A Framework for Formal Reasoning about Open Distributed Systems[⋆]

Lars-åke Fredlund and Dilian Gurov

Swedish Institute of Computer Science,
Box 1263, SE-164 29 Kista, Sweden,
`fred|dilian@sics.se`

Abstract. We present a framework for formal reasoning about the behaviour of distributed programs implementing open distributed systems (ODSs). The framework is based on the following key ingredients: a specification language based on the μ-calculus, a hierarchical transitional semantics of the implementation language used, a judgment format allowing parametrised behavioural assertions, and a proof system for proving validity of such assertions which includes proof rules for property decomposition. This setting provides the expressive power for behavioural reasoning required by the complex open and dynamic nature of ODSs. The utility of the approach is illustrated on a prototypical ODS.

1 Introduction

For a few years now, the Formal Design Techniques group at the Swedish Institute of Computer Science has pursued a programme aimed at enabling formal verification of complex open distributed systems (ODSs) through program code verification. While previous work by the group has been predominantly directed towards establishing the mathematical machinery [5], basic tool support [3], and performing case studies [2], the present paper focuses on methodological aspects by motivating the chosen verification framework and by showing on an example proof how suitable this framework is in practice for formal reasoning about the behaviour of ODSs.

A central feature of open distributed systems as opposed to concurrent systems in general is their reliance on modularity. Large-scale open distributed systems, for instance in telecom applications, must accommodate complex functionality such as dynamic 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 the 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.

The decision to focus on verification of actual program code rather than addressing the easier task of verifying specifications comes from the observation that still, after all these years of advocating formalised specifications as a means to improve the quality of products, in industry today only rarely does one find such formalised specifications.

We summarise the framework in Section 2 as it has developed throughout the project, and then illustrate its merits in Section 3 by focusing on a prototypical distributed systems example where a set data structure is implemented through the coordination of a dynamically changing number of processes. The example is programmed in the Erlang language [1], a functional programming language with support for distribution and concurrency, that is nowadays used in numerous telecommunication products developed by the Ericsson corporation. To illustrate the verification method we formulate and sketch a proof of a key property of the set implementation.

2   Verification of Open Distributed Systems

First we examine the characteristics of programming platforms for open distributed systems, and from this description derive requirements on the formal machinery necessary to permit verification of open distributed systems.

2.1   Programming Platforms for ODSs

Programming platforms provide the necessary functionality for open distributed systems. To name but a few services typically provided:

1. The basic building blocks that can execute concurrently (processes and/or threads, concurrent objects).
2. Facilities for dynamically creating new executing entities.
3. Means for coordination of, and communication between, concurrently executing entities. For example through semaphores, a shared memory, remote method calls, or asynchronous message passing.
4. Support for implicitly or explicitly grouping executing entities into more complex structures such as process groups, rings of processes or hypercubes.
5. Support for fault detection and fault recovery.

Like large software systems in general, ODSs are usually built from libraries of *components*. These ideally use encapsulation to provide clean *interfaces* to the components to prevent their improper use.

2.2   A Framework for Formal Reasoning about ODS Behaviour

Semantics of ODSs. To reason in a formal fashion about the behaviour of an ODS, a formal semantics of the design language in which the system is described is needed. This can be done in different styles, depending on the intended style of reasoning. Our methodology is mainly tailored to *operational semantics*, although other formal notions of behaviour are derivable in our framework, supporting reasoning in different flavours. Operational semantics are usually presented by transition rules involving labelled transitions between structured states [11]. A natural approach to handling the different conceptual layers of entities in the language, supporting modular (i.e. compositional) reasoning, is to organise the semantics hierarchically, in layers, using different sets of transition labels at each layer, and extending at each layer the structure of the state with new components as needed. This approach will be illustrated in the example of the Erlang programming language in Section 2.3.

