Model Checking Erlang Programs – Abstracting the Context-Free Structure

Frank Huch

Institut für Informatik
Christian-Albrechts-Universität Kiel
Kiel, Germany
fhu@informatik.uni-kiel.de

Abstract. We present an approach for the verification of Erlang programs using abstract interpretation and model checking. In previous work we defined a framework for abstract interpretations for Erlang. In this framework it is guaranteed, that the abstract operational semantics preserves all paths of the standard operational semantics. We consider properties that have to hold on all paths of a system, like properties in LTL. If these properties can be proven for the abstract operational semantics, then they also hold for the Erlang program. The proof can be automated with model checking if the abstract operational semantics is a finite transition system. But finiteness cannot be guaranteed because of non–tail recursive function calls. Even for finite domain abstract interpretations we get infinite state systems and model checking is undecidable. In this paper we define an abstraction of the control–flow. It replaces the recursive calls in non-tail positions by jumps to the last call of the same function. The corresponding returns are replace by jumps to the possible return points.
We have implemented this approach as a prototype and are able to prove properties like mutual exclusion or the absence of deadlocks and lifelocks for some Erlang programs.

Keywords: abstraction, model checking, Erlang, distributed system, context-free structure

1 Introduction

Growing requirements of industry and society impose greater complexity of software development. Consequently understandability, maintenance and reliability cannot be warranted. Things get even harder when we leave the sequential territory and develop distributed systems. Here many processes run concurrently and interact via communication. This can e.g. yield problems like deadlocks or lifelocks. To guarantee the correctness of software formal verifiaction is needed.

We propose an extension of model checking to programs written in real programming languages. However the model checking problem in general is undecidable for system implementations using programming languages and properties described in interesting logics. Hence we need abstraction [5, 12, 15].

In industry the programming language Erlang [1] is used for the implementation of distributed systems. We have developed a framework for abstract interpretations for a core fragment of Erlang in [10] with the property that the transition system defined by the abstract operational semantics (*AOS*) includes all paths of the standard operational semantics (*SOS*). Because the AOS can sometimes have more paths than the SOS, it is only possible to prove properties that have to be fulfilled on all paths, like in linear time logic (LTL). If the abstraction fulfills a property expressed in LTL, then also the program fulfills it, but not vice versa. If the AOS is a finite transition system, then model checking is decidable [13, 17].

The defined abstraction does only yield a finite transition system for a subclass of Erlang programs, called hierarchical programs [10]. Recursion is only allowed in tail positions. However, in practice many Erlang programs do not fulfill this restriction. For example, already the standard definitions of `append` or `length` are not hierarchical. Hence programs which use such functions cannot be abstracted to finite state transition system with the presented technique of abstract interpretation. The cause is the context-free structure of functional programs. In this paper we define an abstraction of this context-free structure, to a regular one. We obtain a finite state transition system. Properties of the system can automatically be proven with LTL model checking.

In Section 2 we define the syntax for a core fragment of Erlang. We sketch the operational semantics in Section 3. The framework for the abstract interpretation is shortly introduced in Section 4 and its restrictions are presented in Section 5. In Section 6 we present a graph semantics, on which our abstraction is based. We motivate the idea of our abstraction in Section 7 and formalize it in Section 8. Section 9 presents its use in model checking and finally we conclude and discuss future work in Section 10.

2 Syntax of Core Erlang

Let Σ be a set of predefined function symbols with arity. For example $\texttt{+}/2 \in \Sigma$. Let $\mathit{Var} = \{\texttt{X}, \texttt{Y}, \texttt{Z}, \ldots\}$ be a set of variables and *Atoms* a set of atoms, e.g. $\{1, 2, \texttt{fail}, \texttt{succ}, \ldots\}$. Let C be the set of Erlang constructor functions with arity:

$$C = \{\, \texttt{[.|.]}/2, \texttt{[]}/0\} \cup \{\texttt{\{...\}}/n \mid n \in \mathbb{N}\} \cup \{a/0 \mid a \in \mathit{Atoms}\}, \qquad (1)$$

a constructor for building lists, a constructor for the empty list, constructors for building tuples of any arity and the atoms as constructors with arity 0.

The set of constructor terms is defined as the smallest set $T_C(S)$ such that:

$$S \subseteq T_C(S) \quad \text{and} \quad c/n \in C,\, t_1, \ldots, t_n \in T_C(S) \Longrightarrow c(t_1, \ldots, t_n) \in T_C(S)$$

The syntax of Core Erlang programs is defined as follows:

$$\begin{array}{lcl} p & ::= & f\texttt{(}X_1\texttt{,} \ldots \texttt{,}X_n\texttt{)} \texttt{ -> } e\texttt{.} \mid p\; p \\ e & ::= & \phi\texttt{(}e_1\texttt{,} \ldots \texttt{,}e_n\texttt{)} \mid X \mid \mathit{pat} \texttt{ = } e \mid \texttt{self} \mid e_1\texttt{,}e_2 \mid e_1\texttt{!}e_2 \mid \\ & & \texttt{case } e \texttt{ of } m \texttt{ end} \mid \texttt{receive } m \texttt{ end} \mid \texttt{spawn(}f\texttt{,}e\texttt{)} \\ m & ::= & p_1\texttt{->}e_1\texttt{;} \ldots \texttt{;}p_n\texttt{->}e_n \\ \mathit{pat} & ::= & c\texttt{(}p_1\texttt{,} \ldots \texttt{,}p_n\texttt{)} \mid X \end{array}$$

All defined functions of a program, extended with their arity, built the set $FS(p)$. ϕ/n is an abbreviation for $f/n \in FS(p)$, $F/n \in \Sigma$ and $c/n \in C$. In every Core Erlang program a main function is defined: $\texttt{main}/0 \in FS(p)$.

We call the set of Core Erlang terms e $ET(\emptyset)$. The set $ET(S)$ is defined by adding the grammar rule $e ::= v \in S$ for Core Erlang terms.

