Electronic Notes in Theoretical Computer Science 64 (2002)
URL: `http://www.elsevier.nl/locate/entcs/volume64.html` 25 pages

Model Checking Erlang Programs – Abstracting Recursive Function Calls

Frank Huch [1]

Institute for Computer Science
Christian-Albrechts-University of Kiel
D-24098 Kiel, Germany

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 data abstraction for Erlang. An abstract operational semantics for this framework preserves all paths of the standard operational semantics. Hence, the abstraction is safe for all properties that have to hold on all paths of a system, like properties in LTL. 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 formalize 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 replaced 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.

1 Introduction

For the formal verification of concurrent and distributed systems we propose an extension of model checking to programs written in real programming languages. Direct approaches to this problem, have been described in [16,17]. However, the model checking problem in general is undecidable for system implementations using programming languages and properties described in expressive temporal or modal logics. The direct application of model checking is not possible. Already, simple calculations make the automatic verification undecidable, because the termination of these calculations can be relevant for the verification of the property. Hence, we need abstraction [6,14,19].

[1] Email: `fhu@informatik.uni-kiel.de`

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 [11] 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 [15,21].

The use of the logic LTL is only one possibility. It is also possible to use other logics which are defined over all paths of the underlying Kripke structure. Examples are fragments of the computational tree logics [5] in which properties can only be defined for all paths: ACTL and ACTL*. In our prototype we use the expressive logic LTL and use it as a representative for arbitrary logics with for all quantification.

The defined abstraction does only yield a finite transition system for a subclass of Erlang programs, called hierarchical programs [11]. 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 are non-tail-recursive function calls. In [13] we skeched an abstraction of non-tail-recursive function calls which we formalize in this paper. Non-tail-recursive calls are replaced by jumps to the associated positions in the program. The return is replaced by jumps to all possible continuations of calls of the function. 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 and sketch the operational semantics in Section 3 for this graph representation. 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 the presented abstraction is based. We formalize our abstraction in Section 7. Section 8 presents its use in model checking and finally we conclude and discuss future work in Section 9.

2 Syntax of Core Erlang

Let Σ be a signature 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. $\{\texttt{1}, \texttt{2}, \texttt{fail}, \texttt{succ}, \ldots\}$. Let $\mathcal{C}$ be the signature of Erlang constructor functions: $\mathcal{C} = \{\texttt{[.|.]}/2, \texttt{[]}/0\} \cup \{\texttt{\{...\}}/n \mid n \in \mathbb{N}\} \cup \{a/0 \mid a \in \mathit{Atoms}\}$, 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 $T_{\mathcal{C}}(S)$ such that:

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

The syntax of Core Erlang programs is defined as follows:

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

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 $c/n \in \mathcal{C}$. In every Core Erlang program a main function is defined: `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 2.1 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 (`L`) 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 with the keys in th database, 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 [11]. 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 $p\texttt{!}v$ 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].

The main difference to other functional programming languages is the absence of scoping. Erlang uses bind-once variables: First, a variable can be bound in the call and spawning of a defined function. Second, it can be bound with pattern matching in *pat*=*e*, `case`, and `receive`.

If a bound variable is used in pattern matching, then this is no introduction of a new variable, but a matching against its actual binding. In Erlang programs this technique is commonly used to compare values at runtime. For example the variable `P` in the pattern `{value,V,P}` in the inner receive statement of Example 2.1 is already bound to the pid of a requesting client. This implicit test for the same pid of the client is used to identify messages from the same client and ignore `value`-messages from other clients.

In [11] 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 ($\pi \in \mathit{Pid} := \{@n \mid n \in \mathbb{N}\}$), a Core Erlang evaluation term ($e \in ET(T_\mathcal{C}(\mathit{Pid}))$) and a word over constructor terms, representing the mailbox ($\mu \in T_\mathcal{C}(\mathit{Pid})^*$). For the definition of the leftmost innermost evaluation strategy, we use the technique of evaluation contexts [8]:

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

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 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: $v \in T_{\mathcal{C}}(S)$. In the operational semantics defined in [11] we had $(S = T_{\mathcal{C}}(Pid))$. For the abstraction presented in this paper S will be $T_{\mathcal{C}}(Var)$.

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

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

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 [11] we developed a framework for data abstraction 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 $\leq$5, than $\leq$10 than any number. The last component of $\widehat{\mathcal{A}}$ is the abstraction function: $\alpha : T_{\mathcal{C}}(Pid) \longrightarrow \widehat{A}$ 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 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 $\phi/n \in \Sigma \cup C, v_1, \ldots, v_n \in T_{\mathcal{C}}(Pid)$ 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 [11,12]. 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 Limits of Data Abstraction

Example 5.1 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 finite domains as well. In [11] 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 2.1. 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 are non-tail recursive function calls which result in infinite transition systems with a context-free structure. 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.

A process which behaves like a stack can be programmed as follows:

```
stack(P) -> receive
              pop -> pop;
              X   -> stack(P),P!X,stack(P)
            end.
```

If this process receives the message pop, then it sends the last message stored in the stack to the process with the pid P. All other messages are pushed to the stack. With two processes behaving like this stack it is possible to simulate a Turing machine without the use of any data structures. Therefore, the same Turing machine can also be simulated with an abstract domain containing only five values (pop, a symbol on the tap, the blank, and the pids of the two processes). In LTL it is possible to specify that a special state is reachable. This state can also be the final state of the program and we can specify the termination of the simulated Turing machine. Therefore, the verification of these systems is undecidable in general, even for finite domain abstractions.