Specification Language. Reasoning about complex systems requires *compositional reasoning*, i.e. the capability to reduce arguments about the behaviour of compound entities to arguments about the behaviours of its parts. To support compositional reasoning, a specification language should capture the labelled transitions at each layer of the transitional semantics. Poly-modal logic is particularly suitable for the task, employing box and diamond *modalities* labelled by the transition labels: a structured state s satisfies formula $\langle \alpha \rangle \Phi$ if there is an α-derivative of s (i.e. a state s' such that $s \xrightarrow{\alpha} s'$ is a valid labelled transition) satisfying Φ, while s satisfies $[\alpha]\Phi$ if all α-derivatives of s (if any) satisfy Φ. Additionally, *state predicates* are needed to capture the "local", unobservable characteristics of structured states, such as e.g. the value of a local variable. The presence of recursion on different layers requires also the specification language to be recursive. Adding recursion in the form of least and greatest fixed-points to the modalities described above results in a powerful specification language, broadly known as the μ-*calculus* [10, 8]. Roughly speaking, least fixed-point formulas $\mu X.\phi$ express eventuality properties, while greatest fixed-point formulas $\nu X.\phi$ express invariant properties. Nesting of fixed points allows complicated reactivity and fairness properties to be expressed.

Parametricity. As explained above, reasoning about open systems requires reasoning about their interface behaviour relativised by assumptions about certain system parameters. Technically, this can be achieved by using Gentzen-style proof systems, allowing free parameters to occur within the *proof judgments* of the proof system. The judgments are of the form $\Gamma \vdash \Delta$, where Γ and Δ are sets 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 simply variables ranging over specific types of entities, such as messages, functions, or processes. For example, the proof judgment $x : \Psi \vdash P(x) : \Phi$

states that object P has property Φ provided the parameter x of P satisfies property Ψ.

Compositionality. Reducing an argument about the behaviour of compound entities to arguments about the behaviours of its parts can be achieved through parametricity: We can relativise an assertion $P[Q/x] : \Phi$ about the compound object P to a certain property Ψ of its component Q by considering Q as a parameter for which property Ψ is assumed, provided we can show that Q indeed satisfies the assumed property Ψ. Technically this can be achieved through a *term-cut* proof rule of the shape:

$$\frac{\Gamma \vdash Q : \Psi, \Delta \qquad \Gamma, x : \Psi \vdash P : \Phi, \Delta}{\Gamma \vdash P[Q/x] : \Phi, \Delta}$$

Recursion. When reasoning about programs in the presence of recursion on different layers, one traditionally relies on different forms of *inductive reasoning*, such as mathematical induction, complete induction and well-founded induction. Of these, the latter is the most general one. Through a sophisticated mechanism for generalized-loop detection, fixed-point approximation, and discharge, our proof method supports well-founded inductive reasoning, as well as proofs by *co-induction* [9] which is needed when reasoning about entities of non-well-founded nature such as infinite streams. Recursion on any layer like data, functions, and processes is treated uniformly in this framework.

The mechanism itself is presented and studied in detail in [5]; here we only give an idea of the approach. Assume that we are to prove that repeated popping of elements from a stack must eventually fail unless interleaved with the pushing of new elements. The initial proof goal will roughly have the shape *s:stack* ⊢ *reppop(s):terminates*, where *stack* is the type of stack expressed as a formula describing the transitional semantics of stacks, *reppop* is a function implementing repeated popping, and *terminates* is a formula expressing termination of computation. Since the stack definition and the formulas are recursive, in the process of proof construction we will eventually reach a point where we have to prove the same termination property, but for a modified stack. In fact, this new proof goal will be an *instance* of the more general initial goal, and in this way we will have discovered a generalised loop in the proof tree. But along this loop we will have made progress in that we will have decreased the value of an ordinal approximating a least fixed-point formula describing the stack. This fact will allow the new goal to be discharged with respect to the initial goal, analogously to the way assumptions are discharged in natural deduction, thus terminating successfully the respective branch in the proof tree.

2.3   Programming ODSs in Erlang

Erlang [1] is at its core a functional programming language, extended with a notion of processes and primitives for message passing. Erlang has a small set of

powerful constructs, and is therefore suitable as both a modeling as well as an implementation language for ODSs consisting of a high number of light-weight processes. It is especially suitable for telecommunications software. In contrast to most other functional programming languages, Erlang has seen heavy use in industry. In a recent project at Ericsson where a state-of-the-art high-speed ATM switch was developed [4], figures of 480 000 lines of Erlang source code have been reported, compared to 330 000 lines of C code (most of it in the form of imported protocol libraries), and approximately 5 000 lines of Java code. A frequently voiced opinion is that a chief reason for the quick development of this product, and with resulting excellent quality, is the fast code-debug-replace cycle made possible through the introduction of Erlang. Another important reason for the success of Erlang in such projects clearly are the accompanying libraries which provide support for many aspects of developing and maintaining large telecommunications applications. There is for instance support for distributed data base access, error recovery, and code replacement during runtime. A brief overview of the Erlang fragment used in this paper can be found in Appendix A.1.