Example 1. Let the Core Erlang program p_0 be:

```
main() -> DB = spawn(dataBase,[[]]), spawn(client,[DB]),
          client(DB).

dataBase(L) -> receive
    {allocate,Key,P} -> case lookup(Key,L) of
        fail -> P!free,
                receive
                  {value,V,P} -> dataBase(insert(Key,V,L))
                end;
        {succ,V} -> P!allocated, dataBase(L)
      end;
    {lookup,Key,P} -> P!lookup(Key,L), dataBase(L)
  end.

insert(K,V,L) -> case L of
      []           -> [{K,V}];
      [{K',V'}|L'] -> case K'<K of
            true  -> [{K',V'}|insert(K,V,L')];
            false -> [{K,V}|L]
         end
   end.
```

The program creates a database process holding a state in which the database information is stored. The database is represented by a list of tuples, each consisting of a key and a corresponding value. The interface of the database is given by the messages `{allocate,Key,P}` and `{lookup,Key,P}`. Allocation is done in two steps. First the key is received and checked. If there is no conflict, then the corresponding value can be received and stored in the database. This exchange of messages in more than one step has to guarantee mutual exclusion on the database, because otherwise it could be possible that two client processes send keys and values to the database and they are stored in the wrong combination. A client can be defined accordingly [10]. We will later prove that the database combined with two accessing clients fulfills this property.

3 Semantics of Core Erlang

Erlang is a strict functional programming language. It is extended with processes, that are concurrently executed. With $\texttt{spawn}(f, \texttt{[}a_1, \ldots, a_n\texttt{]})$ a new process can be created anywhere in the program. The process starts with the evaluation of $f(a_1, \ldots, a_n)$. If the second argument of `spawn` is not ground, it is evaluated before the new process is created. The functional result of `spawn` is the process identifier (*pid*) of the newly created process.

With $p!v$ arbitrary values (including pids) can be sent to other processes. The processes are addressed by their pids (p). A process can access its own pid with the Erlang function `self/0`. The messages sent to a process are stored in a mailbox and the process can access them conveniently with pattern matching in the `receive`-statement. Especially, it is possible to ignore some messages and fetch messages from further behind. For more details see [1].

In [10] we presented a formal semantics for Core Erlang. In the following we will refer to it as standard operational semantics (*SOS*). It is an interleaving semantics over a set of processes Π. Formally, a process consists of a pid ($\pi \in Pid := \{@n \mid n \in \mathbb{N}\}$), a Core Erlang evaluation term ($e \in ET(T_C(Pid))$) and a word over constructor terms, representing the mailbox ($q \in T_C(Pid)^*$). For the definition of the leftmost innermost evaluation strategy, we use the technique of evaluation contexts [7]:

$$\begin{aligned} E ::= {} & [\,] \mid \phi(v_1, \ldots, v_i, E, e_{i+2}, \ldots, e_n) \mid E, e \mid p = E \\ & \texttt{spawn}(f, E) \mid E\,!\,e \mid v\,!\,E \mid \texttt{case}\ E\ \texttt{of}\ m\ \texttt{end} \end{aligned}$$

Here v denotes an evaluated expression, E the subterm the redex is in and e and m the parts which cannot be evaluated. $[\,]$ is called the hole and marks the point for the next evaluation. We shall then write $E[e]$ for the context E with the hole replaced by e and the next step of the evaluation takes place here. Analogously to the Core Erlang Terms $ET(S)$ over a set S, we name the Core Erlang contexts $EC(S)$. The set S defines, the set of values: $v \in T_C(S)$. In the operational semantics defined in [10] we had ($S = T_C(Pid)$). For the abstraction presented in this paper S will also contain variables.

The semantics is a non-confluent transition system. The evaluations of the processes are interleaved. Only communication and process creation have side effects. For the modeling of these actions more than one process are involved. To give an impression of the semantics we present the rule for sending a value to another process

$$\frac{v_1 = \pi' \in Pid}{\Pi, (\pi, E[v_1\,!\,v_2], q)(\pi', e, q') \stackrel{!v_2}{\Longrightarrow} \Pi, (\pi, E[v_2], q)(\pi', e, q' : v_2)}$$

The value is added to the mailbox of the process π' and the functional result of the send action is the sent value.

4 Abstract Interpretation of Core Erlang Programs

In [10] we developed a framework for abstract interpretations of Core Erlang programs. The abstract operational semantics (*AOS*) yields a transition system which includes all paths of the SOS. In an abstract interpretation $\widehat{\mathcal{A}} = (\widehat{A}, \widehat{\iota}, \sqsubseteq, \alpha)$ for Core Erlang programs $\widehat{A}$ is the abstract domain, which should be finite for our application in model checking. The abstract interpretation function $\widehat{\iota}$ defines the semantics of predefined function symbols and constructors. Its codomain is $\widehat{A}$. Therefore it is for example not possible to interpret constructors freely in a finite domain abstraction. $\widehat{\iota}$ also defines the abstract behaviour of pattern matching in equations, `case`, and `receive`. Here the abstraction can yield additional non-determinism, because branches can get undecidable in the abstraction. Hence $\widehat{\iota}$ yields a set of results, which define possible successors. Furthermore, an abstract interpretation contains a partial order $\sqsubseteq$, describing which elements of $\widehat{A}$ are more precise than other ones. We do not need a complete partial order, because we do not compute any fixed point. We just evaluate the operational semantics with this abstract interpretation. An example for an abstraction of numbers with an ordering of the abstract representations is: $\mathbb{N} \sqsubseteq \{v \mid v \leq 10\} \sqsubseteq \{v \mid v \leq 5\}$. It is more precise to know, that a value is ≤ 5, than ≤ 10 than any number. The last component of $\widehat{\mathcal{A}}$ is the abstraction function. $\alpha : T_C(Pid) \longrightarrow \widehat{A}$ maps every real value to an abstract representation. Usually this is the most precise representation. Finally, the abstract interpretation has to fulfill five properties, which relate an abstract interpretation to the standard interpretation. They also guarantee that all paths of the SOS are represented in the AOS, for example in branching. An example for these properties is the following

(P1) For all $\phi/n \in \Sigma \cup C, v_1, \ldots, v_n \in T_C(Pid)$ and $\widetilde{v}_i \sqsubseteq \alpha(v_i)$ it holds that $\phi_{\widehat{\mathcal{A}}}(\widetilde{v}_1, \ldots, \widetilde{v}_n) \sqsubseteq \alpha(\phi_{\mathcal{A}}(v_1, \ldots, v_n))$.

