Model Checking Erlang Programs – LTL-Propositions and Abstract Interpretation

Frank Huch
Christian-Albrechts-University of Kiel,Germany
fhu@informatik.uni-kiel.de

Abstract. We present an approach for the formal verification of Erlang programs using abstract interpretation and model checking. In previous work we defined a framework for the verification of Erlang programs using abstract interpretation and LTL model checking. The application of LTL model checking yields some problems in the verification of state propositions, because propositions are also abstracted in the framework. In dependence of the number of negations in front of state propositions in a formula they must be satisfied or refuted. We show how this can automatically be decided by means of the abstract domain.
The approach is implemented as a prototype and we are able to prove properties like mutual exclusion or the absence of deadlocks and lifelocks for some Erlang programs.

1 Introduction

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

In industry the programming language Erlang [1] is used for the implementation of distributed systems. For formal verification, we have developed a framework for abstract interpretations [4, 13, 17] for a core fragment of Erlang [9]. This framework guarantees 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 satisfied on all paths, like in linear time logic (*LTL*). If the abstraction satisfies a property expressed in LTL, then also the program satisfies it, but not vice versa. If the AOS is a finite transition system, then model checking is decidable [15, 18]. For finite domain abstract interpretations and an additional flow-abstraction [11] this finite state property can be guaranteed for Erlang programs which do not create an unbound number of processes and use only mailboxes of restricted size.

However, the application of LTL model checking to the AOS is not so straightforward. LTL is usually defined by means of simple state propositions. In our approach of verification by model checking, the validity of a state proposition depends on the chosen abstraction. This must be taken into account in the implementation of the model checking algorithm. Furthermore, negation in LTL formulas results in additional problems. The equivalence of not validating a state proposition and refuting it does not hold in the context of abstraction. We show how safe model checking with respect to the abstract interpretation can be implemented.

The paper is organized as follows: Section 2 introduces the programming language Erlang and Section 3 shortly presents our framework for abstract interpretation. In Section 4 we add propositions to Erlang programs which can be used for formal verification by LTL model checking, shortly introduced in Section 5. Then, Section 6 motivates how abstract state propositions can be decided, which is formalized in Section 7. Finally, we present a concrete verification in Section 9 and conclude in Section 10.

2 Erlang

Erlang is a strict functional programming language. It is extended with concurrent processes. In Erlang, variables start with an uppercase letter. They can be bound to arbitrary values, with no type restrictions. Basic values are atoms (starting with a lowercase letter), e.g. `true`,`false`. More complex data structures can be created by constructors for tuples of any arity ($\{\ldots\}/n$) and constructors for lists (`[`.`|`.`]`and `[]`). It is also possible to use atoms and these constructors in pattern matching.

In Erlang, a new process can be created by `spawn`$(f, [a_1, \ldots, a_n])$. The process starts with the evaluation of $f(a_1, \ldots, a_n)$. Since Erlang is strict the arguments of `spawn` are evaluated before the new process is created. The functional result of `spawn` is the process identifier (*pid*) of the newly created process. With p`!`v arbitrary values (including pids) can be sent to other processes. The processes are addressed by their pids (p). A process can access its own pid with the Erlang function `self/0`. The messages sent to a process are stored in its mailbox and the process can access them conveniently with pattern matching in a `receive`-statement. Especially, it is possible to ignore some messages and fetch messages from further behind. For more details see [1].

For the formal verification we restrict to a core fragment of Erlang (called *Core Erlang*) which contains the main features of Erlang. The syntax of Core Erlang is defined as follows:

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

where ϕ represents predefined functions, functions of the program and constructors (c).

Example 1. A database process for storing key-value pairs can be defined as follows:

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

dataBase(L) -> prop(top),
   receive
      {allocate,Key,P} ->
          prop({allocate,P}),
          case lookup(Key,L) of
             fail -> P!free,
                     receive
                        {value,V,P} -> prop({value,P}),
                                       dataBase([{Key,V}|L])
                     end;
```

