Soter: An Automatic Safety Verifier for Erlang

Emanuele D'Osualdo
University of Oxford
emanuele.dosualdo@cs.ox.ac.uk

Jonathan Kochems
University of Oxford
jonathan.kochems@cs.ox.ac.uk

C.-H. Luke Ong
University of Oxford
luke.ong@cs.ox.ac.uk

Abstract

This paper presents Soter, a fully-automatic program analyser and verifier for Erlang modules. The fragment of Erlang accepted by Soter includes the higher-order functional constructs and all the key features of *actor concurrency*, namely, dynamic and possibly unbounded spawning of processes and asynchronous message passing. Soter uses a combination of static analysis and infinite-state model checking to verify safety properties specified by the user. Given an Erlang module and a set of properties, Soter first extracts an abstract (approximate but sound) model in the form of an *actor communicating system* (ACS), and then checks if the properties are satisfied using a Petri net coverability checker, BFC. To our knowledge, Soter is the first fully-automatic, infinite-state model checker for a large fragment of Erlang. We find that in practice our abstraction technique is accurate enough to verify an interesting range of safety properties such as mutual-exclusion and boundedness of mailboxes. Though the ACS coverability problem is EXPSPACE-complete, Soter can analyse these problems surprisingly efficiently.

1. Introduction

This paper presents Soter, a tool that automatically verifies safety properties of concurrent Erlang programs, based on the framework of [4]. Erlang is an open-sourced language with support for higher-order functions, concurrency, communication, distribution, fault tolerance, on-the-fly code reloading and multiple platforms [2]. The sequential part of Erlang is a higher order, dynamically typed, call-by-value functional language with pattern-matching algebraic data types. Following the *actor model* [7], a concurrent Erlang computation consists of a dynamic network of processes that communicate by asynchronous message passing. Each process has a unique process identifier (pid), and is equipped with an unbounded mailbox. Message send is non-blocking. Retrieval of messages from the mailbox is not FIFO but First-In-First-Firable-Out (FIFFO) via pattern-matching. A process may block while waiting for a message that matches a certain pattern to arrive in its mailbox. Thanks to a highly efficient runtime system, Erlang is a natural fit for programming multicore CPUs, networked servers, distributed databases, GUIs, and monitoring, control and testing tools. For an introduction to Erlang, see Armstrong's *CACM* article [1].

Safety Verification by Static Analysis and Model Checking

The challenge of verifying Erlang programs is that one must reason about the asynchronous communication of an unbounded set of messages, across an unbounded set of Turing-powerful, higher-order processes. The inherent complexity of the verification task can be seen from several diverse sources of infinity in the state space.

(∞ 1) Function definitions are not necessarily tail-recursive, so a call-stack is needed.

(∞ 2) Higher-order functions are first-class values; closures can be passed as parameters or returned.

(∞ 3) Data domains, and hence the message space, are unbounded: functions may return, and variables may be bound to, terms of an arbitrary size.

(∞ 4) An unbounded number of processes can be spawned dynamically.

(∞ 5) Mailboxes have unbounded capacity.

This motivates our model checking approach: we automatically extract an abstract model that simulates the semantics of the program by construction, then we use decision procedures on the abstract model to prove safety properties.

Our abstract model, called *Actor Communicating System*, is highly expressive: it can model dynamic spawning and unbounded mailboxes. An ACS is defined by a finite set of rules but it is infinite-state i.e. its dynamic semantics includes traces that go through infinitely many different configurations. It follows that one cannot establish reachability by exploring all the possible runs. However ACS are equivalent to Petri nets for which model-checking algorithms do exist. Our tool uses a Petri net coverability checker called BFC [8]. ACS models are described in Section 2.

Overview of Soter

Soter is an experimental, prototype Haskell implementation of the framework of [4]. It accepts a (concurrent) subset of the Erlang language: supported features include algebraic data-types with pattern-matching, higher-order, spawning of new processes, asynchronous communication. See Section 4 for the Erlang constructs that are not currently supported by Soter.

As presented in Figure 1, Soter's workflow has three phases.