It postulates, that evaluating a predefined function or a constructor on abstract values, which are representations of some concrete values yields abstractions of the evaluation of the same function on the concrete values. The other properties postulate correlating properties for matching and pattern matching in `case` and `receive`, and the pids represented by an abstract value. More details and some example abstractions can be found in [10, 11]. We do not define the AOS here again. In the next section we will define a modified version of this semantics, which is more useful for our aims.

5 Restrictions of Data Abstraction

Example 2. Consider the following Core Erlang program:

```
main() -> f(42).                    f(X) -> f(f(X)).
```

The smallest possible abstract domain is the one only containing the element ?, which represents all possible values. With this abstract domain the abstract semantics of the program contains the path:

$$(@1, \texttt{main()}, ()) \longrightarrow (@1, \texttt{f(42)}, ()) \longrightarrow (@1, \texttt{f(?)}, ()) \longrightarrow (@1, \texttt{f(f(?))}, ())$$
$$\longrightarrow \ldots \longrightarrow (@1, \texttt{f}^{\texttt{n}}\texttt{(?)}, ()) \longrightarrow (@1, \texttt{f}^{\texttt{n+1}}\texttt{(?)}, ()) \longrightarrow \ldots$$

which contains infinitely many different states. This abstract semantics is correct with respect to the operational semantics, in the sense, that all paths of the SOS are represented. But we cannot prove properties for this abstract semantics using simple model checking algorithms, because it has an infinite state space.

This example seems to be irrelevant in practice, but commonly used functions like the `append` or the `length` function for lists produce infinite transition systems for the abstract semantics over a finite domain as well. In [10] we defined the class of hierarchical programs, where recursive calls are only allowed in tail positions. For this class we obtain a finite abstract model. However, this restriction is too strong for programmers. A tail recursive version of a function, if it exists, can be very complicated and inefficient. This can also be seen in Example 1. The function `insert/2`, which inserts a new element into the list, with respect to an ordering on the keys, is also non-hierarchical. Hence the abstract domain of this program has an infinite state space for every abstract interpretation.

The source of the problem is the context-free structure of function calls. For special classes of context-free transition systems, it has been shown, that model checking is decidable [4, 3] and it seems that these theoretical results could be used here. But we do not have just one context-free transition system. We have several of them in multiple processes which can communicate with each other. Hence we can simulate several stacks which can exchange data. It is possible to simulate a Turing machine with the use of a finite domain abstraction containing only five values. In LTL it is possible to specify its termination. Therefore the verification of these systems is undecidable in general.

We need an abstraction of the context-free structure to a finite or a context-free model, which results from only one context-free process. The second possibility seems to be complicated for practice and it is not clear, from which process the context-free structure should be kept. Therefore we abstract a finite model. The abstraction must contain all paths of the context-free structure, because we want to prove properties of the program with model checking for linear time logic (LTL).

6 Graph Semantics

In the semantics of Core Erlang as it is defined in [10] we cannot detect which parts of an Erlang term belong to which function call. After a function definition is applied, the right hand-side vanishes in the context, in which it is called. We cannot detect where it ends. The call stack is not explicitly represented. To make these calls and returns more visible we move somewhat closer to the implementation. We split an Erlang term into a stack of Erlang contexts and a term which is actually evaluated. When a function is called, its context is stored on the stack and the corresponding right hand-side is the next term, which has to be evaluated. If the actual value is ground (it cannot be evaluated anymore), then the next context is popped from the stack and the value is put in the hole. The evaluation continues with this Erlang term. These stack representations of evaluation terms are defined by $SR(S) := ET(S) \times (FS(p) \times EC(S))^*$ where S are the possible values. The stack also contains the name of the function, which was called, when this context was pushed. This is superfluous in the graph representation, but we will later use this information for our abstraction.

This technique could be applied to the Erlang semantics. But in the semantics of Core Erlang all processes act interleaved and the critical calls and returns of a process cannot be identified and modified so easily. Here we only represent the behaviour of one process. This makes an analysis easier. We define a pre-compilation, which transforms a Core Erlang function into a transition system which describes the behaviour of a process starting with this function. The idea is that all actions are interpreted freely. The arcs in this transition system are labeled with the behaviour/actions the process may perform. The states are labeled with the Erlang terms, which have to be evaluated. The only difference to the SOS is that also variables may occur in the Core Erlang terms. These

