Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

A Verification Tool for Erlang

Thomas Arts[1], Gennady Chugunov[2], Mads Dam[2], Lars–Åke Fredlund[2], Dilian Gurov[2], Thomas Noll[3]

[1] Ericsson Computer Science Laboratory, Ericsson Utvecklings AB, e-mail: thomas@cslab.ericsson.se
[2] Swedish Institute of Computer Science, e-mail: gena@sics.se,mfd@sics.se,fred@sics.se,dilian@sics.se
[3] Lehrstuhl für Informatik II*, Aachen University of Technology, e-mail: noll@cs.rwth-aachen.de

The date of receipt and acceptance will be inserted by the editor

Abstract. This paper presents an overview of the main results of the project "Verification of Erlang Programs", which is funded by the Swedish Business Development Agency (NUTEK) and by Ericsson within the ASTEC (Advanced Software TEChnology) initiative. Its main outcome is the Erlang Verification Tool (EVT), a theorem prover 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 EVT, discuss reasoning principles essential for successful proofs such as inductive and compositional reasoning, and an efficient treatment of side–effect–free code. The experiences of applying the tool in an industrial case study are summarised, and an approach for supporting verification in the presence of program libraries is outlined.

EVT 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 the 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

Erlang is a programming language developed at the Ericsson corporation for implementing telecommunication systems [1]. It provides a functional sublanguage, enriched with constructs for dealing with side effects such as process creation and inter–process communication. Today many commercially available products offered by Ericsson are at least partly programmed in Erlang. The software of such products is typically organised into many, relatively small source modules, which at runtime execute as a dynamically varying number of processes operating in parallel and communicating through asynchronous message passing. The highly concurrent and dynamic nature of such software makes it particularly hard to debug and test. We therefore explore the alternative of proof system-based verification. The core fragment of the Erlang language is economic and clean, allowing a compact transitional semantics, and component interfaces can be elegantly specified in a modal logic with recursion, suggesting feasibility of such an endeavour.

Rather than working with some abstract model of the Erlang system under consideration, our verification approach is directly based on the code: we show that a concrete Erlang program 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. It semantically subsumes the temporal logics CTL, CTL*, and LTL. By employing a macro mechanism, named formulas with parameters can be introduced to represent typical properties such as safety, liveness, and fairness conditions. The verification problem in this rather general context is not decidable, but can be automated to a considerable extent, requiring human intervention at a few, but critical points.

Verifying recursive temporal properties of systems with dynamically evolving process structures and unbounded data is known to be hard. It requires a rich verification framework supporting reasoning which is para-

* Most of the work was completed during the author's employment at the Department of Teleinformatics, Royal Institute of Technology (KTH), Stockholm.

metric on components, relativised on the properties of components, compositional, and provides support for inductive and co-inductive reasoning about recursively defined components [7,8,12]. Due to the concurrency and dynamism inherent in the systems we address, a variety of (mutual) induction schemes need to be available; at the same time it is often unlikely to foresee which of these might work. We therefore employ symbolic program execution and instance checking to "discover" induction schemes. Our machinery is based on fixed-point ordinal approximation and well-founded ordinal induction, and on a global discharge proof rule for ensuring consistency of the mutual inductions present in a proof structure.

The effort on the verification of Erlang programs is taking 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 is funded by the ASTEC (Advanced Software TEChnology) competence center of the Swedish Business Development Agency (NUTEK). So far, the project activities have 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].

This paper presents an overview of the main results of this project, and focuses in particular on the Erlang Verification Tool (EVT), a theorem prover that assists in establishing formal correctness properties of Erlang applications. Although EVT has been applied in case studies working with real–life software (see below), it must be stressed that it is not intended to be used by the Erlang programmers themselves. In its current state, it should rather be understood as a "proof of concept", demonstrating that applying formal methods to Erlang programs is feasible, at least if done by experts. Efficiency and user-friendliness being crucial aspects of a practical verification tool, we opted for designing a special purpose theorem prover rather than trying to embed our rich and complicated verification framework into some existing general purpose theorem proving environment.

The paper is organized as follows. In Section 2 we summarise the verification framework supported by EVT: 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 which are less often found in comparable tools like Coq [10], Isabelle [25], NuPrl [5], and PVS [23], notably the discharge mechanism which implements a well–founded induction scheme to handle infinitary behaviour. 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 for dealing with such classical functional code is absolutely essential, and is also discussed in Section 4. Section 5 illustrates the reasoning principles using a simple example. Section 6 summarises our experience with using EVT in a typical case study: the verification of a distributed database query evaluation protocol. Section 7 presents an approach for dealing with modularity in an elegant fashion. Typical Erlang applications make extensive use of standard libraries which implement everything from basic operations on lists to complex distributed data bases. The paper proposes a semantic approach to extending the capabilities of EVT for handling library modules without considering the actual implementation of these modules. The next section presents related work. The paper is concluded with a section on discussion of the merits and shortcomings of the tool.