```
            {succ,V} -> P!prop(allocated), dataBase(L)
        end;
    {lookup,Key,P} -> prop(lookup),
                      P!lookup(Key,L), dataBase(L)
end.
```

At this point the reader should ignore the applications of the function `prop`. In Section 9 we will prove mutual exclusion for the database. For this purpose we will need the state propositions introduced by `prop`.

The program creates a database process holding a state (its argument `L`) in which key-value pairs are stored. The database is represented by a list of key-value pairs. The communication interface of the database is given by the messages `{allocate,Key,P}` for allocating a new key and `{lookup,Key,P}` for retrieving the value of a stored key. In both patterns `P` is bound to the requesting client pid to which the answer is sent.

The allocation of a new key is done in two steps. First the key is received and checked. If there is no conflict, then the corresponding value can be received and stored in the database. This exchange of messages in more than one step has to guarantee mutual exclusion on the database. 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 [9]. In Section 9 we will prove that the database combined with two accessing clients satisfies this property.

2.1 Semantics

In [9] 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 $\Pi \in States$. A process ($proc \in Proc$) consists of a pid ($\pi \in Pid := \{\texttt{@n} \mid n \in \mathbb{N}\}$), a Core Erlang evaluation term e and a word over constructor terms, representing the mailbox.

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

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

E is the context of the leftmost-innermost evaluation position, with v_1 a pid and v_2 an arbitrary constructor term. The value v_2 is added to the mailbox μ' of the process π' and the functional result of the send action is the sent value. For more details see [12].

3 Abstract Interpretation of Core Erlang Programs

In [9] we developed a framework for abstract interpretations of Core Erlang programs. The abstract operational semantics (*AOS*) yields a transition system which includes all paths of the SOS. In an abstract interpretation $\widehat{\mathcal{A}} = (\widehat{A}, \widehat{\iota}, \sqsubseteq, \alpha)$ for Core Erlang programs $\widehat{A}$ is the abstract domain which should be finite for our application in model checking. The abstract interpretation function $\widehat{\iota}$ defines the semantics of predefined function symbols and constructors. Its codomain is $\widehat{A}$. Therefore, it is for example not possible to interpret constructors freely in a finite domain abstraction. $\widehat{\iota}$ also defines the abstract behaviour of pattern matching in equations, `case`, and `receive`. Here the abstraction can yield additional non-determinism because branches can get undecidable in the abstraction. Hence, $\widehat{\iota}$ yields a set of results which defines possible successors. Furthermore, an abstract interpretation contains a partial order $\sqsubseteq$, describing which elements of $\widehat{A}$ are more precise than others. We do not

need a complete partial order or a lattice because we do not compute any fixed point. Instead, we just evaluate the operational semantics with respect to the abstract interpretation[1]. This yields a finite transition system, which we use for (on the fly) model checking. An example for an abstraction of numbers with an ordering of the abstract representations is: $\mathbb{N} \sqsubseteq \{v \mid v \leq 10\} \sqsubseteq \{v \mid v \leq 5\}$. It is more precise to know, that a value is ≤ 5, than ≤ 10 than any number. The last component of $\widehat{\mathcal{A}}$ is the abstraction function. α which maps every concrete value to its most precise abstract representation. Finally, the abstract interpretation has to fulfill five properties, which relate an abstract interpretation to the standard interpretation. They also guarantee that all paths of the SOS are represented in the AOS, for example in branching. An example for these properties is the following:

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

It postulates, that evaluating a predefined function or a constructor on abstract values which are representations of some concrete values yields abstractions of the evaluation of the same function on the concrete values. The other properties postulate correlating properties (P2-P5) for pattern matching in equations, `case`, and `receive`, and the pids represented by an abstract value. More details and some example abstractions can be found in [9, 10].

4 Adding Propositions

The semantics of Core Erlang is defined as a labeled transition system where the labels represent the performed actions of the system. We want to prove properties of the system with model checking. It would be possible to specify properties using the labels. Nevertheless, it is more convenient to add propositions to the states of this transition system. With these state propositions, properties can be expressed more easily. As names for the propositions we use arbitrary Core Erlang constructor terms which is very natural for Erlang programmers.

For the definition of propositions we assume a predefined Core Erlang function `prop/1` with the identity as operational semantics. Hence, adding applications of `prop` does not effect the SOS nor the AOS. Nevertheless, as a kind of side-effect the state in which `prop` is evaluated has the argument of `prop` as a valid state proposition. We mark this with the label prop in the AOS:

$$\frac{-}{\Pi, (\pi, E[\texttt{prop}(v)], \mu) \xrightarrow{\mathsf{prop}}_{\widehat{\mathcal{A}}} \Pi, (\pi, E[v], \mu)}$$

The valid state propositions of a process and a state over abstract values can be evaluated by the function prop:

Definition 1. *(Proposition of Processes and States[2])*
The proposition of a process is defined by $\mathsf{prop}_{\widehat{\mathcal{A}}} : \widehat{Proc}_{\widehat{\mathcal{A}}} \longrightarrow \mathcal{P}(\widehat{A})$*:*

$$\mathsf{prop}_{\widehat{\mathcal{A}}}((\pi, E[e], \mu)) := \begin{cases} \{\widehat{v}\} & , \text{ if } e = \texttt{prop}(\widehat{v}) \text{ and } \widehat{v} \in \widehat{A} \\ \emptyset & , \text{ otherwise} \end{cases}$$

The propositions of a state $\mathsf{prop}_{\widehat{\mathcal{A}}} : \widehat{State}_{\widehat{\mathcal{A}}} \longrightarrow \mathcal{P}(\widehat{A})$ *are defined as the union of all propositions of its processes:*

$$\mathsf{prop}_{\widehat{\mathcal{A}}}(\Pi) = \bigcup_{\pi \in \Pi} \mathsf{prop}_{\widehat{\mathcal{A}}}(\pi)$$

[1] Since we do not need a cpo, we use a partial order for the abstract domain. Therefore, the orientation of our abstract domain is upside down compared to standard frameworks using lattices. The least precise abstract value is the least element of the abstract domain.

In Example 1 we have added four propositions to the database which have the following meanings:

`top`	the main state of the database process
`{allocate,P}`	the process with pid `P` tries to allocate a key
`{value,P}`	the process with pid `P` enters a value into the database
`lookup`	a reading access to the database

In most cases, propositions will be added in a sequence as for example the proposition `top`. However, defining propositions by an Erlang function it is also possible to mark existing (sub-)expressions as propositions. As an example, we use the atom `allocated` which is sent to a requesting client, as a state proposition.

5 Linear Time Logic

The abstract operational semantics is defined by a transition system. We want to prove properties (described in temporal logic) of this transition system using model checking. We use *linear time logic* (*LTL*) [6] in which properties have to be satisfied on every path of a given transition system.

Definition 2. *(Syntax of Linear Time Logic (LTL)) Let Props be a set of state propositions. The set of* LTL-formulas *is defined as the smallest set with:*

- $Props \subseteq \text{LTL}$ — *state propositions*
- $\varphi, \psi \in \text{LTL} \Longrightarrow$
  - $\neg\varphi \in \text{LTL}$ — *negation*
  - $\varphi \wedge \psi \in \text{LTL}$ — *conjunction*
  - $X\varphi \in \text{LTL}$ — *in the next state φ holds*
  - $\varphi\ U\ \psi \in \text{LTL}$ — *φ holds until ψ holds*

An LTL-formula is interpreted with respect to an infinite path. The propositional formulas are satisfied, if the first state of a path satisfies them. The next modality $X\varphi$ holds if φ holds in the continuation of the path. Finally, LTL contains a strong until: If φ holds until ψ holds and ψ finally holds, then $\varphi\ U\ \psi$ holds. Formally, the semantics is defined as:

Definition 3. *(Path Semantics of LTL) An infinite word over sets of propositions $\pi = p_0p_1p_2\ldots \in \mathcal{P}(Props)^\omega$ is called a path. A path π satisfiess an LTL-formula φ ($\pi \models \varphi$) in the following cases:*

$$
\begin{array}{rcll}
p_0\pi & \models & P & \textit{iff } P \in p_0 \\
\pi & \models & \neg\varphi & \textit{iff } \pi \not\models \varphi \\
\pi & \models & \varphi \wedge \psi & \textit{iff } \pi \models \varphi \textit{ and } \pi \models \psi \\
p_0\pi & \models & X\varphi & \textit{iff } \pi \models \varphi \\
p_0p_1\ldots & \models & \varphi\ U\ \psi & \textit{iff } \exists i \in \mathbb{N} : p_ip_{i+1}\ldots \models \psi \textit{ and } \forall j < i : p_jp_{j+1}\ldots \models \varphi
\end{array}
$$

Formulas are not only interpreted with respect to a single path. Their semantics is extended to Kripke Structures:

[2] For both functions we use the name $\mathsf{prop}_{\widehat{\mathcal{A}}}$. The concrete instance of this overloading will be clear from the application of $\mathsf{prop}_{\widehat{\mathcal{A}}}$. We will also omit the abstract interpretation in the index, if it is clear from the context. As for the abstract domain, we use a hat to distinguish the abstract variants (which contain abstract instead of concrete values) from the defined concrete structures (e.g. $\widehat{Proc}$).

Definition 4. *(Kripke Structure)* $\mathcal{K} = (S, Props, \longrightarrow, \tau, s_0)$ *with* S *a set of states,* $Props$ *a set of propositions,* $\longrightarrow \subseteq S \times S$ *the transition relation,* $\tau : S \longrightarrow \mathcal{P}(Props)$ *a labeling function for the states, and* $s_0 \in S$ *the initial state is called a* Kripke Structure. *Instead of* $(s, s') \in \longrightarrow$ *we usually write* $s \longrightarrow s'$.

A state path of $\mathcal{K}$ *is an infinite word* $s_0 s_1 \ldots \in S^\omega$ *with* $s_i \longrightarrow s_{i+1}$ *and* s_0 *the initial state of* $\mathcal{K}$. *If* $s_0 s_1 \ldots$ *is a state path of* $\mathcal{K}$ *and* $p_i = \tau(s_i)$ *for all* $i \in \mathbb{N}$, *then the infinite word* $p_0 p_1 \ldots \in \mathcal{P}(Props)^\omega$ *is a* path of $\mathcal{K}$.

Definition 5. *(Kripke-Structure-Semantics of LTL) Let* $\mathcal{K} = (S, \rightarrow, \tau, s_0)$ *be a Kripke structure. It satisfies an LTL-formula* φ *(*$\mathcal{K} \models \varphi$*) iff for all paths* π *of* $\mathcal{K}$: $\pi \models \varphi$.