In phase 1, the input Erlang module with correctness annotations is compiled using the standard Erlang compiler `erlc` to a module of *Core Erlang* — the official intermediate representation of Erlang. The code is then normalised in preparation for the

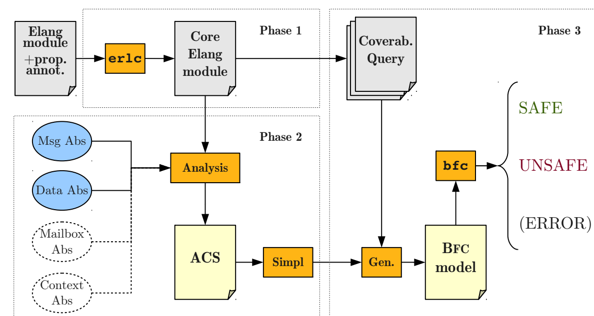

Figure 1. Workflow of Soter

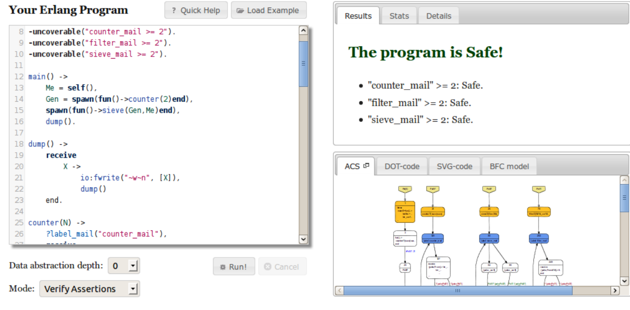

Figure 2. Screenshot of Soter's web interface

next phase. Correctness properties, expressible in various forms, are specified by annotating the program source. The user can insert assertions, or label program points and mailboxes and then state constraints they must satisfy.

The main purpose of phase 2 is to soundly abstract sources of infinity (∞ 1), (∞ 2) and (∞ 3). This is done as follows. A control-flow based analysis is performed on the program, yielding a control flow graph on which we bootstrap the generation of the ACS rules. The analysis is parametric in D and M, the depth of the data and message abstraction respectively. We abstract data by truncating terms at the specified depth. By default D is set to zero, so calls to the same function with different arguments are merged in the abstract model; the runtime of the analysis is exponential in D. M is by default set to $D + P$ where P is the maximum depth of the receive patterns of the program; using large values for M does not incur the same slowdown as adjusting D. In future releases, we plan to introduce parameters to tune the precision of the abstraction so that users can control the context sensitivity of the analysis.

In phase 3, Soter generates a Petri net in the format of BFC [8], which is a fast coverability checker for Petri nets with transfer arcs, developed by Alexander Kaiser. For each property Soter needs to prove, BFC is called internally with the BFC model and a query representing the safety property as input.

Soter can be run in three modes: "analysis only" which produces the ACS, skipping phase 3; "verify assertions" which extracts the properties from user annotations; "verify absence-of-errors" which generates BFC queries asserting the absence of runtime exceptions. Currently not all the exceptions that the Erlang runtime can throw are represented; the supported ones include sending a message to a non-pid value, applying a function with the wrong arity, and spawning a non-functional value. A notable omission is pattern-matching failures which will be supported by the next release of Soter.

Soter is a practical implementation of a highly complex procedure. Phases 1 and 2 are polytime in the size of the input program [4]. Despite the EXPSPACE-completeness of the Petri net coverability problem, phase 3 is surprisingly efficient; see the outcome of the experiments in Table 1.

In addition to a command-line interface, we have built a web interface for Soter at http://mjolnir.cs.ox.ac.uk/soter/. The user interface allows easy input of Erlang programs. A library of annotated example programs is available to be tried and modified. Soter presents the generated abstract model as a labelled graph for easy visualisation, and reports in detail on the performance and results of the verification. A screenshot of the web interface is shown in Figure 2.

Related Work There are a few popular bug-finding tools for Erlang, notably Dyalizer [3, 9] which implements a variety of static analyses. McErlang [5] and EtomCRL2 [6] are model checkers

```
1  main() -> Me = self(),
2            Gen = spawn(fun()->counter(2)end),
3            spawn(fun()->sieve(Gen,Me)end),
4            dump().
5
6  dump() -> receive X -> io:write(X), dump() end.
7
8  counter(N) ->
9     ?label_mail("counter_mail"),
10    receive {poke, From} ->
11       From!{ans, N}, counter(N+1)
12    end.
13
14 sieve(In, Out) ->
15    ?label_mail("sieve_mail"),
16    In!{poke, self()},
17    receive {ans,X} ->
18       Out!X,
19       F = spawn(fun()->
20             filter(divisible_by(X), In)
21          end),
22       sieve(F,Out)
23    end.
24
25 filter(Test, In) ->
26    ?label_mail("filter_mail"),
27    receive {poke, From} ->
28       filter(Test, In, From)
29    end.
30
31 filter(Test, In, Out) ->
32    In!{poke, self()},
33    receive {ans,Y} ->
34       case Test(Y) of
35          false -> Out!{ans,Y}, filter(Test, In);
36          true  -> filter(Test, In, Out)
37       end
38    end.
39
40 -ifdef(SOTER).
41    divisible_by(X) ->
42       fun(Y) -> ?any_bool() end.
43 -else.
44    divisible_by(X) ->
45       fun(Y) -> case Y rem X of
46                    0 -> true;
47                    _ -> false
48                 end
49       end.
50 -endif.
```

Figure 3. Eratosthenes' Sieve, actor style

based on finite-state abstract models, the extraction of which is not automatic. By comparison, Soter is fully automatic and employs infinite-state verification techniques.