2 Foundations

In this section we briefly highlight the foundations of our approach: 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 implementing open distributed (telecom) systems: the language Erlang with support for concurrency, and middleware OTP (Open Telecom Platform) providing ready–to–use components (libraries) and services such as a distributed data base manager, support for "hot code replacement", and design guidelines for using the components.

2.1.1 Syntax of Core Erlang

In the following we consider a core fragment of the Erlang programming language which allows to implement 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, mechanisms for controlling the distribution of processes (onto computation nodes), and support for interoperation with non–Erlang code written in, e.g., C or Java.

Besides Erlang *expressions* e the syntactical categories of *matches* m, *patterns* p, *guards* g, and *values* v are considered. The abstract syntax of Core Erlang expressions is:

$$
\begin{array}{rcl}
e & ::= & bv \mid [e_1 \mid e_2] \mid \{e_1, \ldots, e_n\} \mid var \\
& \mid & e(e_1, \ldots, e_n) \\
& \mid & \texttt{begin}\ e_1, \ldots, e_n\ \texttt{end} \\
& \mid & \texttt{case}\ e\ \texttt{of}\ m\ \texttt{end} \\
& \mid & \texttt{catch}\ e \\
& \mid & \texttt{receive}\ m\ \texttt{end} \\
& \mid & e_1 ! e_2 \\
\\
bv & ::= & atom \mid number \mid pid \mid [] \mid \{\} \\
v & ::= & bv \mid [v_1 \mid v_2] \mid \{v_1, \ldots, v_n\} \\
\\
m & ::= & p_1\ \texttt{when}\ g_1\ \texttt{->}\ e_1\ ;\ \cdots\ ;\ p_n\ \texttt{when}\ g_n\ \texttt{->}\ e_n \\
p & ::= & bv \mid var \mid [p_1 \mid p_2] \mid \{p_1, \ldots, p_n\} \\
g & ::= & e_1, \ldots, e_n
\end{array}
$$

The Erlang values consists of a set of atom literals (with an initial lowercase letter), the numbers (here integers only), process identifier constants ranged over by *pid*, and tuples and conses. The variables (ranged over by *var*) are symbols starting with an uppercase letter.

To support the understanding of the remaining syntactic constructs, we anticipate some elements of the formal semantics which is going to be discussed in Section 2.1.2. An Erlang *process*, here written <e,pid,q>, 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. The semantics of Erlang specifies perfect (non-lossy) communication channels of an unbounded size. The empty queue is `eps`, $[[v]]$ is the queue containing the one element v, and $q_1 @ q_2$ concatenates the queues q_1 and q_2. To express the concurrent execution of two sets of processes s_1 and s_2, the syntax $s_1 \,||\, s_2$ 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. The main choice construct of Erlang is by matching:

$$
\begin{array}{l}
\texttt{case}\ e\ \texttt{of} \\
\quad p_1\ \texttt{when}\ g_1\ \texttt{->}\ e_1\ ; \\
\quad\ \vdots \\
\quad p_n\ \texttt{when}\ g_n\ \texttt{->}\ e_n \\
\texttt{end}
\end{array}
$$

A guard g_i can be omitted; in this case, the trivially true guard `true` is assumed. The value that e evaluates to is matched sequentially against patterns (values that may contain unbound variables) p_i, respecting the optional guard expressions g_i which are expressions that, due to syntactic restrictions, are guaranteed to compute without side effects and terminate.

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 process identifier. More concretely, upon evaluation of the expression $e_1 ! e_2$ 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 some pattern in m. Once such an element v has been found, evaluation proceeds analogously to `case` v `of` m `end`.

In addition side effects are possible through builtin functions like `self`(), yielding the process identifier 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 which executes the function call $f(v_1, \ldots, v_n)$, where the process identifier of the new process is returned by the call to `spawn`.