The technique of *model checking* automatically decides, if a given Kripke structure satisfies a given formula. For finite Kripke structures and the logic LTL model checking is decidable [15]. For the convenient specification of properties in LTL we define some abbreviations:

Definition 6. *(Abbreviations in LTL)*

ff	$:= \neg P \wedge P$	*the boolean value true*
tt	$:= \neg ff$	*the boolean value false*
$\varphi \vee \psi$	$:= \neg(\neg\varphi \wedge \neg\psi)$	*disjunction*
$\varphi \rightarrow \psi$	$:= \neg\varphi \vee \psi$	*implication*
$F\ \varphi$	$:= tt\ U\ \varphi$	*finally* φ *holds*
$G\ \varphi$	$:= \neg F \neg\varphi$	*globally* φ *holds*
$F^\infty\varphi$	$:= G\ F\ \varphi$	*infinitely often* φ *holds*
$G^\infty\varphi$	$:= F\ G\ \varphi$	*only finally often* φ *does not hold*

The propositional abbreviations are standard. $F\varphi$ is satisfied if there exists a position in the path, where φ holds. If in every position of the path φ holds, then $G\varphi$ is satisfied. The formulas φ which have to be satisfied in these positions of the path are not restricted to propositional formulas. They can express properties of the whole remaining path. This fact is used in the definition of $F^\infty\varphi$ and $G^\infty\varphi$. The weaker property $F^\infty\varphi$ postulates, that φ holds infinitely often on a path. Whereas $G^\infty\varphi$ is satisfied, if φ is satisfied with only finitely many exceptions. In other words there is a position, from where on φ always holds.

For the verification of Core Erlang programs we use the AOS respectively the SOS of a Core Erlang program as a Kripke structure. We use the transition system which is spawned from the initial state (`@0`,`main()`,()). As labeling function for the states we use the function prop from the previous section. Extending the semantics by state propositions in the presented way, a small problem might occur. The semantics is slightly modified by adding additional transition steps for state proposition. These additional steps can be relevant in the verification of properties for a fixed number of steps by means of the X operator. However, in practice it is uncommon to verify such formulas. Instead, the fixed-point operators F and G are usually used to specify system properties. The validity of these formulas is not influenced by the additional prop step.

6 Abstraction of Propositions

We want to verify Core Erlang programs with model checking. The framework for abstract interpretations of Core Erlang programs guarantees, that every path of the SOS is also represented in the AOS. If the resulting AOS is finite, then we can use simple model checking

algorithms to check, if it satisfies a property φ expressed in LTL. If φ is satisfied in the AOS, then φ also holds in the SOS. In the other case model checking yields a counter example which is a path in the AOS on which φ is not satisfied. Due to the fact that the AOS contains more paths than the SOS, the counter example must not be a counter example for the SOS. The counter path can be a valid path in the abstraction but not in the SOS. Therefore, in this case it only yields a hint, that the chosen abstraction is too coarse and must be refined.

The application of model checking seems to be easy but proving state propositions yields problems as the following example shows:

Example 2. `main() -> prop(42).`

A possible property of the program could be F **42** (finally 42). To prove this property we use the AOS with an abstract interpretation, for instance the even-odd interpretation, which only contains the values **even** (representing all even numbers), **odd** (representing all odd numbers), and **?** (representing all values). With this abstraction the AOS yields the following transition system:

$$(@0, \texttt{main}(), ()) \longrightarrow (@0, \texttt{prop}(42), ()) \longrightarrow (@0, \texttt{prop}(\textbf{even}), ()) \xrightarrow{\mathsf{prop}} (@0, \textbf{even}, ())$$

Only the state $(@0, \texttt{prop}(\textbf{even}), ())$ has a property, namely **even**. **42** is an even number, but it is not the only even number. For safeness, this property cannot be proven. For instance, we could otherwise also prove the property F **40**.

It is only possible to prove properties for which the corresponding abstract value exclusively represents this value. However, it does not make much sense to abstract from special values and express properties for these values afterwards. Therefore, we only use propositions of the abstract domain, like

$$F\ \textbf{even} \quad \text{(finally even)}$$

In the AOS the state (@0,`prop(`**even**`)`,()) has the property **even**. Therefore, the program satisfies this property. Now we consider a more complicated example:

Example 3. `main() -> prop(84 div 2).`

This system satisfies the property too because $(84 \div 2) = 42$. On the other hand, in the even-odd abstraction we only get:

$$\begin{array}{c} (@0, \texttt{main}(), ()) \longrightarrow (@0, \texttt{prop}(84\ \texttt{div}\ 2), ()) \longrightarrow (@0, \texttt{prop}(\textbf{even}\ \texttt{div}\ 2), ()) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \downarrow \\ (@0, \texttt{?}, ()) \longleftarrow (@0, \texttt{prop}(\texttt{?}), ()) \longleftarrow (@0, \texttt{prop}(\textbf{even}\ \texttt{div}\ \textbf{even}), ()) \end{array}$$

with prop((@0,`prop(?)`,()))= {**?**} and $\emptyset$ as propositions of the other states. The result of the division of two even values must not be even. In a safe abstraction we cannot be sure, that the property F **even** is satisfied. Hence, model checking must yield, that it does not hold. For instance, the program

