Model Checking Erlang Programs – An Abstraction of the Context-Free Structure

Frank Huch
Lehrstuhl für Informatik II, RWTH Aachen,
52056 Aachen, Germany

huch@i2.informatik.rwth-aachen.de

No Institute Given

Abstract. We present an approach for the verification of Erlang programs using abstract interpretation and model checking. In general model checking for temporal logics and Erlang programs is undecidable. Therefore 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, 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 the context-free structure of functional programming languages. Even finite domain abstract interpretations yield infinite state systems. The reduction terms can grown infinitely in non-tail recursive calls. Here we define an abstraction of the context-free structure of Erlang programs, which replaces recursive calls in non-tail positions and the corresponding returns by jumps back to the last call of the same function and corresponding return jumps.
We have implemented this approach 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 can not 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. For the verification of concurrent systems model checking is a powerful technique: A concurrent system is specified in a specification formalism, like

CCS. The properties a system should have are specified in a formula, usually in temporal or modal logic. Under the assumption, that the operational semantics of the CCS specification yields a finite transition system the properties can be verified automatically. In the next steps the specification is more and more refined without changing the concurrent behaviour. The result is the program.

In practice, formal methods are not used in this strict way. First, the system is somewhat specified. Then people implement what they understand of the specification by hand. During the implementation the specifications are even changed again. Another problem is, that specification formalisms like UML or SDL are used, which do have too much expressive power and model checking gets undecidable. They usually do not even have a formalized operational semantics. Hence it is not clear how to build a Kripke structure, on which the formula can be checked.

We extend 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 [CC77,JN94,SS98]. In industry the programming language Erlang [AWV93] is used for the implementation of distributed systems. Erlang is a strict functional language with additional features for the creation of processes and communication between them, independently of a process being executed on the same computer or somewhere else in the Internet. We have developed a framework for abstract interpretations for a core fragment of Erlang in [Huc99a] 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 [LP85,Var96].

The defined abstraction does only yield a finite transition system for a subclass of Erlang programs. We have called this class hierarchical programs. 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 and system properties 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. In Section 5 we present a graph semantics, on which our abstraction is based. We scetch the idea of our abstraction in Section 6. Section 7 presents its use in model checking and finally we conclude and discuss future work in Section 8.

2 Syntax of Core Erlang

Let Σ be a set of predefined function symbols with arity. For example $\texttt{+}/2 \in \Sigma$. Let $Var = \{X, Y, Z, \dots\}$ be a set of variables and $Atom$ a set of atoms, e.g. $\{\texttt{1}, \texttt{2}, \texttt{fail}, \texttt{succ}, \dots\}$. Let CF be the set of Erlang constructor functions with arity: $CF = \{[\,.\,|\,.\,]/2, [\,]/0\} \cup \{\{\dots\}/n \mid n \in \mathbb{N}\} \cup \{a/0 \mid a \in Atom\}$. 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 over a set S is defined as

$$S \subseteq CT_S^{CF},$$
$$c/n \in CF \text{ and } t_1, \dots, t_n \in CT_S^{CF} \implies c(t_1, \dots, t_n) \in CT_S^{CF}$$

In the following we will just write CT_S, because CF is fixed.
The syntax of Core Erlang programs is defined as follows:

$$\begin{array}{lll} p & ::= & f\,(X_1\,,\dots,X_n)\ \texttt{->}\ e\,.\mid p\ p \\ e & ::= & \phi(e_1\,,\dots,e_n) \mid X \mid pat\ \texttt{=}\ e \mid \texttt{self} \mid e_1\,,e_2 \mid e_1\,!\,e_2 \mid \\ & & \texttt{case}\ e\ \texttt{of}\ m\ \texttt{end} \mid \texttt{receive}\ m\ \texttt{end} \mid \texttt{spawn}(f\,,e) \\ m & ::= & pat_1\texttt{->}e_1\,;\dots;pat_n\texttt{->}e_n \\ pat & ::= & c(pat_1\,,\dots,pat_n) \mid X \end{array}$$

All defined functions of a program build, extended with their arity, the set $FS(p)$. ϕ/n is an abbreviation for $f/n \in FS(p)$, $F/n \in \Sigma$ and $c/n \in CF$. 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, which holds 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. The code for the client and the function `lookup` can be found in [Huc99a]. We will later prove that the database combined with two accessing clients fulfils this property.