Expressions are interpreted relative to an environment of "user defined" function definitions of the shape:

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

Example 1. We shall illustrate the intuitive meaning of Erlang programs using a simple but typical example. Consider a concurrent server which repeatedly takes an incoming request from its message queue and spawns off a process to serve it:

```
central_server() ->
  receive
    {request, Request, Client} ->
      spawn(serve, [Request, Client]),
      central_server()
  end.

serve(Request, Client) ->
  Client!{response, handle(Request)}.

handle(Request) ->
  ok.
```

Note above that Erlang variables are always upper–case (`Request` and `Client`) while atoms are lower–case (`central_server`, `request` etc.). The `central_server` function is continuously prepared to receive tuples containing a request `Request` and a process identifier `Client`. It then spawns off a new process evaluating the `serve` function, which simply invokes the `handle` function (just returning an atom here) and sends the result back to the process identified by `Client`.

Since Erlang is not statically typed 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, e.g. in the case that `Client` does not refer to a valid process identifier.

2.1.2 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]. As mentioned earlier, the latter are given by parallel products of processes. 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.

Here we are faced with the question how to handle the different conceptual layers of entities in the language, i.e., functional expressions and concurrent processes, such that modular (i.e., compositional) reasoning is supported. A natural approach 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.

More concretely, 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 $f(v_1, \ldots, v_n) \rightsquigarrow v$ which represents the calling of a builtin function f (like `spawn` for process spawning) with side effects on the process level. Here the non–τ actions denote side effects of expression evaluations on the next level of the semantics.

The second semantics layer concerns concurrent processes executing expressions in the context of a unique process identifier and a mailbox of incoming messages. Their operational behaviour 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 silent 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 builtin function with side effects, like for instance `spawn`. On the level of Erlang expressions the evaluation of such a function is covered by the transition rule

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

where the *isProcFun* predicate recognizes the names of builtin functions with side effects, 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. On the process level spawning is handled more directly in the following rule:

$$\frac{e \xrightarrow{\texttt{spawn}(f,[v_1,\ldots,v_n])\rightsquigarrow pid'} e' \quad pid' \neq pid}{\begin{array}{c}\texttt{<}e, pid, q\texttt{>} \\ \xrightarrow{\tau} \\ \texttt{<}e', pid, q\texttt{>} \;||\; \texttt{<}f(v_1, \ldots, v_n), pid', \texttt{eps>}\end{array}}$$

One of the interleaving rules also makes a provision for process spawning:

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

The condition $pid' \neq pid$ ensures that the process identifier of the newly spawned process is locally unique, and the condition $pids(s_1') \cap pids(s_2) = \emptyset$, where $pids(s)$ returns the process identifiers of processes in s, guarantees the same under parallel composition.

2.2 The Property Specification Language

Verifying properties of 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, offering box and diamond *modalities* employing the transition labels: a structured state s satisfies the 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 application. In the following we let t range over general terms, T over variables representing terms, and S over the sorts (types), although these sorts will usually not be written out in formulas. Sorts are used to distinguish terms of the different syntactical categories of Erlang, such as expressions, process identifiers, atoms, or processes.

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 $\nu X.\phi$ denote invariant (safety) properties. Nesting of fixed points allows complicated reactivity and fairness properties to be specified. 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 as follows:

$$
\begin{array}{lr}
\phi ::= t_1 = t_2 & \text{(equality)} \\
\mid \ \texttt{true} \mid \texttt{false} & \text{(truth values)} \\
\mid \ \neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 & \text{(connectives)} \\
\mid \ \exists T : S.\phi \mid \forall T : S.\phi & \text{(quantifiers)} \\
\mid \ \lambda T : S.\phi \mid \phi\ t & \text{(abstraction/application)} \\
\mid \ \langle\alpha\rangle\,\phi \mid [\alpha]\,\phi & \text{(modalities)} \\
\mid \ \nu X.\phi \mid \mu X.\phi \mid X & \text{(fixed points)} \\
\mid \ \kappa < \kappa' & \text{(ordinal inequations)} \\
\mid \ t_1 \xrightarrow{\alpha} t_2 & \text{(transition assertions)}
\end{array}
$$

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). Moreover we sometimes denote an application of the form $\phi\ t$ by $t : \phi$.

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. 1. The type of natural numbers is the least set which contains zero and which is 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. An interesting property of the concurrent server (cf. Example 1) is *stabilization*, i.e. the convergence on output and silent actions. This liveness property expresses that, assuming that no input is being received, the process is able to execute only a finite number of output and silent steps:

$$stabilizes \ \Leftarrow \ \lambda S.\begin{pmatrix} \forall P.\forall V.[P!V]stabilizes\ S \\ \wedge\ [\tau]stabilizes\ S \end{pmatrix}$$

2.3 The Proof System

Verifying correctness properties of 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 a Gentzen–style proof system, allowing free parameters to occur within the *proof judgments*. The judgments are of the form $\Gamma \ \vdash \ \Delta$ where Γ and Δ are sequences 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:

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

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

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 and the discharge rule 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[1]. This tool has been tailored to this 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 theorem provers like Coq [10], Isabelle [25], NuPrl [5], or PVS [23], 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 tool would permit. Moreover most existing theorem provers are rather inflexible in that they offer a set of predefined induction schemes, from which the user has to choose one at the outset of the proof. This contrasts with our ambition to discover

[1] `http://www.sics.se/fdt/VeriCode/evt.html`

induction schemes through a lazy search procedure in the course of the proof.