```
main() -> prop(42 div 2).
```

has a similar AOS but the property is not satisfied ($42 \div 2 = 21$). Therefore, a property is satisfied in a state, if the property of the state is at least as precise, as the expected property:

$$p_0 p_1 \ldots \models \widetilde{v} \text{ iff } \exists \widetilde{v}' \in p_0 \text{ with } \widetilde{v} \sqsubseteq \widetilde{v}'$$

However, this is not correct in all cases, as the following example shows. We want to prove that the program satisfies the property $G\neg$**even** (always not even) Therefore, one point is to check that the state (`@0`,`prop(?)`,()) models $\neg$**even**. With the definition from above we can conclude:

$$(\texttt{@0},\texttt{prop(?)},()) \not\models \textbf{even} \qquad \text{and hence} \qquad (\texttt{@0},\texttt{prop(?)},()) \models \neg\textbf{even}.$$

However, this is wrong. In Example 3 the property is not satisfied. The SOS has the property 42, which is an even value. The reason is the non-monotonicity of $\neg$. Considering abstraction, the equivalence

$$\pi \models \neg\varphi \text{ iff } \pi \not\models \varphi$$

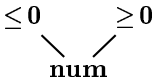
does not hold! $\pi \not\models \varphi$ only means that $\pi \models \varphi$ is not safe. In other words, there can be a concretization which satisfies φ but we cannot be sure that it holds for all concretizations. Therefore, negation has to be handled carefully.

Which value of our abstract domain would fulfill the negated proposition $\neg$**even**? Only the proposition **odd** does. The values **even** and **odd** are incomparable and no value exists, which is more precise than these two abstract values. This connection can be generalized as follows:

$$p_0 p_1 \ldots \models \neg\widetilde{v} \text{ if } \forall \widetilde{v}' \in p_0 \text{ holds } \widetilde{v} \sqcup \widetilde{v}' \text{ does not exist}$$

Note, that this is no equivalence anymore. The non-existence of $\widetilde{v} \sqcup \widetilde{v}'$ does only imply that $p_0 p_1 \ldots \models \neg\widetilde{v}$. It does not give any information for the negation $p_0 p_1 \ldots \models \widetilde{v}$. This (double) negation holds, if $\exists \widetilde{v}' \in p_0$ with $\widetilde{v} \sqsubseteq \widetilde{v}'$.

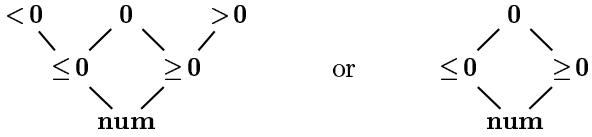
On a first sight refuting a state proposition seems not to be correct for arbitrary abstract interpretations. Consider the abstract domain



where the abstract value $\leq\mathbf{0}$ represents the concrete values $\{\mathbf{0}, -\mathbf{1}, -\mathbf{2}, \ldots\}$, $\geq\mathbf{0}$ represents $\{\mathbf{0}, \mathbf{1}, \mathbf{2}, \ldots\}$, and **num** represents $\mathbb{Z}$. The represented concrete values of $\leq\mathbf{0}$ and $\geq\mathbf{0}$ overlap in the value **0**. Therefore, it would be incorrect that a state with the proposition $\leq\mathbf{0}$ satisfies the formula $\neg\geq\mathbf{0}$.

However, this abstraction is not possible. Any abstraction function $\alpha : A \longrightarrow \widehat{A}$ yields one single abstract representation for a concrete value. Without loss of generality, let $\alpha(\mathbf{0}) = \geq\mathbf{0}$. Abstract values which represent the concrete value **0** can only be the result of the use of the abstract interpretation function $\widehat{\iota}$. However, all these results $\widetilde{v}$ must be less precise: $\widetilde{v} \sqsubseteq \alpha(\mathbf{0}) = \geq\mathbf{0}$ because of the properties claimed by our framework. Hence, this abstract domain can be defined but the value $\leq\mathbf{0}$ does only represent the values $\{-\mathbf{1}, -\mathbf{2}, \ldots\}$. The name of the abstract value is not relevant. However, for understandability it should be renamed to $<\mathbf{0}$.

Alternatively, the abstract domain can be refined. The two overlapping abstract values can be distinguished by a more precise abstract value:



In both cases we must define $\alpha(\mathbf{0}) = \mathbf{0}$ because otherwise we have the same situation as before and the concrete value $\mathbf{0}$ is not represented by both abstract values $\leq\mathbf{0}$ and $\geq\mathbf{0}$.

7 Concretization of Propositions

With the advisement of the previous section we can now formalize whether a state proposition is satisfied or refuted, respectively. Similar results have been found by Clark, Grumberg, and Long [3] and Knesten and Pnueli [14]. In their results Knesten and Pnueli handle a simple toy language in which only propositions on integer values can be made. In our framework state propositions are arbitrary Erlang values and their validity has to be decided with respect to an arbitrary abstract domain. The same holds for the paper of Clark et. al which also does not consider a real programming language. In following, we present how these ideas can be transfered to formal verification of the real programming language Erlang.

