*Electronic Notes in Theoretical Computer Science 55 No. 3 (2001)*

*URL:* [*http://www.elsevier.nl/locate/entcs/volume55.html*](http://www.elsevier.nl/locate/entcs/volume55.html) *18 pages*

*Model Checking Erlang Programs { Abstracting the Context-Free Structure*

*Frank Huch 1 Lehrstuhl fur Informatik* *II*

*RWTH Aachen*

*D-52056 Aachen, Germany*

*Abstract*

*We present an approach for the veri cation of Erlang programs using abstract inter- pretation and model checking. In previous work we de ned a framework for abstract interpretations for Erlang. In this framework it is guaranteed, that the abstract op- erational semantics preserves all paths of the standard operational semantics. We consider properties that have to hold on all paths of a system, like properties in LTL. If these properties can be proven for the abstract operational semantics, then they also hold for the Erlang program. The proof can be automated with model checking if the abstract operational semantics is a nite transition system. But*

*niteness cannot be guaranteed because of non{tail recursive function calls. Even*

*for nite domain abstract interpretations we get in nite state systems and model checking is undecidable. In this paper we de ne an abstraction of the control{ ow. It replaces recursive calls in non-tail positions by jumps to the last call of the same function. The corresponding returns are replace by jumps to the possible return points.*

*We have implemented this approach as a prototype and are able to prove proper- ties 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 re- liability cannot be warranted. Things get even harder when we leave the se- quential territory and develop distributed systems. Here many processes run concurrently and interact via communication. This can e.g. yield problems

*1 Email:* [*huch@informatik.rwth-aachen.de*](mailto:huch@informatik.rwth-aachen.de)

*c 2001 Published by Elsevier Science B. V. Open access under* [*CC BY-NC-ND license.*](http://creativecommons.org/licenses/by-nc-nd/3.0/)

*like deadlocks or lifelocks. To guarantee the correctness of software formal* veri action is needed.

*We propose an extension of model checking to programs written in real* programming languages. However the model checking problem in general is undecidable for system implementations using programming languages and properties described in interesting logics. Hence we need abstraction [5,13,16]. In industry the programming language Erlang [1] is used for the implemen- tation of distributed systems. We have developed a framework for abstract interpretations for a core fragment of Erlang in [10] with the property that the transition system de ned 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 ful lled on all paths, like in linear time logic (LTL). If the abstraction ful lls a property expressed in LTL, then also the program ful lls it, but not vice versa. If the AOS is a nite transition system, then

*model checking is decidable [14,18].*

*The de ned abstraction does only yield a nite transition system for a* subclass of Erlang programs, called hierarchical programs [10]. Recursion is only allowed in tail positions. However, in practice many Erlang programs do not ful ll this restriction. For example, already the standard de nitions of append or length are not hierarchical. Hence programs which use such func- tions cannot be abstracted to nite state transition system with the presented technique of abstract interpretation. The cause is the context-free structure of functional programs. In this paper we de ne an abstraction of this context- free structure, to a regular one. We obtain a nite state transition system. Properties of the system can automatically be proven with LTL model check- ing.

*In Section 2 we de ne the syntax for a core fragment of Erlang. We sketch* the operational semantics in Section 3. The framework for the abstract inter- pretation is shortly introduced in Section 4 and its restrictions are presented in Section 5. In Section 6 we present a graph semantics, on which our ab- straction is based. We motivate the idea of our abstraction in Section 7 and formalize it in Section 8. Section 9 presents its use in model checking and

*nally we conclude and discuss future work in Section 10.*

# *2 Syntax of Core Erlang*

*Let be a set of prede ned function symbols with arity. For example +/2*

*2 . Let Var = fX; Y; Z;:: :g be a set of variables and Atoms a set of atoms,*

*e.g. f1; 2; fail; succ;:: :g. Let C be the set of Erlang constructor functions* with arity:

*C = f[:|:]=2; []=0g[ f{ ::: }=n j n 2 INg [ fa=0 j a 2 Atomsg; (1)*

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

*The set of constructor terms is de ned as the smallest set TC(S) such that:* S TC(S) and c=n 2 C, t1;::: ; tn 2 TC(S) =) c(t1;::: ; tn) 2 TC(S)

