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 widely used at Ericsson for developing telecommunication applications. We consider the challenge of verifying temporal properties of Erlang programs which are used to implement systems with dynamically evolving process structures and unbounded data. This is a hard task, which requires a rich verification framework. Building upon such a framework for goal-directed, proof system–based verification, the paper investigates the problem of semi–automating this task by identifying the proof parameters crucial for successful proof search.

I. Introduction

The *Erlang* programming language [1] is widely 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*. 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 the endeavour.

Verifying recursive temporal properties of systems with dynamically evolving process structures and unbounded data is known to be hard. Handling realistic examples requires a rich verification framework [2], [3], [4] 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 recursively defined components.

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 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 present in a proof structure. Thus our approach is alternative to using abstract interpretation followed by model checking as proposed by Huch [5] (the conclusion gives a brief overview of related approaches).

An efficient implementation of proof search in such a framework requires access to the internal structures of a proof. We therefore opted for developing a special-purpose tool [6], [7] rather than building upon a general-purpose theorem proving tool like PVS [8]. We have focused on implementing the features which are new and specific for our approach; exploring the machine support for symbolic program execution, proof decomposition and induction scheme discovery, and less on "standard" tasks like proving recursive properties of finite components and equational properties of data which could be delegated to external tools.

The effort on the verification of Erlang programs is taking place within a collaborative project between the Swedish Institute of Computer Science[1] and the Computer Science Laboratory of Ericsson. Applications of the proof assistant tool to industrial code [9] have highlighted the need for reasoning about software components on an architecture level (this problem is investigated in in [10]), and for addressing within our framework the problem of providing support for semi-automatic proof search.

It is the latter point which we are going to study in this paper. Its remainder is organized as follows. Section II summarizes the verification framework: the Erlang program-

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

[1] Partly funded by the ASTEC (Advanced Software Technology) competence center

ming language and its formal semantics, the property specification language, and the proof system and its implementation. Section III addresses the problem of proof automation, both in a general setting and using concrete examples. Finally, Section IV draws some conclusions and presents related work.

II. Foundations

A. The Erlang Programming Language

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

In the paper we consider a core fragment of the Erlang programming language with dynamic networks of processes operating on data types such as integers, lists, tuples, or process identifiers (pid's), using asynchronous, call–by–value communication via unbounded ordered message queues called mailboxes.

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

$$
\begin{array}{rcll}
e & ::= & \textit{var} & \\
 & | & \textit{bv} \mid \texttt{[}e_1\texttt{|}e_2\texttt{]} \mid \{e_1\texttt{,}\ \ldots\texttt{,}e_n\} & \\
 & | & e\texttt{(}e_1\texttt{,}\ldots\texttt{,}e_n\texttt{)} & \textit{function call}\\
 & | & \texttt{begin}\ e_1\texttt{,}\ \ldots\ \texttt{,}e_n\ \texttt{end} & \textit{sequence}\\
 & | & \texttt{case}\ e\ \texttt{of}\ m\ \texttt{end} & \textit{matching}\\
 & | & \texttt{exiting}\ e & \textit{throw exception}\\
 & | & \texttt{catch}\ e & \textit{handle exception}\\
 & | & \texttt{receive}\ m\ \texttt{end} & \textit{process input}\\
 & | & e_1\texttt{!}e_2 & \textit{process output}\\
 & & & \\
\textit{bv} & ::= & \textit{atom} \mid \textit{number} \mid \textit{pid} \mid \texttt{[]} \mid \{\} & \\
v & ::= & \textit{bv} \mid \texttt{[}v_1\texttt{|}v_2\texttt{]} \mid \{v_1\texttt{,}\ \ldots\texttt{,}v_n\} & \\
 & & & \\
p & ::= & \textit{bv} \mid \textit{var} \mid \texttt{[}p_1\texttt{|}p_2\texttt{]} \mid \{p_1\texttt{,}\ \ldots\texttt{,}p_n\} & \\
m & ::= & p_1\ \texttt{when}\ g_1\ \texttt{->}e_1\texttt{;}\ \cdots\ \texttt{;}p_n\ \texttt{when}\ g_n\ \texttt{->}e_n & \\
g & ::= & e_1\texttt{,}\ldots\texttt{,}e_n &
\end{array}
$$

