On the Verification of Open Distributed Systems *

Mads Dam, Lars-åke Fredlund
Swedish Institute of Computer Science†

Keywords: Open Distributed Systems; Program Verification; Erlang; Parametric Verification; Agents

Abstract

A logic and proof system is introduced for specifying and proving properties of open distributed systems. Key problems that are addressed include the verification of process networks with a changing interconnection structure, and where new processes can be continuously spawned. To demonstrate the results in a realistic setting we consider a core fragment of the Erlang programming language. Roughly this amounts to a first-order actor language with data types, buffered asynchronous communication, and dynamic process spawning. Our aim is to verify quite general properties of programs in this fragment. The specification logic extends the first-order μ-calculus with Erlang-specific primitives. For verification we use an approach which combines local model checking with facilities for compositional verification. We give a specification and verification example based on a billing agent which controls and charges for user access to a given resource.

1 Introduction

A central feature of open distributed systems as opposed to concurrent systems in general is their reliance on modularity. Open distributed systems must accommodate addition of new components, modification of interconnection structure, and replacement of existing components without affecting overall system behaviour adversely. To this effect it is important that component interfaces are clearly defined, and that systems can be put together relying only on component behaviour along these interfaces. That is, behaviour specification, and hence verification, needs to be parametric on subcomponents. But almost all prevailing approaches to verification of concurrent and distributed systems rely on an assumption that process networks are static, or can safely be approximated as such, as this assumption opens up for the possibility of bounding the space of global system states. Clearly such assumptions square poorly with the dynamic and parametric nature of open distributed systems.

*Work partially supported by the Computer Science Laboratory of Ericsson Telecom AB, Stockholm, EU Esprit BRA project 8130 LOMAPS, and a Swedish Foundation for Strategic Research Junior Individual Grant.

†Address: SICS, Box 1263, S-164 28 Kista, Sweden. Email: mfd@sics.se and fred@sics.se.

Our aim in this paper is to demonstrate an approach to system specification and verification that has the potentiality of addressing open distributed systems in general. We study the issue in terms of a core fragment of Ericsson's Erlang programming language [AVWW96] which we call Core Erlang. Core Erlang is essentially a first-order actor language (cf. [AMST97]). The language has primitives for local computation: data types, first-order abstraction and pattern matching, and sequential composition. In addition to this Core Erlang has a collection of primitives for component (process) coordination: sending and receiving values between named components, and for dynamically creating new components. Similar to [AMST97] an operational semantics is given in terms of a two-level transition semantics: A transition relation on aggregate processes (or: composed system states) on top of one for local computation.

We use a temporal logic based on a first-order extension of the modal μ-calculus for the specification of component behaviour. In this logic it is possible to describe a wide range of important system properties, ranging from type-like assertions to complex interdependent safety and liveness properties. The development of this logic is actually fairly uncontroversial: To adequately describe component behaviour it is certainly needed to express potentialities of actions across interfaces and the (necessary and contingent) effects of these actions, to express properties of data types, and (to accommodate modular reasoning) to determine whether components are fully evaluated to a ground value, to access component names, and to express properties of messages in transit.

The real challenge is to develop techniques that allow such temporal properties to be verified in a parametric fashion in face of the following basic difficulties:

1. Components can dynamically create other components.
2. Component names can be bound dynamically, thus dynamically changing component interconnection structure (similar to the case of the π-calculus [MPW92]).
3. Components are connected through unbounded message queues.
4. Through use of non-tail recursion components can give rise to local state spaces of unbounded size.
5. Basic data types such as natural numbers and lists are also unbounded.

We would expect some sort of uniformity in the answers to these difficulties. For instance techniques for handling dynamic process creation is likely to be adaptable to non-tail recursive constructions quite generally, and similarly message queues is just another unbounded data type.

In [Dam95] an answer to the question of dynamic process creation was suggested, cast in terms of CCS. Instead of closed correctness assertions of the shape $S : \phi$ (S is a system, ϕ its specification) which are the typical objects of state exploration based techniques the paper considered more general *open correctness assertions* of the shape $\Gamma \vdash S : \phi$ where Γ expresses assumptions $s : \psi$ on components s of S. Thus the behaviour of S is specified parametrically upon the behaviour of its component s. To address verification a sound and weakly complete proof system was presented, consisting of proof rules to reduce complex proof goals to (hopefully) simpler ones, and, most importantly, to accommodate proof goal discharge by loop detection.

Our contribution in the present paper is to show how the approach of [Dam95] can be used to address the difficulties enumerated above for a fragment of a *real* programming language, and to show the utility of our approach on a concrete example exhibiting some of those difficulties. Thus many details of the proof system are kept very informal in the present paper, and we concentrate instead on an intuitive explanation of our approach, in particular the rule of discharge. The proof system captures both model checking-like reasoning (by state exploration) and parametric reasoning, and we motivate the discharge rule by first seeing how it reduces to well-known termination conditions in local model checking, and then generalising to open correctness assertions.

The example is based on the following scenario: A user wants to access a resource paying for this using a given account. She therefore issues a request to a resource manager which responds by dynamically creating a billing agent to act as an intermediary between the user, the resource, and the users account. We view this scenario as quite typical of many security-critical mobile agent applications. The user is clearly taking a risk by exposing her account to the resource manager and the billing agent. One of these parties might violate the trust put in him eg. by charging for services not provided, or by passing information to third parties that should be kept confidential. Equally the resource manager need to trust the billing agent (and to some minor extent the user). We show how the system can be represented in Core Erlang, how a few critical properties can be expressed, and outline a proof that the number of transfers from the user account does not exceed the number of requests to use the resource.

2 Core Erlang

We introduce a core fragment of the Erlang programming language with dynamic networks of processes operating on data types such as natural numbers, lists, tuples, or process identifiers (pid's), using asynchronous, first-order call-by-value communication via unbounded fifo queues. Real Erlang has several additional features such as communication guards, exception handling, modules, distribution extensions, and a host of built-in functions.