Formal Semantics Our semantics for Erlang is a small-step operational one [6]. The basic message of the previous section with respect to language semantics was the desirability to mimic the conceptual view that a programmer has of a system built using Erlang in the language semantics. The semantics developed here matches closely the hierarchic structure of the Erlang language. First the Erlang expressions are provided with a semantics that does not require any notion of processes. The actions here are a computation step τ, an output $pid!v$, $read(q, v)$ for reading a value v from the queue of the process in which context the expression executes, and $f(v_1, \ldots, v_n) \rightsquigarrow v$ for calling a builtin function (like $\texttt{spawn}$ for process spawning) with side-effects on the process level state. An example of an expression level transition rule is:

$$\textsf{fun}_1 \ \frac{f(v) \rightsquigarrow v}{f(\tilde{v}) \xrightarrow{\ f(v) \rightsquigarrow v\ } v}$$

The transitional behaviors of Erlang systems are captured separated into two cases: (i) a single process constraining the behaviors of an Erlang expression as illustrated in the following rule for process spawning:

$$\textsf{spawning} \ \frac{e \xrightarrow{\ \texttt{spawn}(module, f, v) \rightsquigarrow pid'\ } e' \quad pid' \neq pid}{\texttt{proc} \langle e, pid, q \rangle \longrightarrow \texttt{proc} \langle e', pid, q \rangle \parallel \texttt{proc} \langle module : f(v), pid', \texttt{eps} \rangle}$$

and (ii) the (parallel) composition of two Erlang systems into a single one exemplified in the following rule for interleaving:

$$\textsf{interleave}_0 \ \frac{s_1 \xrightarrow{\ \tau\ } s_1' \quad wellformed(s_1' \parallel s_2)}{s_1 \parallel s_2 \xrightarrow{\ \tau\ } s_1' \parallel s_2}$$

where $wellformed(s)$ requires that process identifiers of processes in s are unique. The system actions are computation steps τ, input $pid?v$ and output $pid!v$.

2.4   Verifying ODSs in EVT

The Erlang Verification Tool (EVT for short) is a proof editing tool implementing the above described framework: providing a property specification language and an embedding of an operational semantics for Erlang, combined with a general proof system based on the classical first-order sequent calculus.

$$
\begin{aligned}
\text{F} ::= \quad & \texttt{tt} \ \Big|\ \texttt{ff} \ \Big|\ \text{T = T} \ \Big|\ \text{F /\textbackslash\ F} \ \Big|\ \text{F => F} \ \Big|\ \text{F \textbackslash/ F} \ \Big|\ \texttt{not F} \\
& \Big|\ \texttt{forall Var : Type . F} \ \Big|\ \texttt{exists Var : Type . F} \\
& \Big|\ \lambda\text{Var:Type.F} \ \Big|\ \text{F T} \ \Big|\ \text{T : F} \\
& \Big|\ \text{[Action]F} \ \Big|\ \text{<Action>F}
\end{aligned}
$$

PredicateDef ::=   Name : PropType DefSymbol F
PropType ::=   `prop` | Type `->` PropType
DefSymbol ::=   `=>` | `<=` | `=`

Fig. 1. The syntax of logic formulae and definitions

The syntax of the specification logic of EVT is illustrated in Figure 1. In addition to the usual connectives of predicate logic the $\langle\alpha\rangle$F and $[\alpha]$F modalities are available with their usual meaning, defined by referring to the transition relations of the embedded operational semantics. The T : F construct expresses the proposition "T satisfies F" (or "T has type F"). In the following we will refer to a number of predefined types such as `erlangValue` (ground values), `erlangExpression` (expressions), `erlangSystem` (systems), `erlangAction` (actions ranging over computation steps `tau`, output $pid!v$ and input $pid?v$), etc. In the definition of a predicate (PredicateDef) the `=>` symbol selects the greatest fixed point, the `<=` symbol the least fixed point, while `=` is for non-recursive definitions (shorthands). Recursive occurrences of predicates are only permitted under an even number of negations to ensure monotonicity.