$$
\begin{array}{ll}
1. & (E[a,e],W) \xrightarrow{\varepsilon} (E[e],W) \qquad\qquad 2.\ (E[a\,!\,b],W) \xrightarrow{a\,!\,b} (E[b],W) \\
3. & (E[\texttt{self}],W) \xrightarrow{Y\,=\,\texttt{self}} (E[Y],W) \quad \text{where } Y \notin \mathit{Vars}(E) \\
4. & (E[p\texttt{=}a],W) \xrightarrow{p\,=\,a} (E[a],W) \\
5. & (E[\texttt{receive } p_1\texttt{->}e_1\texttt{;}\dots\texttt{;}p_n\texttt{->}e_n \texttt{ end}],W) \xrightarrow{(i,\,?p_i)} (E[e_i],W) \quad \forall 1 \le i \le n \\
6. & (E[\texttt{case } a \texttt{ of } p_1\texttt{->}e_1\texttt{;}\dots\texttt{;}p_n\texttt{->}e_n \texttt{ end}],W) \xrightarrow{(i,\,p_i\,=\,a)} (E[e_i],W) \quad \forall 1 \le i \le n \\
7. & (E[\phi(a_1,\dots,a_n)],W) \xrightarrow{Y\,=\,\phi(a_1,\dots,a_n)} (E[Y],W) \quad \text{where } Y \notin \mathit{Vars}(E) \\
8. & (E[\texttt{spawn}(\texttt{f},a)],W) \xrightarrow{Y\,=\,\texttt{spawn}(f,a)} (E[Y],W) \quad \text{where } Y \notin \mathit{Vars}(E) \\
9. & (f(\overline{a}),W) \xrightarrow{\text{lc}:\,\overline{X}\,=\,\overline{a}} (e_f,W) \quad \text{where } f(\overline{X})\texttt{->}e_f\texttt{.} \in p \\
10. & (E[f(\overline{a})],W) \xrightarrow{\text{c}:\,\overline{X}\,=\,\overline{a}} (e_f,(f,E)W) \quad \text{where } f(\overline{X})\texttt{->}e_f\texttt{.} \in p \text{ and } E \neq [\,] \\
11. & (a,(f,E)W) \xrightarrow{\text{r}:\,Y\,=\,a} (E[Y],W) \quad \text{where } a \in T_C(Vars) \text{ and } Y \notin \mathit{Vars}(E)
\end{array}
$$

Figure 1: The graph representation of Core Erlang with a stack

variables will later be instantiated with values. Hence we can handle variables in our free interpretation as values too. The position, where the next evaluation takes place is independent of the concrete variable bindings. The result is the relation $\longrightarrow \subseteq SR(T_C(\mathit{Var})) \times \mathit{Act} \times SR(T_C(\mathit{Var}))$ defined in Figure 1. The set of all actions Act should be clear from the figure. The first eight rules just perform the free interpretation of the actions. In the rules for `receive` and `case` we have to consider branching. The correct order of the patterns is important. Therefore we number the patterns in the corresponding arcs and preserve their order. If the result of an action has to be used in subsequent states, then we introduce a new variable Y. The result of the action is bound to Y and the redex is replaced by Y. The call of a function yields a new stack frame for the context, in which the function is called (10). In the SOS we also have to push the variable bindings to a runtime stack at this point and proceed with the binding of the parameters of the called function f. This is retained by the transition label c: $\overline{X} = \overline{a}$ [1]. If a function is called in an empty context, we use tail recursion optimization (9).

If we have no evaluation context anymore, in other words, the Core Erlang term is a constructor term over variables, then we have to return to the last context (11). We cannot simply, copy the value a into the hole, because a could contain variables, which also occur in E. In the SOS these variables are usually bound to different values. Hence we introduce a new variable Y, which does not occur in E and bind this variable to the result of the evaluation, which is a. on top of this graph representation an 8abstract) semantics can easily be defined. For Core Erlang programs which use recursion only in tail positions, the graph representation is a finite transition system.

[1] We write $\overline{X}$ as an abbreviation for $X_1,\dots,X_n$, $\overline{a}$ for $\texttt{[}a_1,\dots,a_n\texttt{]}$ and c: $\overline{X} = \overline{a}$ for c: $X_1 = a_1,\dots,X_n = a_n$. n will be clear from the context.

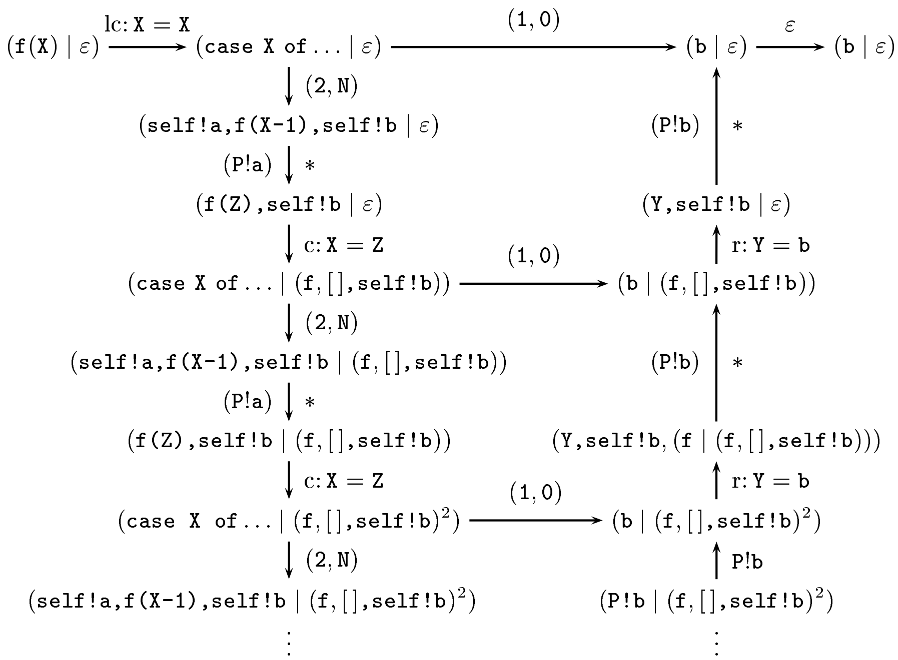

Figure 2: Graph representation of Example 3

We will use this graph representation for our abstraction, but we can also use it for a more efficient implementation of abstraction and model checking. In the first implementation we used Core Erlang evaluation terms to identify the states. Constructing the abstract model, it is necessary to detect cycles. Therefore the states must be stored. For every new state in the transition system, its successors are computed and compared with the stored states. Only for new states further successors must be computed. But the storage of states needs much space and the comparison of states needs much time. Therefore a compact representation of a state is desirable.