Processes A Core Erlang system consists of a number of processes computing in parallel, and communicating by means of fifo ordered, unbounded message queues. Each process is named by a unique pid of which we assume an infinite supply. Associated with a process is an Erlang expression currently being evaluated and a queue of incoming messages, the process mailbox. Messages are sent by addressing a data value to a receiving process mailbox, identified using its pid. We use the notation $\{E, p, Q\}$ for a process with pid p, Erlang expression E, and queue Q. *System states*, or *aggregate processes*, S, are sets of processes, written using the grammar (where "$\|$" expresses parallel execution):

$$S ::= \{E, p, Q\} \mid S \parallel S$$

Abstract Syntax We operate with the syntactical categories of *expressions* E, *matches* M, *patterns* P, and *values* V. The abstract syntax of Core Erlang is summarised as follows:

$$
\begin{array}{rcl}
E & ::= & \Omega(E_1, ..., E_n) \mid X \mid \texttt{self} \mid \texttt{case}\ E\ \texttt{of}\ M \mid E\ E \mid \\
 & & \texttt{spawn}(E, E) \mid E!E \mid \texttt{receive}\ M\ \texttt{end} \mid E, E \\
M & ::= & P_1\texttt{->}E_1;\ \cdots\ ; P_n\texttt{->}E_n \\
P & ::= & \Omega(P_1, ..., P_n) \mid X \\
V & ::= & \Omega(V_1, ..., V_n)
\end{array}
$$

X is an Erlang variable, and Ω ranges over a set of primitive constants and operations including zero 0, successor $E+1$, tupling $\{E_1, E_2\}$, the empty list $[\,]$, list prefix $[E_1|E_2]$, pid constants ranged over by p, and atom constants ranged over by a, f, and g. In addition we need constants and operations for message queues: ϵ is the empty queue and $Q_1 \cdot Q_2$ is queue concatenation.

Atoms are used to name functions. Expressions are interpreted relative to an environment of function definitions $f(X_1, .., X_n)$ `->` E. Such definitions should be considered sugared versions of definitions $f = \{X_1, \{\ldots, X_n\}\}$ `->` E assigning matches to function atoms. Each function atom f is assumed to be defined at most once in this environment.

Intuitive Semantics The intuitive behaviour of most operators should not be too surprising:

- Ω is a data type constructor: To evaluate $\Omega(E_1, ..., E_n)$, E_1 to E_n are evaluated in left-to-right order.
- `self` extracts the pid of the current process.
- `case` E `of` M is evaluated by first evaluating E to a value V, then matching V using M. If several patterns in M match the first one is chosen. Matching a pattern P_i of M against V can cause unbound variables to become bound in E_i[1]. In a function definition $f = M$ all free variables are considered as unbound.
- $E_1\ E_2$ is application: First E_1 is evaluated to a function atom f (we have no λ's), then E_2 is evaluated to a value V, then the function definition of f is looked up and matched to V.
- `spawn`(E_1, E_2) is evaluated by first evaluating E_1 to a function atom f, then E_2 to a value V, a new pid p is generated, and a process with that pid is spawned evaluating $f\ V$. The value of the spawn expression is then p.
- $E_1!E_2$ evaluates E_1 to a pid p, then E_2 to a value V, then V is sent to p, resulting in V as the value of the send expression.
- `receive` M `end` inspects the process mailbox Q and retrieves the first element in Q that matches any pattern of M. Once such an element V has been found, evaluation proceeds analogously to `case` V `of` M.
- E_1, E_2 is sequential composition: First E_1 is evaluated to a value, and then evaluation proceeds with E_2.

[1] This is not quite the binding the convention of Erlang proper: There the first occurrence of X in (case E_1 of X `->` E_2), X can bind the second.

Operational Semantics Expression evaluation takes place in the context of a *process configuration* $\langle E, p, Q\rangle$ [2]. The intuitive meaning of the operators is easily formalised as a Plotkin-style SOS semantics. Transitions are labelled by triples "*pre*,α,*post*" where *pre* is a necessary precondition for performing the transition (formulated in the logic introduced in Section 3), α is the action causing the transition and *post* is the result (in terms of variable bindings etc.) of taking the transition.

For instance we have rules like

$$\textit{input}\quad \frac{X \ \textit{fresh}}{\langle E, p, Q\rangle \xrightarrow{\textit{true}, p?X, \textit{true}} \langle E, p, Q\cdot X\rangle}$$

reflecting that input transitions are always enabled,

$$\textit{spawn1}\quad \frac{\langle E_1, p, Q\rangle \xrightarrow{\textit{pre}, \alpha, \textit{post}} \langle E_1', p, Q'\rangle}{\langle \texttt{spawn}(E_1, E_2), p, Q\rangle \xrightarrow{\textit{pre}, \alpha, \textit{post}} \langle \texttt{spawn}(E_1', E_2), p, Q'\rangle}$$

to allow for expression evaluation in contexts, and

$$\textit{spawn3}\quad \frac{p' \ \textit{fresh}}{\langle \texttt{spawn}(f, V), p, Q\rangle \xrightarrow{\textit{true}, \texttt{spawn}(f, V, p'), \textit{true}} \langle p', p, Q\rangle}$$

Here α is a metavariable over transition types, either τ (corresponding to an internal, spontaneous computation step), output $p!V$, input $p?X$, or spawn, $\texttt{spawn}(f, V, p')$. Rules for the remaining operators are left out for concerns of space. They are, however, quite trivial given the intuition stated above.

