System Description: Verification of Distributed Erlang Programs

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

[1] Computer Science Laboratory, Ericsson Telecom AB, 126 25 Stockholm, Sweden,
E-mail: thomas@cslab.ericsson.se
[2] Swedish Institute of Computer Science, Box 1263, S-164 28 Kista, Sweden,
E-mail: {mfd,fred,dilian}@sics.se

1 Introduction

Software written for telecommunication applications has to meet high quality demands. *Correctness* is one major concern; the activity of proving formally that a system is correct is called *verification*. Telecommunications software is highly concurrent, and *testing* is often not capable of guaranteeing correctness to a satisfactory degree. The software we are faced with consists of many, relatively small modules, written in the functional language Erlang [AVWW96]. These modules define the behaviour of a number of processes operating in parallel and communicating through asynchronous message-passing. New processes can be generated during execution. Because of the complexity of such software, our approach to verification is to prove that the software satisfies a set of properties called *specification* and formalized in a suitable logic language. The specification language we use is based on Park's μ-calculus [Par76,Koz83], extended with Erlang-specific features. This is a very powerful logic, due to the presence of least and greatest fixed point recursion, allowing the formalization of a wide range of behavioural properties. Verification in this context is not decidable, but can be automated to a large extent, requiring human intervention in a few, but crucial points.

To facilitate verification of Erlang programs of realistic size we are developing a verification tool implementing a tableau-based proof system described in [DFG98]. Our main objectives are to achieve a satisfactory degree of automation, proof reuse, easy navigation through proof tableaux, and meaningful feedback about the current proof state, so as to require user intervention only when this is really necessary, and to assist in taking informed proof decisions.

2 The Erlang Programming Language

We consider a core fragment of the Erlang programming language with dynamic networks of processes operating on data types such as numbers, lists, tuples, or process identifiers (pid's), using asynchronous, first-order call-by-value communication via unbounded ordered message queues called mailboxes. Real Erlang has several additional features such as communication guards, exception handling, modules, and a host of built-in functions. The abstract syntax of Core Erlang expressions is summarised as follows:

$$
\begin{array}{ll}
\text{Var} & := \texttt{X} \mid \texttt{Y} \mid \texttt{Z} \mid \ldots \\
\text{Atom} & := \texttt{a} \mid \texttt{b} \mid \texttt{c} \mid \ldots \\
\text{Pid} & := < \texttt{1} > \mid < \texttt{2} > \mid < \texttt{3} > \mid \ldots \\
\text{Pattern} & := \text{Var} \mid \text{Atom} \mid \text{Pid} \mid \{\text{Pattern},\ \text{Pattern}\} \\
\text{Match} & := \text{Pattern} \texttt{->} \text{Expr} \mid \text{Match};\ \text{Pattern} \texttt{->} \text{Expr} \\
\text{Expr} & := \text{Var} \mid \text{Atom} \mid \text{Pid} \mid \{\text{Expr},\ \text{Expr}\} \mid [\text{Expr} \mid \text{Expr}] \mid \\
 & \quad\ \texttt{case}\ \text{Expr}\ \texttt{of}\ \text{Match}\ \texttt{end} \mid \text{Expr},\ \text{Expr} \mid \\
 & \quad\ \text{Atom}(\text{Expr}) \mid \texttt{spawn}(\text{Atom},\ \text{Expr}) \mid \\
 & \quad\ \texttt{self} \mid \text{Expr}\,!\,\text{Expr} \mid \texttt{receive}\ \text{Match}\ \texttt{end}
\end{array}
$$

Core Erlang expressions are built from variables, atoms (like integers and operations on integers), process identifiers (pid's), and patterns by forming tuples (pairs), lists, case expressions, sequential composition, function application, generating new processes (`spawn`), obtaining the identity of the current process (`self`), as well as constructs for input and output to a specified recipient.

3 The Specification Language

The property specification logic we use can be summarised as a first-order predicate logic, extended with labelled "box" and "diamond" modalities, least and greatest fixed point recursion, and some Erlang-specific atomic predicates. 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. For example, the formula $\mu X.(n = 0 \vee \exists n'.(X(n') \wedge n = n'+1))$ defines the type of natural numbers, i.e. the least predicate which is true at zero and is closed under successor. As another example, $\nu X.(\forall x.[p?x]\ (\exists y. < q!y > X))$ expresses the capability of a system to react to messages received by process p by sending replies to process q. Far more complicated properties can be expressed by alternating least and greatest fixed points, such as fairness and response properties.