Two notable releases of EVT exist. 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, an experimental embedding of a variant of the value–passing Calculus of Communicating Systems [19] (CCS for short) exists. The current tool is, like the theorem provers HOL [14] and Isabelle [25], implemented in Standard ML [20].

3.1 Terms, Variables, Formulas, and Proofs

EVT 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 (e.g., to support Erlang syntax). Likewise derived formula constructs, with language–specific semantics, can be defined. The introduction of subtyping in the underlying theory can, as usual, introduce typing–related proof obligations during parsing of terms and formulas.

For types considered to be freely generated (intuitively those types where "semantic equality" coincides with the syntactic notion of equality) such as 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 associating indices with variables, based on [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 EVT involves the "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 a proof rule 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 the node of which it is an instance, called the *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], which generally require the proof representation to be independent of the underlying machinery).

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 the 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. A finished proof graph is a proof graph that contains no open goals.

The application of a proof rule can be cancelled (undone), resulting potentially in non–local cancellation effects on the proof tree when e.g. the companion node of a copydischarge node is cancelled, naturally also causing the copydischarge to fail. Another such problematic case is when a meta variable is assigned or cancelled in one proof branch, but where this variable is also present in another branch. In such a situation both the assignment and the 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`, meta variables are of type `var`, and if 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. Assertions, on both sides of a sequent, are numbered starting from one. Thus, for instance, the tactic implementing the proof rule $\forall_R$ has the signature

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

Being applied to a position i where, in the current goal sequent, the ith assertion on the right–hand side is universally quantified, the quantified variable is replaced by a fresh variable.

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 are

```
t_compose: tactic -> (tactic list) -> tactic
t_orelse:  tactic -> tactic -> tactic
t_fix:     'a ->
           ('a -> ('a -> tactic) -> tactic) ->
           tactic
```

The tactical `t_compose t tl` applies the tactic `t` to the current sequent and then applies the tactics in the list `tl` to the corresponding resulting goals, failing if `t` does so or if the number of goal sequents does not match the number of tactics in `tl`. To evaluate `t_orelse t1 t2`, first `t1` is applied, and `t2` is applied only if `t1` fails. 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.

3.3 User Interface 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 "the current proof graph" and "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 through proof graphs (i.e., to redefine the current proof node), to navigate through the hierarchy of proof graphs, to extend (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 on 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.

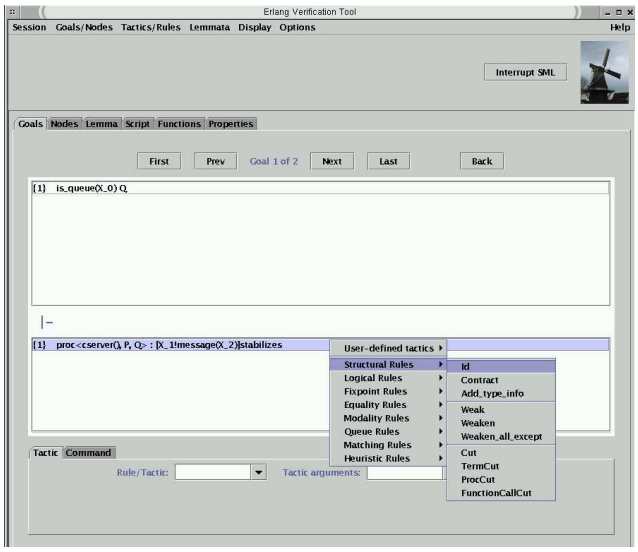

Fig. 1. The Graphical User Interface of EVT

A second, graphical, user interface is also available. 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. A screenshot of a proof session using the graphical user interface is shown in Figure 1. 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] graph visualisation system. Experiences with the graphical interface indicate that the initial training period required to become familiar with the tool is considerably shortened. However, for experienced users the command-line interface sofar remains the interface of choice.