It remains to state the rules governing computation steps for (aggregate) processes. These we state in full:

$$\textit{proc1}\quad \frac{\langle E, p, Q\rangle \xrightarrow{\textit{pre}, \tau, \textit{post}} \langle E', p, Q'\rangle}{\{E, p, Q\} \xrightarrow{\textit{pre}, \tau, \textit{post}} \{E', p, Q'\}}$$

$$\textit{proc2}\quad \frac{\langle E, p, Q\rangle \xrightarrow{\textit{pre}, p'!?V, \textit{post}} \langle E', p, Q'\rangle}{\{E, p, Q\} \xrightarrow{\textit{pre}, p'!?V, \textit{post}} \{E', p, Q'\}}$$

$$\textit{proc3}\quad \frac{\langle E, p, Q\rangle \xrightarrow{\textit{pre}, \texttt{spawn}(f, V, p'), \textit{post}} \langle E', p, Q'\rangle}{\{E, p, Q\} \xrightarrow{\textit{pre}, \tau, \textit{post}} \{E', p, Q'\} \parallel \{f\ V, p', \epsilon\}}$$

$$\textit{com}\quad \frac{S_1 \xrightarrow{\textit{pre}_1, p!V, \textit{post}_1} S_1' \qquad S_2 \xrightarrow{\textit{pre}_2, p'?X, \textit{post}_2} S_2'}{S_1 \parallel S_2 \xrightarrow{p = p' \wedge \textit{pre}_1 \wedge \textit{pre}_2, \tau, \textit{post}_1 \wedge \textit{post}_2 \wedge X = V} S_1' \parallel S_2'}$$

$$\textit{interleave1}\quad \frac{S_1 \xrightarrow{\textit{pre}, \tau, \textit{post}} S_1'}{S_1 \parallel S_2 \xrightarrow{\textit{pre}, \tau, \textit{post}} S_1' \parallel S_2}$$

[2]Process configurations are introduced mainly to handle process spawning in subexpressions.

$$\textit{interleave2} \quad \frac{S_1 \xrightarrow{\textit{pre}, p!?V, \textit{post}} S_1'}{S_1 \parallel S_2 \xrightarrow{\texttt{foreign}(p)(S_2) \wedge \textit{pre}, p!?V, \textit{post}} S_1' \parallel S_2}$$

Here we use $p!?V$ as a wildcard among $\{p!V, p?V\}$, and $\texttt{foreign}(p)$ for the predicate on system states S stating that p is not a pid of a process in S. $\texttt{local}(p)$ is the dual of $\texttt{foreign}(p)$.

Example: Billing Agents As an example Core Erlang program, a function for managing accesses to private resources (a resource manager) is shown below.

```
rm(ResList, Bank, RAcc) ->
  receive
    {contract, {Pu, UAcc}, From}} ->
      case lookup(Pu, ResList)) of
        {ok, Pr} -> From!{agent,spawn(billagent, (Pr, Bank, RAcc, UAcc))};
        nok -> From!{agent, nok}
      end
  end, rm(ResList, Bank, RAcc).
```

The resource manager rm accepts a resource list, the pid of a trusted bank agent, and a private account as arguments. The resource list contains pairs of public and private "names". The function $\textit{lookup}(\textit{Pu}, \textit{ResList})$ (not shown) searches for the public name $\textit{Pu}$ in $\textit{ResList}$, and if found, returns the matching private name. After receiving a contract offer (identifying the paying account $\textit{UAcc}$) a billing agent is spawned.

```
billagent(Res, Bank, RAcc, UAcc) ->
  receive
    {use, From} ->
      Res!{use, self},
      receive
        {res, ok, Value} ->
          Bank!{{trans, UAcc, RAcc}, self},
          receive
            {{trans, UAcc, RAcc}, ok} -> From!{use, ok, Value};
            {{trans, UAcc, RAcc}, nok} -> From!{use, nok}
          end;
        {res, nok} -> From!{use, nok}
      end
  end, billagent(Res, Bank, RAcc, UAcc).
```

The billing agent coordinates access to the resource with withdrawals from the account. Upon receiving a request for the resource $\{\textit{use}, \textit{From}\}$ it acquires the resource, attempts to transfer money from the user account to resource manager account $\textit{Bank}!\{\{\textit{trans}, \textit{UAcc}, \textit{RAcc}\}, \texttt{self}\}$, and then sends the resource to the process with pid $\textit{From}$.

3 The Property Specification Logic

In this section we introduce a specification logic for Core Erlang. The logic is based on a first-order μ-calculus, extended with Erlang-specific features. Thus the logic is based on the first-order language of equality, extended with recursive (minimal and maximal) definitions, modalities reflecting the transition capabilities of processes and process configurations, along with a few additional primitives.

Syntax The abstract syntax of formulas is given as follows:

$$\begin{array}{rcl} \phi & ::= & P = P \mid P \neq P \mid \mathtt{term}(P) = P \mid \mathtt{queue}(P) = P \mid \\ & & \mathtt{local}(P) \mid \mathtt{foreign}(P) \mid \mathtt{atom}(P) \mid \mathtt{unevaluated}(P) \mid \\ & & f(P_1, ..., P_n) \mid \phi \wedge \phi \mid \phi \vee \phi \mid \forall X.\phi \mid \exists X.\phi \mid \\ & & <>\phi \mid <P!?P>\phi \mid <\phi>\phi \mid [\,]\phi \mid [P!?P]\phi \mid [\phi]\phi \end{array}$$

Formula atoms are defined by parametrised recursive definitions. As is by now well known monotone recursive definitions in a complete boolean lattice have minimal and maximal solutions. For readability we use the notation $f(X_1, \ldots, X_n) \Rightarrow \phi$ for maximal definitions, and $f(X_1, \ldots, X_n) \Leftarrow \phi$ for minimal ones. Maximal solutions are used, typically, for safety, or invariant properties, while minimal solutions are used for liveness, or eventuality properties. We also use standard abbreviations like *true*, *false*, $\forall X_1, \ldots, X_n.\phi$, etc.

Intuitive Semantics To limit space requirements we refrain from giving a formal semantics of formulas and make do instead with an informal one:

- Equality, inequality, boolean operators and the quantifiers take their usual meaning.
- The purposes of $\mathtt{term}(P_1) = P_2$ and $\mathtt{queue}(P_1) = P_2$ are to pick up the values of terms and queues associated with given pid's. $\mathtt{term}(P_1) = P_2$ requires P_1 to be the pid of a process which is part of the system state being predicated (alt. is the pid of the predicated process configuration), and the Erlang expression associated with that pid to be identical to P_2. Similarly $\mathtt{queue}(P_1) = P_2$ holds if the queue associated with P_1 is P_2.
- $\mathtt{atom}(P)$ holds if P is an atom.
- `local(P)` holds if `P` is the pid of a process in the system state being predicated, and analogously `foreign(P)` holds if `P` is a pid and there is no process with pid `P` in the predicated system state.
- $\mathtt{unevaluated}(P)$ holds if $\mathtt{local}(P)$ does and the Erlang expression associated with P is not a ground value.
- $<>\phi$ holds if an internal transition is enabled to a system state (process configuration) satisfying ϕ. $[\,]\phi$ is the complement of $<>\phi$ (ie. all states following an internal transition satisfy ϕ). $<P_1!P_2>\phi$ holds if an output transition with appropriate parameters is enabled to a state (configuration) satisfying ϕ. $<P_1?P_2>\phi$ is used for

input transitions. Finally $<\phi_1>\phi_2$ predicates process configurations only and holds if a process satisfying ϕ_1 can be spawned such that the ensuing configuration satisfies ϕ_2. The boxed modalities, as before, are dual.