First we define the concretization of an abstract value. This is the set of all concrete values which have been abstracted to the value, or a more precise value.

Definition 7. *(Concretization of Abstract Values)*
Let $\widehat{\mathcal{A}} = (\widehat{A}, \widehat{\iota}, \sqsubseteq, \alpha)$ be an abstract interpretation. The concretization function $\gamma : \widehat{A} \longrightarrow \mathcal{P}(T_{\mathcal{C}}(Pid))$ is defined as $\gamma(\widetilde{v}) = \{v \mid \widetilde{v} \sqsubseteq \alpha(v)\}$.

For the last example we get the following concretizations:

$$\begin{array}{lcllcl} \gamma(\mathbf{0}) & = & \{\mathbf{0}\} & \gamma(\geq\mathbf{0}) & = & \{\mathbf{0}, \mathbf{1}, \mathbf{2}, \ldots\} \\ \gamma(\leq\mathbf{0}) & = & \{\mathbf{0}, -\mathbf{1}, -\mathbf{2}, \ldots\} & \gamma(\mathbf{num}) & = & \mathbb{Z} \end{array}$$

Lemma 1. *(Connections between γ and α)*
Let $\widehat{\mathcal{A}} = (\widehat{A}, \widehat{\iota}, \sqsubseteq, \alpha)$ be an abstract interpretation and γ the corresponding concretization function. Then the following properties hold:

1. $\forall v \in \gamma(\widetilde{v}) : \widetilde{v} \sqsubseteq \alpha(v)$
2. $\bigsqcap\{\alpha(v) \mid v \in \gamma(\widetilde{v})\} = \widetilde{v}$

Proof.
1. $v \in \gamma(\widetilde{v})$ iff $v \in \{v' \mid \widetilde{v} \sqsubseteq \alpha(v')\}$ iff $\widetilde{v} \sqsubseteq \alpha(v)$
2. $\bigsqcap\{\alpha(v) \mid v \in \gamma(\widetilde{v})\} = \bigsqcap\{\alpha(v) \mid v \in \{v' \mid \widetilde{v} \sqsubseteq \alpha(v')\}\} = \bigsqcap\{\alpha(v) \mid \widetilde{v} \sqsubseteq \alpha(v)\} = \widetilde{v}$

With the concretization function we can define whether a state proposition of a state satisfies a proposition in the formula or refutes it.

Definition 8. *(Semantics of a State Proposition)*
Let $\widehat{\mathcal{A}} = (\widehat{A}, \widehat{\iota}, \sqsubseteq, \alpha)$ be an abstract interpretation. A set of abstract state propositions satisfies or refutes a proposition of a formula in the following cases:

$$\begin{array}{l} p \models \widetilde{v} \quad \textit{if} \quad \exists \widetilde{v}' \in p \textit{ with } \gamma(\widetilde{v}') \subseteq \gamma(\widetilde{v}) \\ p \not\models \widetilde{v} \quad \textit{if} \quad \forall \widetilde{v}' \in p \textit{ holds } \gamma(\widetilde{v}) \cap \gamma(\widetilde{v}') = \emptyset \end{array}$$

Similarly to these definitions for the concretization, we can decide whether a state proposition if satisfied or refuted for abstract values. For finite domain abstractions, this can be decided automatically.

Lemma 2. *(Deciding Propositions in the abstract domain)*
Let $\widehat{\mathcal{A}} = (\widehat{A}, \widehat{\iota}, \sqsubseteq, \alpha)$ be an abstract interpretation. A set of abstract state propositions satisfies or refutes a proposition of a formula in the following cases:

$$\begin{array}{l} p \models \widetilde{v} \quad \textit{if} \quad \exists \widetilde{v}' \in p \textit{ with } \widetilde{v} \sqsubseteq \widetilde{v}' \\ p \not\models \widetilde{v} \quad \textit{if} \quad \forall \widetilde{v}' \in p \textit{ holds } \widetilde{v} \sqcup \widetilde{v}' \textit{ does not exist} \end{array}$$

Proof. We show: $\widetilde{v} \sqsubseteq \widetilde{v}'$ implies $\gamma(\widetilde{v}') \subseteq \gamma(\widetilde{v})$ and the non-existence of $\widetilde{v} \sqcup \widetilde{v}'$ implies $\gamma(\widetilde{v}) \cap \gamma(\widetilde{v}') = \emptyset$:

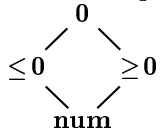
- $\widetilde{v} \sqsubseteq \widetilde{v}'$
  $\gamma(\widetilde{v}) = \{v \mid \widetilde{v} \sqsubseteq \alpha(v)\}$ and $\gamma(\widetilde{v}') = \{v \mid \widetilde{v}' \sqsubseteq \alpha(v)\}$.
  $(\widehat{A}, \sqsubseteq)$ is a partial order. Hence, it is transitive. This implies $\gamma(\widetilde{v}') \subseteq \gamma(\widetilde{v})$.
- $\widetilde{v} \sqcup \widetilde{v}'$ does not exist $\Longrightarrow \gamma(\widetilde{v} \sqcup \widetilde{v}') = \emptyset \Longrightarrow \{v \mid (\widetilde{v} \sqcup \widetilde{v}') \sqsubseteq \alpha(v)\} = \emptyset$
  $\Longrightarrow \{v \mid \widetilde{v}' \sqsubseteq \alpha(v) \text{ and } \widetilde{v} \sqsubseteq \alpha(v)\} = \emptyset \Longrightarrow \gamma(\widetilde{v}') \cap \gamma(\widetilde{v}) = \emptyset$

Note, that we only show an implication. We can define unnatural abstract domains in which a property is satisfied or refuted with respect to Definition 8 but using only the abstract domain, we cannot show this. We consider the following abstract domain:

$$\mathbf{num} \sqsubseteq \mathbf{zero} \sqsubseteq \mathbf{0} \qquad \text{with } \alpha(v) = \begin{cases} \mathbf{0} & \text{, if } v = \mathbf{0} \\ \mathbf{num} & \text{otherwise} \end{cases}$$