We need an abstraction of the non-tail recursive function calls to a finite or a context-free model which results from only one context-free process. The second possibility seems to be too complicated for practice, because 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 infinite model, 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 [11] 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

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. $(E[\texttt{receive}\ p_1\texttt{->}e_1\texttt{;}\ldots\texttt{;}p_n\texttt{->}e_n\ \texttt{end}],W) \xrightarrow{(i,?p_i)} (E[e_i],W)$ $\forall 1 \leq i \leq n$
6. $(E[\texttt{case}\ a\ \texttt{of}\ p_1\texttt{->}e_1\texttt{;}\ldots\texttt{;}p_n\texttt{->}e_n\ \texttt{end}],W) \xrightarrow{(i,p_i = a)} (E[e_i],W)$ $\forall 1 \leq i \leq n$
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. $(E[\texttt{spawn(f},a\texttt{)}],W) \xrightarrow{Y = \texttt{spawn(}f,a\texttt{)}} (E[Y],W)$ where $Y \notin \mathit{Vars}(E)$
9. $(f(\overline{a}),W) \xrightarrow{\mathrm{lc}:\overline{X} = \overline{a}} (e_f,W)$ where $f(\overline{X})\texttt{->}e_f\texttt{.} \in p$
10. $(E[f(\overline{a})],W) \xrightarrow{\mathrm{c}:\overline{X} = \overline{a}} (e_f,(f,E)W)$ where $f(\overline{X})\texttt{->}e_f\texttt{.} \in p$ and $E \neq [\,]$
11. $(a,(f,E)W) \xrightarrow{\mathrm{r}:Y = a} (E[Y],W)$ where $a \in T_{\mathcal{C}}(Vars)$ and $Y \notin \mathit{Vars}(E)$

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

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 position, where the next evaluation takes place is independent of the concrete variable bindings. The result is the relation $\longrightarrow \subseteq SR(T_{\mathcal{C}}(\mathit{Var})) \times \mathit{Act} \times SR(T_{\mathcal{C}}(\mathit{Var}))$ 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 bindings of the parameters of the called function f. This is retained by the transition label $\mathrm{c}:\overline{X} = \overline{a}$ [2]. If a function is called in an empty context, we use tail recursion optimization (9) to guarantee a finite graph representation for programs with only tail-recursion, like

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

$$\frac{s \xrightarrow{\varepsilon} s'}{\Pi, (\pi, s, \sigma, \Sigma, \mu) \rightsquigarrow_{\widehat{\mathcal{A}}} \Pi, (\pi, s', \sigma, \Sigma, \mu)}$$

$$\frac{s \xrightarrow{Y = \phi(a_1, \ldots, a_n)} s'}{\Pi, (\pi, s, \sigma, \Sigma, \mu) \rightsquigarrow_{\widehat{\mathcal{A}}} \Pi, (\pi, s', \sigma[Y/\phi_{\mathcal{A}}(\sigma(a_1), \ldots, \sigma(a_n))], \Sigma, \mu)}$$

$$\frac{s \xrightarrow{Y = \texttt{self}} s'}{\Pi, (\pi, s, \sigma, \Sigma, \mu) \rightsquigarrow_{\widehat{\mathcal{A}}} \Pi', (\pi, s', \sigma[Y/\pi], \Sigma, \mu)}$$

Figure 2: Graph Semantics – Sequential Evaluation

`main() -> main().`

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, here 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 (*GOS*) can be defined as the AOS respectively SOS 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 $\mathit{Subst} : \mathit{Var} \longrightarrow \widehat{A}$ and a stack of substitutions Subst^* for the frames on the call stack. The state space of the GOS is defined as

$$\begin{aligned}
&\widehat{\mathit{State}} := \mathcal{P}_{fin}(\widehat{\mathit{Proc}}),\\
&\widehat{\mathit{Proc}} := \mathit{Pid} \times SR(T_{\mathcal{C}}(\mathit{Var})) \times \mathit{Subst} \times \mathit{Subst}^* \times \widehat{\mathrm{M}b}\\
&\widehat{\mathrm{M}b} := \widehat{A}^*\\
&\widehat{\mathit{Label}} := \{\,!\widehat{v} \mid \widehat{v} \in \widehat{A}\} \cup \{?\widehat{v} \mid \widehat{v} \in \widehat{A}\} \ \cup \{\varepsilon\}
\end{aligned}$$

We define the GOS as $\rightsquigarrow_{\widehat{\mathcal{A}}} \subseteq \widehat{\mathit{State}} \times \widehat{\mathit{Label}} \times \widehat{\mathit{State}}$ in dependence of the labelings of $\longrightarrow$. It is defined in Figures 2 – 5. The rules just bind 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 $\mathsf{allSuccs} : SR(T_{\mathcal{C}}(\mathit{Var})) \longrightarrow \mathcal{P}_{fin}(SR(T_{\mathcal{C}}(\mathit{Var})))$:

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

$$\frac{s \xrightarrow{p = a} s' \text{ and } \text{match}_{\widehat{\mathcal{A}}}(p, \sigma(a)) = \rho}{\Pi, (\pi, s, \sigma, \Sigma, \mu) \rightsquigarrow_{\widehat{\mathcal{A}}} \Pi, (\pi, s', \sigma \uplus \rho, \Sigma, \mu)}$$

$$\frac{\begin{array}{c}\mathsf{allSuccs}(s) = \{s_1, \ldots, s_m\} \text{ and } s \xrightarrow{(1, p_1 = a_1)} s_1, \ldots, s \xrightarrow{(m, p_m = a_m)} s_m \\ \text{and } (i, \rho) \in \mathsf{casematch}_{\widehat{\mathcal{A}}}((p_1, \ldots, p_m), v)\end{array}}{\Pi, (\pi, s, \sigma, \Sigma, \mu) \rightsquigarrow_{\widehat{\mathcal{A}}} \Pi, (\pi, s_i, \sigma \uplus \rho, \Sigma, \mu)}$$

