cut–property ψ. It should capture the essence of the behaviour of Q needed for completing the proof. In some special cases we can give a concrete structure to the formation of ψ, as illustrated in the next subsection, and give (and support through tactics) more high–level decomposition principles exploiting this additional structure.

4.3 Dealing with Side–effect–free Erlang Code

A frequent case in practice is dealing with function calls where the body of the definition of the function is side–effect–free, i.e., is evaluated purely for its value, and does not affect the environment in which it is evaluated in terms of sending messages, reading from the message queue, or process spawning. The libraries offer a large number of such functions. For example, if we want to prove a property of an Erlang process the expression of which involves a function call to a list–sorting library function `sort` with argument L, we would like to reason high–level and replace this function call with a value variable V, by adding the assumption that V is a sorted permutation of L. What we abstract from in this case are the internal steps required to evaluate the function call. It is safe to do so, since this computation does not affect the rest of the system. It only affects the number of silent (i.e., side–effect–free) steps, therefore such a decomposition assumes the property we are analysing to be "insensitive" to silent actions.

In general, under the assumptions:

- e is a function call;
- the body of the definition of c is side–effect–free;
- ϕ is *insensitive* to (the number of) side–effect–free actions.

the following decomposition principle is applicable:

$$\text{(SefCut)}\ \frac{\begin{array}{c}\Gamma \vdash c : \mathit{prepost}(\psi,\theta), \Delta \\ \Gamma \vdash \psi, \Delta \\ \Gamma, V : \theta \vdash \langle r[V], \mathit{pid}, q\rangle : \phi, \Delta\end{array}}{\Gamma \vdash \langle r[e], \mathit{pid}, q\rangle : \phi, \Delta}$$

where $r[\cdot]$ denotes a reduction context, *prepost* is as defined below, and ϕ is a formula insensitive to silent actions.

Once a side–effect–free function call has been factored out, it can be specified and verified using well–known techniques. A classical method for verification of sequential programs is the axiomatic method of Hoare [16]. It is based on assertions of the shape $\{\psi\}e\{\theta\}$, the intuitive semantics of which, in our context, is: given the parameters of e satisfy the pre–condition ψ, then execution of e, provided it terminates, results in a value satisfying the post–condition θ. We follow the same idea, but require termination; a correctness notion known as *total correctness*.

We use assertions $c : \mathit{prepost}(\psi,\theta)$ where:

$$e : \mathit{prepost}(\psi,\theta) \; = \; (\psi \rightarrow e : \mathit{eval}\,\theta)$$

$$\begin{array}{l} e : \mathit{eval}\,\theta \;\Leftarrow \\ \qquad \exists V : \mathit{ErlangValue}.(e = V \;\wedge\; V : \theta) \\ \quad \vee\, e : \langle\tau\rangle\, \mathit{true} \wedge [\tau]\, \mathit{eval}\,\theta \end{array}$$

For example, a list sorting function `sort` can be specified as a satisfaction pair $\texttt{sort}(L) : \mathit{prepost}(L : \mathit{list}, \theta_{sort}\ L)$ where type *list* is defined by:

$$
\begin{aligned}
& L : \mathit{list} \Leftarrow \\
& \quad\ L = \texttt{[]} \\
& \ \vee \exists P, R : \texttt{ErlangValue}. \\
& \qquad L = \texttt{[}P|R\texttt{]} \wedge R : \mathit{list}
\end{aligned}
$$

and type $\theta_{sort}\ L$ is defined by:

$$
\begin{aligned}
& V : \theta_{sort}\ L = \\
& \quad\ \mathit{isSorted}\, V \\
& \ \wedge \mathit{isPermutation}\ V\ L
\end{aligned}
$$

A more detailed account of how to deal with side-effect-free Erlang code can be found in [15].

5 Case Study: A Distributed Database Lookup Manager

Erlang is used extensively for writing robust distributed telecommunication applications. Central in many of these applications is a distributed database, Mnesia [28], also written in Erlang. The Mnesia system is crucial to the robustness of many Erlang based products 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. 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 [22].

