# The Erlang Verification Tool

Thomas Noll[1], Lars–åke Fredlund[2], and Dilian Gurov[2]

[1] Lehrstuhl für Informatik II***, Aachen University of Technology, Aachen, Germany, `noll@cs.rwth-aachen.de`

[2] Swedish Institute of Computer Science (SICS), Kista, Sweden, `{fred,dilian}@sics.se`

## 1 Introduction

The functional programming language Erlang was developed by the Ericsson corporation to address the complexities of developing large–scale programs within a concurrent and distributed setting. It is successfully used in the design and implementation of telecommunication systems.

Software written for this application domain usually has to meet high quality demands such as correctness. Due to the high degree of concurrency and to the dynamic behaviour of systems, testing is generally not sufficient to guarantee these properties to a satisfactory degree. We therefore follow a verification approach, i.e., we employ formal methods to prove that a telecommunication system implemented in Erlang has certain properties specified in a suitable logic.

In view of the complexity of the verification problem in general it is mandatory to provide the user with powerful tool support. Therefore, in 1997 the development of the EVT Erlang Verification Tool was started at the Swedish Institute of Computer Science in collaboration with and with financial support from the Ericsson Computer Science Lab and the Swedish ASTEC (Advanced Software TEChnology) competence centre.

To cope with the challenges of software verification in a highly dynamic setting, a semi–automatic theorem–proving approach was chosen as the underlying framework. EVT is a proof assistant which offers powerful induction techniques to handle dynamic process creation and unbounded data structures. Its graphical user interface provides comfortable access to proof resources. Several case studies such as a billing agent [3] and a distributed database lookup manager [2] have demonstrated the usefulness of the tool.

## 2 Foundations

### 2.1 The Erlang Programming Language

Erlang [1] is a concurrent functional programming language which allows to implement dynamic networks of processes operating on data types such as integers, lists, tuples, or process identifiers (pids), using asynchronous, call–by–value communication via unbounded ordered message queues called mailboxes.

The following code fragment specifies a simple concurrent server which repeatedly accepts an incoming query in form of a triple which is tagged by the

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

`request` constant, and which contains the request itself (matched by the variable `Request`) and the pid of the client process (`Client`). It then spawns off a process to serve the request (by using the `handle` function which is not considered here), and sends the result back to the client as a tuple tagged by the `response` constant.

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

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

The starting point of any kind of rigorous verification is a formal semantics. Here we use an operational semantics (a variant of the semantics presented in [3]) by associating a transition system with an Erlang program, giving a precise account of its possible computations. The states are parallel products of processes, each of the form $\langle e, p, q\rangle$, where $e$ is an Erlang expression to be evaluated in the context of a unique pid $p$ and a mailbox $q$ for incoming messages. A set of rules is provided to derive labelled transitions between such states.

This semantics is embedded in the proof system by allowing transition assertions to be used as atomic propositions, and by considering the modalities as derived formulae that are defined in terms of the transition relation (analogously to the treatment of CCS in [5,4]).

### 2.2 The Property Specification Logic

The logic that we use to capture the desired behaviour of Erlang programs and their components combines both temporal and functional features. It can be characterized as a many–sorted first–order predicate logic, extended with temporal modalities, least and greatest fixed–point operators, and some Erlang–specific atomic predicates. Due to its generality, it can be used to express a wide range of important system properties, ranging from more static, type–like assertions to complex safety and liveness features of the interaction between processes. An example for the latter is the following property of our concurrent server. It expresses that the process stabilizes on internal ('$\tau$') and output ('!') actions, that is, only a finite number of these can occur consecutively:

$$\texttt{stabilizes} = \mu Z.[\tau, !]Z.$$

### 2.3 The Proof System

At the very heart of our approach is a tableau–based proof system [3,4] embodying the basic proof rules by which complex correctness assertions can be reduced to (hopefully) less complex ones. It operates on Gentzen–style sequents of the form $\Gamma \vdash \Delta$ where $\Gamma$ and $\Delta$ are sets of assertions representing the premises and the conclusions, respectively, of the proof. For example, the following sequent expresses that the server process has the `stabilizes` property provided that the same holds, given any request $r$, for the `handle` function:

$$\forall q'.\forall r.\langle \mathtt{handle}(r), p, q'\rangle : \mathtt{stabilizes} \vdash \forall q.\langle \mathtt{server}(), p, q\rangle : \mathtt{stabilizes}$$

Summarizing, our proof system can be characterized by the following attributes:

– *laziness*: only those parts of the state space and of the formula are taken into account which are needed to establish the desired property, and so–called metavariables are used to postpone the choice of witnesses in a proof rule.
– *parametricity*: by representing parts of the system by placeholder variables (parameters), a relativised reasoning style for open systems is supported.
– *compositionality*: using a 'cut' rule, the end–system requirements can be decomposed in a modular and iterative way into properties to hold of the component processes.
– *induction*: to support reasoning about unbounded (or even non–well–founded) structures, inductive and co–inductive reasoning is provided based on the explicit approximation of fixed points and on well–founded ordinal induction. Inductive assertions are discovered during the proof rather than being enforced right from its beginning.
– *reuse*: by sharing subproofs and using lemmata, already established properties can be reused in different settings.