3.4 Checking Discharge Conditions

Of particular interest is the implementation of the discharge proof rule. Consider a proof node N_d, henceforth called the *discharge node*, representing an open proof goal of the form $\Gamma_d \vdash \Delta_d$. Assume that there exists an ancestor node N_c in the proof tree, henceforth called the *companion node*, labelled by a sequent $\Gamma_c \vdash \Delta_c$. Here the discharge proof rule can be used to check whether the companion node, and some auxiliary conditions formulated on the global proof graph, motivate the discharging of the discharge node. 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 given in [7,9], here only a sketch is given.

The overall idea is to keep track of unfoldings of fixed–point definitions by annotating fixed points with ordinal variables, representing 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) results in the replacement of the 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.

For example, the rules for manipulating a greatest fixed point on the right–hand side, occurring under applications, are:

$$\frac{\Gamma \vdash ((\mu X.\phi)^\kappa)\ t_1\ \ldots\ t_n, \Delta}{\Gamma \vdash (\mu X.\phi)\ t_1\ \ldots\ t_n, \Delta}\,\kappa \text{ fresh}$$

$$\frac{\Gamma, \kappa' < \kappa \vdash (\phi\{(\mu X.\phi)^{\kappa'}/X\})\ t_1\ \ldots\ t_n, \Delta}{\Gamma \vdash ((\mu X.\phi)^\kappa)\ t_1\ \ldots\ t_n, \Delta}\,\kappa' \text{ fresh}$$

Above, κ ranges over ordinal variables. Intuitively the first rule corresponds to commencing a co-induction (on the unfolding of the fixed point), and the second records the existence of an lesser ordinal as the inequation $\kappa' < \kappa$. As a side-effect the term vector $t_1 \ldots t_n$ is kept in the unfolded fixed point (as in Winskel's [29] *tagging* technique). This is used in proof search to heuristically determine whether unfolding is a progressing proof step.

The discharge proof rule then comprises checking three conditions, given a proof node $N_d \equiv \Gamma_d \vdash \Delta_d$ and a candidate companion node $N_c \equiv \Gamma_c \vdash \Delta_c$:

- Is there a mapping from N_c to N_d? That is, does a substitution ρ exists such that (i) for each $\phi \in \Gamma_c$, $\phi\rho \in \Gamma_d$ and (ii) for each $\phi \in \Delta_c$, $\phi\rho \in \Delta_d$
- Does some ordinal decrease on the path between N_c and N_d? That is, is there some ordinal variable κ occurring in N_c such that $\Gamma_d \vdash \kappa\rho < \kappa$
- The previous two conditions where local, i.e., involving only one pair of discharge and companion nodes. The third condition is a global one which examines all related discharges throughout the proof tree to ensure that discharges cannot cancel each other (theoretical details are elaborated in [7,9]). In essence this corresponds to checking whether the global-proof tree defines a proper simultaneous fixed point induction scheme.

3.5 Embedding of Erlang

The Erlang program constructs are encoded as terms of the many–sorted first–order logic. The tool 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 (T is assumed fresh in Γ, Δ):

$$\|\ ?_L\ \frac{\Gamma, s_1 \xrightarrow{pid?v} T, s' = T \parallel s_2 \vdash \Delta}{\Gamma, s_1 \parallel s_2 \xrightarrow{pid?v} s' \vdash \Delta}$$

In general the handling of the operational semantics in EVT is split into two parts: a 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 try to achieve the result of the rules $\langle\,\rangle_r$ and $[\,]_r$ given below. An underlying assumption of these rules is that the program term t has a sequence of transitions $t \xrightarrow{\alpha} t_1, \ldots, t \xrightarrow{\alpha} t_n$ under the action α, and that no other such continuation state t_x exists.

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

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

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 such as splitting conjunctions and term equality reasoning. In addition language dependent tactics and lemmas for handling data are appealed to.

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 taken which prohibits the verification of industrial–scale software. To make our verification method applicable in this setting, we have to lift the reasoning to a suitably high level of abstraction. *High–level*

reasoning means proving "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 of 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. Section 4.3 gives 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 Section 6:

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

Induction on the number of evaluation 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 a considerable extent. To handle these complications, most parts of the proof will involve induction and co–induction at many levels simultaneously, 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 fact the discharge mechanism as described in the previous section attempts to cast the proof as constructed so far 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 at all. All this is managed by the tool. Furthermore, the tool supports, through the discharge mechanism, the discovery of successful induction schemes; for making informed decisions, however, the user will need to have a basic understanding of the general principles of simultaneous fixed-point induction.