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

Figure 3: Graph Semantics – Matching

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

$$\frac{s \xrightarrow{Y = \texttt{spawn}(f, a)} s', \quad \mathsf{init}(f) = (s_f, (X_1, \ldots, X_n)) \text{ and } \sigma(a) = [v_1, \ldots, v_n]}{\begin{array}{ll} & \Pi, (\pi, s, \sigma, \Sigma, \mu) \\ \rightsquigarrow_{\widehat{\mathcal{A}}} & \Pi, (\pi, s', \sigma[Y/\pi'], \Sigma, \mu), (\pi', s_f, [X_1/v_1, \ldots X_n/v_n], \varepsilon, ())\end{array}}$$

Figure 4: Graph Semantics – Concurrent Evaluation

If a new process is spawned, then this process starts with the initial state of the graph representation of the spawned function. The function $\mathsf{init} : FS(p) \longrightarrow (SR(T_{\mathcal{C}}(Var)) \times Var^*)$ yields this state and also the variables which have to be bound in the function call:

$$\mathsf{init}(f) := ((e_f, \varepsilon), \overline{X}), \text{ if } f(\overline{X})\texttt{->}e_f\texttt{.} \in p$$

In the rules for function calls and returns (Figure 5) we push or pop the actual substitution to respectively from the stack. This stack has always the

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

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

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

Figure 5: Graph Semantics – Function Calls

same size as the one we use in the graph representation. Pushing unfinished evaluations to a stack also assures the correct scoping of variables. Renaming is superfluous. We omit the rules for runtime errors here. According to [11] they can easily be added.

The graph semantics and the AOS are equivalent for arbitrary Core Erlang programs. This is not surprising, because our translation corresponds to standard techniques in compiler construction. We use the stacks similarly to the implementation of procedure calls. Therefore, the equivalence should be intuitively clear. This equivalence is no major point of this paper. Already, the formalization of the equivalence is technically expensive. The formal proof of the equivalence relates to proofs for the correctness of compilers which are technically expensive. Therefore, we omit the formalization and the proof.

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, see Figures 2 – 5. 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

$$\begin{tikzcd}
(\mathtt{f(X)} \mid \varepsilon) \arrow[r, "\mathrm{lc}\colon \mathtt{X} = \mathtt{X}"] & (\mathtt{case\ X\ of} \ldots \mid \varepsilon) \arrow[r, "(1{,}0)"] \arrow[d, "(2{,}\mathtt{N})"] & (\mathtt{b} \mid \varepsilon) \arrow[r, "\varepsilon"] & (\mathtt{b} \mid \varepsilon) \\
& (\mathtt{self!a,f(X-1),self!b} \mid \varepsilon) \arrow[d, "(\mathtt{P!a})"', "*"] & & \\
& (\mathtt{f(Z),self!b} \mid \varepsilon) \arrow[d, "\mathrm{c}\colon \mathtt{X} = \mathtt{Z}"] & (\mathtt{Y,self!b} \mid \varepsilon) \arrow[uu, "(\mathtt{P!b})", "*"'] & \\
& (\mathtt{case\ X\ of} \ldots \mid (\mathtt{f,[\,],self!b})) \arrow[r, "(1{,}0)"] \arrow[d, "(2{,}\mathtt{N})"] & (\mathtt{b} \mid (\mathtt{f,[\,],self!b})) \arrow[u, "\mathrm{r}\colon \mathtt{Y} = \mathtt{b}"'] & \\
& (\mathtt{self!a,f(X-1),self!b} \mid (\mathtt{f,[\,],self!b})) \arrow[d, "(\mathtt{P!a})"', "*"] & & \\
& (\mathtt{f(Z),self!b} \mid (\mathtt{f,[\,],self!b})) \arrow[d, "\mathrm{c}\colon \mathtt{X} = \mathtt{Z}"] & (\mathtt{Y,self!b},(\mathtt{f} \mid (\mathtt{f,[\,],self!b}))) \arrow[uu, "(\mathtt{P!b})", "*"'] & \\
& (\mathtt{case\ X\ of} \ldots \mid (\mathtt{f,[\,],self!b})^2) \arrow[r, "(1{,}0)"] \arrow[d, "(2{,}\mathtt{N})"] & (\mathtt{b} \mid (\mathtt{f,[\,],self!b})^2) \arrow[u, "\mathrm{r}\colon \mathtt{Y} = \mathtt{b}"'] & \\
& (\mathtt{self!a,f(X-1),self!b} \mid (\mathtt{f,[\,],self!b})^2) & (\mathtt{P!b} \mid (\mathtt{f,[\,],self!b})^2) \arrow[u, "\mathtt{P!b}"'] & \\
& \vdots & \vdots &
\end{tikzcd}$$

Figure 6: Graph Representation of Example 6.1

a state can be evaluated more efficiently.

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

Example 6.1 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 6. 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. To keep the figure more compact only the interesting transitions are displayed. Transitions like the introduction of new variables are omitted.