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. Whenever a subquery process receives a message, it extracts the corresponding data from the database tables and sends it 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 reader is referred to [2] for a more detailed description of this case study.

5.1 The Distributed Query Evaluation Protocol

The database tables in which the requested information is stored are distributed over several computers. Whenever a query is formulated for the database the Erlang function `query_setup` is called to analyse the query and to divide it into subqueries each addressing only one table. The subqueries are distributed over a network (by the Erlang function `mk_ring`) 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.

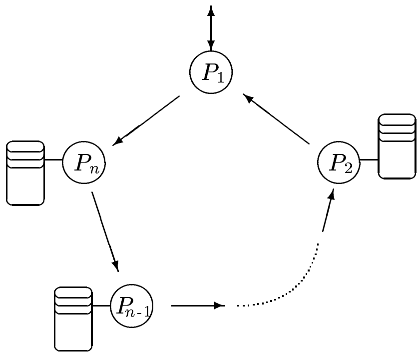

Fig. 1. Ring of processes attached to tables, with P_1 the initial process

We identify an initial process P_1 taking care of a query by partitioning 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 (viz. `process_in_ring`), having the Erlang functions that represents the subquery as an argument. The processes are spawned (by `mk_ring`) in a ring configuration (Fig. 1) and the initial process may be seen as a distinguished member of this ring.

For example, in a telephony accounting system one might wish to charge clients for their telephone calls. Here P_1 could split the initial query, which specifies e.g. a list of client names and asks for their phone costs, into three subqueries. The first one manages the original input list. The second one receives a list of client names and returns a list of pairs of the form {`Name, Number`}, enumerating every phone number that is owned by the respective client. And finally, for each of the numbers, the third subquery collects the target number and the fee of every phone call that was originating from it, returning a list of quadruples of the form {`Name, Number, Target, Fee`}.

After spawning the ring, the initial process 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.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 given by *PacketSize*, which is dynamically determined by P_1, 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 contains less than the demanded number of answers (`NrSolutions`) a message will be sent to the process P_n requesting to produce new answers.

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

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?

5.2 Outline of the Proof

According to the informal property as stated above, 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.

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, a ring segment which includes the initial process.

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 $P_2, \ldots, P_n$.
2. A sort of structural and temporal invariant for a ring segment of the shape $P_1, P_n, \ldots, P_i$ with $2 \leq i \leq n$.

Let us call the first invariant *proc_wait_for_input* and the second invariant for *rootspec*.

The Ring Invariant 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

$$x : \mathit{rootspec}(\cdots) \vdash x : \mathit{spec}(\cdots) \tag{1}$$

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

$$\begin{array}{l}\text{some assumptions} \vdash \\ \quad \mathit{proc}(\texttt{query\_setup}(\cdots), \cdots) : \mathit{rootspec}(\cdots)\end{array} \tag{2}$$

Here *proc* denotes the Erlang process for which *rootspec* has to be proved. Such a process consists of the Erlang term to be evaluated, the process identifier and the message queue associated with the process. Using straightforward, and fully automatable, state exploration techniques we can reduce (2) first to a subgoal of the shape

$$\begin{array}{l}\text{some assumptions} \vdash \\ \quad \mathit{proc}(\texttt{mk\_ring}(\cdots), \cdots) : \mathit{rootspec}(\cdots)\end{array} \tag{3}$$

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

$$\begin{aligned}&\text{some assumptions} \vdash \\ &\quad \mathit{proc}(\texttt{mk\_ring}(\cdots),\cdots) \parallel \\ &\quad \mathit{proc}(\texttt{process\_in\_ring}(\cdots),\cdots) : \mathit{rootspec}(\cdots)\end{aligned} \tag{4}$$

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

$$\begin{aligned}&\text{some assumptions} \vdash \\ &\quad \mathit{proc}(\texttt{process\_in\_ring}(\cdots)) : \mathit{proc\_wait\_for\_input}(\cdots)\end{aligned} \tag{5}$$

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

$$\begin{aligned}&C1 : \mathit{rootspec}(\cdots), \\ &C2 : \mathit{proc\_wait\_for\_input}(\cdots) \vdash \\ &\quad C1 \parallel C2 : \mathit{rootspec}(\cdots)\end{aligned} \tag{6}$$

Subgoal (6) 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, (5) and (6) are not sufficient to conclude (4). However, using (5) and (6) it is possible to reduce to a goal which is actually an instance of the goal (3), 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. [7]).

