Semi-Automated Verification of Erlang Code

Lars-Åke Fredlund Dilian Gurov
Swedish Institute of Computer Science
fred@sics.se dilian@sics.se

Thomas Noll
Aachen University of Technology
noll@cs.rwth-aachen.de

Abstract

Erlang is a functional programming language with support for concurrency and message passing communication that is used at Ericsson for developing telecommunication applications. We consider the challenge of verifying temporal properties of systems programmed in Erlang with dynamically evolving process structures. To accomplish this a rich verification framework for goal-directed, proof system–based verification is used. This paper investigates the problem of semi–automating the verification task by identifying the proof parameters crucial for successful proof search.

I. Introduction

The *Erlang* programming language [1] is used at Ericsson for programming telecommunication applications. Such software is usually of a highly *concurrent* and *dynamic* nature, and is therefore hard to debug and test. We explore the alternative of proof system-based Erlang *code verification*. Verifying temporal properties of systems with dynamically evolving process structures and unbounded data is hard, requiring a framework [2], [3] which

- is *parametric* on components and *relativised* on their properties, i.e., does not necessarily require all parts of the Erlang system in question to be fully specified;
- is *compositional*, i.e., allows to reduce a property of a compound Erlang program to arguments about the properties of its components; and
- provides support for *inductive* and *co-inductive* reasoning about the infinitary behaviour of components.

Due to the concurrency and dynamism inherent in the systems addressed, a variety of induction schemes are required. However, it is often difficult to foresee which of these might work. We therefore employ *symbolic program execution* and *instance checking* to "discover" induction schemes lazily. Our machinery is based on ordinal approximation of fixed points and on well-founded ordinal induction, and on a global discharge proof rule for ensuring consistency of the mutual inductions in a proof structure.

A. The Erlang Programming Language

We consider a core fragment of the Erlang programming language with dynamic networks of processes operating on data types using asynchronous, call–by–value communication. Besides Erlang *expressions* e the syntactical categories of *matches* m, *patterns* p, and *guards* g are considered:

e	::=	*var*	
	|	*bv* | `[`e_1`|`e_2`]` | `{`e_1`, ..., `e_n`}`	
	|	e`(`e_1`, ..., `e_n`)`	*function call*
	|	`begin` e_1`, ..., `e_n `end`	*sequence*
	|	`case` e `of` m `end`	*matching*
	|	`exiting` e	*throw exception*
	|	`catch` e	*handle exception*
	|	`receive` m `end`	*process input*
	|	e_1`!`e_2	*process output*
bv	::=	*atom* | *number* | *pid* | `[]` | `{}`	
v	::=	*bv* | `[`v_1`|`v_2`]` | `{`v_1`, ..., `v_n`}`	
p	::=	*bv* | *var* | `[`p_1`|`p_2`]` | `{`p_1`, ..., `p_n`}`	
m	::=	p_1 `when` g_1 `->` e_1`;` ⋯ `;`p_n `when` g_n `->` e_n	
g	::=	e_1`, ..., `e_n	

The Erlang values consists of a set of atom literals (with an initial lowercase letter), the numbers, pid constants ranged over by *pid*, tuples, and lists. The variables (ranged over by *var*) are symbols starting with an uppercase letter. An Erlang *process*, here written `proc<`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 binary, with one process sending a message to a second process identified by its pid. Messages sent to a process are put in its mailbox q, queued in arriving order. Non-lossy communication channels of an unbounded size are assumed. The empty queue is `eps`, `[[`v`]]` is the queue containing the one element v, and $q_1@q_2$ concatenates 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 main choice construct of Erlang is by matching:

`case` e `of` p_1 `when` g_1 `->` e_1`; ... ;` p_n `when` g_n `->` e_n `end`

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. The expression $e_1!e_2$ represents sending (the value of e_2 is sent to the process with process identifier e_1) whereas `receive` m `end` inspects the process mailbox q and retrieves the first element v in q that matches any pattern in m. Then evaluation proceeds analogously to `case` v `of` m. Expressions are interpreted relative to an environment of "user defined" function definitions of the shape:

$f(p_{11},\ldots)$ `when` g_1 `->` e_1`;` $\ldots$`;` $f(p_{n1},\ldots)$ `when` g_n `->` e_n.