3   Verifying a Prototypical Open Distributed System

Active data structures, i.e., collections of processes that by coordinating their activities mimic in a concurrent way some data structure, are frequently used in telecommunication software. In a previous study [2] a protocol for responding to database queries, directed to the distributed database manager Mnesia, was verified. Internally the protocol built up a ring like structure of connected processes in order to answer queries efficiently. In the current example we examine a scheme for a set implementation inspired by a set-as-process example of Hoare [7]. Here the active data structure is a linked list, but the similarities with the database query example are striking.

3.1   An Implementation of a Persistent Set

As an abstract mathematical notion, a *set* is simply a collection of objects (taken out of an universe of objects), characterized by the membership relation "$\in$": if s is an object and S is a set, then the statement $s \in S$ is either true or false. Using the membership relation, one can define sets as unions, intersections, or differences of other sets, or in other ways.

Computer scientists have also another view of sets, namely as *mutable* objects: a set, when manipulated by adding or removing elements, still keeps its "identity", e.g. through an identifier. Any data-structure for manipulating collections of objects, which does not impose an order on its elements (i.e. hides this order through its interface), can be understood as implementing a set.

The objects to be manipulated can be distributed in space, and if the objects themselves are large, it is conceivable, that we might want each object to be maintained by a separate process. A further reason for implementing a set as an active data structure is to permit concurrent access to multiple elements.

A complete implementation of a set, without a possibility to remove elements, by means of a collection of interacting processes is given in Appendix A.2, where a module `persistent_set_adt` is defined. Internally the module implements two functions - one for maintaining of single elements, and one for the empty set. A set is identified by an Erlang process identifier. When creating a new set, it initially consists of a single process executing the `empty_set` function; it is the process identifier of this process by which the set is to be identified from hence on. When an element is added, a new process is spawned off to store the element if it is not already present in the set. Internally, when a new element is added to a set, it is "pushed downwards" through the list of processes representing set elements, until it reaches the emptyset process, which spawns off another emptyset process, and becomes itself a process maintaining the new element. So, as a result, a set is implemented as a unidirectional linked collection of processes referenced by a process identifier.

To encapsulate the set against improper use, we provide a controlled interface to the set module, consisting of a function for set creation `mk_empty`, testing for membership `is_member`, addition of elements `add_element`, etc. The set creation function, for example, spawns off a process executing the `empty_set` function, and returns the process identifier of the newly spawned process. This process identifier has then to be provided as an argument to all the other interface functions. The implementation of the two set functions and the interface prevents the user of the set module from having to notice that sets are internally represented by processes, and moreover prevents direct access to any other process identifiers created internal to the linked list of processes.

Note however that any process, given knowledge of the process identifier of a persistent set, can choose to circumvent the interface functions and directly communicate (through message passing) with the set process. As we shall see in the proof such "protocol abuse" can lead to program errors.

3.2   A Persistent Set Property

To check the correctness of a persistent set implementation, we have to specify those properties of sets which we consider paramount for correct behaviour. Ideally, one would like such a specification to be complete, i.e. a system should satisfy the specification exactly when it implements such a set. Completeness, however, is usually difficult to achieve in practice, since such a specification would be very detailed and the resulting proofs could easily become unmanageably complex.

One crucial property of persistent sets is naturally that they retain any element added to them. For simplicity, we will here prove a simpler property, that once any element has been added to such a set the set will forever be non-empty. The main predicates are:

```
ag_non_empty: erlangPid -> erlangSystem -> prop =>
  \SetPid:erlangPid. \SetSys:erlangSystem.
    ((SetSys : non_empty SetPid) /\
     (SetSys : forall Alpha:erlangAction.[Alpha](ag_non_empty SetPid)));

persistently_non_empty: erlangPid -> erlangSystem -> prop =>
  \SetPid:erlangPid. \SetSys:erlangSystem.
    (((SetSys : non_empty SetPid) /\ (SetSys : ag_non_empty SetPid)) \/
      (SetSys : empty SetPid) /\ (SetSys : forall Alpha:erlangAction.
                                [Alpha](persistently_non_empty SetPid)));
```

Intuitively the `persistently_non_empty` predicate expresses an automaton that, when applied to a process identifier `SetPid` and an Erlang system `SetSys` representing a set, checks that `empty SetPid` remains true until `non_empty SetPid` becomes true, after which `non_empty SetPid` must remain continuously true forever (definition `ag_non_empty`). Note that this is, in some respect, a challenging property since it contains both a safety part (*non-empty sets never claim to be empty*) and a liveness part (*all sets eventually answer queries whether they are empty*).