Properties of the Separate Processes We are thus left with two main subgoals, one of the shape (5), and one of the shape (6). We do not comment further on (6) other than observe that the ring process property *proc_wait_for_input* we are looking for must be strong enough to permit (6) 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 sent 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, model checking 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 ($P_2, \dots, P_n$) 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 a non–*end*–message and sends a message to the next process,
3. the process receives an *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.

6 Extension: Support for Program Libraries

As pointed out in the preceding explanations, the verification of complex distributed systems strongly demands for compositional reasoning methods. Aiming to bring verification technology into industrial applications and to support research on industrially relevant problems in software development, it is neither meaningful nor manageable to start completely from scratch when a new or modified verification problem is being addressed. Instead it should be possible to exploit known properties of subsystems by reusing their proofs.

A compositional reasoning framework will turn out to be useful especially in connection with standard program libraries. Since these are developed to be used frequently, it is worth spending much effort in analysing and describing their properties since many applications will potentially benefit from this knowledge.

In our approach we try to capture the behaviour of library functions by specifying their operational semantics on an abstract level, regardless of their concrete implementation. To this aim we provide rules in the style of Section 2.1 which describe the possible transitions that any Erlang process evaluating the respective function can take, restricted by the shape of the environment if necessary. Adding these rules to the general proof system of Section 2.3 enables us to argue about any program that uses the library module without having to consider the module's source code. In this way we support a compositional style of reasoning which is relativised by the assumption that the concrete implementation of a library follows its specification.

From a pragmatical point of view we can argue that such assumptions are justified since software libraries are usually well–tested, and since their frequent use uncovers unexpected behaviour very soon. From a conceptual point of view however, the consistency between the library–specific transition rules and the concrete implementation with respect to the general proof system is an issue: do the specific rules fully reflect the behaviour of the library functions, or are they too abstract in the sense that certain details of the implementation are ignored although they have an impact on the verification problem? Or, in other words: is the (low–level) implementation of the library module correct with respect to the (high–level) specification?

We now concretely demonstrate our ideas using a specific class of programs which plays an important rôle in open distributed applications. The essential characteristics of this class are described in the following subsection.

6.1 Generic Client–Server Implementations

To support the software development process, the Erlang OTP Team has devised a wide range of design principles which describe how to structure a concrete Erlang software architecture. In particular several kinds of *behaviour* modules are offered as templates to build concrete systems. Among these one finds the `gen_server` behaviour which is widely used to implement client–server applications in a standardized way.

The `gen_server` module offers a number of interface functions which provide synchronous communication, debugging support, error handling, and other administrative tasks. The actual, application–specific implementation of the server has to be provided by the user in a separate module, called the *callback module*. Whenever the generic part of a server receives a request, the corresponding callback function is being invoked.

For example, `gen_server` provides the `call` function which can be invoked in the user process to send a request to a server:

```
Answer = gen_server:call(Server, Req)
```

This request is handled by the generic server process by executing the corresponding callback function:

$$\textit{callback}\texttt{:handle\_call(Req, User, State)}$$

Here, *callback* is the name of the callback module, `User` identifies the user process, and `State` is a term representing the current state of the server. The callback function now decides whether the user should receive a reply immediately (`{reply, Answer, NewState}`), later (`{noreply, NewState}`), or whether the server process should terminate as a result of the request (`{stop, Reason, NewState}`). In the first case, `Answer` is the `call` return value.