3 Semantics of Core Erlang

Erlang is a strict functional programming language. It is extended with processes, that are concurrently executed. With `spawn`$(f, [a_1, \ldots, a_n])$ 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`. 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 [AWV93].
In [Huc99a] 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. Formally, a process consists of a pid ($Pid := \{@n \mid n \in \mathbb{N}\}$), a Core Erlang evaluation term ($ET(CT_{Pid})$) and a word over constructor terms, representing the mailbox (CT^*_{Pid}). For the definition of the leftmost innermost evaluation strategy, we use the technique of evaluation contexts [FFKD87]:

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

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: $v \in CT_S$. In the operational semantics defined in [Huc99a] we had ($S = CT_{Pid}$). For the abstraction presented in this paper S will also contain variables.

To give an impression of the semantics we present the rules for function application

$$\frac{f(X_1,\ldots,X_n)\texttt{->}e\,.\in p \quad \text{and} \quad e' = \text{"}e \text{ with fresh free variables"}}{\Pi,(p,E[f(v_1,\ldots,v_m)],q) \implies \Pi,(p,E[e'[X_1/v_1,\ldots,X_m/v_m]],q')}$$

and sending of a value

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

4 Abstract Interpretation of Core Erlang Programs

In [Huc99a] we developed a framework for abstract interpretation 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 the 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`. In the last two cases the abstraction can yield additional non-determinism. Hence $\widehat{\iota}$ yields a set of results, which define possible successors. Furthermore, an abstract interpretation contains a partial order $\sqsubseteq$. This ordering defines 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. The abstract interpretation of pattern matching in `case`, `receive` and sending to values that represent more than one pid are sources of additional non-determinism. The values can be ordered arbitrarily, e.g. we could define an abstract interpretation, in which numbers have the following representations: $\mathbb{N} \sqsubseteq \{v \mid v \leq 10\} \sqsubseteq \{v \mid v \leq 5\}$. It is more precise to know, that a value is $\leq$5, than $\leq$10 than any number. The last component of $\widehat{\mathcal{A}}$ is the abstraction function. $\alpha : CT_{Pid} \longrightarrow \widehat{A}$ 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 aos to the sos. They guarantee that all path of the sos are represented in the aos, for example in branching. An example for these properties is the following

(P1) For all $\phi/n \in \Sigma \cup CF, v_1,\ldots,v_n \in CT_{Pid}$ 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 and some examples can be found in [Huc99a]. 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.

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 $\perp$, which represents all possible values. With this abstract domain the abstract semantics of the program contains the path:

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

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 model checking, 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. In [Huc99a] we defined the class of hierarchical programs, where recursive calls are only allowed in tail positions. For this class we obtain a finite abstract semantics. However, this restriction is too strong for programmers. A tail recursive version of a functionm , if it exists, can be very complicated and inefficient. This can also be seen in Examples 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. Model checking cannot be used to prove any properties of the program.
The source of the problem is the context-free structure of function calls. Hence we need an abstraction of this context-free structure to a regular one. 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).

5 Graph Semantics

In the semantics of Core Erlang as it is defined in [Huc99a] we cannot detect which parts of the 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. We do not have an explicite call stack. 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

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

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

is the set of 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 so easily be identified and modified. 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 variables will later be instantiated with values. Hence we can handle variables in our free interpretation as values too. The result is the relation $\longrightarrow \subseteq SR(CT_{Var}) \times Act \times SR(CT_{Var})$ defined in Figure 1. The set of all actions Act should be clear from Figure 1. 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 at this point we also have to push the variable bindings to a runtime stack 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). Consider the function

```
main() -> main().
```

In the semantics without stacks, we have only one state, the system can be in. This shall also be guaranteed in our graph representation with stacks. Therefore, if a function call has an empty context, then we do not extend the stack but just jump with a last call to the corresponding right hand-side. In the sos the old variable bindings are replaced by the new bindings of the parameters of the function to the arguments, that f is called with.
If we have no evaluation context anymore, in other words, the Core Erlang term is a constructor term over variables, 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.
The semantics over this graph representation can be defined as the aos, except that we replace the evaluation term by a state in the graph representation and a corresponding environment, representing the variable bindings. This environment consists of a substitution for the actual variable bindings $Subst : Var \longrightarrow \widehat{A}$ and a stack of substitutions $Subst^*$ for the frames on the call stack.