2. Actor Communicating Systems

The abstract model we extract from the input Erlang program is an *Actor Communicating System* (ACS), which models the interaction of an unbounded set of communicating processes. An ACS has a finite set Q of *control states*, a finite set P of *pid classes*, a finite set M of message kinds and a finite set of rules. An ACS rule has the shape $\iota\colon q \xrightarrow{\ell} q'$ which means that a process of pid class ι can transition from state q to state q' with (possible) *communication side effect* ℓ, of which there are four kinds:

(i) the process makes an internal transition,
(ii) it extracts and reads a message m from its mailbox,
(iii) it sends a message m to a process of pid class ι' and
(iv) it spawns a process of pid class ι'.

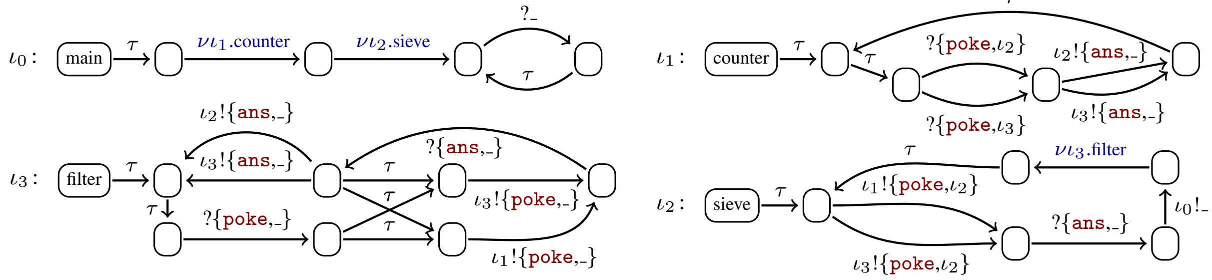

Figure 4. The ACS graph generated by Soter from the sieve example. The ι_0 component represents the starting process which sets up the counter agent (ι_1) and the sieve agent (ι_2) and then becomes the dump agent. During its execution, the sieve agent spawns new filter agents, all represented by the ι_3 component.

To help user visualise the model, we present an ACS as a collection of label graphs (called components), each showing a pid-class, where the vertices are the control states and the labelled edges are the rules. An example of such a graphical presentation is shown in Figure 4.

The semantics of ACS is as follows. Each control state is a counter holding tokens; when a $\iota\colon q \xrightarrow{\tau} q'$ rule is executed a token is extracted from q and transferred to q'; if q contains no tokens the rule is not enabled. Spawn rules insert a new token in the control state of the process to be created while making the transition. Message passing is dealt with analogously: for each component there is a counter for each message; these counters keep track of the number of messages that have been sent to that component so far, thus merging all the mailboxes of the processes of the component. When a message is sent, a token is inserted in the relevant counter. A receive rule can fire only when the counter for the message to be extracted contains at least one token; when fired, a token gets consumed. Note that the order of arrival of messages is not recorded.

An ACS can be interpreted naturally as a *vector addition system* (VAS), or equivalently Petri net. Recall that a VAS of dimension n is given by a set of n-long vectors of integers regarded as transition rules. A VAS defines a state transition graph whose states are just n-long vectors of non-negative integers. There is a transition from state $\mathbf{v}$ to state $\mathbf{v}'$ just if $\mathbf{v}' = \mathbf{v}+\mathbf{r}$ for some transition rule $\mathbf{r}$. In this paper, we are concerned with the EXPSPACE-complete decision problem *Coverability* [10]: given a VAS, a start vector s and a target non-negative vector t of the same dimension, is it possible to reach some v that covers t (i.e. $v \geq t$)? Note that *LTL Model Checking* is also EXPSPACE-complete for VAS; *Reachability* is decidable but its complexity is open.