The graph representation is a transition system, where the transitions represent the behaviour of a process. The labels of the states have only been used for its construction, but they are superfluous after that.E.g. we can replace them by numbers. Then we construct the interleaving transition system with these numbers as names of the states a process is in. This is a much more compact representation of a state and allows a faster verification of even larger systems. Furthermore we do not have to descend the evaluation context during the generation of the model. The successors of a state can be evaluated more efficiently.

But for non-hierarchical Core Erlang programs this graph representation is infinite:

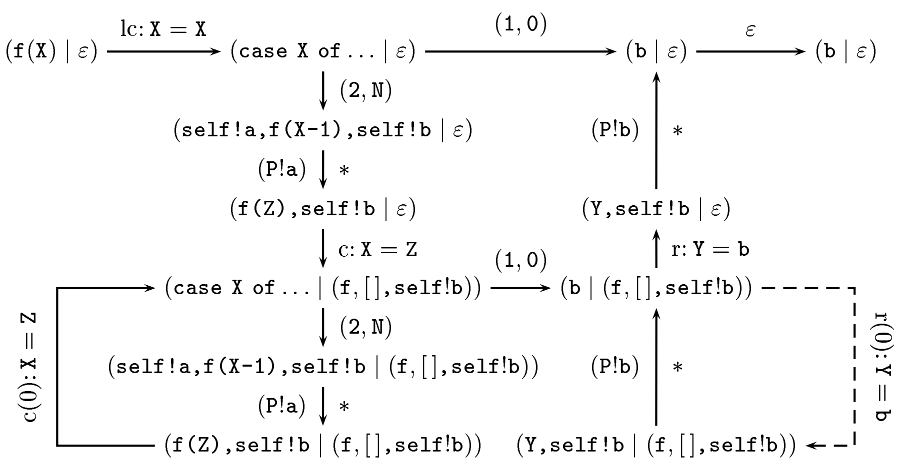

Figure 3: Abstract graph representation of Example 3

Example 3. Consider the following function definition:

```
f(X) -> case X of
          0 -> b;
          N -> self!a, f(X-1), self!b
        end.
```

A process executing this function sends `X` times the atom `a` to itself and after that `X` times `b`. The resulting graph representation is sketched in Figure 2. For a better distinction of the commas in the Core Erlang terms and the stacks, we have used | to separate the evaluation term from the stack of contexts.

7 Abstracting the Context-Free Structure

As discussed in Section 4 we need abstraction of the context-free structure of Core Erlang programs. We use the same basic idea as for the abstract interpretation. We construct an abstract graph representation of the program, which is finite state. The construction guarantees, that its semantics is safe with respect to the SOS.

Our approach is a kind of call-string approach [16] on program level. The main idea of the abstraction is to replace the calls and the returns by jumps. For Example 3 a good abstraction is, that first n times an `a` is sent and after that m times a `b`. A property like "no `a` is sent after a `b`" could then be proven automatically.

The idea of the abstraction is to replace the calls of `f` (see Figure 2) by jumps to a predecessor node, where `f` was already called. Hence we replace the second non-tail call by the following arc:

$$(\texttt{f(Z),self!b} \mid (\texttt{f}, [\,], \texttt{self!b})) \xrightarrow{\text{c: X=Z}} (\texttt{case X} \ldots \mid (\texttt{f}, [\,], \texttt{self!b}))$$

The states underneath $(\texttt{f(Z),self!b} \mid (\texttt{f}, [\,], \texttt{self!b}))$ in Figure 2 must not be considered.

But how can we perform the corresponding return step? We know the stack of the state we jumped to instead of calling `f`. Hence the evaluation of this call will be terminated, if the Core Erlang term is evaluated to a value with the same stack as the one we jumped to instead of the call. These are all states of the form $(a \mid (\texttt{f}, [\,], \texttt{self!b}))$ with $a \in T_C(\mathit{Var})$. In our example this is only the state $(\texttt{b} \mid (\texttt{f}, [\,], \texttt{self!b}))$. The destination of this returning jump is defined by the state where the call was initiated. The result of the call is `b`:

$$(\texttt{b} \mid (\texttt{f}, [\,], \texttt{self!b})) \xrightarrow{\text{r}:\,\texttt{Y=b}} (\texttt{Y,self!b} \mid (\texttt{f}, [\,], \texttt{self!b}))$$

We do not pop the top-level context, as usually in a return step. The context stack is not modified. The result is a finite graph representation, in which n times an `a` is sent and then m times a `b` The abstract graph representation of Example 3 in Figure 3. The added return jump is drawn with a dashed line.

When we generalize this technique, some problems appear. In general, we do not have only one function which calls itself recursively. We have multiple functions. Therefore we have extended the call stack with the names of the called functions. We can distinguish the different function calls. Thus we only jump back to states which correspond to the right hand-side of the function we are calling. Another feature of this extension is that we can detect, if a function was already called. If it was not called, then it does not appear in the stack. A sub-evaluation, which terminates and does not recursively call something outside itself, will not be abstracted. The abstract graph representation is similar to the non-abstract one. No calls are converted into jumps and no additional paths are added. Only if we detect recursion in a non-tail position, then we cut of the transition system and jump back.

Example 4. Another problem is exposed by a modified version of Example 3. We send the value of the variable `X` instead of the atom `b`:

```
f(X) -> case X of
          0 -> b;
          N -> self!a, f(X-1), self!X
        end.
```

First, the process sends n times an `a` to itself, and then it sends the numbers $1, \ldots, n$, where $n \in \mathbb{N}$ is the value, `f` is called with.