*The syntax of Core Erlang programs is de ned as follows:*

*p ::= f (X1, ::: ,Xn) -> e. j p p*

*e ::= (e1, ::: ,en) j X j pat = e j self j e1,e2 j e1!e2 j case e of m end j receive m end j spawn(f ,e)*

*m ::= p1->e1; ::: ;pn->en*

*pat ::= c(p1, ::: ,pn) j X*

*All de ned functions of a program, extended with their arity, built the set* FS(p). =n is an abbreviation for f =n 2 FS(p), F=n 2 and c=n 2 C. In every Core Erlang program a main function is de ned: main=0 2 FS(p).

*We call the set of Core Erlang terms e ET (;). The set ET (S) is de ned* by adding the grammar rule e ::= v 2 S for Core Erlang terms.

*Example 2.1 Let the Core Erlang program p0 be: main() -> DB = spawn(dataBase,[[]]),*

*spawn(client,[DB]),*

*client(DB).*

*dataBase(L) -> receive*

*{allocate,Key,P} -> case lookup(Key,L) of fail -> P!free,*

*receive*

*{value,V,P} -> dataBase(insert(Key,V,L)) end;*

*{succ,V} -> P!allocated, dataBase(L) end;*

*{lookup,Key,P} -> P!lookup(Key,L), dataBase(L) end.*

*insert(K,V,L) -> case L of*

*[] -> [{K,V}];*

*[{K',V'}|L'] -> case K'<K of*

*true -> [{K',V'}|insert(K,V,L')]; false -> [{K,V}|L]*

*end*

*end.*

*The program creates a database process holding a state in which the* database information is stored. The database is represented by a list of tu- ples, 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 con ict, 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 de ned accordingly [10]. We will later prove that the database combined with two accessing clients ful lls this property.

# *3 Semantics of Core Erlang*

*Erlang is a strict functional programming language. It is extended with pro-* cesses, that are concurrently executed. With spawn(f; [a1;::: ; an]) a new process can be created anywhere in the program. The process starts with the evaluation of f (a1;::: ; an). 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 identi er (pid) of the newly created process.

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

*In [10] we presented a formal semantics for Core Erlang. In the following* we will refer to it as standard operational semantics (SOS). It is an interleaving semantics over a set of processes . Formally, a process consists of a pid ( 2 Pid := f@n j n 2 INg), a Core Erlang evaluation term (e 2 ET (TC(P id))) and a word over constructor terms, representing the mailbox (q 2 T (P id) ). For the de nition of the leftmost innermost evaluation strategy, we use the technique of evaluation contexts [7]:

*C*

*E ::= [] j (v1, ::: ,vi,E,ei+2, ::: ,en) j E,e j p = E spawn(f ,E) j E!e j v!E j case E of m 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 de nes, the set of values: v 2 TC(S). In the operational semantics de ned in [10] we had (S = TC(P id)). For the abstraction presented in this paper S will also contain variables.

*The semantics is a non-con uent transition system. The evaluations of the* processes are interleaved. Only communication and process creation have side e ects. 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*

*v = 0 2 Pid*

*1*

*!v2*

*; ( ; E[v1!v2]; q)( 0; e; q0) =) ; ( ; E[v2]; q)( 0; e; q0 : v2)*

*The value is added to the mailbox of the process 0 and the functional result* of the send action is the sent value.

# *4 Abstract Interpretation of Core Erlang Programs*

*In [10] we developed a framework for abstract interpretations of Core Er-* lang programs. The abstract operational semantics (AOS) yields a transition system which includes all paths of the SOS. In an abstract interpretation A = (A; ; v; ) for Core Erlang programs A is the abstract domain, which

*b b b b*

*pretation function b de nes the semantics of prede ned function symbols and* should be nite for our application in model checking. The abstract inter-

*constructors. Its codomain is Ab. Therefore it is for example not possible*

*to interpret constructors freely in a nite domain abstraction. also de nes*

*b*

*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 yields a set of results, which de ne possible successors. Furthermore, an abstract interpretation contains a