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 s containing component t can be represented through term substitution as $s[t/T]$, where T is a variable ranging over entities of the type of t. We can relativise an assertion $s[t/T] : \phi$ about the compound object $s[t/T]$ to a certain property ψ of its component t by considering t as a parameter for which property ψ is assumed, provided we can show that t indeed satisfies the assumed property ψ. Technically, we achieve this through a *term–cut* proof rule of the shape:

$$(\mathsf{Term\,Cut})\ \frac{\Gamma \vdash t : \psi, \Delta \qquad \Gamma, T : \psi \vdash s : \phi, \Delta}{\Gamma \vdash s[t/T] : \phi, \Delta}$$

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 s has been spawned off by a recursive process t one can apply the above term–cut rule to relativise the proof on the specification of t rather than on its implementation, thus avoiding new processes from being generated by t 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 t 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, the list–sorting function `sort` could be used by the `handle` function in the concurrent server (cf. Example 1) to process list sorting requests:

```
handle ({srt, L}) ->
  sort (L).
```

If we want to prove a property of such a sorting server, we would like to reason on a high–level and replace function calls to the list–sorting library function `sort` with argument L, 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 `sort` 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 that

- e is a call of a function f, that
- the body of the definition of f is side–effect–free, and that
- ϕ is *insensitive* to (the number of) side–effect–free actions,

the following decomposition principle is applicable:

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

where *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 the 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: if the parameters of e satisfy the precondition ψ, the execution of e, provided it terminates, results in a value satisfying the postcondition θ. We follow the same idea, but require termination; a correctness notion known as *total correctness*.

We use assertions of the form $e : prepost(\psi,\theta)$ where

$$e : prepost(\psi,\theta) = (\psi \to e : eval\ \theta)$$

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

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

$$\begin{array}{l} L : list \Leftarrow \\ \qquad L = [] \\ \quad \vee \exists P, R : \texttt{erlangValue}. L = [P|R] \wedge R : list \end{array}$$

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

$$\begin{array}{l} V : \theta_{sort}\, L\ = \\ \qquad isSorted\, V \\ \quad \wedge\, isPermutation\ V\ L \end{array}$$

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

5 Example

We shall illustrate the ideas presented above using the Erlang program from Example 1. Recall the definition of the concurrent server which repeatedly takes a request from its message queue and spawns off a process to serve it by handling the request, here always assumed to succeed, and responding with the obtained result to the client specified in the request:

```
central_server() ->
  receive
    {request, Request, Client} ->
      spawn(serve, [Request, Client]),
      central_server()
  end.

serve(Request, Client) ->
  Client!{response, handle(Request)}.

handle(Request) ->
  ok.
```

The property we consider is the liveness property from Example 3, namely *stabilization*, i.e., the convergence on output and silent (`estep`) actions. We recall its definition from Example 3:

$$\begin{array}{l} stabilizes\ \Leftarrow \\ \qquad \lambda S. \begin{pmatrix} \forall P.\forall V.[P!V] stabilizes\ S \\ \wedge\ [\tau] stabilizes\ S \end{pmatrix} \end{array}$$

So, the initial proof goal is declared as:

```
declare P:erlangPid, Q:erlangQueue in
  |- <central_server(), P, Q> : stabilizes
```

In the proof sketch below we illustrate the interplay between automated proof search – leading to discovery of proof structures such as induction strategies – and

manual proof steps realising the discoveries in a revised proof attempt.

The following proof search script results in a symbolic execution of the process until either a system which is not a singleton process, or a repetition of the same control state is encountered:

```
loop
(case_by
 [(sp_and (sp_sat_sysproc_r 1)
          (sp_not (sp_sat_is_queue_var_r 1)),
   t_queue_flat_r 1),
  (sp_and (sp_sat_sysproc_r 1)
          (sp_unfoldable_r 1),
   t_gen_unfold_r 1)
 ]);
```

In the first case, if the first right–hand side formula is a satisfaction pair the first part of which is a single process the queue term of which is not a variable, the `t_queue_flat_r` tactic is applied which replaces the term with a fresh variable and adds an equation to the left equating this fresh variable with the queue term. This is done to insure that, in the second case, the pre-instance checking mechanism based on `sp_unfoldable_r` detects control-point repetition. Execution of the above proof search script terminates because a new process was spawned (and thus `sp_sat_sysproc_r` failed). The result is the sequent:

```
Q = Q2@[[{request,Req,ClPid}]]@Q3,
Q1 = Q2@Q3, not (P = P1) |-
   <begin P1, central_server() end, P, Q1>
|| <serve (Req, ClPid), P1, eps> : stabilizes
```