We advocate an observational approach to specification, through invocation of the interface functions, as evidenced in the definition of the `empty` predicate:

```
empty: erlangPid -> erlangSystem -> prop =
  \SetPid:erlangPid. \SetSys:erlangSystem.
    (forall Pid:erlangPid.
     ((not (Pid = SetPid)) =>
      (proc<is_empty(SetPid), Pid, eps> || SetSys : (evaluates_to Pid true))));
```

The `empty` predicate expresses that `proc<is_empty(SetPid), Pid, eps>`, an observer process, will eventually (in a finite number of steps) terminate with the value `true`, if executing concurrently with the observed set `SetSys`. For lack of space the definition of the `evaluates_to` predicate has been omitted.

3.3   A Proof Sketch

Expressed in the syntax of EVT the main proof obligation becomes:

```
prove "declare P:erlangPid in |- proc<empty_set(), P, eps> : persistently_non_empty P";
```

That is the Erlang system `proc<empty_set(), P, eps>`, an initially empty set, satisfies the `persistently_non_empty` P property. In fact we will prove a slightly stronger property:

```
Goal #0: not(add_in_queue Q) |- proc<empty_set(), P, Q> : persistently_non_empty P;
```

where the `not(add_in_queue Q)` assumption expresses that the queue `Q` does not contain an `add_element` message. This proof goal is reduced by unfolding the definition of the `persistently_non_empty` predicate, choosing to show that the set process will signal that it is empty when queried, and performing a few other trivial proof steps. There are two resulting proof goals:

```
#1: not(add_in_queue Q) |- proc<empty_set(), P, Q> : empty P
#2: not(add_in_queue Q) |- proc<empty_set(), P, Q> :
         forall Alpha:erlangAction. [Alpha](persistently_non_empty P)
```

Goal `#1` reduces to (after unfolding `empty` and rewriting):

```
not(add_in_queue Q), not(P=P') |- proc<is_empty(P), P', eps> || proc<empty_set(), P, Q> :
                               evaluates_to P' true
```

That is, an observer process calling the interface routine `is_empty` with the set process identifier `P` as argument will eventually (in a finite number of steps) evaluate to the value `true` (meaning that the set is empty). Here the proof strategy is to symbolically "execute" the two processes together with the formula, and observe that in all possible future states the observer process terminates with `true` as the result. Note however that the assumption `not(add_in_queue Q)` is crucial due to the Erlang semantics of queue handling. If the queue `Q` contains an `add_element` message the observer process will instead return `false` as a result, since its `is_empty` message would be stored after the `add_element` message in the queue and thus be serviced only after an element is added to the set.

The second proof goal `#2` is reduced by eliminating the universal quantifier, and computing the next state under all possible types of actions. Since the process is unable to perform an output action there are two resulting goals, one which corresponds to the input of a message `V` (note the resulting queue `Q@V`) and the second a computation step (applying the `empty_set` function).

```
#3: not(add_in_queue Q) |- proc<empty_set(), P, Q@V> : persistently_non_empty P
#4: not(add_in_queue Q) |- proc<receive ... end, P, Q> : persistently_non_empty P
```

Proceeding with goal `#4` either the first message to be read from the queue is `is_empty` or `is_member` (the possibility of an `add_element` message can be discarded due to the queue assumption). Handling these two new goals presents no major difficulties.