*b*

*b*

*partial order v, describing which elements of A are more precise than other* ones. We do not need a complete partial order, because we do not compute any xed 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: IN v fv j v 10g v fv j v 5g. It is more precise to know, that a value is 5, than 10 than any number. The last component of A is the abstraction function: : TC(P id) ! A maps every real value to an abstract representation. Usually this is the most precise rep- resentation. Finally, the abstract interpretation has to ful ll ve 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

*b b*

*(P1) For all =n 2 [ C; v1;::: ; vn 2 TC(P id) and* vi v (vi) it holds that b(v1;::: ; vn) v ( A(v1;::: ; vn)).

*A*

*e e e*

*It postulates, that evaluating a prede ned function or a constructor on ab-* stract values, which are representations of some concrete values yields abstrac- tions of the evaluation of the same function on the concrete values. The other properties postulate correlating properties for matching and pattern matching in case and receive, and the pids represented by an abstract value. More details and some example abstractions can be found in [10,11]. We do not de-

*ne the AOS here again. In the next section we will de ne a modi ed 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; main(); ()) ! (@1; f(42); ()) ! (@1; f(?); ()) ! (@1; f(f(?)); ())*

*! ::: ! (@1; fn(?); ()) ! (@1; fn+1(?); ()) ! :::*

*which contains in nitely many di erent 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 in nite state space.

*This example seems to be irrelevant in practice, but commonly used func-* tions like the append or the length function for lists produce in nite transition systems for the abstract semantics over nite domains as well. In [10] we de-

*ned the class of hierarchical programs, where recursive calls are only allowed* in tail positions. For this class we obtain a nite 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 ineÆcient. 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 in nite state space for every abstract interpretation.

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

*We need an abstraction of the context-free structure to a nite or a context-* free model, which results from only one context-free process. The second pos- sibility seems to be complicated for practice and it is not clear, from which process the context-free structure should be kept. Therefore we abstract a

*nite model. The abstraction must contain all paths of the context-free struc-* ture, 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 de ned in [10] we cannot detect* which parts of an Erlang term belong to which function call. After a function de nition 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 de ned by SR(S) := ET (S) (FS(p) 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 super uous 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 se-* mantics of Core Erlang all processes act interleaved and the critical calls and returns of a process cannot be identi ed and modi ed so easily. Here we only represent the behaviour of one process. This makes an analysis easier. We de-

*ne a pre-compilation, which transforms a Core Erlang function into a transi-*

*tion system which describes the behaviour of a process starting with this func-* tion. The idea is that all actions are interpreted freely. The arcs in this transi- tion 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 di erence 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 ! SR(TC(Var)) Act SR(TC(Var)) de ned in Figure 1. The set of all actions Act should be clear from the gure. The rst eight rules just perform the free interpretation of the actions. In the rules for receive and case we have to consider branching. The correct order of the patterns is important. Therefore we number the patterns in the corresponding arcs and preserve their order. If the result of an action has to be used in subsequent states, then we introduce a new variable Y . The result of the action is bound to Y and the redex is replaced by Y . The call of a function yields a new stack frame for the context, in which the function is called (10). In the SOS we also have to push the variable bindings to a runtime stack at this point and proceed with the binding of the parameters of the called function f . This is retained by the transition label c: X = a *2 .*

*If a function is called in an empty context, we use tail recursion optimization* (9).

*2 We write X as an abbreviation for X ;:::;X , a for [a ;::: ;a ] and c: X = a for*