The abstract value **zero** is superfluous because it represents exactly the same values, as the abstract value **0**. However, this abstract domain is valid. Using the definition of the semantics of a state proposition from Definition 8, we can show that $\{\mathbf{zero}\} \models \mathbf{0}$ because $\gamma(\mathbf{zero}) = \gamma(\mathbf{0}) = \{\mathbf{0}\}$. However, $\mathbf{zero} \sqsubseteq \mathbf{0}$ and we cannot show that $\{\mathbf{zero}\} \models \mathbf{0}$ just using the abstract domain.

The same holds for refuting a state proposition:



$$\text{with } \alpha(v) = \begin{cases} \geq \mathbf{0} & \text{, if } v \geq \mathbf{0} \\ \leq \mathbf{0} & \text{otherwise} \end{cases}$$

In this domain the abstract value **0** is superfluous. Its concretization is empty. Hence, $\gamma(\leq \mathbf{0}) = \{-\mathbf{1}, -\mathbf{2}, \ldots\}$ and $\gamma(\geq \mathbf{0}) = \{\mathbf{0}, \mathbf{1}, \mathbf{2}, \ldots\}$. $\gamma(\leq \mathbf{0}) \cap \gamma(\geq \mathbf{0}) = \emptyset$ and $\leq \mathbf{0} \models \neg \geq \mathbf{0}$. However, this proposition cannot be refuted with this abstract domain because $\leq \mathbf{0} \sqcup \geq \mathbf{0} = \mathbf{0}$ exists.

These examples are unnatural because the domains contain superfluous abstract values. Nobody will define domains like these. Usually, the concretization of an abstract value is nonempty and differs from the concretizations of all other abstract values. In this case deciding propositions in the abstract domain is complete with respect to the semantics of propositions. Although it is not complete in general, it is safe. If we can prove a property with the abstract values, then it is also correct for its concretizations.

8 Proving LTL Formulas

So far we have discussed whether a state proposition is satisfied or refuted. However, in LTL negation is not only allowed in front of state propositions. Arbitrary sub-formulas can be negated. For the decision of arbitrary LTL formulas with respect to an abstract interpretation two approaches are possible: In the first approach, all negations can be pushed into the formula until they only occur in front of state propositions. Since there exists no equivalent representation of $\neg(\varphi U \psi)$ which uses negation only in front of φ and ψ, we must extend LTL with the release modality $\varphi R \psi$, the dual modality of until: $\neg(\varphi U \psi) \sim \neg\varphi R \neg\psi$. Its semantics is defined as:

$$p_0 p_1 \ldots \models \varphi R \psi \text{ iff } \forall i \in \mathbb{N} : p_i p_{i+1} \ldots \models \psi \text{ or } \exists j < i : p_j p_{j+1} \ldots \models \varphi$$

There is no intuitive semantics of release, except that it can be used for the negation of until. However, it can also be automatically verified in model checking.

Furthermore, we must add $\vee$ to LTL. After pushing the negations in front of the state propositions we can use standard model checking algorithms in combination with Lemma 2 for deciding state propositions.

Standard model checking algorithms work with a similar idea but they do not need the release modality. For example, in [18] an alternating automaton is constructed, that represents the maximal model which satisfies the formula. The states correspond to the possible sub-formulas and their negations. For every negation in the formula the automaton switches to the corresponding state which represents the positive or negative sub-formula. With this alternation negations are pushed into the automaton representing the formula, like in the first approach. We use a similar idea as in the second approach and decide the state propositions in dependence of the negation in front. We distinguish positive (marked with $+$) and negative (marked with $-$) state propositions in dependence on an even respectively odd number of negations in front of them. The advantage of this approach is that the formula is not transformed and its semantics stays more intuitive. The two kinds of state propositions can then be decided as follows:

$$\begin{array}{ll} p_0 p_1 \ldots \models^+ \widetilde{v} & \text{if } \exists \widetilde{v}' \in p_0 \text{ with } \widetilde{v} \sqsubseteq \widetilde{v}' \\ p_0 p_1 \ldots \not\models^- \widetilde{v} & \text{if } \forall \widetilde{v}' \in p_0 \text{ holds } \widetilde{v} \sqcup \widetilde{v}' \text{ does not exist} \end{array}$$

We implemented this model checker as a prototype. By means of this prototype we are able to analyze small systems with up to 30000 states in the abstract model (AOS). The prototype also includes some standard abstractions (for which we proved correctness with respect to the claimed properties P1-P5 of [9]) and a simple partial order reduction optimization. For the verification of real systems, this prototype is not sufficient and a connection to an efficient model checker like Spin [8] is necessary.

9 Verification of the Database