A wide range of properties can be encoded as coverability queries on the ACS. Examples include reachability of error states, mutual exclusion, bounds on the number of enqueued messages in a mailbox. Some of these correctness properties can be exploited by optimising compilers. Bounds on mailboxes of a class of processes, for example, allow the compiler to allocate a fixed number of cells for that mailbox, resulting in programs that can be efficiently garbage-collected.

Liveness properties such as deadlock freedom cannot currently be checked by Soter because there are no efficient implementations of LTL model checking for Petri nets. Should such implementations become available, Soter can quickly take advantage of them.

3. Demo: A Concurrent Eratosthene's Sieve

We illustrate the workings of Soter by an example. Figure 3 shows an implementation of Eratosthenes' sieve inspired by a NewSqueak program by Rob Pike.[1] The actor defined by `counter` provides the sequence of natural numbers as responses to `poke` messages, starting from 2; the `dump` actor prints everything it receives. The `sieve` actor's goal is to send all prime numbers in sequence as messages to the `dump` actor; to do so it pokes its current `In` actor waiting for a prime number. After forwarding the received prime number, it creates (`spawn`) a new `filter` process, which becomes its new `In` actor. The filter actor, when poked, queries its `In` actor until a number satisfying `Test` is received and then it forwards it; the test (an higher-order parameter) is initialized by `sieve` to be a divisibility check that tests if the received number is divisible by the last prime produced. The overall effect is a growing chain of `filter` actors each filtering multiples of the primes produced so far; at one end of the chain there is the counter, at the other the sieve that forwards the results to `dump`.

Since Soter does not have native support for arithmetic operations, line 41 defines a stub to be used by Soter that returns true or false non-deterministically, thus soundly approximating the real definition based on division given in line 44.

The communication here is synchronous in spirit: whenever a message is sent, the sender actor blocks waiting for a reply. To check this is the case, we can verify the property that every mailbox contains in fact at most one message at any time. To be able to express this constraint we label the mailboxes we are interested in with the `?label_mail()` macro: the instructions in lines 9, 15 and 26 mark the mailbox of any process that may execute them with the corresponding label.

Then we can insert the following lines at the beginning of the module

```
-uncoverable("counter_mail >= 2").
-uncoverable("filter_mail >= 2").
-uncoverable("sieve_mail >= 2").
```

which state the property we want to prove. The `-uncoverable` directive is ignored by the Erlang compiler but it is interpreted by Soter as a property to be proved: all the states satisfying the constraint are considered to be “bad states”. These inequalities state that if the total number of messages in the labelled mailboxes exceed the given bound, we are in a bad state.

Soter allows user-defined labels for program locations as well with the macro `?label()`; the inequalities in this case state that the total number of processes executing the labelled instruction at the same time must be less that the given bound.

When executed on the code in Figure 3, Soter will compute the ACS in Figure 4; its semantics is a sound approximation of the actual semantics of the program. A VAS description of it,

[1] see “Concurrency/message passing Newsqueak”, http://video.google.com/videoplay?docid=810232012617965344

Example	LOC	PRP	SAFE?	ABSTR		ACS SIZE		TIME			
				D	M	Places	Ratio	Analysis	Simpl	BFC	Total
reslockbeh	507	1	yes	0	2	40	4%	1.94	0.41	0.85	3.21
reslock	356	1	yes	0	2	40	10%	0.56	0.08	0.82	1.48
sieve	230	3	yes	0	2	47	19%	0.26	0.03	2.46	2.76
concdb	321	1	yes	0	2	67	12%	1.10	0.16	5.19	6.46
state_factory	295	2	yes	0	1	22	4%	0.59	0.13	0.02	0.75
pipe	173	1	yes	0	0	18	8%	0.15	0.03	0.00	0.18
ring	211	1	yes	0	2	36	9%	0.55	0.07	0.25	0.88
parikh	101	1	yes	0	2	42	41%	0.05	0.01	0.07	0.13
unsafe_send	49	1	no	0	1	10	38%	0.02	0.00	0.00	0.02
safe_send	82	1	no*	0	1	33	36%	0.05	0.01	0.00	0.06
safe_send	82	4	yes	1	2	82	34%	0.23	0.03	0.06	0.32
firewall	236	1	no*	0	2	35	10%	0.36	0.05	0.02	0.44
firewall	236	1	yes	1	3	74	10%	2.38	0.30	0.00	2.69
finite_leader	555	1	no*	0	2	56	20%	0.35	0.03	0.01	0.40
finite_leader	555	1	yes	1	3	97	23%	0.75	0.07	0.86	1.70
stutter	115	1	no*	0	0	15	19%	0.04	0.00	0.00	0.05
howait	187	1	no*	0	2	29	14%	0.19	0.02	0.00	0.22

Table 1. Soter Benchmarks. The number of lines of code refers to the compiled Core Erlang. The PRP column indicates the number of properties which need to be proved. The columns D and M indicate the data and message abstraction depth respectively. In the "Safe?" column, "no*" means that the program satisfies the properties but the verification was inconclusive; "no" means that the program is not safe and Soter finds a genuine counterexample. "Places" is the number of places of the underlying Petri net after the simplification; "Ratio" is the ratio of the number of places of the generated Petri net before and after the simplification. All times are in seconds.

incorporating the property, is then generated and fed to BFC to check for the uncoverability of bad states; in this instance BFC is successful in proving the program safe.