$$\begin{tikzcd}
(\mathtt{f(X)} \mid \varepsilon) \arrow[r, "\mathrm{lc}: \mathtt{X} = \mathtt{X}"] & (\mathtt{case\ X\ of} \ldots \mid \varepsilon) \arrow[r, "\mathtt{(1,0)}"] \arrow[d, "\mathtt{(2,N)}"] & (\mathtt{b} \mid \varepsilon) \arrow[r, "\varepsilon"] & (\mathtt{b} \mid \varepsilon) \\
 & (\mathtt{self!a,f(X-1),self!b} \mid \varepsilon) \arrow[d, "*", "\mathtt{(P!a)}"'] & & \\
 & (\mathtt{f(Z),self!b} \mid \varepsilon) \arrow[d, "\mathrm{c}: \mathtt{X} = \mathtt{Z}"] & (\mathtt{Y,self!b} \mid \varepsilon) \arrow[uu, "*", "\mathtt{(P!b)}"'] & \\
 & (\mathtt{case\ X\ of} \ldots \mid (\mathtt{f},[\,],\mathtt{self!b})) \arrow[r, "\mathtt{(1,0)}"] \arrow[d, "\mathtt{(2,N)}"] & (\mathtt{b} \mid (\mathtt{f},[\,],\mathtt{self!b})) \arrow[u, "\mathrm{r}: \mathtt{Y} = \mathtt{b}"] \arrow[dd, dashed, bend left=60, "\mathrm{r}(0): \mathtt{Y} = \mathtt{b}"] & \\
 & (\mathtt{self!a,f(X-1),self!b} \mid (\mathtt{f},[\,],\mathtt{self!b})) \arrow[d, "*", "\mathtt{(P!a)}"'] & & \\
 & (\mathtt{f(Z),self!b} \mid (\mathtt{f},[\,],\mathtt{self!b})) \arrow[uu, bend left=60, "\mathrm{c}(0): \mathtt{X} = \mathtt{Z}"] & (\mathtt{Y,self!b} \mid (\mathtt{f},[\,],\mathtt{self!b})) \arrow[uu, "*", "\mathtt{(P!b)}"'] &
\end{tikzcd}$$

Figure 7: Abstract graph representation of Example 6.1

7 Abstraction of Non-tail Recursive Function Calls

The idea of the abstraction of non-tail recursive function calls is similar to the idea of data abstraction. We construct an abstract transition system which includes all paths of the GOS. Hence, the abstraction is safe with respect to properties expressed in LTL. Our approach is a kind of call-string approach [20] on program level. The abstraction is defined for the graph representation of Core Erlang programs and was informally described and motivated with multiple examples in [13]. The idea is to cut off infinite recursion in the graph representation. We can detect this recursion, because we keep track of the called functions in the graph representation. If a function is called for a second time, we do not unroll the recursion. Instead we jump back to the state in the graph representation, in which this function was already called before. See the backwards directed call transition in Figure 7.

Correspondingly, we add return jumps from states in which the execution of the abstracted function call terminates (these states have the same stack as the destination state of the abstract call and their body is reduced to a value or a variable) to states in which the abstracted function call is finished (see the dashed return transition in Figure 7). The graph representation induced by these states represent the actions of the context of the abstracted function call. If a function is recursively called from multiple points in the program, then this results in non-determinism in the possible return-jumps. However, this is necessary due to the safeness of our control-flow abstraction.

In this example we did not considered possible variable bindings in the context of the abstracted function call.

Example 7.1 Instead of `b`, we send the value of the variable `X`:

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

This 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 this example, we add a binding for the free variables of the context of the abstracted function call in the abstract return (in this case the variable `X`). Therefore, we use the least element ? of the abstract domain [3]:

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

In Example 6.1 the recursive call of the function `f` is direct. No functions are called in between. But in general also indirect recursion is possible. In this case the stack of called functions must be shortened in the abstracted call and reconstructed in the corresponding return jump. Therefore, we extend the call and return labels in the abstract graph representation with the number of stack elements which are removed respectively added in the GOS. For direct recursive function calls they are zero. In the reconstruction of the stack we additionally must reconstruct the bindings of the variables. We again use the least element of the abstract domain for these bindings, because the concrete bindings cannot be reconstructed. For a more detailed discussion of the idea of our flow-abstraction see [13].

To distinguish variables which are already bound to values and unbound variables we mark variables with a tag ($'$) when they are bound. This is necessary, because Erlang has no scoping but bind-once variables. We define a function *tag* which tags a set of variables.

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

This function is also canonically extended to Core Erlang terms and contexts. The graph representation of Core Erlang with a stack and tagging of bound variables is defined in Figure 8. Every variable which is bound to a value is tagged. 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. They are superfluous for the GOS.

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

[3] In our framework the abstract domain must not contain a least element, but it can always be added.

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. $(E[p\texttt{=}a],W) \xrightarrow{p\,=\,a} (\mathit{tag}(\mathit{Vars}(p),E[a]),W)$
5. $(E[\texttt{receive}\ p_1\texttt{->}e_1\texttt{;}\ \dots\texttt{;}p_n\texttt{->}e_n\ \texttt{end}],W) \xrightarrow{(i,?p_i)} (\mathit{tag}(\mathit{Vars}(p_i),E[e_i]),W)$
6. $(E[\texttt{case}\ a\ \texttt{of}\ p_1\texttt{->}e_1\texttt{;}\ \dots\texttt{;}p_n\texttt{->}e_n\ \texttt{end}],W)$
$\xrightarrow{(i,p_i\,=\,a)} (\mathit{tag}(\mathit{Vars}(p_i),E[e_i]),W)$ $\forall 1 \le i \le n$
7. $(E[\phi(a_1,\dots,a_n)],W) \xrightarrow{Y\,=\,\phi(a_1,\dots,a_n)} (E[Y'],W)$ 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. $(f(\overline{a}),W) \xrightarrow{\text{lc:}\,\overline{X}\,=\,\overline{a}} (\mathit{tag}(\{\overline{X}\},e_f),W)$ where $f(\overline{X})\texttt{->}e_f\texttt{.} \in p$
10. $(E[f(\overline{a})],W) \xrightarrow{\text{c:}\,\overline{X}\,=\,\overline{a}} (\mathit{tag}(\{\overline{X}\},e_f),(f,E)W)$ $f(\overline{X})\texttt{->}e_f\texttt{.} \in p \wedge E \neq [\,]$
11. $(a,(f,E)W) \xrightarrow{\text{r:}\,Y\,=\,a} (E[Y'],W)$ where $a \in T_{\mathcal{C}}(Vars)$ and $Y \notin \mathit{Vars}(E)$

Figure 8: The graph representation of Core Erlang with a stack and tagging of instantiated variables

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 is constructed by a stepwise execution of abstract calls. If the same function was already called, then the stack decreases.

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

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 a total function and defined for all call stacks.