$$\leadsto_{\widehat{\mathcal{A}}} \subseteq State \times Label \times State, \text{ where}$$

$$\begin{aligned}
State &:= \mathcal{P}_{fin}(Proc),\\
Proc &:= Pid \times SR(CT_{Var}) \times Subst \times Subst^* \times Que\\
Que &:= \widehat{A}^*\\
Label &:= \{!\, v \mid v \in \widehat{A}\} \cup \{?v \mid v \in \widehat{A}\} \cup\\
&\qquad \{\texttt{spawn}(f) \mid f/n \in FS(p)\} \cup \{\varepsilon\}
\end{aligned}$$

$\leadsto_{\widehat{\mathcal{A}}}$ is defined in dependence of the labeling of $\longrightarrow$. Its definition is scetched in Figure 2. The rules just instantiate the actions with the abstract interpretation. In the case of branching we have to consider all successors of s. For their evaluation we use the function allSuccs : $SR(CT_{Var}) \longrightarrow \mathcal{P}(SR(CT_{Var}))$:

$$\text{allSuccs}(s) := \{t \mid s \xrightarrow{l} t\}$$

In the rules for function calls and returns, we also push respectively pop the actual substitution on, respectively from the stack. This stack has always the same size as the one we use in the graph representation. The stack assures the correct scoping of variables. Renaming is superfluous.
For Core Erlang programs which uses recursion only in tail positions, the graph representation is a finite transition system. In the aos we do not need the state labels of the graph representation. It only depends on the transition labels. Therefore we can use this graph representation for a more compact implementation of

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

$$\frac{\text{allSuccs}(s) = \{s_1, \ldots, s_m\} \text{ and } s \xrightarrow{(1,?p_1)} s_1 \ \ldots \ s \xrightarrow{(m,?p_m)} s_m \quad \text{and } (i,j,\rho) \in \text{queuematch}_{\widehat{\mathcal{A}}}((p_1,\ldots,p_m),(v_1,\ldots,v_u))}{\Pi,(\pi,s,\sigma,\Sigma,(v_1,\ldots,v_j,\ldots,v_u)) \overset{?v_j}{\rightsquigarrow}_{\widehat{\mathcal{A}}} \Pi,(\pi,s',\sigma \uplus \rho,\Sigma,(v_1,\ldots,v_{j-1},v_{j+1},\ldots,v_u))}$$

$$\frac{s \xrightarrow{a!b} s' \quad \text{and} \quad \pi' \in \text{pid}_{\widehat{\mathcal{A}}}(\sigma(a))}{\Pi,(\pi,s,\sigma,\Sigma,q)(\pi',t,\Sigma',q') \overset{!\sigma(b)}{\rightsquigarrow}_{\widehat{\mathcal{A}}} \Pi,(\pi,s',\sigma,\Sigma,q)(\pi',t,\Sigma',q':\sigma(b))}$$

$$\frac{s \xrightarrow{\text{c}:\overline{X}=\overline{a}} s'}{\Pi,(\pi,s,\sigma,\Sigma,q) \rightsquigarrow_{\widehat{\mathcal{A}}} \Pi,(\pi,s',[\overline{X}/\overline{a}],\sigma:\Sigma,q)}$$

$$\frac{s \xrightarrow{\text{lc}:\overline{X}=\overline{a}} s'}{\Pi,(\pi,s,\sigma,\Sigma,q) \rightsquigarrow_{\widehat{\mathcal{A}}} \Pi,(\pi,s',[\overline{X}/\overline{a}],\Sigma,q)}$$

$$\frac{s \xrightarrow{\text{r}:Y=a} s'}{\Pi,(\pi,s,\sigma,(\sigma':\Sigma),q) \rightsquigarrow_{\widehat{\mathcal{A}}} \Pi,(\pi,s',\sigma'[Y/a],\Sigma,q)}$$

Figure 2: Graph semantics

abstract interpretation and model checking. There we used Core Erlang evaluation terms to identify the states. This is especially necessary to detect cycles. In the graph representation we have a transition system, where the transitions represent the behaviour of a process. The labels of the states are not important anymore, see Figure 2. We can throw them away and replace them by numbers. This can be done as a pre-compilation for every function, a process can be started with (including `main`). Then we can construct the interleaving transition system with these numbers as names of the states a process is in. Therefore we can also replace the evaluation term in a process in the aos by a state-number, a substitution for the bound variables and a stack of context state-numbers and substitutions. This is a much smaller representation of a state and allows us to check larger systems with the same memory. For non-hierarchical programs the graph representation is infinite. Therefore this graph representation can also be use for the implementation This is exactly the class of hierarchical Core Erlang programs.