Some Simple Examples The combination of recursive definitions with data types makes the logic very expressive. For instance the type of natural numbers is defined as a minimal recursion:

$$nat(P) \Leftarrow P = 0 \vee \exists X.P = X + 1 \wedge nat(X)$$

Using this idea all involved data types can be defined quite easily. This in turn can be used to define a De-Morganised negation *not*. We can define "weak" modalities that are insensitive to the specific number of internal transitions in the following style:

$$\begin{array}{cc} [[]]\phi \Rightarrow \phi \wedge [\,][[]]\phi & [[X!Y]]\phi = [[]][X!Y][[]]\phi \\ \ll\gg\phi \Leftarrow \phi \vee <>\ll\gg\phi & \ll X!Y\gg\phi = \ll\gg<X!Y>\ll\gg\phi \end{array}$$

Observe the use of formula parameters, and the use of "=" for non-recursive definitions. A temporal property like *always* is also easily defined:

$$always(\phi) \Rightarrow \phi \wedge [\,]always(\phi) \wedge \forall X, Y.[X?Y]always(\phi) \wedge \forall X, Y.[X!Y]always(\phi) \qquad (1)$$

Eventuality operators are more delicate as progress is in general only made when internal or output transitions are taken. This can easily be handled, though, by nesting minimal and maximal definitions.

Billing Agents: Specification We enumerate some desired properties of the billing agent system introduced on Page 6.

Disallowing spontaneous account withdrawals. The first correctness property forbids spontaneous withdrawals from the user account by the billing agent. That is, the number of transfers from the user account should be less than or equal to the number of requests to use the resource.

$$\begin{array}{ll} \multicolumn{2}{l}{safe(Ag, Bank, UAcc, N) \quad \Rightarrow} \\ & [\,]safe(Ag, Bank, UAcc, N) \\ \wedge & \forall P, V.[P?V] \\ & \left(\begin{array}{ll} & P = Ag \wedge \exists Pid.V = \{use, Pid\} \wedge safe(Ag, Bank, UAcc, N+1) \\ \vee & safe(Ag, Bank, UAcc, N) \vee contains(V, UAcc) \end{array}\right) \\ \wedge & \forall P, V.[P!V] \\ & \left(\begin{array}{ll} & \left(\begin{array}{ll} & P = Bank \wedge \exists Pid, Acc.\ V = \{\{trans, UAcc, Acc\}, Pid\} \\ \wedge & N > 0 \wedge safe(Ag, Bank, UAcc, N-1)) \end{array}\right) \\ \vee & \left(\begin{array}{ll} & (P \neq Bank \vee not(\exists Pid, Acc.\ V = \{\{trans, UAcc, Acc\}, Pid\})) \\ \wedge & safe(Ag, Bank, UAcc, N) \end{array}\right) \end{array}\right) \end{array}$$

The predicate $contains(Y, A)$ is defined via structural induction over an Erlang value expression Y and holds if no subexpression of Y is equal to A (definition omitted for lack of space).

Expected Service is Received. Other intersting properties concern facts like: Denial of service responses correspond to failed money transfers, and returning the resource to the proper user. These sorts of properties are not hard to formalise in a style similar to the first example.

Preventing Abuse by a Third Party. The payment scheme presented here depends crucially on the non-communication of private names. For instance, even if we can prove that a resource manager or billing agent does not make illegal withdrawals nothing stops the resource manager from communicating the user account key to a third party, that can then access the account in non-approved ways.

Thus we need to prove at least that the system communicates neither the user account key nor the agent process identifier. Perhaps the service user also requests that her identity not be known outside of the system, in such a case the return process identifiers may not be communicated either. As an example, the property that the system does not communicate the user account key is captured by $\mathit{notrans}(\mathit{UAcc})$ given the definition below.

$$\begin{array}{rcl} \mathit{notrans}(A) & \Rightarrow & [\,]\mathit{notrans}(A) \\ & \wedge & \forall X,Y.[X?Y]\,(\mathit{contains}(Y,A) \vee \mathit{notrans}(A)) \\ & \wedge & \forall X,Y.[X!Y]\,(\mathit{not}(\mathit{contains}(Y,A)) \wedge \mathit{notrans}(A)) \end{array}$$

4 Toward Parametric Verification

Consider a correctness assertion of the shape

$$\Gamma \vdash S : \phi. \tag{2}$$

where S is an arbitrary system state, ϕ an arbitrary specification, and Γ consists of first-order assumptions (like: $V_1 = V_2$) on value variables V in S and ϕ (a restriction that will be lifted in later sections). We wish to devise an adequate and general set of proof rules that allow us to prove such sequents. A natural starting point for this is the classical sequent calculus with equality. A general sequent will have the shape $\Gamma \vdash S : \Delta$ where Δ is a finite set, interpreted disjunctively, of formulas. Classical sequent calculus provides us with a standard collection of rules, easily adaptable to the sequent format of (2), for manipulating sequents, and for introducing connectives to the left and the right of the turnstile. For instance the rule for introducing conjunction to the right becomes

$$\mathsf{AndR}\quad \frac{\Gamma \vdash S : \phi, \Delta \qquad \Gamma \vdash S : \psi, \Delta}{\Gamma \vdash S : \phi \wedge \psi, \Delta}$$