The queue `Q2@[[{request,Req,ClPid}]]@Q3` is built from the concatenation of three parts, `Q1`, the value `[[{request,Req,ClPid}]]` and `Q3`. We have now a clear indication that the number of processes in the system will grow without bound, so a blind proof search is bound to fail. Rather, one has to proceed by *induction on the system structure*. This is achieved through compositional reasoning by abstracting away the first process component which is responsible for the unbounded dynamic process creation, and relativising the argument on a property of this component. The choice of a suitable property is crucial, of course, for the induction to succeed. In our particular example it happens that `stabilizes` composes. We apply the term-cut rule to obtain the two new goals:

```
|- <begin P1, central_server() end, P, Q1> :
   stabilizes

X : stabilizes |-
  X || <serve (Req, ClPid), P1, eps> :
  stabilizes
```

the first of which corresponding to the induction basis, and the second corresponding to the induction step. The first of these can be be analysed by the script presented above, terminating with the goal

```
|- <central_server(), P, Q1> : stabilizes
```

because of detecting a pre-instance (we looped back to the initial control point), causing `sp_unfoldable_r` to fail. One might expect to be able to discharge here w.r.t. the initial goal, but this fails. The reason is that no ordinal has been decreased. However, by inspecting the proof state we realize that the length of the queue of the process has decreased, and that indeed stabilization of the server is a consequence of the well–foundedness of message queues. We therefore return to the initial goal and re-declare it by adding an explicit assumption on the well-foundedness of the queue, which will be maintained throughout the proof:

```
declare P:erlangPid, Q:erlangQueue in
  Q : queue |-
  <central_server(), P, Q> : stabilizes
```

$$\begin{array}{l} \mathit{queue} \Leftarrow \\ \quad \lambda Q : \texttt{erlangQueue}. \\ \left(\begin{array}{l} \quad Q = \texttt{eps} \\ \vee\ \exists V. \exists Q_1. \exists Q_2. Q = Q_1 \texttt{@} [[V]] \texttt{@} Q_2 \wedge \mathit{queue}\ Q_1 \texttt{@} Q_2 \end{array} \right) \end{array}$$

The revised proof will turn out to be, at least partly, by *induction on the queue-term structure*. All we have to change in the beginning is to approximate the left formula, resulting in `Q : queue` being replaced by `Q : queue(K)` where `K` is an approximation ordinal, and to proceed as before. This eventually results in:

```
Q2@[[{request,Req,ClPid}]]@Q3 : queue(K),
Q1 = Q2@Q3 |-
  <central_server(), P, Q1> : stabilizes,
```

in place of the unsuccessful goal we ended up with earlier. This goal is "almost" dischargable w.r.t. the initial goal after approximation. For the instance check to go through, one needs `Q1 : queue(K1)`, for some ordinal variable `K1<K`, instead of `Q2@[[{request,Req,ClPid}]]@Q3 : queue(K)` to appear as an assumption in the sequent. We therefore unfold `queue(K)` via `t_gen_unfold_l`, followed by transferring the queue-term assumption via `t_queue_invar_l` to obtain a dischargable goal.

The important goal we are left with is the sequent corresponding to the induction step. Fortunately, it can be dealt with by the same proof script as the initial goal, with the important difference that no new processes will be spawned. Parameter-assumption transfer, however, concerns in this case not the queue but the process parameter `X`. And the number of control states will grow due to the presence of two concurrent processes.

6 Report on a Verification Experience: the Analysis of 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 (see Figure 2). The task we set ourselves was to prove that the implementation provided a responsiveness property: that input queries are eventually being replied to.

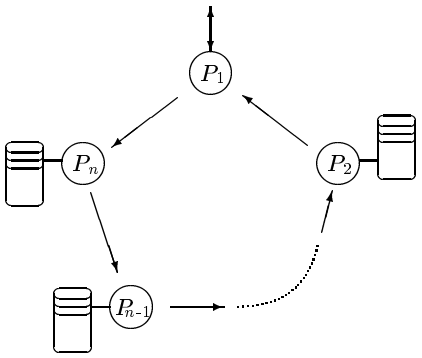

Fig. 2. Ring of processes attached to tables, with P_1 the initial process

6.1 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 of 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. The most sophisticated of those proofs contains about 2000 proof nodes, of which two-third has been generated automatically. To help visualisation the daVinci graph display facility [13] was used. 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 (which is not very well supported currently) or in segments, to avoid excessive delays.

6.2 Conclusions on the Database Case Study

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 code.

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 rôle not unlike names in the π-calculus. Even though our handling of process identifiers and their 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.

The reader is referred to [2] for a more detailed description of this case study.

7 Extension: Support for Program Libraries

As pointed out in the preceding explanations, the verification of complex distributed systems requires 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 and programming techniques. Since these are developed to be used frequently, it is worth spending a considerable effort in analysing and describing their properties since many applications will potentially benefit from this knowledge.