The operational semantics for Erlang developed in [4] forms the basis for program verification.

B. The Property Specification Language

Behavioural properties of Erlang programs, and the structure of program data, are characterised in a many-sorted first-order logic with explicit fixed point operators. To reason about behaviour the modalities $\langle\alpha\rangle\,\phi$ and $[\alpha]\,\phi$ are available. The addition of least and greatest fixed point operators results in a powerful specification language, known as the *μ–calculus* [5]. In the following we let α range over a set of program actions, t range over general terms, T over sort names and X ranges over the term and fixed point variables. The abstract syntax of logic formulae ϕ is:

$$\begin{array}{lll}
\phi ::= & t_1\texttt{=}t_2 \mid \texttt{tt} \mid \texttt{ff} \mid \texttt{not}\ \phi \mid \phi_1\ \texttt{and}\ \phi_2 \mid \phi_1\ \texttt{or}\ \phi_2 & \\
\mid & \texttt{exists}\ X\texttt{:}T\texttt{.}\phi \mid \texttt{forall}\ X\texttt{:}T\texttt{.}\phi & \textit{quantifiers} \\
\mid & \texttt{\textbackslash}X\texttt{:}T\texttt{.}\phi \mid \phi\ t & \textit{abstraction/application} \\
\mid & \texttt{<}\alpha\texttt{>}\phi \mid \texttt{[}\alpha\texttt{]}\phi & \textit{modalities} \\
\mid & \texttt{gfp}\ X\texttt{.}\phi \mid \texttt{lfp}\ X\texttt{.}\phi \mid X & \textit{fixed points} \\
\mid & \kappa < \kappa' & \textit{ordinal inequations} \\
\mid & t_1 \xrightarrow{\alpha} t_2 & \textit{transition assertions}
\end{array}$$

The syntactic form $t\texttt{:}\phi$ is an alternative for an application $\phi\ \ t$. Fixed point formulas can be named, e.g., `name` $\Leftarrow \phi$ abbreviates the least fixed point `lfp` $X\texttt{.}\phi\{\texttt{X/name}\}$ and `name` $\Rightarrow \phi$ abbreviates a greatest fixed point.

C. The Proof System

Program verification uses a Gentzen–style proof system, allowing free parameters to occur within the *proof judgments* of the proof system. The judgments are of the form Γ `|-` Δ, where Γ and Δ are sequences of assertions. A judgment is *valid* if, for any interpretation of the free variables, some assertion in Δ is valid whenever all assertions in Γ are valid. Parameters are variables ranging over specific types of entities, such as messages, functions, or processes. The proof rules of the proof system are standard from first-order logic, with the addition of rules for fixed point manipulation, a cut–like rule for decomposing proofs about a compound system to proofs about the components, and a rule for discharging loops in a proof, via fixed point induction.

The fixed point rules govern the unfolding of fixed points, and the annotation of fixed points with ordinal variables to represent the number of such unfoldings. These ordinal variables are examined by the global discharge rule to determine whether a proof structure contains a proper inductive or co-inductive argument. Consider two example rules

$$\mathsf{Apprx}_R\quad \frac{\Gamma\ \texttt{|-}\ ((\texttt{gfp}\ X\texttt{.}\phi)^{\kappa})\ t_1\ \ldots\ t_n, \Delta}{\Gamma\ \texttt{|-}\ (\texttt{gfp}\ X\texttt{.}\phi)\ t_1\ \ldots\ t_n, \Delta}$$

$$\mathsf{Unf1}_R\quad \frac{\Gamma, \kappa' < \kappa\ \texttt{|-}\ (\phi\{(\texttt{gfp}\ X\texttt{.}\phi)^{\kappa'}/X\})\ t_1\ \ldots\ t_n, \Delta}{\Gamma\ \texttt{|-}\ ((\texttt{gfp}\ X\texttt{.}\phi)^{\kappa})\ t_1\ \ldots\ t_n, \Delta}$$

The rule Apprx_R commences a co-induction (on the unfolding of the fixed point) and introduces a fresh ordinal variable κ. The rule $\mathsf{Unf1}_R$ unfolds the fixed point and records the existence of a lesser ordinal as the inequation $\kappa' < \kappa$. As a side-effect the term vector $t_1 \ldots t_n$ is recorded and used in proof search to heuristically determine whether unfolding is a progressing proof step.