The Erlang values consists of a set of atom literals (with an initial lowercase letter), the numbers (here integers only), 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 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. As in the informal Erlang semantics, perfect (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 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 main choice construct of Erlang is by matching:

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

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

$$
\begin{array}{l}
f(p_{11},\ldots,p_{1k})\ \texttt{when}\ g_1\ \texttt{->}\ e_1\texttt{;}\\
\quad\vdots\\
f(p_{n1},\ldots,p_{nk})\ \texttt{when}\ g_n\ \texttt{->}\ e_n.
\end{array}
$$

To support compositional reasoning the operational semantics is organised hierarchically, in layers, using different sets of transition labels at each layer. Thus first the Erlang expressions are provided with a semantics that does not require any notion of processes but does represent the strict, eager evaluation strategy of Erlang, on top of which a process level semantics is built [4].

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 programs the usual modalities $\langle\alpha\rangle\,\phi$ and $[\alpha]\,\phi$ are available (as derived operators, referring to recursive predicates encoding the transitional semantics of Erlang). Adding least and greatest fixed point operators results in a powerful specification language, broadly known as the *μ–calculus* [11], [12].

In the following we let α range over a set of actions (labels in the operational semantics), t range over general terms, T over sort names (including the Erlang expressions, systems, and various classes of data), and X range over the term and fixed point variables. The abstract syntax of logic formulae ϕ is:

$\phi ::= t_1 = t_2$	*equality*
$\mid$ `tt` $\mid$ `ff`	*truth values*
$\mid$ `not` $\phi \mid \phi_1$ `and` $\phi_2 \mid \phi_1$ `or` ϕ_2	*connectives*
$\mid$ `exists` $X{:}T.\phi \mid$ `forall` $X{:}T.\phi$	*quantifiers*
$\mid \backslash X{:}T.\phi \mid \phi\ t$	*abstraction/application*
$\mid$ `<`α`>`$\phi \mid$ `[`α`]`ϕ	*modalities*
$\mid$ `gfp` $X.\phi \mid$ `lfp` $X.\phi \mid X$	*fixed points*
$\mid \kappa < \kappa'$	*ordinal inequations*
$\mid t_1 \xrightarrow{\alpha} t_2$	*transition assertions*

To reinforce the connection with the modal μ-calculus the syntactic form $t{:}\phi$ is accepted as an alternative for an application $\phi\ t$. In the following fixed point formulas are named, e.g., `name` $\Leftarrow \phi$ abbreviates the least fixed point `lfp` $X.\phi\{$`X/name`$\}$ and `name` $\Rightarrow \phi$ abbreviates a greatest fixed point.

The semantics of a formula in the logic is defined in the usual (denotational) fashion, as the set of (Erlang) terms that satisfy the formula (see [2] for details).

C. The Proof System

Reasoning about Erlang programs requires the ability to specify their observable behaviour relativised by assumptions about certain system parameters. Technically, this is achieved using a Gentzen–style proof system, allowing free parameters to occur within the *proof judgments* of the proof system. The judgments are of the form $\Gamma \mid\!\text{-}\ \Delta$, 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 variables ranging over specific types of entities, such as messages, functions, or processes. For example, the proof judgment $x : \psi \mid\!\text{-}\ P(x) : \phi$ states that P has property ϕ provided the parameter x of P satisfies property ψ.

The proof rules of the proof system are standard from accounts of 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 the proof structure, via fixed point induction. In combination, these additional proof rules permit 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 [2].