4 The Proof System

A large number of algorithms, tableau systems and proof systems for verifying processes against modal μ-calculus specifications can be found in literature, e.g. [EL86,SW91,Gur98] to cite but a few. However, most of these approaches are only applicable for finite-state processes, or at least processes where properties depend only on a finite portion of a potentially infinite-state process. The complexity of the software we consider and the properties we want to verify demand a new approach.

We build upon work begun by Dam in [Dam95], where instead of closed correctness assertions of the shape $S : \phi$ (where S is a system and ϕ a specification), *open correctness assertions* of the shape $\Gamma \vdash S : \phi$, where Γ expresses a set of assumptions $s : \psi$ on components s of S, are considered. Thus, the behaviour of S is specified parametrically upon the behaviour of its components.

This idea of open correctness assertions gave rise to the development of a Gentzen-style proof system [DFG98] that serves a the basis for the implementation of the verification tool. On top of a fairly standard proof system we added two rules: the first a "cut" rule for decomposing proofs of a system with multiple processes 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 safely unfolded. The discharge rule thus takes into account the history of assertions in the proof tree. In terms of the implementation this unfortunately requires the preservation of the proof tree during proof construction. Combined, the 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.

5 The Erlang Verification Tool

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 tree (tableau). The user is provided with commands for defining the initial node of the proof tree, for expanding a proof tree node ('the current proof node can be considered proved if the following nodes are proved instead'), for navigating through the proof tree, for checking whether the discharge rule is applicable, and for visualizing the current state of the proof tree using the daVinci graph manipulation tool [FW94]. Since the whole proof tree is maintained, proof reuse and sharing is greatly facilitated. The verification tool provides also a scripting language which can be used for automating several proof tasks, such as model-checking of simple formulas.

As an example, consider a resource managing process *rm*, which accepts requests *req* from users *u* for using resources. The resource manager reacts to each such request by generating a new resource handling process *rh*, the only task of which is to serve this special request by sending a reply *rep* to the corresponding user. Naturally, such a system should not send spontaneous replies without having received initiating requests. To keep the example simple, we shall formalise an approximation of this property, namely that the system can never engage in an infinite sequence of output actions. This property (let us denote it with ϕ) can be expressed as $\nu X.\mu Y.(\forall u.\forall req.[u?req]X \wedge \forall u.\forall rep.[u!rep]Y)$. Our initial proof obligation is "$\vdash rm : \phi$". By applying a command which attempts to model-check process *rm* until some new process is generated, we automatically obtain a new proof obligation of the shape "$\vdash rm \parallel rh : \phi$", namely that the system after generating one request handler also has the same property. So, some form of induction on the global process structure is necessary here. This is easily achieved by applying (manually) the cut rule, reducing the previous obligation to "$s : \phi \vdash s \parallel rh : \phi$" (denote this proof obligation by $(*)$), namely to proving that *any* process s satisfying ϕ, when put in parallel with process rh, also satisfies ϕ. In fact, this is the only point at which human intervention is required. By invoking the same command, the tool explores the possible actions of s and rh, and ultimately completes the proof. If $s \parallel rh$ performs an input action, this can only be because of s, and if s evolves thereby to s', then the resulting proof obligation becomes "$s' : \phi \vdash s' \parallel rh : \phi$" which is automatically discharged against $(*)$. Similarly, if $s \parallel rh$ performs an output action, this can only be because of rh, and since after this action rh ceases to exist, the resulting proof obligation becomes "$s : \phi \vdash s : \phi$" which is an instance of the usual identity axiom of Gentzen-style proof systems.

At the present point in time a first prototype tool has been completed with the functionality described above. A number of smallish examples have been completed, including, as the largest, a correctness proof concerning trust of a mobile billing agent reported in [DFG98]. Further information on the project and the prototype implementation can be found at `http://www.sics.se/fdt/erlang/`. We expect to announce a public release of the system by the end of 1998. Future work includes bigger case studies, increased support for proof automation, and better handling of fairness.

References

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

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

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

[EL86] E.A. Emerson and C. Lei, Efficient model checking in fragments of the propositional mu-calculus. In *Proceedings LICS'86*, p. 267–278, 1986.

[Gur98] D. Gurov, Specification and Verification of Communicating Systems with Value Passing. Ph.D. Thesis, Department of Computer Science, University of Victoria, March 1998.

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

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

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

[SW91] C. Stirling and D. Walker, Local model checking in the modal mu-calculus. *Theoretical Computer Science*, 89(1):161–177, 1991.