In the abstraction above we replace the communication by sending n times `a` and m times `b`. But what can we do here? In the abstract domain these values are represented by abstract values, which must not be an infinite set (especially in a finite domain abstraction). Jumping back instead of calling, we cannot know to which value `X` is bound. Hence we bind `X` to the value ?, which represents every value in the abstract domain $\widehat{A}$. We claim, that such a value exists in our abstract domain. Otherwise we can always add ? with $? \sqsubseteq v \ \forall v \in \widehat{A}$. Additionally, we annotate the label of the return arc with this substitution:

$$(\texttt{b} \mid (\texttt{f}, [\,], \texttt{self!X})) \xrightarrow{\text{r}:\,\texttt{Y=b}, [\texttt{X}/?]} (\texttt{Y,self!X} \mid (\texttt{f}, [\,], \texttt{self!X}))$$

In this abstract return jump we do not remove the top element of the call stack. It is even possible, that we have to add more entries, if the recursive call is indirect, that is, it calls some other functions in between. When we return from the function call we have to reconstruct the call stack to the old stack, because in the AOS these stored contexts still have to be executed. But with the variable bindings in these contexts we have the same problem, as with variables in the Core Erlang term, the evaluation returns with. The solution is to add bindings for the variables of these contexts to ?.

We also have to note these changes of the call stack in the label, because in the AOS we stack the substitutions in the same manner as in the graph representation. Hence we annotate the number of stack elements, which are removed instead of pushing a new block, in an abstract call. Analogously we note the number of stack elements, which have to be added in the abstract return jump and add the substitutions to ? for these frames. For corresponding calls and returns these numbers coincide. In our example it is zero, because no functions were called in between

$$(\texttt{f(Z),self!X} \mid (\texttt{f},[\,],\texttt{self!X})) \xrightarrow{\text{c}(0):\ \texttt{X=Z}} (\texttt{case X of}\ldots \mid (\texttt{f},[\,],\texttt{self!X}))$$

$$(\texttt{b} \mid (\texttt{f},[\,],\texttt{self!X})) \xrightarrow{\text{r}(0):\ \texttt{Y=b},[\texttt{X}/?],()} (\texttt{Y,self!X} \mid (\texttt{f},[\,],\texttt{self!X}))$$

So far we bind all variables to ? in an abstract return jump. This is safe, but not necessary. It is sufficient to bind only the bound variables to ?. The variables which will later be bound by pattern matching need not to be replaced. Since Erlang has no scoping, we do not know if a variable occurring in a subterm is free or bound. We need an analysis, which marks the variables which are already bound to values. This analysis can be combined with the construction of the abstract graph representation. Building the graph representation we can detect, when a variable is instantiated. We mark it with a tag ($'$). When we simulate the return of an abstracted call, we can bind all tagged variables to ?. The others can be left unchanged in the return jump:

```
f(X) -> case X of
          0 -> b;
          N -> self!a, f(X-1), B=b, self!B
        end.
```

In this example the variable `B` is not instantiated before the recursive call. We can leave it unchanged:

$$\begin{array}{l} (\texttt{b} \mid (\texttt{f},[\,],\texttt{B=b,self!B})) \\ \xrightarrow{\text{r}(0):\texttt{Y=b,[],()}} (\texttt{Y}',\texttt{B=b,self!B} \mid (\texttt{f},[\,],\texttt{B}=\texttt{b},\texttt{self!B})) \end{array}$$

8 Formal Description

In the last section we have motivated our abstraction with some examples. Now we present the formal definition and prove its correctness with respect to its use in formal verification with model checking.

First we define a function *tag*, which tags a set of variables.

$$\mathsf{tag}(V,X) = \begin{cases} X' & \text{, if } X \in V \\ X & \text{, otherwise} \end{cases}$$

It is canonically extended to Core Erlang terms and contexts. In the graph representation we tag the variables, which get bound in a label. For example

$$4'.\quad (E[p\text{=}a], W) \xrightarrow{p = a} (tag(\mathit{Vars}(p), E[a]), W)$$

This tagging is just an additional information and tagged variables are treated like un-tagged ones. In the transition labels we use only the names of the variables and ignore the tags.

Recursion is abstracted by jumps back to the last call of the same function. It is detected in the call stack, if the same function was already called. The destination state of this jump has a smaller call stack, than the call would yield. To relate call stacks in the graph representation with their abstract representation, we define an abstraction function α. This function yields the call stack, which grows by stepwise extension of the call stack and replacement of the occurring recursion by decreasing. This decreasing represents jumps back to previous calls.

$$\begin{aligned} \alpha(\varepsilon) &= \varepsilon \\ \alpha((f,E)W) &= \begin{cases} (f,E)\alpha(W), & \text{if } |\alpha(W)|_f = 0 \\ (f,E')V & \text{, if } \alpha(W) = U(f,E')V \\ & \quad \text{with } |U|_f = |V|_f = 0 \end{cases} \end{aligned}$$

From the definition it is not directly clear that α is total. With the following lemma, we see that always one of the two cases for $\alpha((f,E)W)$ matches. Hence α is defined for all call stacks.

Lemma 1. *$|\alpha(W)|_f \leq 1$ for all call stacks W and all functions $f \in FS(p)$.*

This abstraction function can now be used for the analysis of a given call stack, when calling a function. We can define the abstract graph representation directly with this abstraction function.

$$\Longrightarrow \subseteq SR(T_C(\mathit{Var})) \times \widehat{Act} \times SR(T_C(\mathit{Var}))$$

The actions $\widehat{Act}$ are the ones from Act plus the ones for abstract calls and returns. $\Longrightarrow$ is defined by the rules (1)-(9) and (11) of $\longrightarrow$. Instead of call stacks (10) we use their abstract representations:

$$\begin{gathered} (E[f(\overline{a})], \alpha(W)) \xRightarrow{\mathrm{c}(n)\colon \overline{X} = \overline{a}} (tag(\{\overline{X}\}, e_f), \alpha((f,E)W)) \\ \text{where } f(\overline{X})\texttt{->}e_f\texttt{.} \in p \text{ and } E \neq [\,] \text{ and } n = |\alpha(W)| - |\alpha((f,E')W)| \end{gathered}$$