*c: X1 = a1;:::; Xn = an . n will be clear from the context.*

*1*

*n*

*1*

*n*

*" a!b*

*1. (E[a; e];W ) ! (E[e];W ) 2. (E[a!b];W ) ! (E[b];W )*

*3. (E[self*

*Y = self*

*];W ) ! (E[Y ];W )*

*p = a*

*where Y*

*2= Vars(E)*

*4. (E[p=a];W ) ! (E[a];W )*

*(i; ?pi)*

*5. (E[receive p1->e1; ::: ;pn->en end];W ) ! (E[ei];W ) 81 i n*

*(i; pi = a)*

*6. (E[case a of p1->e1; ::: ;pn->en end];W ) ! (E[ei];W ) 81 i n*

*Y = (a1;::: ; an)*

*7. (E[ (a1 ;::: ; an)];W ) ! (E[Y ];W ) where Y 2= Vars(E)*

*Y = spawn(f; a)*

*8. (E[spawn(f; a)];W ) ! (E[Y ];W ) where Y*

*lc: X = a*

*9. (f (a);W ) ! (ef ;W ) where f (X)->ef . 2 p*

*c: X = a*

*2= Vars(E)*

*10. (E[f (a)];W ) ! (ef ; (f; E)W ) where f (X)->ef . 2 p and E 6= [ ]*

*r: Y = a*

*11. (a; (f; E)W ) ! (E[Y ];W ) where a 2 TC(V ars) and Y 2= Vars(E)*

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

*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 di erent values. Hence we introduce a new variable Y , which does not occur in E and bind this variable to the result of the evaluation, which is a. on top of this graph representation an (abstract) semantics can easily be de ned. For Core Erlang programs which use recursion only in tail positions, the graph representation is a nite transition system.

*We will use this graph representation for our abstraction, but we can also* use it for a more eÆcient implementation of abstraction and model checking. In the rst 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 rep-* resent the behaviour of a process. The labels of the states have only been used for its construction, but they are super uous after that. E.g. we can replace them by numbers. Then we construct the interleaving transition system with these numbers as names of the states a process is in. This is a much more compact representation of a state and allows a faster veri cation of even larger systems. Furthermore we do not have to descend the evaluation context dur- ing the generation of the model. The successors of a state can be evaluated more eÆciently.

*lc: X = X*

*(f(X) j ") * *(case X of ::: j ")*

*(2; N)*

*(1; 0)*

*(b j ")*

*"*

 *(b j ")*

*(self!a,f(X-1),self!b j ") (P!b)*

*(P!a)*

*(f(Z),self!b j ") (Y,self!b j ")*

*c: X = Z*

*(1; 0)*

*r: Y = b*

*(case X of ::: j (f; [ ],self!b)) (b j (f; [ ],self!b))*

*(2; N)*

*(self!a,f(X-1),self!b j (f; [ ],self!b)) (P!b)*

*(P!a)*

*(f(Z),self!b j (f; [ ],self!b)) (Y,self!b; (f j (f; [ ],self!b)))*

*c: X = Z*

*(1; 0)*

*r: Y = b*

*(case X of ::: j (f; [ ],self!b)2) (b j (f; [ ],self!b)2)*

*(2; N)*

*P!b*

*(self!a,f(X-1),self!b j (f; [ ],self!b)2) (P!b j (f; [ ],self!b)2)*

*. .*

*Figure 2: Graph representation of Example 6.1*

*But for non-hierarchical Core Erlang programs this graph representation* is in nite:

*Example 6.1 Consider the following function de nition: f(X) -> case X of*

*0 -> b;*

*N -> self!a, f(X-1), self!b end.*

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

# *7 Abstracting the Context-Free Structure*

*As discussed in Section 4 we need abstraction of the context-free structure* of Core Erlang programs. We use the same basic idea as for the abstract

*(f(X) j ")*

*lc: X = X*

*(case X of ::: j ")*

*(2; N)*

*(1; 0)*

*(b j ")*

*"*

*(b j ")*

*(self!a,f(X-1),self!b j ") (P!b)*

*(P!a)*

*(f(Z),self!b j ") (Y,self!b j ")*

*c: X = Z*

*(1; 0)*

*r: Y = b*

*(case X of ::: j (f; [ ],self!b)) * *(b j (f; [ ],self!b))*

*(2; N)*

*r(0): Y = b*

*c(0): X = Z*

*(self!a,f(X-1),self!b j (f; [ ],self!b)) (P!b)*

*(P!a)*

*(f(Z),self!b j (f; [ ],self!b)) (Y,self!b j (f; [ ],self!b))*

*Figure 3: Abstract graph representation of Example 6.1*

*interpretation. We construct an abstract graph representation of the program,* which is nite state. The construction guarantees, that its semantics is safe with respect to the SOS.