6.2 The Approach

To support the verification of client–server systems that employ the above generic implementation scheme, one might think of the following strategies:

- The complete system specification including the callback and the `gen_server` module(s) is fed into the Erlang Verification Tool.
- The generic server implementation in background is eliminated by deriving a standalone Erlang program which reflects the essential behaviour of the system.

The first approach requires no intermediate translation of the program system, but the proof will become much too complicated due to the necessity to consider the details of the generic server implementation. Using the second idea, the proof system has to deal with only one, comparatively simple piece of software. However, the source code has to be translated, and synchronous communication has to be implemented by asynchronous messages, involving a potential state–space overhead.

As explained above, we follow a third approach here. We facilitate the proofs by ignoring the concrete implementation of the `gen_server` module. Instead, we specify its abstract behaviour by including its syntactic constructs in the Erlang syntax, and by adding appropriate transition rules to the proof system. So the intuitive meaning of the `call`/`handle_call` mechanism as described above gives rise to the following set of rules.

A call in the user process can be handled by the server if it is in an idle state, as indicated by the `loop` atom. In this case, the server process executes the `handle_call` callback function, and the user process is put into a wait state until the request has been answered. Formally, this is reflected by the following rule:

$$\begin{aligned}&\langle r[\texttt{call}(\mathit{pid}', \mathit{req})], \mathit{pid}, q\rangle \parallel \langle \texttt{loop}(\mathit{state}), \mathit{pid}', q'\rangle\\ &\quad\longrightarrow \langle r[\texttt{wait}(\mathit{pid}')], \mathit{pid}, q\rangle \parallel\\ &\qquad\quad \langle \texttt{handle\_call}(\mathit{req}, \mathit{from}, \mathit{data}), \mathit{pid}', q'\rangle\end{aligned}$$

Here, $r[\cdot]$ again denotes an arbitrary reduction context.

If the `handle_call` function yields an answer, it is immediately returned to the waiting user process, and the server changes into the idle state again:

$$\begin{aligned}&\langle r[\texttt{wait}(pid')], pid, q\rangle \parallel \\ &\langle \{\texttt{reply}, answer, newstate\}, pid', q'\rangle \\ \longrightarrow\ &\langle r[answer], pid, q\rangle \parallel \langle \texttt{loop}(newstate), pid', q'\rangle\end{aligned}$$

The remaining cases are handled in a similar fashion.

We are currently in the process of extending the proof system by appropriate transition rules and applying it to simple examples, starting with systems which consist of a finite number of clients and servers. Later, for more elaborated case studies, we will try to identify tactics and tacticals which automatically take (most of) the decisions described in Section 3, and we will try to extend the method to programs which involve dynamic process creation. The whole approach is also easily adaptable to several other libraries in the Erlang distribution, like systems of finite–state machines implemented by the generic `gen_fsm` module.

7 Conclusion

The present paper gives an overview of the main results obtained in the ASTEC project Verification of Erlang Programs, focusing in particular on the Erlang verification tool, a theorem-proving tool which assists in obtaining proofs that Erlang applications satisfy their correctness requirements formulated in a specification logic. We gave a summary of the verification framework as supported by the tool, discussed reasoning principles essential for successful verification such as inductive and compositional reasoning and reasoning about side-effect-free code, summarized our experience from a larger industrial case study, and suggested a practical approach for supporting verification in the presence of program libraries.

The experience gained in the project shows clearly the potential of the chosen approach to verification of Erlang programs. We were able to verify Erlang systems which are beyond the scope of most other existing verification approaches due to their dynamic nature. The price to pay is undecidability of the general verification problem. The verification task has to be split into automatable and manually assisted parts. Thus, the success of the approach depends crucially on the efficiency of the decision procedures employed and on the support provided for minimizing the need for human intervention in terms of high-level reasoning principles and user interface.

To make the presented verification method practically useful a considerable additional effort is required in several research directions. These include providing automatic support for identifying appropriate induction schemes, providing easy and context-sensitive access to the available proof machinery through the GUI, and designing efficient decision procedures automating the straightforward low-level reasoning and finite state space exploration.