Goal `#3` is reduced by analysing the value of `V`. If it is not an `add_element` message then we can easily extend the assumption about non-emptiness of `Q`:

```
#5: not(add_in_queue Q@V) |- proc<empty_set(), P, Q@V> : persistently_non_empty P
```

Goal #5 is clearly an instance of goal #0, i.e., we can find a substitution of variables that when applied to the original goal will result in the current proof goal (the identity substitution except that it maps the queue Q to the queue Q@V). Since we have at the same time unfolded a greatest fixed point on the right hand side of the turnstile (the definition of persistently_non_empty) we are allowed to discharge the current proof goal at this point. If, on the other hand, V is an add_element message the next goal becomes:

```
#6: add_in_queue Q@V |- proc<empty_set(), P, Q@V> : persistently_non_empty P
```

At this point we cannot discharge the proof goal, since there is no substitution from the original proof goal to the current one. Instead we repeat the steps of the proof of goal #0 but taking care to show non_empty P instead of empty P. Also, we cannot discard the possibility of receiving an add_element message and the resulting goal is (after weakening out the queue assumption):

```
#7: |- proc<set(Element,mk_empty(...)),P,Q'> : ag_non_empty P
```

By repeating the above pattern of reasoning with regards to goal #7 we eventually reach the proof state:

```
#8: not(P=P') |- proc<set(Element,P'),P,Q''> || proc<empty_set,P',eps> : ag_non_empty P
```

The Erlang components of the proof states of the proof, up to the point of the spawning off of the new process, are illustrated in Figure 2.
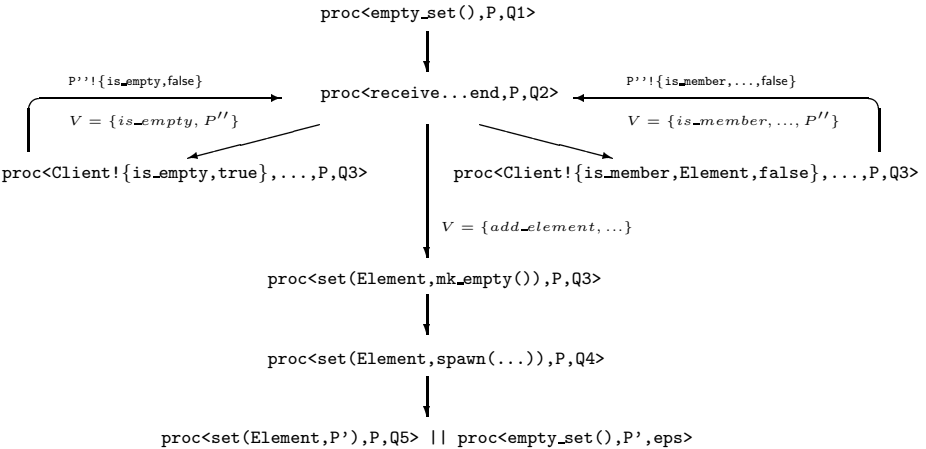
proc<empty_set(),P,Q1>

$P''!\{is\_empty,false\}$ proc<receive...end,P,Q2> $P''!\{is\_member,...,false\}$

$V = \{is\_empty, P''\}$ $V = \{is\_member, ..., P''\}$

proc<Client!{is_empty,true},...,P,Q3>   proc<Client!{is_member,Element,false},...,P,Q3>

$V = \{add\_element, ...\}$

proc<set(Element,mk_empty()),P,Q3>

proc<set(Element,spawn(...)),P,Q4>

proc<set(Element,P'),P,Q5> || proc<empty_set(),P',eps>

Fig. 2. Erlang components of initial proof states

At this point we have reached a critical point in the proof where some manual decision is required. Clearly we can repeat the above proof steps forever,

never being able to discharge all proof goals, due to the possibility of spawning new processes. Instead we apply the *term-cut* proof rule, to abstract the freshly spawned processes with a formula `psi` ending up with two new proof goals:

```
#9: not(P=P') |- proc<empty_set(), P', eps> : psi P P'
#10: not(P=P'), X:psi P P' |- proc<set(Element,P), P, Q''> || X : ag_non_empty P
```

How should we choose `psi`? The cut formula must be expressive enough to characterise the `proc<empty_set(), P', eps>` process, in the context of the second process and for the purpose of proving the formula `ag_non_empty P`. Here it turns out that the following formula is sufficient:

```
psi: erlangPid -> erlangPid -> erlangSystem -> prop =>
  \P:erlangPid . \P':erlangPid .
  (   (forall P:erlangPid . forall V:erlangValue . [P?V]((not(is_empty V)) => psi P P'))
   /\ (forall P:erlangPid . forall V:erlangValue . [P!V](not(is_empty V)))
   /\ (converges P P') /\ (foreign P) /\ (local P'));
```

Intuitively `psi` expresses:

- Whenever a new message is received, and it is not an `is_empty` message then `psi` continues to hold.
- An `is_empty` reply is never issued.
- The predicated system can only perform a finite number number of internal and output steps (definition of `converges` omitted).
- Process identifier `P` is foreign (does not belong to any process in the predicated system) and process identifier `P'` is local (definitions omitted).