Lemma 7.2 $|\alpha(W)|_f \le 1$ *for all call stacks W and all functions $f \in FS(p)$.*

Proof. A simple induction on W:

- $W = \varepsilon$. Trivial
- $W = (f,E)W'$. By induction hypothesis we know, that $|\alpha(W')|_f \le 1$. We distinguish two cases:

- $|\alpha(W')|_f = 0$. Hence, $\alpha(W) = (f,E)\alpha(W')$ and $|\alpha(W)|_f = 1$. And for all $g \neq f$ $|\alpha(W)|_g = |\alpha(W')|_g \leq 1$ by induction hypothesis.
- $\alpha(W') = U(f,E')V$ with $|U|_f = |V|_f = 0$. Then $\alpha(W) = (f,E')V$ and $|\alpha(W)|_f = 1$. And again $|\alpha(W)|_g = |\alpha(V)|_g \leq 1$ by induction hypothesis.

□

We use this abstraction function for the analysis of a given call stack, when calling a function. The abstract graph representation $\rightharpoondown \subseteq SR(T_{\mathcal{C}}(Var)) \times \widehat{Act} \times SR(T_{\mathcal{C}}(Var))$ can be defined with this abstraction function. The actions $\widehat{Act}$ contain Act and the actions for abstract calls and returns. $\rightharpoondown$ is defined by the rules (1)–(9) and (11) of $\longrightarrow$. Instead of call stacks (10) we use their abstract representations:

$$(E[f(\overline{a})], \alpha(W)) \xrightarrow{\mathrm{c}(n)\colon \overline{X} = \overline{a}} (\mathsf{tag}(\{\overline{X}\}, e_f), \alpha((f,E)W))$$

$$\text{where } f(\overline{X})\texttt{->}e_f\,. \in p \text{ and } E \neq [\,] \text{ and } n = |\alpha(W)| - |\alpha((f,E')W)|$$

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

$$(a, \alpha((f,E)W)) \xrightarrow{\begin{array}{l}\mathrm{r}(n)\colon\ Y = a\\ \quad [\mathsf{tagged}(E)/?]\\ \quad (\sigma_1,\ldots,\sigma_n)\end{array}} (E[Y'], (W_1 \ldots W_k))$$

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

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 bound variables in these blocks and in E cannot be known. We bind them with ?, the least element of the abstract domain, in the evaluation. The function tagged yields all tagged variables. For these we can define substitutions which bind them with ?. These are the substitutions $[\mathsf{tagged}(E)/?]$ and $(\sigma_1,\ldots,\sigma_n)$. We add them to the label.

$$\begin{tikzcd}
(\mathtt{f(X')} \mid \varepsilon) \arrow[r, "\mathrm{lc}\colon \mathtt{X}=\mathtt{X}"] & (\mathtt{case\ X'\ of}\ldots \mid \varepsilon) \arrow[r, "(\mathtt{1},\mathtt{0})"] \arrow[d, "(\mathtt{2},\mathtt{N})"] & (\mathtt{b} \mid \varepsilon) \\
& (\mathtt{g(X'\text{-}1),self!X'} \mid \varepsilon) \arrow[d, "\mathtt{Z}=\mathtt{X}-\mathtt{1}"] & \\
& (\mathtt{g(Z'),self!X'} \mid \varepsilon) \arrow[d, "\mathrm{c}\colon \mathtt{X}=\mathtt{Z}"] & \\
& (\mathtt{f(X'\text{-}1),self!X'} \mid (\mathrm{g},[\,],\mathtt{self!X'})) \arrow[d, "\mathtt{Z}=\mathtt{X}-\mathtt{1}"] & \\
& (\mathtt{f(Z'),self!X'} \mid (\mathrm{g},[\,],\mathtt{self!X'})) \arrow[d, "\mathrm{c}\colon \mathtt{X}=\mathtt{Z}"] & \\
& (\mathtt{case\ X'\ of}\ldots \mid (\mathtt{f},[\,],\mathtt{self!X'})(\mathrm{g},[\,],\mathtt{self!X'})) \arrow[d, "(\mathtt{2},\mathtt{N})"] \arrow[ddd, bend right=60, "(\mathtt{1},\mathtt{0})"'] & \\
& (\mathtt{g(X'\text{-}1),self!X'} \mid (\mathtt{f},[\,],\mathtt{self!X'})(\mathrm{g},[\,],\mathtt{self!X'})) \arrow[d, "\mathtt{Z}=\mathtt{X}-\mathtt{1}"] & \\
& (\mathtt{g(Z'),self!X'} \mid (\mathtt{f},[\,],\mathtt{self!X'})(\mathrm{g},[\,],\mathtt{self!X'})) \arrow[uuuu, dashed, bend right=60, "\mathrm{c}(1)\colon \mathtt{X}=\mathtt{Z}"'] & \\
& (\mathtt{b} \mid (\mathtt{f},[\,],\mathtt{self!X'})(\mathrm{g},[\,],\mathtt{self!X'})) \arrow[d, "\mathrm{r}\colon \mathtt{Y}=\mathtt{b}"'] & (\mathtt{X'} \mid \varepsilon) \\
& (\mathtt{Y',self!X'} \mid (\mathrm{g},[\,],\mathtt{self!X'})) \arrow[d, "\mathtt{P}=\mathtt{self}"'] & (\mathtt{P!X'} \mid \varepsilon) \arrow[u, "\mathtt{P!X}"'] \\
& (\mathtt{P'!X'} \mid (\mathrm{g},[\,],\mathtt{self!X'})) \arrow[d, "\mathtt{P!X}"'] & (\mathtt{Y',self!X'} \mid \varepsilon) \arrow[u, "\mathtt{P}=\mathtt{self}"'] \\
& (\mathtt{X'} \mid (\mathrm{g},[\,],\mathtt{self!X'})) \arrow[ur, "\mathrm{r}\colon \mathtt{Y{=}X}"'] \arrow[d, dashed, "\mathrm{r}(1)\colon \mathtt{Y}=\mathtt{X}\ \ \mathtt{X}=?\ \ ([\mathtt{X}/?])"'] & \\
& (\mathtt{Y,self!X'} \mid (\mathtt{f},[\,],\mathtt{self!X'})(\mathrm{g},[\,],\mathtt{self!X'})) \arrow[d, "\mathtt{P}=\mathtt{self}"'] & \\
& (\mathtt{P!X'} \mid (\mathtt{f},[\,],\mathtt{self!X'})(\mathrm{g},[\,],\mathtt{self!X'})) \arrow[d, "\mathtt{P!X}"'] & \\
& (\mathtt{X'} \mid (\mathtt{f},[\,],\mathtt{self!X'})(\mathrm{g},[\,],\mathtt{self!X'})) \arrow[uuuu, bend left=60, "\mathrm{r}\colon \mathtt{Y}=\mathtt{X}"] &
\end{tikzcd}$$