We thus need only consider additions needed for the Erlang-specific primitives, for formula definitions, and for the modal operators. For the former we give a single example for the `term` construction:

$$\mathsf{Term}\quad \frac{\Gamma \vdash \{V, P_1, Q\} \parallel S : P_1 = P_2, \Delta \qquad \Gamma \vdash \{V, P_1, Q\} \parallel S : V = P_3, \Delta}{\Gamma \vdash \{V, P_1, Q\} \parallel S : \mathtt{term}(P_2) = P_3, \Delta}$$

For recursively defined formulas we need rules for definition unfolding, for instance:

$$\text{UnfoldR} \quad \frac{\Gamma \vdash S : \phi[A_1/X_1, ..., A_n/X_n], \Delta \qquad f(X_1, \ldots, X_n) \Rightarrow \phi}{\Gamma \vdash S : f(A_1, \ldots, A_n), \Delta}$$

This leaves the modal operators. For these one option is to explore the transition relation, by rules of the following shape[3]:

$$\text{Diamond} \quad \frac{\Gamma, pre, post \vdash S' : \phi \qquad \Gamma \vdash pre \qquad S \xrightarrow{pre, \alpha, post} S'}{\Gamma \vdash S : <\alpha>\phi}$$

$$\text{Box} \quad \frac{\{\Gamma, pre, post \vdash S' : \phi_1, \ldots, \phi_n \mid S \xrightarrow{pre, \alpha, post} S'\}}{\Gamma \vdash S : [\alpha]\phi_1, \ldots, [\alpha]\phi_n}$$

Proofs built up using the proof rules introduced so far will rarely terminate, as both processes and their specifications are given recursively. We thus need a way of safely discharging an assumption once it is seen to be an instance of a proof goal which has already been encountered during proof construction. We discuss the ideas on the basis of an example.

Example 4.1 Consider the following Core Erlang function:

$$\mathit{stream}(N, \mathit{Out}) \texttt{ -> } \mathit{Out}!N, \mathit{stream}(N+1, \mathit{Out}).$$

which outputs the increasing stream N, $N+1$, $N+2$, ... along Out. The specification of the program could be

$$\mathit{stream\_spec}(\mathit{Out}) = \mathit{always}(\exists X. \ll \mathit{Out}!X \gg \mathit{true})$$

The goal sequent takes the shape

$$\mathit{Out} \neq P \vdash \{\mathit{stream}(N, \mathit{Out}), P, \epsilon\} : \mathit{stream\_spec}(\mathit{Out}). \tag{3}$$

That is, assuming $\mathit{Out} \neq P$ (since otherwise output will go to the stream itself), and started with pid P and the empty input queue, the property *stream_spec*(Out) will hold. One problem with (3) is to avoid the queue component growing in an unbounded manner, due to the input modality of *always* (see Page 8). In fact the goal sequent holds for any initial queue, and we can thus use a rule of substitution to replace (3) by the goal

$$\mathit{Out} \neq P \vdash \{\mathit{stream}(N, \mathit{Out}), P, Q\} : \mathit{stream\_spec}(\mathit{Out}) \tag{4}$$

The first step is to unfold the formula definition. Then, using the proof rules (4) is easily reduced to subgoals of the shapes

$$\mathit{Out} \neq P \vdash \{\mathit{stream}(N, \mathit{Out}), P, Q\} : \ll \mathit{Out}!N \gg \mathit{true} \tag{5}$$

$$\mathit{Out} \neq P \vdash \{\mathit{stream}((N+1), \mathit{Out}), P, Q\} : \mathit{stream\_spec}(\mathit{Out}) \tag{6}$$

$$\mathit{Out} \neq P \vdash \{\mathit{stream}(N, \mathit{Out}), P, Q :: Y\} : \mathit{stream\_spec}(\mathit{Out}). \tag{7}$$

[3]Another option is exemplified by the DiaSeq1 proof rule on Page 14.

Discharge We see that (5) can be proved without recourse to discharge. However, in subgoals (6) and (7) we have arrived at sequents which are both instances of a sequent already "reduced", namely (4). Discharge at these points is indeed sound, for reasons explained below.

Consider a proof constructed using the proof rules introduced so far. Suppose an internal node of the proof is labelled $\Gamma \vdash S : \phi$, and suppose the proof has a leaf which is labelled $\Gamma' \vdash S' : \phi'$ such that $S' : \phi'$ is a substitution instance of $S : \phi$ under a substitution, say, ρ. That is, we have a scenario like the one in ex. 4.1. Suppose also that $\Gamma' \vdash \rho(\phi)$ whenever $\phi \in \Gamma$. That is, with the leaf node $\Gamma' \vdash S' : \phi'$ we have arrived at a situation which is a special case of a situation we have already considered. We wish to devise safe conditions for discharging the leaf node $\Gamma' \vdash S' : \phi'$. Informally it suffices, during proof construction, to tag each formula identifier with a unique label whenever it is looked up by applications of UnfoldR while not yet having been assigned a tag, and to assign another unique tag, say, a *colour*, to each member of the right-hand formula set Δ in a manner which is preserved by all applications of proof rules. Then, if substitutions are required to respect colours, we can introduce a notion of *formula regeneration path* by tracking, from any given $\phi \in \Delta$, using the colouring information, a member of any subsequent right-hand formula set with the same colour, until the leaf has been reached. In that situation we know for the set Δ' that we have tracked a formula ψ which, up to tagging of formula identifiers, is identical to $\rho(\phi)$. We can now inspect the formula regeneration path to pick the latest tagged formula identifier which is unfolded along the regeneration path and which occurs, with that same tag, in ψ. If, for some such ϕ we can find such a formula identifier which is defined using a maximal definition then discharge is permitted, otherwise it is not.

What we have described is a proof system for local model checking in the style of the tableau systems of eg. [SW91] or [Win91], extended to cater for symbolic reasoning on value parameters. A more formal treatment of a similar proof system can be found in [Dam95].

Parametricity Unfortunately the proof rules introduced so far are incapable of dealing satisfactorily with the complications discussed in the introduction. Features like dynamic process spawning or non-tail recursion cause state spaces to grow in an essentially unbounded manner which can not in general be captured by the discharge conditions given in the previous section. This applies, for instance, to the resource manager on Page 6. Moreover, many proof goals require data type induction, mechanisms for which have not been considered yet. In this section we introduce a generalisation of the model checking-like approach of the previous section that permits us to address these issues in a very general way.