The proof of goal `#9` is straightforward up to reaching the goal:

```
#11: not(P'=P'') |- proc<set(Element,P''), P', Q'''> || proc<empty_set(), P'', eps>: psi P P'
```

Here we once again apply the term-cut rule to obtain the following goals:

```
#12: not(P'=P'') |- proc<empty_set(), P'', eps> : psi P' P''
#13: Y:psi P' P'' |- proc<set(Element,P''),P',Q'''>||Y : psi P P'
```

Goal `#12` can be discharged immediately due to the fact that it is an instance of goal `#9`. Goal `#13` involves symbolically executing the `proc<set(Element,P''),P',Q'''>` process together with the (abstracted) process variable `Y`, thus generating their combined proof state space. Since both these systems generate finite proof state spaces this construction will eventually terminate. The proof of goal `#10` is highly similar to the proof of goal `#13` above, and is omitted for lack of space.

3.4   A Discussion of the Proof

The proof itself represented a serious challenge in several respects:

- The modelled system is an open one in which at any time additional set elements can be added outside of the control of the set implementation itself. The state space of the set implementation is clearly non finite state: both the number of processes and the size of message queues can potentially grow without bound.

– The queue semantics of Erlang has some curious effects with regards to an observer interacting with the set implementation. It is for instance not sufficient to consider only the program states of the set process to determine whether an observer will recognise a set to be empty or not; also the contents of the input message queue of the set process has to be taken into account.

Although the correctness of the program may at first glance appear obvious, a closer inspection of the source code through the process of proving the implementation correct revealed a number of problems.

For instance, in an earlier version of the set module the guards `pid(Client)` in the `empty_set` and `set` functions were missing. These guards serve to ensure that any received `is_empty` or `is_member` message must contain a valid process identifier. Should these guards be removed a set process will terminate due to a runtime (typing) error if, say, a message `{is_empty,21}` is sent to it.

In most languages adding such guards would not be needed since usage of the interface functions should ensure that these kinds of "typing errors" can never take place. In Erlang, in contrast, it is perfectly possible to circumvent the interface functions and communicate directly with the set implementation.

4   Conclusion

We have introduced an ambitious proof system based verification framework that enables formal reasoning about complex open distributed systems (ODSs) as programmed in real programming languages like Erlang. The proof method was illustrated on a prototypical example where a set library was implemented as a process structure, and verified with respect to a particular correctness property formulated in an expressive specification logic. Parts of the proof were checked using the Erlang Verification Tool, a proof editing tool with specific knowledge about Erlang syntax and operational semantics.

In conclusion we have clearly illustrated the great potential of the approach: we were able to verify non-trivial properties of real code in spite of difficulties such as the essentially non-finite state nature of the class of open distributed systems studied in the example. In addition the approach is promising because of its generality: we are certainly not tied for all future to the currently studied programming language (Erlang) but can, by providing alternative operational semantics, easily target other programming languages. Still numerous improvements of the framework are necessary, perhaps at the moment most importantly with respect to the interaction with the proof editing tool. We are currently forced to reason at a detail level where too many manual proof steps are required to complete proofs. How to rectify this situation by providing high-level automated proof tactics remains an area of active research.