Figure 9: Abstract graph representation ($\Longrightarrow$) of Example 7.3

Example 7.3 As a more complex example, we consider the following program

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

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

The abstract graph representation of this program is presented in Figure 9. The arcs in this graph are only drawn with one line, but the displayed graph presents the relation $\rightharpoondown$. In the abstraction, parts of the call structure of the program are preserved. First, we have an even number of function calls[4] and then also an even number of times the value `X` is send. In the AOS over this abstract graph representation `X` will be bound to ?, with the exception of the first send operation.

Theorem 7.4 *(Safeness of the abstract graph representation)*

If $(e_1, W_1) \xrightarrow{l} (e_2, W_2)$ *then* $(e_1, \alpha(W_1)) \xrightharpoondown{\widehat{l}} (e_2, \alpha(W_2))$, *where* $\widehat{l}$ *has one of the following forms:*

if $l = \mathrm{r}\colon Y = a$ *then* $\widehat{l} = \mathrm{r}(n)\colon Y = a, \sigma, \overline{\sigma}$

or $\widehat{l} = \mathrm{r}\colon Y = a$

if $l = \mathrm{c}\colon \overline{X} = \overline{a}$ *then* $\widehat{l} = \mathrm{c}(n)\colon \overline{X} = \overline{a}$

otherwise $\widehat{l} = l$

Proof. We distinguish the cases 1. to 11. from the definition of $\longrightarrow$. The cases 1. to 9. are trivially fulfilled.

10. $(E[f(\overline{a})], W) \xrightarrow{\mathrm{c}\colon \overline{X} = \overline{a}} (e_f, (f, E) : W)$
and also
$$(E[f(\overline{a})], \alpha(W)) \xrightharpoondown{\mathrm{c}(n)\colon \overline{X} = \overline{a}} (e_f, \alpha((f, E)W)).$$
11. $(a, (f, E)W) \xrightarrow{\mathrm{r}\colon Y = a} (E[Y], W)$
We distinguish two cases:
   · $|\alpha(W)|_f = 0$
   Then $\alpha((f, E)W)) = (f, E)\alpha(W)$ and also
   $$(a, (f, E)W) \xrightharpoondown{\mathrm{r}\colon Y = a} (E[Y], \alpha(W)).$$
   · $|\alpha(W)|_f = 1$
   Then $\alpha((f, E)W)) = (f, E')V$ with $\alpha(W) = U(f, E')V$. Because W is a stack, we know that there must also be a state with
   $$(E[f(\overline{a})], W) \xrightarrow{\mathrm{c}\colon \overline{X} = \overline{a}} (e_f, (f, E) : W) \longrightarrow^m (a, (f, E)W)$$

[4] Apart from the initial tail recursive call.

With a simple induction over the length m of this derivation we can conclude, that also

$$(E[f(\overline{a})], \alpha(W)) \xrightarrow{\mathrm{c}(n)\colon \overline{X} = \overline{a}} (e_f, \alpha((f,E)W)) \rightharpoondown^m (a, \alpha((f,E)W))$$

and $(a, \alpha((f,E)W)) = (a, (f,E)V)$. Hence

$$(a, \alpha((f,E)W)) \xrightarrow{\mathrm{r}(n)\colon Y = a, \sigma, \overline{\sigma}} (E[Y], \alpha(W))$$

with σ and $\overline{\sigma}$ as defined above.

□

For a Core Erlang program p we define the complete abstract graph representation $\mathcal{AG}_p := (S, \rightharpoondown, \mathsf{init})$ as the transition system which is the restriction of $\rightharpoondown$ to the states which can be reached from the state $(\mathtt{main()}, \varepsilon)$ and the initial states of all functions which are spawned in p. $\mathsf{init} : FS(p) \longrightarrow S$ yields the initial state for the spawned functions and the variables which have to be bound in their calls. It was already defined in Section 6.

Lemma 7.5 *The abstract graph representation $\mathcal{AG}_p := (S, \rightharpoondown, \mathsf{init})$ is finite for every Core Erlang program p.*

Proof. This is easy to see, because the stacks can only grow to finite depth and also only finitely many terms and contexts may occur. They are restricted in size, because we only have a finite set of rules. Furthermore, only finitely many different functions can be spawned, because $FS(p)$ is finite. Therefore, S is a finite set of states and also $\rightharpoondown$ can be restricted to this set. □

It is also possible to define an algorithm which computes $\mathcal{AG}_p$. States are successively added using $\rightharpoondown$, until no more states can be added. Furthermore, the algorithm memorizes which abstract calls have been performed. This yields the stacks, for which abstract return jumps must be added. If the construction of $\mathcal{AG}_p$ reaches a state with an evaluated expression, then it adds return jumps to the corresponding memorized states.