*Our approach is a kind of call-string approach [17] on program level. The* main idea of the abstraction is to replace the calls and the returns by jumps. For Example 6.1 a good abstraction is, that rst n times an a is sent and after that m times a b. A property like "no a is sent after a b" could then be proven automatically.

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

*(f(Z),self!b j (f; [ ]; self!b)) c: X=Z*

*!*

*(case X ::: j (f; [ ]; self!b))*

*The states underneath (f(Z),self!b j (f; [ ]; self!b)) in Figure 2 must not* be considered.

*But how can we perform the corresponding return step? We know the* stack of the state we jumped to instead of calling f. Hence the evaluation of this call will be terminated, if the Core Erlang term is evaluated to a value with the same stack as the one we jumped to instead of the call. These are all states of the form (a j (f; [ ]; self!b)) with a 2 TC(Var). In our example this is only the state (b j (f; [ ]; self!b)). The destination of this returning jump

*is de ned by the state where the call was initiated. The result of the call is b:*

*(b j (f; [ ]; self!b)) r: Y=b*

*!*

*(Y,self!b j (f; [ ]; self!b))*

*We do not pop the top-level context, as usually in a return step. The context* stack is not modi ed. The result is a nite graph representation, in which n times an a is sent and then m times a b.

*The abstract graph representation of Example 6.1 is presented in Figure 3.*

*The added return jump is drawn with a dashed line.*

*When we generalize this technique, some problems appear. In general, we* do not have only one function which calls itself recursively. We have multiple functions. Therefore we have extended the call stack with the names of the called functions. We can distinguish the di erent 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 7.1 Another problem is exposed by a modi ed version of Exam-* ple 6.1. 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;::: ; n, where n 2 IN is the value, f is called with.

*In the abstraction above we replace the communication by sending n times* a and m times b. But what can we do here? In the abstract domain these values are represented by abstract values, which must not be an in nite set (especially in a nite domain abstraction). Jumping back instead of calling, we cannot know to which value X is bound. Hence we bind X to the value ?, which represents every value in the abstract domain A. We claim, that such a value exists in our abstract domain. Otherwise we can always add ? with

*b*

*b*

*? v v 8v 2 A. Additionally, we annotate the label of the return arc with this* substitution:

*r: Y=b; [X=?]*

*(b j (f; [ ]; self!X)) ! (Y,self!X j (f; [ ]; self!X))*

*In this abstract return jump we do not remove the top element of the call* stack. It is even possible, that we have to add more entries, if the recursive call is indirect, that is, it calls some other functions in between. When we return from the function call we have to reconstruct the call stack to the old

*stack, because in the AOS these stored contexts still have to be executed.* But with the variable bindings in these contexts we have the same problem, as with variables in the Core Erlang term, the evaluation returns with. The solution is to add bindings for the variables of these contexts to ?.

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

*c(0): X=Z*

*(f(Z),self!X j (f; [ ]; self!X)) ! (case X of ::: j (f; [ ]; self!X))*

*r(0): Y=b; [X=?]; ()*

*(b j (f; [ ]; self!X)) ! (Y,self!X j (f; [ ]; self!X))*

*So far we bind all variables to ? in an abstract return jump. This is* safe, but not necessary. It is suÆcient to bind only the bound variables to

*?. The variables which will later be bound by pattern matching need not* to be replaced. Since Erlang has no scoping, we do not know if a variable occurring in a subterm is free or bound. We need an analysis, which marks the variables which are already bound to values. This analysis can be combined with the construction of the abstract graph representation. Building the graph representation we can detect, when a variable is instantiated. We mark it with a tag ( 0 ). When we simulate the return of an abstracted call, we can bind all tagged variables to ?. The others can be left unchanged in the return jump:

*f(X) -> case X of*

*0 -> b;*

*N -> self!a, f(X-1), B=b, self!B end.*

*In this example the variable B is not instantiated before the recursive call.*

*We can leave it unchanged:*

*r(0): Y=b,[ ],()*

*(b j (f; [ ],B=b,self!B))*

*0*

*! (Y ,B=b,self!B j (f; [ ]; B = b; self!B))*

# *8 Formal Description*

*In the last section we have motivated our abstraction with some examples.* Now we present its formal de nition.