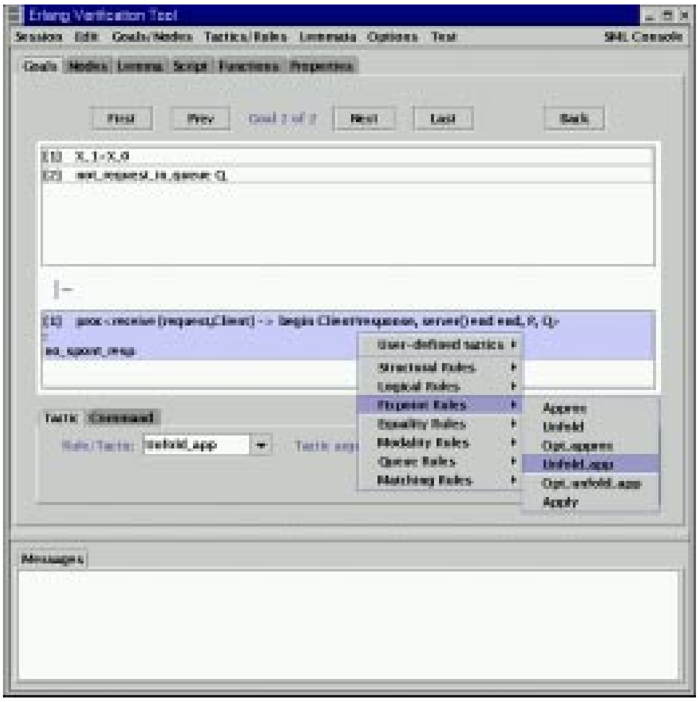
## 3 The EVT Tool

The verification task in our setting amounts to the goal–directed construction of a proof tree. Starting from the root with a goal sequent like the stabilization property of the server, the proof tree is built up by successively applying tactics until every leaf is labeled by an axiom. These tactics can attempt to construct complete proofs, or they can return partial proofs, stopping once in a while to prompt the user for guidance. The atomic tactics are given by the rules of the proof system, and more complex ones can be built up using special combinators called tacticals to obtain more powerful proof steps and, thus, to automate proofs, and to raise the abstraction level of reasoning.

Besides the actual construction of the proof tree, EVT provides support for proof reuse, navigation, and visualization. Due to the global character of the discharge rule which checks whether the current node of the proof tree is labeled by an instance of a previous proof sequent, the whole proof tree has to be maintained. EVT can thus be understood as a proof tree editor, extended with facilities for semi–automatic proof search.

A screen snapshot of EVT's graphical user interface is given in Fig. 1. It provides an easy and context–sensitive access to the proof resources like proof trees and their nodes, tactics, lemmata, and others. In the concrete situation, an example goal sequent is shown (with the premises and the conclusions above and below, respectively, the turnstile), and the user is just about to select a tactic which should be applied to the highlighted assertion.

A binary version of EVT can be downloaded from



[1] X_1=X_0
[2] not_request_in_queue Q

|-

[1] proc : receive {request,Client} -> begin Client!response, serve() end end, P, Q :
no_spont_msg

Fig. 1. The graphical user interface

```
ftp://ftp.sics.se/pub/fdt/evt/index.html
```

It is available for Intel x86 and Sparc architectures running under Linux and Solaris, respectively.

## References

1. J.L. Armstrong, S.R. Virding, M.C. Williams, and C. Wikström. *Concurrent Programming in Erlang*. Prentice Hall International, 2nd edition, 1996.
2. T. Arts and M. Dam. Verifying a distributed database lookup manager written in Erlang. In *Proceedings of the World Congress on Formal Methods in the Development of Computing Systems (FM'99)*, volume 1708 of *Lecture Notes in Computer Science*, pages 682–700. Springer–Verlag, 1999.
3. M. Dam, L.-å. Fredlund, and D. Gurov. Toward parametric verification of open distributed systems. In *Compositionality: the Significant Difference*, volume 1536 of *Lecture Notes in Computer Science*, pages 150–185. Springer–Verlag, 1998.
4. M. Dam and D. Gurov. Compositional verification of CCS processes. In *Perspectives of System Informatics: Proceedings of PSI'99*, volume 1755 of *Lecture Notes in Computer Science*, pages 247–256. Springer–Verlag, 1999.
5. A. Simpson. Compositionality via cut–elimination: Hennessy–Milner logic for an arbitrary GSOS. In *Proc. LICS*, pages 420–430. IEEE Computer Society Press, 26–29 1995.