D. Fixed Point Manipulation

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 the proof structure contains a proper inductive or co-inductive argument. For example, the rules for manipulating a greatest fixed point on the right–hand side, occurring under applications, are (the ordinal variables κ and κ' are assumed fresh):

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

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

Intuitively the rule (Apprx_R) corresponds to commencing a co-induction (on the unfolding of the fixed point), and $(\mathsf{Unf1}_R)$ 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 [13] *tagging* technique). This is used in proof search to heuristically determine whether unfolding is a progressing proof step.

E. 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. This is achieved through a *term–cut* proof rule of the following shape (technically the rule is derived from the normal cut rule of Gentzen proof systems).

$$(\mathsf{TermCut})\ \frac{\Gamma \mid\!\text{-}\ p{:}\psi, \Delta \qquad \Gamma, X{:}\psi \mid\!\text{-}\ s{:}\phi, \Delta}{\Gamma \mid\!\text{-}\ s\{p/X\}{:}\phi, \Delta}$$

F. Checking Discharge Conditions

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. The discharge rule thus takes into account the history of assertions in the proof tree. A thorough investigation of the conditions regulating when such a discharge step is sound is given in [2], [14], here only a sketch is given.

Consider a proof node N_d, henceforth called the *discharge node*, representing an open proof goal of the form $\Gamma_d \mid\!\text{-}\ \Delta_d$. Assume that there exists an ancestor node N_c in the proof tree, henceforth called the *companion node*, labelled by a sequent $\Gamma_c \mid\!\text{-}\ \Delta_c$.

The discharge proof rule comprises checking three conditions, under which the node N_d may be discharged due to the presence of the ancestor node N_c:

• Is there a mapping from N_c to N_d? That is, does a substitution ρ (mapping parameters to terms) exist 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 $\Gamma_d \mid\!- \kappa\rho < \kappa$
• The previous two conditions are local, i.e., involve 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 [2], [14]). In essence this corresponds to checking whether the global-proof tree defines a proper simultaneous fixed point induction scheme.

Combined, the term–cut rule and the discharge rule permit to handle unbounded recursion (e.g. caused by process spawning) as is illustrated in Section III-C.

G. The Erlang Verification Tool

The proof system is realised in a proof assistant – the "Erlang verification tool" (EVT) [6], [7]. EVT is implemented in Standard ML and offers a command-line interface and a graphical user interface in Java.[2]

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 side conditions of the discharge rule. The main reason for developing a new proof assistant tool prototype is our desire to experiment with different implementation strategies for the rule of discharge and the underlying proof graph representation. 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 induction schemes through a lazy search procedure in the course of the proof.

Proving a property of an Erlang program involves "backward" (i.e., goal-directed) construction of a proof graph. The basic proof rules are implemented as *tactics*, which are, somewhat simplified, functions (in the Standard ML sense) 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 sound tactics from basic tactics. For instance, a number of derived tactics implement more practical rules for deriving transitions of Erlang components. An additional concept is that of *proof scripts*. These are "commands" which may destructively update the proof graph, e.g. discharging an open proof goal due to a proof loop.

[2]Further information about EVT is available at the location `http://www.sics.se/fdt/VeriCode/evt.html`

III. 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.

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 (see below for details). 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: $\Gamma \mid\!- s : \phi, \Delta$ 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, and Δ are alternative properties to prove. 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. Taking this into account, and with the experience of a substantial number of Erlang case studies, we have identified the following main proof parameters which are crucial for successful semi-automatic (pre-)proof search:
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 term–cut proof rule, either as an abstraction mechanism to abstract away from a concrete program term (to reduce the proof-state space), or to continue an inductive argument. In the example below, both usages of the term–cut mechanism are illustrated.
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 term–cut 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, for example, reasoning about the transitions of an Erlang component using the operational semantics.
5. *Maintaining proof invariants:* in a typical Erlang proof sequent assumptions record facts about unknown program parameters, or relationships between program variables. For example, classical program invariants are represented in this fashion. During an automated proof search such assumptions need to be updated, typically 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. This task is made easier the more detail about the above proof parameters is provided prior to the pre-proof search. 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.