The semantics over this abstract graph representation ($\hookrightarrow$) is defined analogously to the semantics over the graph representation ($\leadsto$). We only have to add the rules for the abstract calls and returns:

$$\frac{s \xrightarrow{\mathrm{c}(n)\colon \overline{X} = \overline{a}} s' \text{ and } n \geq 0}{\begin{array}{c}\Pi, (\pi, s, \sigma, (\sigma_1 : \ldots : \sigma_k), q) \\ \hookrightarrow_{\widehat{\mathcal{A}}} \Pi, (\pi, s', [\overline{X}/\overline{a}], (\sigma_{n+1} : \ldots : \sigma_k), q)\end{array}}$$

$$\frac{s \xrightarrow{\mathrm{r}(n)\colon Y = a, \sigma, \overline{\sigma}} s'}{\Pi, (\pi, s, \sigma', \Sigma, q) \hookrightarrow_{\widehat{\mathcal{A}}} \Pi, (\pi, s', \sigma[\overline{Y}/\overline{a}], (\overline{\sigma} : \Sigma), q)}$$

Theorem 7.6 *(Safeness of the Abstraction) Let p be a Core Erlang program, $\mathcal{AG}_p = (S, \rightharpoondown, s_0)$ the corresponding abstract graph representation, and $\widehat{\mathcal{A}} = (\widehat{A}, \widehat{\iota}, \sqsubseteq, \alpha)$ an abstract interpretation for Core Erlang programs with least element $\mathbf{?} \in \widehat{A}$. Then for all $s \overset{a}{\rightsquigarrow}_{\widehat{\mathcal{A}}} t$ and $\widehat{s} \preceq \alpha(s)$, there exists $\widehat{a} \sqsubseteq \alpha(a)$ and $\widehat{t} \sqsubseteq \alpha(t)$ such that $\widehat{s} \overset{\widehat{a}'}{\hookrightarrow}_{\widehat{\mathcal{A}}} \widehat{t}$.*

Proof. With Theorem 7.4 the proof is a straightforward case analysis on $\rightsquigarrow$. In the abstract call we lose some variable bindings, but in the corresponding return jump these variables are bound to ?. Therefore, a variable is either bound to the same value as in the full graph representation or it is bound to ?. The latter results from performing an abstracted call, in which the variable is abstracted. ? is the least element of $\widehat{A}$ and the theorem is fulfilled. □

For practical verification this abstraction is sufficient. However it is possible to precise the abstraction: instead of jumping back to the state where the same function was called for the first time we can allow $k \in \mathbb{N}$ calls of the same function in between and also accept an initial part with $n \in \mathbb{N}$ calls. We only have to modify the conditions in the definition of α:

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

The rest of the control flow abstraction can be left unchanged and we obtain more precise abstractions. The abstraction presented above is obtained by $k = 1$ and $n = 0$.

Non-tail recursive calls are in most cases only used for pure calculations without communication. In the verification we are more interested in communication parts which are usually programmed tail-recursively. The presented technique is necessary to obtain a finite model for the verification, but a high precision of the abstraction of non-tail recursive calls is not needed. More precision only results in a blow-up of the state space which makes model checking less efficient or even impossible because of memory restrictions.

8 Verification

We now return to Example 2.1 from the beginning of the paper. We want to prove that the database combined with two clients guarantees mutual exclusion for the writing access to the data. This means: if a process allocates a key, then 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 [12]. This guarantees 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 non-tail recursive calls, we obtain a finite state transition system and can prove the formula automatically. The abstract graph representation of the non-tail recursive function `insert` is presented in Figure 10.

9 Conclusions

For the formal verification of concurrent and distributed systems, which are implemented in real programming languages, abstraction is needed. We have presented an abstraction of non-tail recursive function calls 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. The abstraction preserves enough structure to check interesting properties in practice. For tail recursion and non-recursive function calls in non-tail positions 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 non-tail recursive calls, the graph semantics has another important advantage for the implementation. It also