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`.

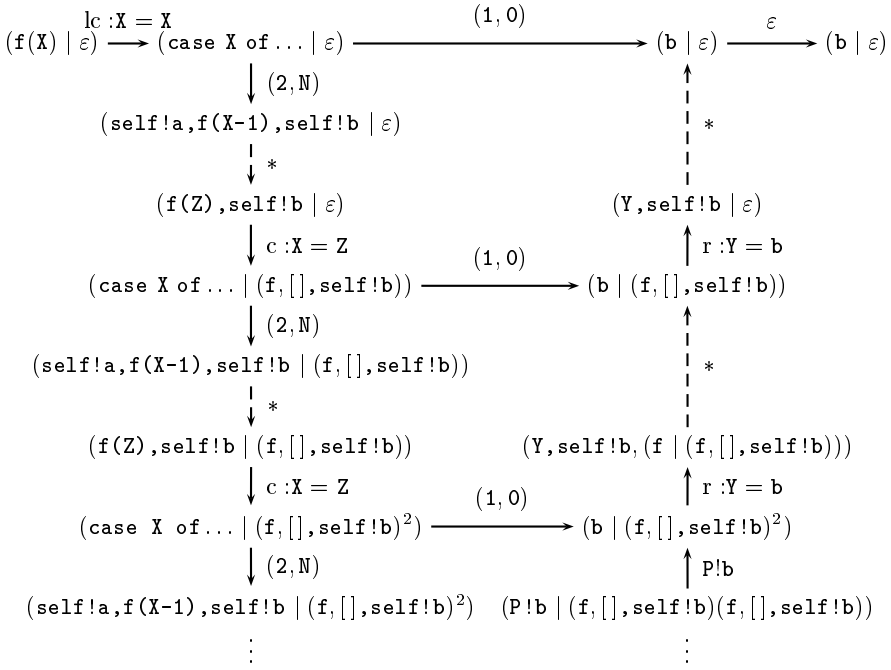

Figure 3: Graph representation of Example refnAsnBsExample

The resulting graph representation is sketched in Figure 3. 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. $(\texttt{f},[],\texttt{self!b})^2$ is an abreviation for $(\texttt{f},[],\texttt{self!b})(\texttt{f},[],\texttt{self!b})$.

6 Abstraction of the Context-Free Structure

The last example shows the graph representation of a non-hierarchical Core Erlang function. It is an infinite transition system. All possible paths of the evaluation are paths where n times `a` and afterwards n times `b` is send. From automata theory we know, that this behaviour cannot be modeled with a finite state system. We need a context-free one. For special classes of context-free transition systems it has been shown, that model checking is decidable [BS92] and it seems that these theoretical results could now be used in practice. 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 have multiple stacks which can exchange data and it would be possible to encode a Turing machine into this structure. Therefore these model checking techniques cannot be applied. We have to find another solution.

We use the same basic idea as for the abstract interpretation. We create a finite abstraction, which guarantees that all paths of the real evaluation are represented in the abstracted transition system.
The main idea of the abstraction is to replace the calls and the returns by jumps. For our example 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 3) 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 3 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. This are all states of the form $(a \mid (\texttt{f}, [], \texttt{self!b}))$ with $a \in CT_{Var}$. 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`:

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

We do not pop the top-level context, as usually in a return step. Here the context stack is unchanged.
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 we replace the communication by sending n times `a` and m times `b`. But what can we do here? In a finite domain abstraction not all values in $\mathbb{N}$ can be represented uniquely. Therefore it is not possible to send the values $1, \ldots, m$ for all $m \in \mathbb{N}$. Jumping back instead of calling, we cannot know to which value `X` is bound. Hence we bind `X` to the value $\bot$, 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 $\bot$ with $\bot \sqsubseteq v \; \forall v \in \widehat{A}$. We annotate the label of the return arc with this substitution:

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

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 $\bot$.
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 $\bot$ for these frames. For corresponding calls and returns these numbers coincide. In our example it is zero, because no functions were called in between

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

and

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

So far we bind all variables to $\bot$ in an abstract return jump. This is safe, but not necessary. It is sufficient to bind only the bound variables to $\bot$. The variables which will later be bound by pattern matching do not have 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 $\bot$. The others can be left unchanged in the return jump:

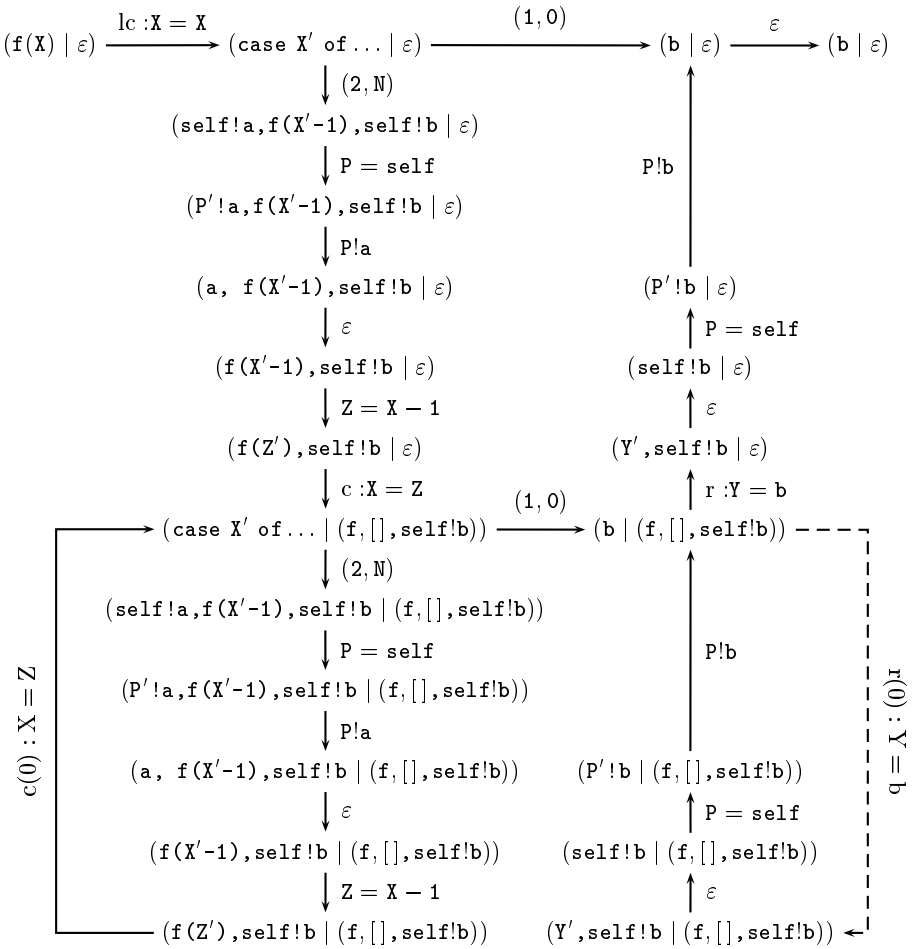

Figure 4: Abstract graph representation of Example 3

```
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:

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

The abstract graph representation of Example 3 with tagging is presented in Figure 4. We see that n times the atom `a` can be sent to the process itself and then m times a `b`. The added return jump is drawn with a dashed line.

7 Verification

In the last sections we defined an abstraction of the context-free structure of Core Erlang programs. The context-free structure is abstracted to a regular one. This abstraction combined with a finite domain abstract interpretation guarantees us a finite state abstraction for many concurrent Core Erlang programs. It only yields an infinite state space in the following two cases:

- an infinite number of processes is created
- the mailbox of a process grows infinitely

The first case is a standard problem for model checkers, which we do not try to solve with more restrictive abstractions here. Like in the concurrency work bench [CPS90], the verification process does not terminate in this case. The user has to know, that only a finite number of processes may be spawned. We have not defined an abstraction here, because we think that it would damage too much, to prove interesting properties.
The second cause of an infinite state space is related to the first one. Asynchronous communication can be simulated by synchronous communication and an arbitrary number of processes. If a mailbox of the system grows infinitely, this hints at problems in the design. These systems can work correct in practice. But using an unfair scheduler or executing the processes on different computers with different performances can result in an overflow of a mailbox. Hence a warning, if a mailbox grows above a given size, is an interesting hint for the programmer. She can add synchronization messages, to restrict the size of the mailboxes. Then she can use model checking for the verification. Again an abstraction seems to damage too much.
We now return to Example 1 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:

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

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 here use the label of an arc to a state as proposition of the state. 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 an easy abstraction in which the depth of constructor terms is restricted to two [Huc99b]. This guarantees us a finite transition system and the property can automatically be proven.
If we would only use the depth two abstract interpretation, 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.

8 Conclusions and Future Work

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 automatically abstract from the context-free structure of Erlang programs. 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. This abstraction preserves enough structure to check interesting properties in practice. For tail recursive calls the abstraction does not even add any paths.