Consider again a proof goal $\Gamma \vdash S : \phi$. Assume that S contains a recursively defined expression which when executed can spawn off some process E. Then, attempting to build a proof using the rules already introduced, we will eventually encounter a sequent of the shape

$$\Gamma \vdash S \parallel \{E, p, \epsilon\} : \psi. \tag{8}$$

Clearly we have no hope of terminating proof construction with only the proof rules introduced so far. The solution we propose is to replace components of the agent being predicated by variables of which sufficiently powerful properties can be assumed, for the entire system to be verifiable. This amounts to a cut, using a rule of the following shape:

$$\mathsf{Cut}\quad \frac{\Gamma \vdash S_1 : \gamma' \qquad \Gamma, s : \gamma' \vdash s \parallel S_2 : \gamma}{\Gamma \vdash S_1 \parallel S_2 : \gamma}$$

This rule allows us, intuitively, to abstract from computational details of a component process that are irrelevant for desired property of the system as a whole. Now, to handle (8) we can then try to guess properties γ_1 and γ_2 for each of the component processes, and then cut out these components. First cutting out S results in the following two subgoals:

$$\Gamma \vdash S : \gamma_1 \tag{9}$$

$$\Gamma, s_1 : \gamma_1 \vdash s_1 \parallel \{E, p, \epsilon\} : \psi. \tag{10}$$

Observe that if the outermost specification ϕ is well chosen it will often be possible to choose γ_1 identical to ϕ itself. Moving on we would then use another cut to reduce (10) to two further subgoals

$$\Gamma \vdash \{E, p, \epsilon\} : \gamma_2 \tag{11}$$

$$\Gamma, s_1 : \gamma_1, s_2 : \gamma_2 \vdash s_1 \parallel s_2 : \psi. \tag{12}$$

We have thus arrived in a situation where a proof goal involving a composite system (8) has been reduced to proof goals for the components, along with an open correctness assertion (10) which does not depend on the system components, only on their properties.

Proving properties of open correctness assertions, however, is a very different task from the problem of proving closed ones in that eg. the modal rules Diamond and Box do not apply. Moreover, the rule of discharge has to revised in light of the more general sequent shape, and new sets of proof rules are needed.

Discharge Revisited Observe that in case S can recursively spawn new processes, as we assumed, the reduction of (8) to (9) may not have actually achieved anything, as S can just continue to spawn off new processes. The task is to terminate the loop. Assume for simplicity that we could choose $\gamma_1 = \phi$. In this case (9) and (2) are identical, and we would then hope to be able to discharge (9) for the same reasons as discharge for (6) was seen to be admissible (even though the path from (2) up to (9) arises for very different reasons (structural ones, the cut) than the corresponding path for (6) which uses the modal rules introduced above). In fact this hope turns out to be justified.

Since formulas are are defined recursively also the subgoal (12) may give rise to a recursive proof structure. That is, in building a proof of (12) we may arrive at a situation where the goal sequent has the shape $\Gamma', s_1 : \gamma_1', s_2 : \gamma_2' \vdash s_1 \parallel s_2 : \psi'$ which is an instance of (12). In this case we will have the possibility of discharging when ψ was regenerated because of a maximal formula identifier, but in addition we will have the possibility of discharging when one of the γ_i' are regenerated because of a minimal formula identifier. Intuitively the assumption of a minimally defined property of some s permits us to define an approximation ordinal for which the assumption continues to hold, and when loops are unrolled this ordinal will decrease strictly.

The difficulty is to devise side conditions that ensure that this intuition remains sound even when loops are nested. This issue was addressed in the paper [Dam95] to which we refer for details. For the examples given here, however, the basic intuition is sufficient and sound, whence we leave aside the wider issue for now.

To illustrate the power of the discharge principles notice how, using the representation of data types of section 3, it is possible to capture data type induction by unfolding minimally defined formula identifiers such as

$$nat(P) \Leftarrow P = 0 \vee \exists X.P = X + 1 \wedge nat(X)$$

to the left of the turnstile.

Proof Rules for Open Correctness Assertions So far we have not discussed proof rules that handle the existence of left hand hypotheses of the shape $s : \phi$. What remains, besides the classical sequent calculus rules, are rules to to handle modal operators to the left of the turnstile, and to unfold formula definitions (similar to UnfoldR). We need two monotonicity rules:

$$\mathsf{Mon1}\quad \frac{\Gamma, s : \phi, s : \phi_1, \ldots, s : \phi_n \vdash s : \psi_1, \ldots, \psi_m}{\Gamma, s : {<}\alpha{>}\phi, s : [\alpha]\phi_1, \ldots, s : [\alpha]\phi_n \vdash s : {<}\alpha{>}\psi_1, \ldots, {<}\alpha{>}\psi_m}$$

$$\mathsf{Mon2}\quad \frac{\Gamma, s : \phi_1, \ldots, s : \phi_n \vdash s : \psi}{\Gamma, s : [\alpha]\phi_1, \ldots, s : [\alpha]\phi_n \vdash s : [\alpha]\psi}$$

and then the following rules for combinations of $\|$ with the modalities:

$$\mathsf{DiaPar1}\quad \frac{\begin{array}{c}\Gamma, s_1 : \phi, s_2 : \psi \vdash P_1 = P_3 \qquad \Gamma, s_1 : \phi, s_2 : \psi \vdash P_2 = P_4 \\ \Gamma, s_1 : \phi, s_2 : \psi \vdash s_1 \parallel s_2 : \gamma\end{array}}{\Gamma, s_1 : {<}P_1!P_2{>}\phi, s_2 : {<}P_3?P_4{>}\psi \vdash s_1 \parallel s_2 : {<}{>}\gamma}$$

$$\mathsf{DiaPar2}\quad \frac{\Gamma, s : \phi \vdash s \parallel S : \psi}{\Gamma, s : {<}{>}\phi \vdash s \parallel S : {<}{>}\psi}$$