In the following section we provide concrete tactics, tacticals, and proof scripts, that help automate the search for proofs of Erlang components. Then, in Section III-C, we explain the semi-automated proof of a concurrent Erlang process with process spawning, illustrating the typical steps in the proof search.

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 [13].

The `case_by` script takes as an argument a list of pairs with first entry a sequent predicate and second entry 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, ClientPid} ->
      begin
        spawn (serve, [Request, ClientPid]),
        central_server ()
      end
  end.

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

handle (Request) -> 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
```

An Erlang *process*, here written as `proc<central_server (), P, Q>`, is a container for the evaluation of an expression. A process has a unique process identifier `P` 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.

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 term-cut 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:

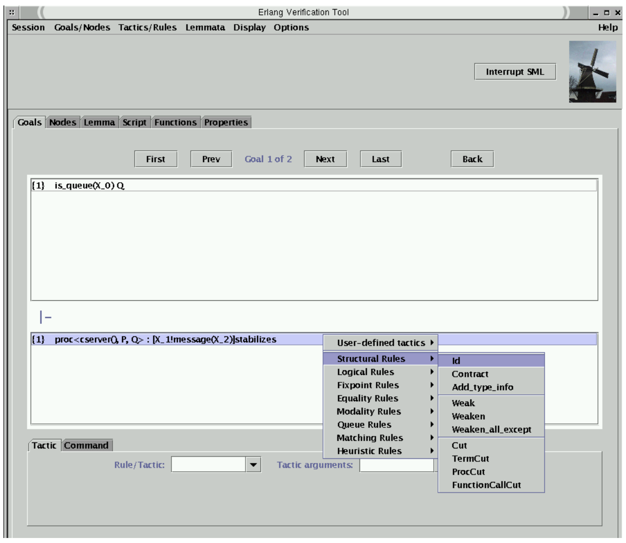
```
declare P:erlangPid, Q:erlangQueue in Q : queue
|- proc<central_server (), P, Q> : stabilizes

queue: erlangQueue -> prop <=
  \Q:erlangQueue .
       Q = eps \/
       (exists V:erlangValue .
        exists Q1:erlangQueue .
        exists 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



```
{1}   is_queue(X_0) Q
|-
{1}   proc<cserver(), P, Q> : [X_1!message(X_2)]stabilizes
```

Fig. 1. The Graphical User Interface of EVT
figure

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.

A screenshot of a proof session using the graphical user interface of EVT, to prove the stabilization property, is shown in Figure 1.

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 to `central_server` 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
```

For this example we will indicate the parameters (standard automated tactics) needed to produce a proof.

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 term–cut) resulting in goals (g1,g2,g3)

```
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 term-cut. For instance, when reducing the goal g3 above 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, as discovered using our semi-automated proof method, to take into account the return values of function applications.

IV. 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 [9], [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 (implying increased difficulties in finding induction schemes), but rather just to additional (tedious) proof obligations.

However, a real complication when addressing production code is that additional program structuring concepts, such as generic components, are utilised. Either program abstraction techniques will have to be adapted, or, as in [10], the new concepts have to be modelled directly.

A. Related Work

In this section we briefly review the support for automation that is offered by other verification frameworks than the one represented by EVT. Of course, due to lack of space, our exposition cannot be exhaustive in any way. We exclude theorem–proving systems designed for the formalization of classical or constructive mathematics, such as Coq, HOL, or Nuprl. Rather we concentrate on (meta–)logical frameworks which support (the specification of and) the formal reasoning using deductive systems tailored towards programming languages.

The first category of frameworks we consider can be characterized as being based solely on theorem–proving methods. Probably the most popular representative of this class 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.

Isabelle's principal proof method is *resolution*, involving higher–order unification. Proof automation is provided by *tactics*, which support backward reasoning by refining goals into subgoals, and *tacticals*, which combine tactics. Examples for the latter are `THEN`, which sequentially composes two tactics, `ORELSE`, which chooses between tactics, and `REPEAT`, which iteratively applies a tactic.