In compositional verification an argument about the behaviour of a compound system is reduced to arguments about the behaviour of its components, which is achieved through a *term–cut* proof rule of the shape

$$\mathsf{TermCut}\frac{\Gamma\ \texttt{|-}\ p\texttt{:}\psi, \Delta \qquad \Gamma, X\texttt{:}\psi\ \texttt{|-}\ s\texttt{:}\phi, \Delta}{\Gamma\ \texttt{|-}\ s\{p/X\}\texttt{:}\phi, \Delta}$$

The global discharge rule is the crucial proof rule on which inductive and co-inductive reasoning relies. Roughly, the goal is to identify situations where a latter proof node can be discharged since is an instance of an earlier one on the same proof branch, and since appropriate fixed points have been unfolded [2].

D. The Erlang Verification Tool

The proof system is realised in the Erlang Verification Tool (EVT) proof assistant [6].[1] EVT has been tailored to the underlying proof system; rather than working with a set of open goals, the underlying data structure is an acyclic proof graph to account for the checking of the discharge rule. Proving a property of an Erlang program involves goal-directed construction of a proof graph. The basic proof rules are implemented as *tactics*, which are functions from a sequent (the current goal, forming the conclusion of the rule) to a list of sequents (the subgoals, given by the premises of the rule). As most proof assistants, EVT provides *tactic combinators* or *tacticals*, for deriving new tactics. A number of higher-level tactics provide practical proof rules for deriving transitions of Erlang components.

II. Proof Organization and Automation

The general verification problem of proving that an Erlang system satisfies a μ-calculus property is not decidable. Therefore, it is crucial to identify the proof tasks that can be automated, and to organize proofs in a manner which combines in the most suitable way the automatable activities with the human-guided ones.

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

A. Proofs and Proof Discovery

In EVT a proof is a tree with some leaves being axiom instances, and the rest being instances of predecessor sequents and satisfying the global discharge condition. In practice, searching for such proofs is computationally too expensive, and moreover the search is not likely to terminate. Instead, we consider here a more relaxed notion of a proof, which is, intuitively, a proof tree that exhibits the essential structure of a complete proof, but where not all proof-branches necessarily are completed or even valid. As we have found in practice, such a "pre-proof" forms a good starting point for obtaining a successful proof, and is relatively cheap to search for; in particular, proof-search can terminate.

Consider the usual shape of a proof goal about Erlang programs $\Gamma \mid\!\!- s : \phi$ where s is an Erlang behavioural component (e.g., process, system, expression), ϕ is the behavioural property the component should satisfy, Γ are assumptions about program parameters. The proof structure representing the proof of such a sequent is governed mainly by two parameters: (i) the behavioural patterns of the Erlang component s (e.g., for a system its communication and network topology, for a functional expression its call graph), and (ii) the fixed point structure of the formula ϕ. Thus the following proof parameters crucial for successful semi-automatic (pre-)proof search can be identified:

1. *Setting up the main (co-)induction structure*: deciding when to approximate and unfold fixed points.
2. *Combatting state-explosion in the proof structure*: deciding where to apply the termcut rule, either as a mechanism to abstract away from a concrete program term to reduce the proof-state space, or to continue an inductive argument.
3. *Terminating (pre-)proof search:* here one has to balance between how often to invoke human intervention and the need to avoid non-terminating or large redundant computations. A good heuristic is to terminate proof search when "growing" program components are detected (and no termcut policy is in place), notably after process spawning, which cause the instance checking to fail and can thus give rise to non-terminating proof branches. Function calls are yet another place to stop proof search, usually to allow for better structuring and reuse of proofs, but also indispensable in the analysis of non-tail-recursive Erlang functions.
Proof search should also be terminated whenever a leaf is encountered which is either a "pre-axiom" (for example suspected to be propositionally valid), or it is a "pre-instance" of some predecessor sequent (for example the main assertion in the sequent is an instance of the corresponding assertion in the predecessor). The second case represents a strong indication that an (co-)inductive argument should be performed, and thus indicates how to transform the pre-proof to a proper proof.
4. *Choosing locally the next proof rule to be applied:* under this item any non-strategic proof rule application falls such as reasoning about the transitions of an Erlang component using the operational semantics.
5. *Maintaining proof invariants:* in an Erlang proof sequent assumptions record facts about unknown program parameters, or relationships between program variables, in the form of program invariants. During an automated proof search such assumptions need to be updated, after a symbolic program step has been taken.

Once a pre-proof has been found, the task of converting it into a proper proof remains. In this paper it is left to the user, who should modify the parameters of the proof search (the proof schema) by, for instance, adding additional inductions (1), or by adding and maintaining proof invariants (5), and then repeat the search for a pre-proof.

B. Proof Search Facilities

We describe some of the tactics and scripts supporting the approach to proof search outlined above. Their use is illustrated in the next subsection. Given an index i, tactic (`t_choiceless_r` i) is used for local proof search. It begins with the i-th formula to the right, and recursively applies the tactic corresponding to the outermost connective of the formula as long as no choice and no fixed-point unfolding or approximation is involved. The `t_gen_unfold_r` tactic combines one unfolding with `t_choiceless_r`.

Sequent predicates are functions from sequents to booleans. These can be combined using the functors `sp_not`, `sp_or` and `sp_and`. An important use of sequent predicates is to capture proof-search termination conditions. For example, (`sp_unfoldable_r` i) checks whether the term appearing as the first component of the satisfaction pair at position i is not an instance of some term at which the fixed-point formula, which is the second component, has already been unfolded. This is a much weaker condition than the instance condition of the discharge rule, and is very useful in practice. The approach is inspired by the fixed-point *tagging* technique of Winskel [7].

The `case_by` script takes as argument a list of pairs consisting of a sequent predicate and a tactic. It executes the tactic corresponding to the first predicate (if any) which holds for the current sequent. The `loop` script takes as an argument a script such as `case_by` and applies it recursively until no new nodes are generated. As an example for invariant maintenance, the (`t_queue_invar` i_l i_r) tactic transfers the queue assumption residing at the left index i_l to the queue term of the process at the right index i_r.

C. Example

We shall illustrate the ideas presented above on a simple but typical example. Consider a 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,ClPid} ->
      begin
        spawn(serve,[Request,ClPid]),
        central_server()
      end
  end.

serve(Request,ClPid) -> ClPid!{response,ok}.
```

C.1 Stabilization

The first formula we consider gives a liveness property of the server, namely *stabilization*, i.e. the convergence on output and silent (`estep`) actions. It 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: erlang_system -> prop <=
   (forall Pid:erlangPid. forall V:erlangValue.
    [Pid!message(V)]stabilizes)
/\ ([estep] stabilizes);
```

So, the initial proof goal is declared as:

```
declare P:erlangPid, Q:erlangQueue in
  |- proc<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)
|- proc<begin P1, central_server() end, P, Q1> ||
   proc<serve(Req, ClPid), P1, eps> : stabilizes
```

The queue `Q2@[[{request,Req,ClPid}]]@Q3` is built from the concatenation of three parts: `Q2`, 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 termcut rule to obtain the two new goals:

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

X : stabilizes |-
X || proc<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 analysed by the script presented above, terminating with the goal

```
|- proc<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. Therefore we add 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
|- proc<central_server(), P, Q> : stabilizes
```

given the definition

```
queue: erlangQueue -> prop <=
\Q:erlangQueue .
  Q = eps \/
  (exists V:erlangValue, Q1,Q2:erlangQueue .
   Q = Q1@[[V]]@Q2 /\ (is_queue Q1@Q2))
```

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
|- proc<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.