$$\mathsf{DiaPar3}\quad \frac{\Gamma, s : \phi \vdash s \parallel S : \psi \qquad \Gamma \vdash S : \mathtt{foreign}(P_1)}{\Gamma, s : {<}P_1!?P_2{>}\phi \vdash s \parallel S : {<}P_1!?P_2{>}\psi}$$

$$\mathsf{BoxPar1}\quad \frac{\begin{array}{c}\Gamma, s_1 : \phi_1, s_2 : [\,]\psi_1, s_2 : \forall X, Y.[X!Y]\psi_2, s_2 : \forall X, Y.[X?Y]\psi_3 \vdash s_1 \parallel s_2 : \gamma \\ \Gamma, s_1 : [\,]\phi_1, s_1 : \forall X, Y.[X!Y]\phi_2, s_1 : \forall X, Y.[X?Y]\phi_3, s_2 : \psi_1 \vdash s_1 \parallel s_2 : \gamma \\ \Gamma, s_1 : \phi_2, s_2 : \psi_3 \vdash s_1 \parallel s_2 : \gamma \\ \Gamma, s_1 : \phi_3, s_2 : \psi_2 \vdash s_1 \parallel s_2 : \gamma\end{array}}{\begin{array}{c}\Gamma, s_1 : [\,]\phi_1, s_1 : \forall X, Y.[X!Y]\phi_2, s_1 : \forall X, Y.[X?Y]\phi_3, \\ s_2 : [\,]\psi_1, s_2 : \forall X, Y.[X!Y]\psi_2, s_2 : \forall X, Y.[X?Y]\psi_3 \vdash s_1 \parallel s_2 : [\,]\gamma\end{array}}$$

$$\mathsf{BoxPar2}\quad \frac{\begin{array}{c}\Gamma, s_1 : \phi, s_2 : \mathtt{foreign}(P_1), s_2 : [P_1!?P_2]\psi \vdash s_1 \parallel s_2 : \gamma \\ \Gamma, s_1 : [P_1!?P_2]\phi, s_1 : \mathtt{foreign}(P_1), s_2 : \psi \vdash s_1 \parallel s_2 : \gamma\end{array}}{\Gamma, s_1 : [P_1!?P_2]\phi, s_2 : [P_1!?P_2]\psi \vdash s_1 \parallel s_2 : [P_1!?P_2]\gamma}$$

For these five rules we assume that Γ mentions neither s_1, s_2, nor s. A similar assumption applies to the monotonicity rules. In actual proofs we tend to use slight generalisations of the last two proof rules where the left hand assumptions like $s_1 : [\,]\phi_1$ are allowed to be vectors.

So far we have presented only rules to reason compositionally about system states. To prove general properties also of non-tail recursive processes another layer is needed in the proof system, for sequents of the form $\Gamma \vdash \langle E, p, Q\rangle : \Delta$. This extension is quite voluminous though it offers no significant new problems. As an example one of the rules for handling sequences is shown below.

$$\mathsf{DiaSeq1} \quad \frac{\Gamma, \langle x, p, q\rangle : \phi \vdash \langle (x, E), p, q\rangle : \psi}{\Gamma, \langle x, p, q\rangle : <\alpha>\phi \vdash \langle (x, E), p, q\rangle : <\alpha>\psi}$$

5 Verifying the Resource Manager

In this section the proof system is demonstrated by outlining a proof that the resource manager on Page 6 satisfies the *safe* specification defined on Page 8. For simplicity it is assumed that the manager knows of only one resource, with public name P_u and private P_r. The list $[\{P_u, P_r\}]$ is denoted with R_L. Furthermore R_P denotes the process identifier of the resource manager process, R_Q its input queue.

Since the definition of *safe* is parametrised on a billing agent and a user account the formula must be preceded by an initialisation phase:

$$\begin{array}{l}\forall \mathit{PubRes}, \mathit{UAcc}, \mathit{From}, \mathit{Agent}.\\ \quad [R_P?\{\mathit{contract}, \{\mathit{PubRes}, \mathit{UAcc}\}, \mathit{From}\}]\\ \qquad [[\mathit{From}!\{\mathit{agent}, \mathit{Agent}\}]]\mathit{safe}(\mathit{Agent}, \mathit{Bank}, \mathit{UAcc}, 0)\end{array}$$

So we set out to prove the following sequent:

$$\begin{array}{rcl}\Gamma & \vdash & \{\mathit{rm}(R_L, \mathit{Bank}, \mathit{RAcc}), R_P, \epsilon\}\\ & : & \forall \mathit{PubRes}, \mathit{UAcc}, \mathit{From}, \mathit{Agent}.[R_P?\{\mathit{contract}, \{\mathit{PubRes}, \mathit{UAcc}\}, \mathit{From}\}] \ldots\end{array} \tag{1}$$

The necessary assumptions on the non-equivalence of process identifiers (e.g., $R_P \neq P_r$) are collected in Γ. By application of simple proof steps – four applications of $\mathsf{ForallR}$ and then repeated applications of the rules for unfolding, elimination of conjunctions, and the rules for the box modality – the following proof state is reached:

$$\begin{array}{rcl}\Gamma' & \vdash & \{\mathit{rm}(R_L, \mathit{Bank}, \mathit{RAcc}), R_P, \epsilon\} || \{\mathit{billagent}(P_u, \mathit{Bank}, \mathit{RAcc}, \mathit{UAcc}), B_P, \epsilon\}\\ & : & \mathit{safe}(B_P, \mathit{Bank}, \mathit{UAcc}, 0)\end{array} \tag{2}$$

where Γ' is Γ extended with the fact that B_P is a fresh process identifier. This is a critical proof state, where we must come up with a property of the resource manager, and a property of the billing agent, that are sufficiently strong to prove that their parallel composition satisfies the *safe* property. In general such a proof step may be very difficult, but here we can simply choose to prove the *safe* property of the *billagent* process, and for the resource manager process that it never communicates with the user account. The result of applying the Cut rule twice is the following proof obligations:

$$\Gamma' \vdash \{billagent(P_u, Bank, RAcc, UAcc), B_P, \epsilon\} \ : \ safe(B_P, Bank, UAcc, 0) \quad (3)$$

$$\Gamma' \vdash \{rm(R_L, Bank, RAcc), R_P, \epsilon\} \ : \ nocomm(UAcc) \quad (4)$$

$$\Gamma',\ s_1 : safe(B_P, Bank, UAcc, 0),\ s_2 : nocomm(UAcc) \vdash s_1||s_2 : safe(B_P, Bank, UAcc, 0) \quad (5)$$

where

$$\begin{aligned} nocomm(P) \ \Rightarrow \ & [\,]nocomm(P) \\ \wedge \ & \forall X, Y.[X?Y]\,(contains(Y, P) \vee nocomm(P)) \\ \wedge \ & \forall X, Y.[X!Y]\,(P \neq X \wedge nocomm(P)) \end{aligned}$$