4. Experiments, Limitations and Extensions

Evaluation In Table 1 we summarise our experimental results. Soter is a fully automatic tool. All our example programs are higher-order and use dynamic (and unbounded) process creation and non-trivial synchronisation. The properties checked fall into three groups: mutual exclusion, unreachability of error states, and bounds on mailboxes. The annotated example programs in Table 1 can all be viewed and verified using Soter at the web interface `http://mjolnir.cs.ox.ac.uk/soter/`. As indicated by the experimental outcome, the abstractions employed by Soter are sufficiently precise to prove safety for a wide variety of examples. We observe that the ACS simplification is especially effective in reducing the problem size. BFC implements an algorithm that is EXPSPACE-hard in the ACS size. The experiments show there are other factors such as transition structure that strongly influence the runtime complexity of BFC, although it is not yet clear what these parameters are. In conclusion, despite the worst-case exponential complexity of the underlying algorithm, Soter is surprisingly efficient. We believe that the experimental outcome justifies further development of the tool.

Limitations Features of Erlang currently unsupported by Soter can be organised into three groups: (i) constructs such exceptions, arithmetic primitives, built-in data types and the module system are not difficult to integrate into the current framework; (ii) features such as time-outs in receives, registered processes, input-output and type guards could be supported by providing specific abstractions; (iii) the monitor / link primitives and the multi-node semantics. These features need to be supported explicitly by the abstract model for them to be usefully approximated, and may require a major extension of the theory. How to extend our framework to explicitly and precisely model the last two groups of features is an interesting research problem. Despite these limitations, it is usually possible to adapt existing programs so that they are accepted by Soter: often it is sufficient to provide "dummy" implementations of the unsupported functions as it has been done in line 41 of the example code in Figure 3.

In addition certain problem instances remain out of Soter's scope: if the proof of safety for a program requires accurate modelling of the stack or precise sequencing information on the arrival order of messages, then our abstractions are not suitable; further our analysis assumes a closed program — the ability to analyse and model-check open programs would enable a compositional approach which we expect would enhance Soter's scalability.

Extensions and Future Directions We plan the following extensions: (i) handle arbitrary Core Erlang programs (ii) formalise and implement specific abstractions for time-outs and I/O (iii) develop fine-tuned, flexible and refineable abstractions for data, mailboxes and context-sensitivity, which would facilitate the construction of a CEGAR loop (iv) exploit the decidability of LTL-properties for ACS to enable Soter to prove liveness and other path properties.

References

[1] J. Armstrong. Erlang. *CACM*, 53(9):68, 2010.

[2] F. Cesarini and S. Thompson. *Erlang Programming - A Concurrent Approach to Software Development*. O'Reilly, 2009.

[3] M. Christakis and K. Sagonas. Detection of asynchronous message passing errors using static analysis. *PADL*, pages 5–18, 2011.

[4] E. D'Osualdo, J. Kochems, and C.-H. L. Ong. Automatic verification of Erlang-style concurrency. Technical report, 2012. `http://mjolnir.cs.ox.ac.uk/soter/soterpaper.pdf`.

[5] L. Fredlund and H. Svensson. McErlang: a model checker for a distributed functional programming language. In *ICFP*, pages 125–136, 2007.

[6] Qiang Guo and John Derrick. Verification of timed erlang/otp components using the process algebra mucrl. *Erlang Workshop*, pages 55–64. 2007.

[7] C. Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. In *IJCAI*, pages 235–245, 1973.

[8] A. Kaiser, D. Kroening, and T. Wahl. Efficient coverability analysis by proof minimization. In *CONCUR*, 2012. `http://www.cprover.org/bfc/`.

[9] T. Lindahl and K. Sagonas. Practical type inference based on success typings. In *PPDP*, pages 167–178, 2006.

[10] C. Rackoff. The covering and boundedness problems for vector addition systems. *TCS*, 6:223–231, 1978.