C.2 Absence of Exceptions

The second property we consider is a safety property, namely, that calls to the central_server function do not cause runtime exceptions, terminating the execution of the process in whose context the call is executed (unless the exception is explicitly handled). Exceptions are caused by e.g., typing errors discovered at runtime, invocation of undefined functions, etc. The property can be specified as

```
no_exceptions : erlangExpression -> prop =>
  forall A:erlangIntAction .
  [A](not(exists V:erlangValue . A=exiting(V)) /\
      no_exceptions);
```

where exiting(V) represents a runtime exception action. The goal to prove is:

```
|- central_server() : no_exceptions
```

The main proof structure (1) will be a co-induction on the no_exceptions property (a greatest fixed point). Thus, first the no_exceptions property is approximated with an ordinal variable K. The reason for the state explosion (proof parameter 2) in this example are non-tail recursive function calls, in particular the call to spawn. Here we simply cut all function calls using the current approximation of no_exceptions, which is always a good first approximation. That is, the goal

```
K1<K,K2<K1 |-
begin
 spawn(serve,[Request,ClientPid]),central_server()
end : no_exceptions(K2)
```

is reduced by an automated tactic (applying termcut) to

```
K1<K,K2<K1 |-   spawn(serve, [Request, ClientPid])
              : no_exceptions(K2)

K1<K,K2<K1 |- central_server() : no_exceptions(K2)

K1<K,K2<K1,
X1 : no_exceptions(K2), X2 : no_exceptions(K2)
|-  begin X1, X2 end : no_exceptions(K2)
```

Pre-proof search (3) is terminated when a pre-instance is found, i.e., an instance of the current expression has already been considered. In this case the discharge rule is applied. For local reasoning (4) we apply a simple tactic similar to t_choiceless_r to reduce the proof state. The proof state invariants to maintain (5) are the result of applications of termcut. For instance, when reducing the third goal the assumptions

```
X1 : no_exceptions(K2), X2 : no_exceptions(K2)
```

act as invariants that have to be maintained in order to complete the proof. With this machinery in place the resulting, automatically obtained, proof tree has 12 nodes, of which 3 are discharged with respect to ancestor proof node instances. Moreover the proof is linear in the size of the program (the functions) – when one employs a clever representation of the ordinal inequations. To scale up this example, a more involved cut-formula is needed, to take into account the return values of function applications.

III. Conclusion

We have demonstrated an approach to semi-automated verification of program code – for a language used in critical industrial applications – which combines proof discovery (finding induction schemes, perhaps partly manually) with proof automation. The setting is general and rich, admitting the use of the same machinery for addressing both program and data behaviours. Previous experiences [8], [2] indicate that proof graphs of a size up to 10^5 nodes can be handled. In our experience, larger programs do usually not lead to more difficult proof structures, but rather just to additional proof obligations.

References

[1] J. Armstrong, R. Virding, C. Wikström, and M. Williams, *Concurrent Programming in Erlang (Second Edition)*, Prentice-Hall, 1996.

[2] 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, vol. 1536, pp. 150–185, 1998.

[3] L.-å. Fredlund and D. Gurov, "A framework for formal reasoning about open distributed systems," In *Proc. ASIAN'99*, Lecture Notes in Computer Science, vol. 1742, pp. 87–100, 1999.

[4] L. Fredlund, *A Framework for Reasoning about Erlang Code*, Ph.D. thesis, Royal Institute of Technology, Stockholm, Sweden, 2001, To be defended on September 14, 2001.

[5] D. Kozen, "Results on the propositional μ-calculus," *Theoretical Computer Science*, vol. **27**, pp. 333–354, 1983.

[6] L. Fredlund, D. Gurov, and T. Noll, "The Erlang verification tool," 2001, vol. 2031, pp. 582–585.

[7] G. Winskel, "A note on model checking the modal ν-calculus," *Theoretical Computer Science*, vol. 83, pp. 157–187, 1991.

[8] 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, vol. 1708, pp. 682–700, 1999.