The proof of (3) involves essentially well-known techniques for proving correctness of sequential programs. Instead we concentrate on (4), or rather, the stronger statement that rm satisfies $nocomm$ even when its input queue is non-empty, as long as no element in the queue contains $UAcc$:

$$\Gamma', not(contains(R_Q, UAcc)) \vdash \{rm(R_L, Bank, RAcc), R_P, R_Q\} : nocomm(UAcc) \quad (6)$$

To prove (6) we first unfold the definition of $nocomm$ and eliminate the conjunctions. In case of an input step ($[X?Y]$) either we are done immediately (if $contains(Y, P)$). Otherwise the resulting proof state is

$$\Gamma', not(contains(R_Q, UAcc)), not(contains(Y, UAcc)) \ \vdash \{rm(R_L, Bank, RAcc), R_P, R_Q \cdot Y\} : nocomm(UAcc) \quad (7)$$

which can be rewritten into a proof state that can be discharged (see discussion on Page 11) against (6). The rm process can clearly not perform any output step so that part of the conjunction is trivially true. Thus only the internal step remains, and such a step must correspond to retrieving a value $\{contract, \{{P_u}', UAcc'\}, From\}$ from the queue. The resulting proof state is:

$$\Gamma', not(contains(R_Q, UAcc)) \ \vdash \{\texttt{case}\ lookup({P_u}', R_L) \ldots, R_P, {R_Q}'\} : nocomm(UAcc) \quad (8)$$

where ${R_Q}'$ is the queue resulting from the reception of the value. By repeating the above steps, i.e., handling input, output and internal steps eventually one reaches the goal:

$$\Gamma'' \vdash \{rm(R_L, Bank, RAcc), R_P, {R_Q}''\} \,||\, \{billagent(P_u, Bank, RAcc, UAcc'), {B_P}', \epsilon\} \ : \ nocomm(UAcc) \quad (9)$$

where Γ'' is $\Gamma', not(contains(R_Q, UAcc))$ together with inequalities involving the new process identifier ${B_P}'$. This goal is handled by applying Cut to the parallel composition using $nocomm$ as the cut formula both to the left and to the right. The resulting goals are:

$$\Gamma'' \vdash \{rm(R_L, Bank, RAcc), R_P, {R_Q}''\} : nocomm(UAcc) \quad (10)$$

$$\Gamma'' \vdash \{billagent(P_u, Bank, RAcc, UAcc'), {B_P}', \epsilon\} : nocomm(UAcc) \quad (11)$$

$$\Gamma'', Z : nocomm(UAcc), V : nocomm(UAcc) \vdash Z \parallel V : nocomm(UAcc) \quad (12)$$

Goal (10) can be discharged against (6). Goal (11) is easy to prove, since no new processes are created (proof sketch omitted). Finally goals (5) and (12) are handled by applications of BoxPar1, BoxPar2, Unfold and discharging against previously seen goals.

6 Concluding Remarks

We have introduced a specification logic and proof system for the verification of programs in a core fragment of Erlang, and illustrated its application on a small, but quite delicate, agent-based example. Our approach is quite general both regarding the kinds of languages and models that can be addressed, and the kinds of assertions that can be formulated[4]. In addition our approach permits the treatment of programming language constructs such as dynamic process creation, non-tail recursion and inductive data type definitions in a uniform way, via a powerful rule of discharge.

As the goal of this work is quite ambitious, i.e., to enable verification of open distributed systems as implemented using real programming languages, there remains a number of shortcomings in the current work. First of all, we need to investigate the proof system in real applications. For this purpose a prototype proof checking tool has been produced based on the approach presented here, that can handle programs of a moderate size[5]. Some support for automation of proof steps already exists (e.g., for the model checking fragment), but we also need to identify other classes of sequents that can be proved algorithmically. Other ongoing work focuses on integrating the operational semantics of Erlang more tightly with the proof systems (along the lines of [Sim95]), to improve the handling of process identifier scoping (but see [AD96] for an approach to this in the context of the π-calculus), and to handle fairness in a more elegant manner.

References

[AD96] R. Amadio and M. Dam. A modal theory of types for the π-calculus. In *Proc. FTRTFT'96,* Lecture Notes in Computer Science, 1135:347–365, 1996.

[AMST97] G. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. A foundation for actor computation. *J. Functional Programming*, 7:1–72, 1997.

[AVWW96] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang (Second Edition).* Prentice-Hall International (UK) Ltd., 1996.

[Dam95] M. Dam. Compositional proof systems for model checking infinite state processes. In *Proc. CONCUR'95,* Lecture Notes in Computer Science, 962:12–26, 1995.

[MPW92] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, I and II. *Information and Computation*, 100(1):1–40 and 41–77, 1992.

[Sim95] A. Simpson. Compositionality via cut-elimination: Hennessy-Milner logic for an arbitrary GSOS. In *Proceedings, Tenth Annual IEEE Symposium on Logic in Computer Science*, pages 420–430, San Diego, California, 26–29 June 1995. IEEE Computer Society Press.

[4] For instance we are not restricted, as in many other approaches to compositional verification, to linear-time logic, neither does the proof system rely on auxiliary features like history or prophecy variables.

[5] The major novelty of the proof checker lies in the handling of the rule of discharge.

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

[Win91] G. Winskel. A note on model checking the modal ν-calculus. *Theoretical Computer Science*, 83:157–187, 1991.