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

*8<X0 , if X 2 V*

*tag(V; X) =*

*:X , otherwise*

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

*40: (E[p=a];W ) p = a*

*!*

*(tag(Vars(p); E[a]);W )*

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

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

*(") = "*

*8*

*((f; E)W ) =*

*<(f; E0)V ; if (W ) = U (f; E0 )V*

*>: with jU jf = jV jf = 0*

*>(f; E) (W ); if j (W )jf = 0*

*From the de nition it is not directly clear that is total. But with the* following lemma, we see that always one of the two cases for ((f; E)W ) matches. Hence is de ned for all call stacks.

*Lemma 8.1 j (W )jf 1 for all call stacks W and all functions f 2 FS(p).*

*This abstraction function can now be used for the analysis of a given call* stack, when calling a function. We can de ne the abstract graph representa- tion directly with this abstraction function.

*=) SR(TC(Var)) dAct SR(TC(Var))*

*The actions Act are the ones from Act plus the ones for abstract calls and* returns. =) is de ned by the rules (1)-(9) and (11) of !. Instead of call stacks (10) we use their abstract representations:

*d*

*c(n): X = a*

*(E[f (a)]; (W )) ========) (tag(fXg; ef ); ((f; E)W ))*

*where f (X)->ef . 2 p and E 6= [ ] and n = j (W )j j ((f; E0)W )j*

*(f(X0) j ")*

*lc: X = X*

*(case X0 of ::: j ")*

 *(2; N)*

*(g(X0-1),self!X0 j ")*

 *Z = X 1*

*(g(Z0),self!X0 j ")*

*c: X = Z*

*(1; 0)*

*(b j ")*

*(1; 0)*

*(f(X0-1),self!X0 j (g; [ ],self!X0))*

 *Z = X 1*

*(f(Z0),self!X0 j (g; [ ],self!X0))*

 *c: X = Z*

*(case X0 of ::: j (f; [ ],self!X0)(g; [ ],self!X0))*

 *(2; N)*

*(g(X0-1),self!X0 j (f; [ ],self!X0)(g; [ ],self!X0))*

 *Z = X 1*

*(g(Z0),self!X0 j (f; [ ],self!X0)(g; [ ],self!X0))*

*c(1): X = Z*

*(b j (f; [ ],self!X0)(g; [ ],self!X0)) (X0 j ")*

*r: Y = b P!X*

*(Y0,self!X0 j (g; [ ],self!X0)) (P!X0 j ")*

*P = self P = self*

*(P0!X0 j (g; [ ],self!X0)) (Y0,self!X0 j ")*

*P!X *

*(X0 j (g; [ ],self!X0))*

*r: Y = X*

*r(1): Y = X*

*X = ? ([X=?])*

*(Y,self!X0 j (f; [ ],self!X0)(g; [ ],self!X0))*

*P = self *

*(P!X0 j (f; [ ],self!X0)(g; [ ],self!X0))*

*P!X *

*(X0 j (f; [ ],self!X0)(g; [ ],self!X0))*

*r: Y=X*

*Figure 4: Abstract graph representation (=)) of Example 8.2*

*If the function call is not abstracted by a jump we get n = 1. This means* that we can add the actual context to the call stack, as we would do without abstraction. In this case we will just write c instead of c(-1). Otherwise we also add a jump back. This means, we detect recursion and j (W )jf = 1. For all a 2 TC(Var):

*(a; ((f; E)W ))*

*r(n): Y = a*

*[tagged(E)=?]*

*( 1;::: ; n)*

*============) (E[Y 0]; (W ::: W ))*

*1 k*

*where Wn+1 ::: Wk = ((f; E)W ), W1 ::: Wn+1 ::: Wk = (W ),*

*Wi = (fi; Ei), and i = [tagged(Ei)=?] 81 i n*

*Note that still n = j (W )j j ((f; E0)W )j and n 0 always holds, if* j (W )jf = 1. In this case W1 ::: Wn are the blocks which have to be restored in this return jump. The instantiated variables in these blocks and in E cannot be known. We have to instantiate them with ? in the evaluation. The func- tion tagged yields all tagged variables. For these we can de ne substitutions, which instantiate them with ?. These are the substitutions [tagged(E)=?] and ( 1;::: ; n). We add them to the label.