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

With regard to proof automation, theorem–proving systems which integrate model–checking methods deserve our special attention. Their motivation is to benefit from both the automation of model checking and the generality of theorem proving. Here (at least) two approaches can be distinguished. The first one embeds model checking as a decision procedure within a deductive framework. The latter is used to decompose a verification goal into model–checkable subgoals, and to apply inductive reasoning methods to "lift" the result to the whole structure.

[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/

In the *PVS* theorem prover[6], for example, a BDD–based model checker is used as a decision procedure for the μ–calculus restricted to finite types, and CTL model checking is implemented as a derived proof procedure on top of that. The PVS specification language 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 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.

The basic means for high–level reasoning in PVS are *proof strategies*. They are intended to combine primitive proof rules by capturing patterns of inference steps. A *defined proof rule* is a strategy that is applied in a single atomic step so that only the final effect is visible and the intermediate results are hidden from the user. Thus it represents the PVS counterpart of EVT's tactics.

Access to the μ–calculus model checker is provided by the `musimp` command. In combination with induction it can be used to, e.g., verify systems composed of networks of finite–state processes. A similar mechanism could be integrated in the EVT tool where the `term-cut` could be used to isolate a model–checkable property of a finite–state subsystem.

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.

In the second combined approach, theorem–proving methods are applied to formally justify abstractions which

[6]http://pvs.csl.sri.com/

map infinite state spaces to finite–state form, followed by model checking. The crucial point here is to find abstractions which, on the one hand, are powerful enough to yield model–checkable systems and, on the other hand, preserve the property to be verified. A corresponding approach to the verification of Erlang programs is presented in [5] where abstract interpretation is used to derive a finite–state abstraction such that all computation paths are maintained, which means that linear–time properties are preserved. Property–preserving abstractions for branching–time logics are thoroughly studied in [15].

A special kind of abstraction is also supported by the PVS system in form of the `abstract` command. It constructs a Boolean abstraction, where Boolean variables replace the concrete predicates occurring in the respective program. The combination with model checking is provided by the `abstract-and-mc` command (see also [16]).

References

[1] J.L. Armstrong, M.C. Williams, C. Wikström, and S.R. Virding, *Concurrent Programming in Erlang,* Prentice Hall, 2nd edition edition, 1995.

[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] M. Dam and D. Gurov, "Compositional verification of CCS processes," In *Proc. PSI'99,* Lecture Notes in Computer Science, vol. 1755, pp. 247–256, 2000.

[4] 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.

[5] F. Huch, "Verification of Erlang programs using abstract interpretation and model checking," *ACM SIGPLAN Notices*, vol. 34, no. 9, pp. 261–272, 1999, Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '99).

[6] 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, vol. 1421, pp. 38–41, 1998.

[7] T. Noll, L.-Å. Fredlund, and D. Gurov, "The Erlang verification tool," in *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01)*. 2001, vol. 2031 of *Lecture Notes in Computer Science*, pp. 582–585, Springer.

[8] 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, vol. 1102, pp. 411–414, 1996.

[9] 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.

[10] T. Arts and T. Noll, "Verifying generic Erlang client-server implementations," *Proc. IFL 2000*, vol. Aachener Informatik-Berichte, no. 00-7, pp. 387–402, 2000.

[11] D. Park, "Finiteness is mu-Ineffable," *Theoretical Computer Science*, vol. **3**, pp. 173–181, 1976.

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

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

[14] M. Dam and D. Gurov, "μ-calculus with explicit points and approximations," In: *Proc. FICS'2000*, 2000.

[15] C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem, "Property preserving abstractions for the verification of concurrent systems," *Formal Methods in System Design: An International Journal*, vol. 6, no. 1, pp. 11–44, Jan. 1995.

[16] H. Saïdi and N. Shankar, "Abstract and model check while you prove," in *Proc. 11th International Conference on Computer–Aided Verification (CAV'99)*, 1999, vol. 1633 of *Lecture Notes in Computer Science*, pp. 443–454.