Besides enabling the abstraction of the context-free structure, the graph semantics has also another important advantage for the implementation. We can represent the states in the aos much more compact. We can rename the states of the abstract graph representation with numbers. In the aos we can then use these numbers instead of the Core Erlang terms and contexts. This is a much smaller representation of a state and allows use to verify larger systems with the same memory.
For future work we plan to precise the presented abstraction. Here we instantiated all bound variables of an abstracted call with $\perp$. But sometimes a function is always called with same argument, e.g. the same variable. Then we can be more precise and restore this value in the jump back from a abstracted call. We could prove more properties. This would also be a first step to allow higher order functions in our abstraction. Erlang has no scoping. Hence argument functions are passed through the calls and a variable is bound to this function. But what can we bind this variable to after an abstracted return jump? $\perp$ is not possible, because the function could have side effects, like sending values. An analysis that tells us that the higher order function is always called with the same parameter could help. We do not loose the information, which function is passed through and can handle most applications of higher order functions. But for practice first order is sufficient, because most Erlang programs do not use higher order functions.
It would also be interesting to implement our approach as a translation to Promela, the specification language of SPIN [Hol95], as it was done for Java/Ada with Java PathFinder [HP99] and the Bandera Tool [DP98,HDL98]. A large

problem here will be the fact, that the languages Erlang and Promela are completely different, in contrast to Java and Promela Another problem is the generation of counter examples and their translation back to Erlang.

References

[AWV93] J. Armstrong, M. Williams, and R. Virding. *Concurrent Programming in Erlang*. Prentice-Hall, Englewood Cliffs, NJ, 1993.

[BS92] O. Burkart and B. Steffen. Model checking for context-free processes. *LNCS*, 630:123–??, 1992.

[CC77] P. Cousot and R. 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.

[CPS90] R. Cleaveland, J. Parrow, and B. Steffen. The concurrency workbench. In J. Sifakis, editor, *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, volume 407 of *LNCS*, pages 24–37, Berlin, June 1990. Springer.

[DP98] 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.

[FFKD87] M. Felleisen, D. P. Friedman, E. Kohlbecker, and B. Duba. A syntactic theory of sequential control. *Theoretical Computer Science*, 52(3):205–237, 1987.

[HDL98] J. Hatcliff, M. Dwyer, and S. Laubach. Staging static analyses using abstraction-based program specialization. *LNCS*, 1490:134–148, 1998.

[Hol95] G. J. Holzmann. Proving properties of concurrent systems with SPIN. *LNCS*, 962:453–455, 1995.

[HP99] K. Havelund and T. Pressburger. Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer – STTT*, to appear, 1999.

[Huc99a] 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).

[Huc99b] Frank Huch. Verification of Erlang programs using abstract interpretation and model checking – extended version. Technical Report 99–02, RWTH Aachen, 1999.

[JN94] 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.

[LP85] 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.

[SS98] D. Schmidt and B. Steffen. Program analysis as model checking of abstract interpretations. *LNCS*, 1503:351–380, 1998.

[Var96] M. Y. Vardi. An automata-theoretic approach to linear temporal logic. *LNCS*, 1043:238ff., 1996.