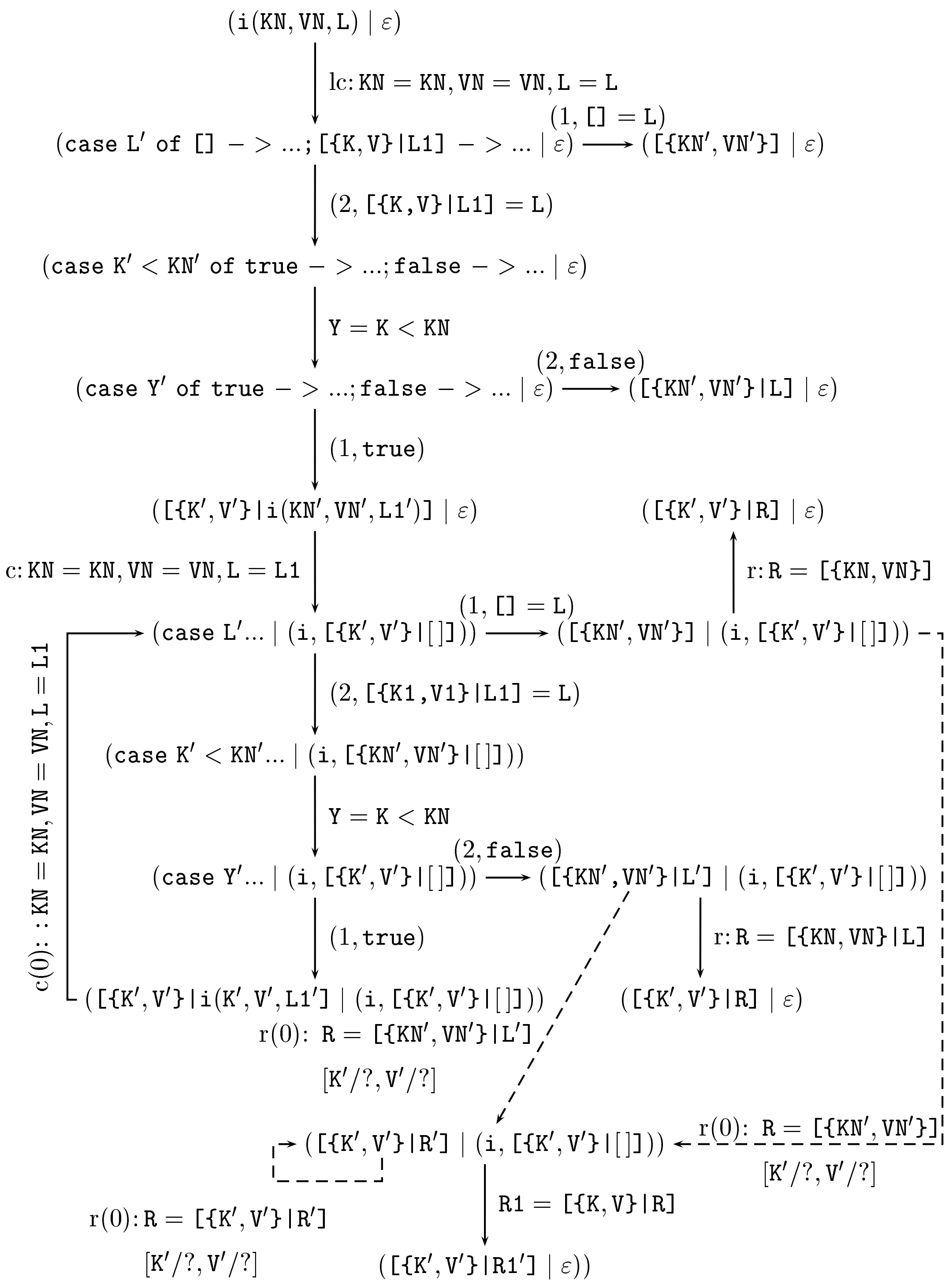
(i(KN, VN, L) | ε)
lc: KN = KN, VN = VN, L = L
(case L′ of [] − > ...; [{K, V}|L1] − > ... | ε)
(1, [] = L)
([{KN′, VN′}] | ε)
(2, [{K,V}|L1] = L)
(case K′ < KN′ of true − > ...; false − > ... | ε)
Y = K < KN
(case Y′ of true − > ...; false − > ... | ε)
(2, false)
([{KN′, VN′}|L] | ε)
(1, true)
([{K′, V′}|i(KN′, VN′, L1′)] | ε)
([{K′, V′}|R] | ε)
c: KN = KN, VN = VN, L = L1
r: R = [{KN, VN}]
(case L′... | (i, [{K′, V′}|[]]))
(1, [] = L)
([{KN′, VN′}] | (i, [{K′, V′}|[]]))
(2, [{K1,V1}|L1] = L)
(case K′ < KN′... | (i, [{KN′, VN′}|[]]))
Y = K < KN
(case Y′... | (i, [{K′, V′}|[]]))
(2, false)
([{KN′, VN′}|L′] | (i, [{K′, V′}|[]]))
(1, true)
r: R = [{KN, VN}|L]
([{K′, V′}|i(K′, V′, L1′] | (i, [{K′, V′}|[]]))
([{K′, V′}|R] | ε)
c(0): : KN = KN, VN = VN, L = L1
r(0): R = [{KN′, VN′}|L′]
[K′/?, V′/?]
([{K′, V′}|R′] | (i, [{K′, V′}|[]]))
r(0): R = [{KN′, VN′}]
[K′/?, V′/?]
R1 = [{K, V}|R]
r(0): R = [{K′, V′}|R′]
[K′/?, V′/?]
([{K′, V′}|R1′] | ε))

Figure 10: Abstract graph representation ($\rightharpoondown$) of the function `insert`

yields a much more compact representation of the AOS which allows us to verify larger systems with the same memory. We have implemented the presented abstraction as a prototype and are able to prove properties like the one above with model checking. The prototype also provides partial order reduction as an optimization of model checking, extensions for the detection of deadlocks and state proposition for a convenient specification of system properties in LTL. In this paper we focus on the control flow abstraction and omit details about these aspects of the prototype.

Another approach for the verification of Erlang programs is the Erlang Verification Tool [18] 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 [10], as it was done for Java/Ada with Java PathFinder [9] and the Bandera Tool [7]. 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. In addition, 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 *Lecture Notes in Computer Science*, pages 123–137, Stony Brook, New York, 24–27August 1992. Springer.

[5] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking.* The MIT Press, Cambridge, Massachusetts, 1999.

[6] 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.

[7] 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.

[8] 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.

[9] Klaus Havelund and Thomas Pressburger. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4), April 1998.

[10] Gerard J. Holzmann. Proving properties of concurrent systems with SPIN. *Lecture Notes in Computer Sience*, 962:453–455, 1995.

[11] 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).

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

[13] Frank Huch. Model checking Erlang programs - abstracting the context-free structure. In Scott D. Stoller and Willem Visser, editors, *Electronic Notes in Theoretical Computer Science*, volume 55. Elsevier Science Publishers, 2001.

[14] 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.

[15] 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.

[16] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification.* Springer, 1992.

[17] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems (Safety).* Springer, 1995.

[18] 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 *Lecture Notes in Computer Science*, pages 582–585. Springer, 2001.

[19] David Schmidt and Bernhard Steffen. Program analysis as model checking of abstract interpretations. *Lecture Notes in Computer Sience*, 1503:351–380, 1998.

[20] 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.

[21] Moshe Y. Vardi. *An Automata-Theoretic Approach to Linear Temporal Logic*, volume 1043 of *Lecture Notes in Computer Science*, pages 238–266. Springer, New York, NY, USA, 1996.