If the function call is not abstracted by a jump we get $n = -1$. This means that we can add the actual context to the call stack, as we would do without abstraction. In this case we will just write c instead of c(-1). Otherwise we also

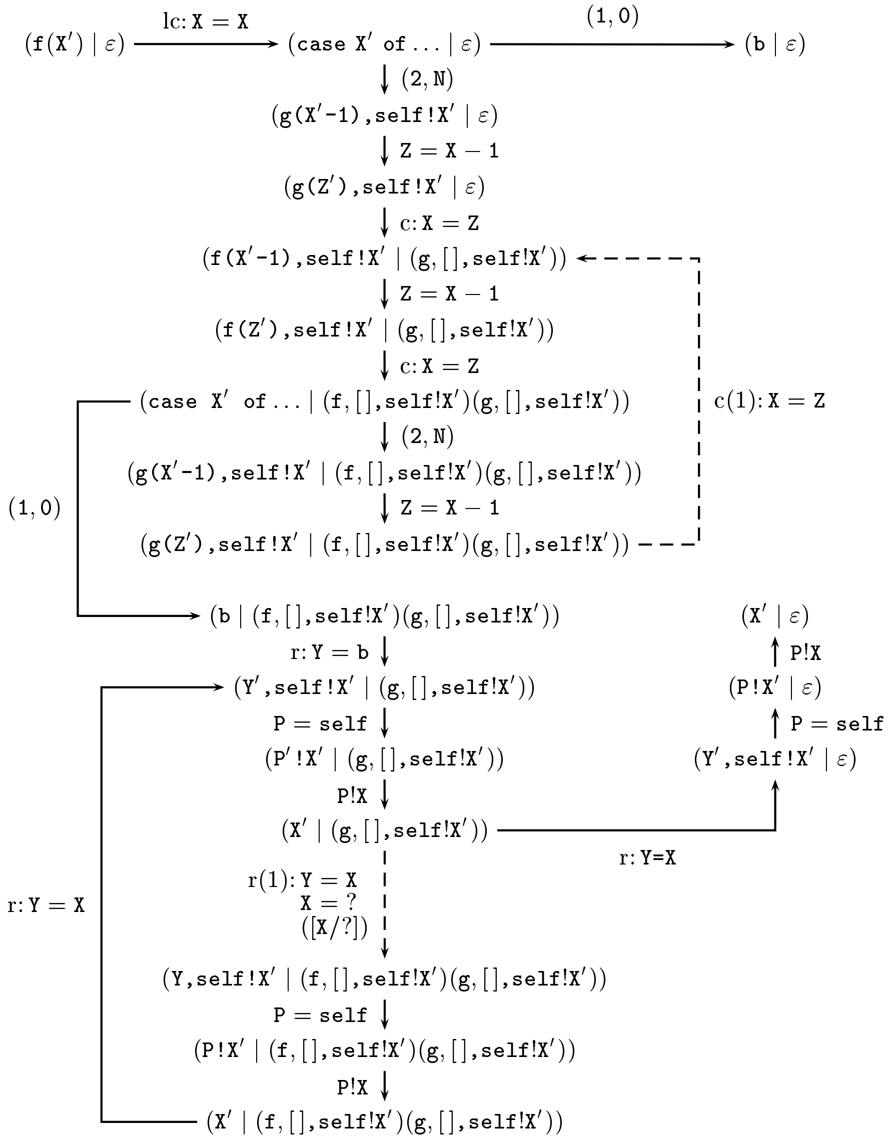

Figure 4: Abstract graph representation ($\Longrightarrow$) of Example 5

add a jump back. This means, we detect recursion and $|\alpha(W)|_f = 1$. For all $a \in T_C(Var)$:

$$(a, \alpha((f,E)W)) \xRightarrow[(\sigma_1,\ldots,\sigma_n)]{\substack{\mathrm{r}(n):\ Y = a \\ [\mathsf{tagged}(E)/?]}} (E[Y'], (W_1 \ldots W_k))$$

$$\text{where } W_{n+1} \ldots W_k = \alpha((f,E)W),\ W_1 \ldots W_{n+1} \ldots W_k = \alpha(W),$$
$$W_i = (f_i, E_i), \text{ and } \sigma_i = [\mathsf{tagged}(E_i)/?] \quad \forall 1 \le i \le n$$

Note that still $n = |\alpha(W)| - |\alpha((f,E')W)|$ and $n \geq 0$ always holds, if $|\alpha(W)|_f = 1$. In this case $W_1 \ldots W_n$ are the blocks which have to be restored in this return jump. The instantiated variables in these blocks and in E cannot be known. We have to instantiate them with ? in the evaluation. The function tagged yields all tagged variables. For these we can define substitutions, which instantiate them with ?. These are the substitutions $[\mathsf{tagged}(E)/?]$ and $(\sigma_1, \ldots, \sigma_n)$. We add them to the label.

The presented abstraction is safe with respect to the graph representation. For the limitations of space, we cannot present the details here. Instead, we present the abstract graph representation of a program using indirect recursion in Figure 4:

Example 5.

```
f(X) -> case X of
   0 -> b;
   N -> g(X-1), self!X
 end.
```

```
g(X) -> f(X-1), self!X.
```

9 Verification

We now return to Example firstProgram from the beginning of the paper. We want to prove that this database combined with two clients guarantees mutual exclusion for the writing access to the data. This means, when a process allocates a key no other process instantiates this key. This can be expressed with the following extended LTL formula:

$$\varphi = \bigwedge_{\substack{p \in Pid \\ p' \neq p}} G\ (\texttt{?\{allocate,\_,}p\texttt{\}} \rightarrow (\neg\texttt{?\{value,\_,}p'\texttt{\}}\ U(\texttt{?\{value,\_,}p\texttt{\}}) \vee \texttt{?allocated})$$

This formula can automatically be translated into a pure LTL formula, because we know that only three pids occur in the transition system. Hence we can replace the conjunction over pids by a conjunction of six instantiations of the formula, where p and p' are replaced by the possible pids.