An alternative to the approach we advocated in Section 4.2 is 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 2.1.2 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?

Pragmatically, our concern is to provide a framework in which we can prove properties of the code in an abstract setting, where we use one abstraction for all possible properties. This abstraction is very close to the real implementation, but there will always exist properties for which it turns out to be too general. However, if we can prove a certain property about the abstraction, then we increased the level of confidence in the code; if we find that a certain property does not hold by reasoning in this abstracted setting, then, most likely, this corresponds to an error in the real program.

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.

7.1 Generic Client–Server Implementations

To support the software development process, the Erlang/OTP Development 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:

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

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

```
callback: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.

7.2 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` modules is fed into EVT.
- 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:

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

When 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:

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

As can be seen, the asynchronous communication actions that are used in the `gen_server` module to implement synchronous message passing are collapsed into an atomic handshake. The remaining functions are represented in a similar fashion.

So far we have extended the proof system by appropriate transition rules and applied it to simple examples, starting with systems which consist of a finite number of clients and servers. Currently, for more elaborated case studies, we are trying 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 should also be easily adaptable to several other libraries in the Erlang distribution, like systems of finite–state machines implemented by the generic `gen_fsm` module.

8 Related Work

In this section we shortly review other verification frameworks which support deductive systems tailored towards formal reasoning about programming languages, ignoring theorem–proving systems designed for the formalization of classical or constructive mathematics, such as Coq, HOL, or Nuprl.

ACL2[2], the successor of the "Boyer–Moore theorem prover" Nqthm, supports the first–order logic of total recursive functions with equality, offering mathematical induction on ordinals as the main proof method. Within this framework it is possible to define models of various kinds of computing systems and to prove theorems about them. Succesful industrial–scale applications of this approach include correctness proofs of several assembler programs for a Motorola signal processor and of the floating–point division unit of an AMD microprocessor.

Another popular system is the *Isabelle* generic theorem proving environment[3]. Its meta logic, called Isabelle/Pure, is used to declare the (concrete and abstract) syntax and the semantics (i.e., the inference rules) of a concrete logic. Moreover it allows to instantiate generic proof tools such as a general tableau prover to obtain a specific prover, or to manually code specialized proof procedures. Concrete programming–oriented applications of this framework comprise verification tools for the Java programming language, for distributed systems specified using I/O automata or the UNITY language, and for object–oriented programs.

Examples for other verification systems of this kind are ELAN[4] and Larch[5].

The specification language of the *PVS* theorem prover[6] is based on classical, typed higher–order logic supporting functions, sets, records, tuples, enumerations, recursively–defined abstract data types, predicate subtypes, and dependent typing. PVS provides a collection of proof rules that are applied interactively under user guidance within a sequent calculus framework. Just like EVT the prover

[2] `http://www.cs.utexas.edu/users/moore/acl2/`
[3] `http://www.cl.cam.ac.uk/Research/HVG/Isabelle/`
[4] `http://www.loria.fr/ELAN/`
[5] `http://www.sds.lcs.mit.edu/spd/larch/`
[6] `http://pvs.csl.sri.com/`

maintains a proof tree where the nodes are labeled by sequents. The primitive proof rules include propositional and quantifier rules, equational reasoning, induction, rewriting, and decision procedures for linear arithmetic.

All of the above frameworks could be applied, at least in principle, to the verification of Erlang programs as well. To this aim, the syntactic constructs and their meaning have to be defined in the corresponding specification formalism. With regard to the logic, however, one would be dependent on those proof methods which are predefined in the respective system. For example this means that, at the outset of a proof, the user has to choose from a collection of predefined induction schemes. This requirement is in contradiction to our intention to support the lazy discovery of complicated induction schemes through symbolic program execution, which is essential for the practical verification of temporal properties of programs with dynamic behaviour.

Of course the price to be paid for this flexibility is the missing generality of our system with respect to the specification language, which makes it a special–purpose theorem prover tailored towards the Erlang language.

An alternative approach to the verification of Erlang programs is the use of abstract interpretation techniques to create a finite–state model of the given program which can be handled 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.

9 Conclusion

We have given 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 presented a summary of the verification framework as supported by EVT, 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 method for supporting verification in the presence of program libraries.

The experience gained in the project clearly shows the potential of the chosen framework. We were able to verify Erlang systems which, due to their dynamic nature, are beyond the scope of most other existing verification approaches. The price to pay is the 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 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 graphical user interface, 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. In: *Proc. 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.
29. G. Winskel. A note on model checking the modal ν-calculus. *Theoretical Computer Science*, 83:157–187, 1991.