*The presented abstraction is safe with respect to the graph representation* [12]. For limitations of space, we cannot present the details here. Instead, we present the abstract graph representation of a program using indirect recursion

*in Figure 4:*

*Example* *8.2*

*f(X) -> case X of g(X) -> f(X-1), self!X.*

*0 -> b;*

*N -> g(X-1), self!X end.*

# *9 Veri cation*

*We now return to Example 2.1 from the beginning of the paper. We want to* prove that this database combined with two clients guarantees mutual exclu- sion 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:

*' = ^*

*p2Pid*

*p0 6=p*

*G (?{allocate, ,p}*

*! (:?{value, ,p0} U (?{value, ,p}) \_ ?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 p0 are replaced by the possible pids.

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

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

# *10 Conclusions*

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

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

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

*Another approach for the veri cation of Erlang programs is the Erlang* Veri cation Tool [15], 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 veri cation 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. xed 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 rst step to allow higher order functions in our abstraction. In many higher or- der functions the argument functions are just reached through, without any modi cations. But for practice rst order is suÆcient, 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 speci cation language of SPIN [9], as it was done for Java/Ada with Java PathFinder [8] and the Bandera Tool [6]. But we rst 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 com- pletely di erent. Additionally, this is relevant for the generation of counter examples, which have to be retranslated to Erlang.

# *References*

*[1] Joe Armstrong, Robert Virding, and Mike Williams. Concurrent Programming in Erlang. Prentice-Hall, Englewood Cli s, NJ, 1993.*

*[2] Thomas Arts and Clara Benac Earle. Development of a veri ed Erlang program for resource locking. In Formal Methods in Industrial Critical Systems, Paris, France, July 2001.*

*[3] Olaf Burkart and Javier Esparza. More in nite results. Bulletin of the European Association for Theoretical Computer Science, 62:138{159, June 1997. Columns: Concurrency.*

*[4] Olaf Burkart and Bernhard Ste en. 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] Patrick Cousot and Radhia Cousot. Abstract interpretation: A uni ed lattice model for static analysis of programs by construction of approximation of*

*xed points. In Proceedings of the 4th ACM Symposium on Principles of Programming Languages, Los Angeles, pages 238{252, New York, NY,* *1977.*

*ACM.*

*[6] Matthew B. Dwyer and Corina S. Pasareanu. Filter-based model checking of partial systems. In Proceedings of the ACM SIGSOFT Sixth International Symposium on the Foundation of Software Engineering, November 1998.*

*[7] Matthias Felleisen, Daniel P. Friedman, Eugene E. Kohlbecker, and Bruce Duba. A syntactic theory of sequential control. Theoretical Computer Science, 52(3):205{237,* *1987.*

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

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

*[10] Frank Huch. Veri cation of Erlang programs using abstract interpretation and model checking. ACM SIGPLAN Notices, 34(9):261{272, September* *1999.*

*Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '99).*

*[11] Frank Huch. Veri cation of Erlang programs using abstract interpretation and model checking { extended version. Technical Report 99{02, RWTH Aachen, 1999.*

*[12] Frank Huch. Veri cation of Erlang programs using abstract interpretation and model checking. Technical report, RWTH Aachen, 2001. PhD Thesis, to be published.*

*[13] 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.*

*[14] Orna Lichtenstein and Amir Pnueli. Checking that nite state concurrent programs satisfy their linear speci cation. 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.*

*[15] Thomas Noll, Lars- ake Fredlund, and Dilian Gurov. The erlang veri cation 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.*

*[16] David Schmidt and Bernhard Ste en. Program analysis as model checking of abstract interpretations. Lecture Notes in Computer Sience, 1503:351{380, 1998.*

*[17] Micha Sharir and Amir Pnueli. Two approaches to interprocedural data*

*ow 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 Cli s , NJ , USA, 1981.*

*[18] 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.*