References

1. J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang (Second Edition).* Prentice-Hall International (UK) Ltd., 1996.
2. T. Arts and M. Dam. Verifying a distributed database lookup manager written in Erlang. In *Proc. Formal Methods Europe'99,* Lecture Notes in Computer Science, 1708:682–700, 1999.
3. T. Arts, M. Dam, L.-å. Fredlund, and D. Gurov. System description: Verification of distributed Erlang programs. In *Proc. CADE'98,* Lecture Notes in Artificial Intelligence, 1421:38–41, 1998.
4. Y. Bertot and L. Thery. A generic approach to building user interfaces for theorem provers. *Journal of Symbolic Computation*, 25(7):161–194, February 1998.
5. R.L. Constable, S.F. Allen, H.M Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice Hall, 1986.
6. M. Dam. Proving properties of dynamic process networks. *Information and Computation*, 140:95–114, 1998.
7. M. Dam, L.-å. Fredlund, and D. Gurov. Toward parametric verification of open distributed systems. In *Compositionality: the Significant Difference,* H. Langmaack, A. Pnueli and W.-P. de Roever (eds.), Springer, 1536:150–185, 1998.
8. M. Dam and D. Gurov. Compositional verification of CCS processes. In *Proc. PSI'99,* Lecture Notes in Computer Science, 1755:247–256, 2000.
9. M. Dam and D. Gurov. μ-calculus with explicit points and approximations. To be presented at *FICS'2000*, 2000.
10. G. Dowek, A. Felty, H. Herbelin, G. Huet, C. Murthy, C. Parent, C. Paulin-Mohring, and B. Werner. The Coq proof assistant user's guide version 5.8. Technical Report 154, INRIA, 1993.
11. L.-å. Fredlund. Towards a semantics for Erlang. Unpublished manuscript*, Swedish Institute of Computer Science*, 1999.
12. L.-å. Fredlund and D. Gurov. A framework for formal reasoning about open distributed systems. In *Proc. ASIAN'99,* Lecture Notes in Computer Science, 1742:87–100, 1999.
13. M. Fröhlich and M. Werner. The graph visualization system daVinci – a user interface for applications. Technical Report 5/94, Department of Computer Science; Universitt Bremen, 1994.
14. M.J.C. Gordon and T.F.Melham (eds.). *Introduction to HOL: a theorem proving environment for higher order logic*. Cambridge Press, 1993.
15. D. Gurov and G. Chugunov. Verification of Erlang programs: Factoring out the side-effect-free fragment. In *Proc. FMICS 2000, GMD Report No.91*, pages 109–122, 2000.
16. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, 1969.
17. F. Huch. Verification of Erlang programs using abstract interpretation and model checking. In *Proc. ICFP '99, ACM SIGPLAN Notices*, 34(9):261–272, 1999.
18. D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, **27**:333–354, 1983.
19. R. Milner. *Communication and Concurrency.* Prentice Hall International, 1989.
20. R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML – Revised.* MIT Press, 1997.

21. George C. Necula. Proof-carrying code. In *Proc. POPL'97*, 1997.
22. H. Nilsson. Patent Application, 1999.
23. S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. K. Srivas. PVS: Combining specification, proof checking, and model checking. In *Proc. CAV'96,* Lecture Notes in Computer Science, 1102:411–414, 1996.
24. D. Park. Finiteness is mu-Ineffable. *Theoretical Computer Science*, **3**:173–181, 1976.
25. L.C. Paulson. *Isabelle: A Generic Theorem Prover.* Springer Verlag (LNCS 828), 1994.
26. G. D. Plotkin. A structural approach to operational semantics. Aarhus University report DAIMI FN-19, 1981.
27. D. Sahlin, T. Franzén, and S. Haridi. An intuitionistic predicate logic theorem prover. In *Journal of Logic and Computation*, 2(5):619–656, October 1992.
28. C. Wikström, H. Nilsson, and H. Mattson. Mnesia database management system. In *Open Telecom Platform Users Manual.* Open Systems, Ericsson Utvecklings AB, Stockholm, Sweden, 1997.