References

[1] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang (Second Edition)*. Prentice-Hall International (UK) Ltd., 1996.
[2] T. Arts and M. Dam. Verifying a distributed database lookup manager written in Erlang. To appear in *Proc. Formal Methods Europe'99*, 1999.
[3] 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.
[4] S. Blau and J. Rooth. AXD 301 - a new generation ATM switching system. *Ericsson Review*, 1:10–17, 1998.
[5] 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, 1536:150–185, 1998.
[6] L. Fredlund. Towards a semantics for Erlang. Unpublished manuscript*, Swedish Institute of Computer Science*, 1999.
[7] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.
[8] D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, **27**:333–354, 1983.
[9] R. Milner and M. Tofte. Co-induction in relational semantics. *Theoretical Computer Science*, 87:209–220, 1991.
[10] D. Park. Fixpoint induction and proof of program semantics. *Machine Intelligence*, **5**:59–78, 1970.
[11] G. D. Plotkin. A structural approach to operational semantics. Aarhus University report DAIMI FN-19, 1981.

A   Appendix

A.1   A Short Introduction to Erlang

An Erlang *process*, here written $\texttt{proc}\langle e, pid, q\rangle$, is a container for evaluation of functional expressions e, that can potentially have side effects (e.g., communicate). 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. The empty queue will be denoted with $\texttt{eps}$, and $q@v$ is a queue composed of the subqueue q and the value v. To express the concurrent execution of two sets of processes s_1 and s_2, the syntax $s_1 \parallel s_2$ is used.

The functional sublanguage of Erlang is rather standard: atoms, integers, lists and tuples are value constructors; $e_1(\tilde{e_2})$ is a function call; e_1, e_2 is sequential composition; $\texttt{case } e \texttt{ of } p_1[\texttt{when } e_{1g}]\texttt{->}e_1; \ldots; p_n[\texttt{when } e_{1g}]\texttt{->}e_n \texttt{ end}$ is matching: the value that e evaluates to is matched sequentially against patterns (values that may contain unbound variables) p_i, respecting the optional guard expressions e_{ig}. $e_1!e_2$ is sending whereas $\texttt{receive } m \texttt{ end}$ 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 $\texttt{case } v \texttt{ of } m$. Finally $\texttt{if } e_1\texttt{->}\tilde{e'_1}; \cdots e_n\texttt{->}\tilde{e'_n} \texttt{ end}$ is sequential choice.

Expressions are interpreted relative to an environment of "user defined" function definitions $f(\tilde{p_1})$ ->$\tilde{e_1}; \cdots; f(\tilde{p_k})$ ->$\tilde{e_k}$. The Erlang functions are placed in *modules* using the `-module` construct. Functions not explicitly exported using the `-export` construct are unavailable outside the body of the module.

A number of builtin functions are used in the paper. `self` returns the pid of the current process. `spawn`(e_1, e_2, e_3) creates a new process, with empty mailbox, executing the expression $e_2(\tilde{e_3})$ in the context of module e_1. The pid of the new process is returned. `pid`(e) evaluates to `true` if e is a pid and `false` otherwise. Syntactical equality (inequality) is checked by `==` (`/=`).

A.2   The Set Erlang Module

```
-module(persistent_set_adt).
-export([mk_empty/0, is_empty/1, is_member/2, add_element/2, empty_set/0]).

empty_set () ->
    receive
        {is_empty, Client} when pid(Client) ->
            Client ! {is_empty, true}, empty_set ();
        {is_member, Element, Client} when pid(Client) ->
            Client ! {is_member, Element, false}, empty_set ();
        {add_element, Element}->
            set (Element, mk_empty ())
    end.

set (Element, Set) ->
    receive
        {is_empty, Client} when pid(Client) ->
            Client ! {is_empty, false}, set (Element, Set);
        {is_member, SomeElement, Client} when pid(Client) ->
            if
                SomeElement == Element ->
                    Client ! {is_member, SomeElement, true}, set (Element, Set);
                SomeElement /= Element ->
                    Set ! {is_member, SomeElement, Client}, set (Element, Set)
            end;
        {add_element, SomeElement} ->
            if
                SomeElement == Element ->
                    set (Element, Set);
                SomeElement /= Element ->
                    Set ! {add_element, SomeElement}, set (Element, Set)
            end
    end.

%% MODULE INTERFACE FUNCTIONS

mk_empty () -> spawn (persistent_set_adt, empty_set, []).

is_empty (Set) ->
    Set ! {is_empty, self ()},
    receive
        {is_empty, Value} -> Value
    end.

is_member (Element, Set) ->
    Set ! {is_member, Element, self ()},
    receive
        {is_member, Element, Value} -> Value
    end.

add_element (Element, Set) -> Set ! {add_element, Element}.
```