Usually LTL is defined on state propositions. For understandability, we use the label of an arc to a state as its proposition here. In the implemented prototype we can add state propositions to the program, which makes it easier to express properties. For shortness we omit the details here.

To prove this property we use a simple abstraction in which the depth of constructor terms is restricted to two [11]. This guarantees us a finite transition system and the property can automatically be proven. Without the abstraction presented in this paper we could not prove this property for the program, because the function `insert` contains a non-tail recursive call. The transition system generated by any abstract interpretation is infinite. But with the presented abstraction of the context-free structure, we obtain a finite state transition system and can prove the formula automatically.

10 Conclusions

For the formal verification of concurrent and distributed systems, which are implemented in real programming languages, abstraction is needed. We have presented a technique, which allows us to abstract from the context-free structure of Erlang programs automatically. The result is a finite graph representation of the possible evaluations a process may perform. The graph includes all paths of the SOS. It can be used to verify properties of Erlang programs with model checking. The abstraction preserves enough structure to check interesting properties in practice. For tail recursion the abstraction does not even add any paths.

Non-tail recursive calls do not only occur in functional languages like Erlang. The use of recursion in imperative languages has the same problem. But the presented abstraction can be used here too.

Besides enabling the abstraction of the context-free structure, the graph semantics has also another important advantage for the implementation. It also yields a much more compact representation of the AOS, which allows us to verify larger systems with the same memory. We have implemented the abstraction of the context-free structure as a prototype and are able to prove properties like the one above with model checking.

Another approaches for the verification of Erlang programs is the Erlang Verification Tool [14], which uses theorem proving. For more convenience, the developers want to integrate model checking in their tool. At the moment they only consider pure model checking without any abstraction [2]. We think that for the verification of real systems abstractions is needed and the presented techniques should be considered for the integration of model checking.

For future work we plan to precise the presented abstraction. Here we instantiated all bound variables of an abstracted call with ?. But often a function is always called with the same arguments, e.g. fixed variables. Then we can be more precise and restore these values in the jump back from an abstracted call. We could prove more properties. This would also be a first step to allow higher order functions in our abstraction. In many higher order functions the argument functions are just reached through, without any modifications. But for practice first order is sufficient, because most Erlang programs do not contain higher order functions.

It would also be interesting to implement our approach as a translation to Promela, the specification language of SPIN [9], as it was done for Java/Ada with Java PathFinder [8] and the Bandera Tool [6]. But we first concentrated on the formal analysis to understand what happens in the abstraction of Core Erlang programs. A large problem in the translation to Promela will be the fact, that the languages Erlang (in contrast to Java) and Promela are completely different. Additionally, this is relevant for the generation of counter examples, which have to be retranslated to Erlang.

References

1. Joe Armstrong, Robert Virding, and Mike Williams. *Concurrent Programming in Erlang*. Prentice-Hall, Englewood Cliffs, NJ, 1993.
2. Thomas Arts and Clara Benac Earle. Development of a verified Erlang program for resource locking. In *Formal Methods in Industrial Critical Systems*, Paris, France, July 2001.
3. Olaf Burkart and Javier Esparza. More infinite results. *Bulletin of the European Association for Theoretical Computer Science*, 62:138–159, June 1997. Columns: Concurrency.
4. Olaf Burkart and Bernhard Steffen. Model checking for context-free processes. In W. R. Cleaveland, editor, *CONCUR '92: Third International Conference on Concurrency Theory*, volume 630 of *LNCS*, pages 123–137, Stony Brook, New York, 24–27August 1992. Springer.
5. Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages, Los Angeles*, pages 238–252, New York, NY, 1977. ACM.
6. Matthew B. Dwyer and Corina S. Pasareanu. Filter-based model checking of partial systems. In *Proceedings of the ACM SIGSOFT Sixth International Symposium on the Foundation of Software Engineering*, November 1998.
7. Matthias Felleisen, Daniel P. Friedman, Eugene E. Kohlbecker, and Bruce Duba. A syntactic theory of sequential control. *Theoretical Computer Science*, 52(3):205–237, 1987.
8. Klaus Havelund and Thomas Pressburger. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4), April 1998.
9. Gerard J. Holzmann. Proving properties of concurrent systems with SPIN. *LNCS*, 962:453–455, 1995.
10. Frank Huch. Verification of Erlang programs using abstract interpretation and model checking. *ACM SIGPLAN Notices*, 34(9):261–272, September 1999. Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '99).
11. Frank Huch. Verification of Erlang programs using abstract interpretation and model checking – extended version. Technical Report 99–02, RWTH Aachen, 1999.
12. Neil D. Jones and Flemming Nielson. Abstract interpretation: a semantics-based tool for program analysis. In *Handbook of Logic in Computer Science*. Oxford University Press, 1994. 527–629.
13. Orna Lichtenstein and Amir Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 97–107, New Orleans, Louisiana, January 13–16, 1985. ACM SIGACT-SIGPLAN, ACM Press.
14. Thomas Noll, Lars-åke Fredlund, and Dilian Gurov. The erlang verification tool. In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01)*, volume 2031 of *LNCS*, pages 582–585. Springer, 2001.
15. David Schmidt and Bernhard Steffen. Program analysis as model checking of abstract interpretations. *LNCS*, 1503:351–380, 1998.
16. Micha Sharir and Amir Pnueli. Two approaches to interprocedural data flow analysis. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications*, Prentice-Hall Software Series, pages 189–233. Prentice-Hall, Englewood Cliffs , NJ , USA, 1981.
17. Moshe Y. Vardi. *An Automata-Theoretic Approach to Linear Temporal Logic*, volume 1043 of *LNCS*, pages 238–266. Springer, New York, NY, USA, 1996.