Now we want to verify the system of Example 1. A database process and two clients are executed. We want to guarantee, that the process which allocates a key also sets the corresponding value:

> If a process π allocates a key, then no other process π' sets a value before π sets a value, or the key is already allocated.

This can for arbitrary processes be expressed in LTL as follows:

$$\bigwedge_{\substack{\pi \in Pid \\ \pi' \neq \pi}} G\ (^-\{\mathbf{allocate}, \pi\} \longrightarrow (\neg\, ^-\{\mathbf{value}, \pi'\})\ U\ (^+\{\mathbf{value}, \pi\} \vee\ ^+\mathbf{allocated}))$$

In our system only a finite number of pids occurs. Therefore, this formula can be translated into a pure LTL-formula as a conjunction of all possible permutations of possible pids which satisfy the condition. This is

$$(\pi, \pi') \in \{(\texttt{@0}, \texttt{@1}), (\texttt{@0}, \texttt{@2}), (\texttt{@1}, \texttt{@2}), (\texttt{@1}, \texttt{@0}), (\texttt{@2}, \texttt{@0}), (\texttt{@2}, \texttt{@1})\}.$$

We have already marked the propositions in the formula with respect to the negations in front of them. Considering this marking the proposition $^-\{\mathbf{allocate}, \pi\}$ must be refuted. This is for

example the case for the abstract values **top** and **{lookup,?}**. However, **?** and **{allocate,p}** with p the pid of the accessing client do not refute the proposition. The right side of the implication must be satisfied. Similar conditions must hold for the other propositions.

We can automatically verify this property using a finite domain abstraction, in which only the top-parts of depth 2 of the constructor terms are considered. The deeper parts of a constructor term are cut off and replaced by **?**. For more details see [10]. Our framework guarantees that the property also holds in the SOS and we have proven mutual exclusion for the database program.

10 Related Work and Conclusion

There exist two other approaches for the formal verification of Erlang: EVT [16] is a theorem prover especially tailored for Erlang. The main disadvantage of this approach is the complexity of proves. Especially, induction on infinite calculations is very complicated and automatization is not support yet. Therefore, a user must be an expert in theorem proving and EVT to prove system properties. We think, for the practical verification of distributed systems push-button techniques are needed. Such a technique is model checking, which we use for the automated verification of Erlang programs. This approach is also pursued by part of the EVT group in [2]. They verified a distributed resource locker written in Erlang with a standard model checker. The disadvantage of this approach is that they can only verify finite state systems. However, in practice many systems have an infinite (or for model checkers too large) state space. As a solution, we think abstraction is needed to verify larger distributed systems. A similar approach for Java is made by the Bandera tool [7]. They provide abstraction, but the user is responsible to define consistent abstractions by explicitly defining abstracted methods in the source code. Hence, the user can define wrong abstractions and the correctness of their approach is not guaranteed. Furthermore, they have the same problems with negation as we discussed in this paper, but they do not explicitly discuss them and the user is responsible to solve them. The Bandera tool uses standard model checkers in which the presented problems of abstraction are not considered.

Our approach for the formal verification of Erlang programs uses abstract interpretation and LTL model checking. The main idea is the construction of a finite model by the AOS by means of a finite domain abstract interpretation. The abstraction is safe in the sense, that all paths of the SOS are also represented in the AOS.

For convenient verification we have added state propositions to the AOS. Considering the abstract interpretation, problems in the semantics of these propositions arise. In this paper we solved these problems by distinguishing positive and negative propositions of the formula. For these we can decide, if a state satisfies or refutes it by means of the abstract domain. Finally, we used this technique in the formal verification of the database process: we proved mutual exclusion for two accessing clients.

For future work, we want to transfer the presented and prototypically implemented results to a tool using existing model checkers like Spin [8]. Here a problem is the integration of verifying or refuting properties in dependence of their positive or negative occurrences in formulas. α-Spin [5] is a first approach tackling this problem. However, using their abstraction techniques will be difficult, because Spins specification language Promela and Erlang are quite different (imperative vs functional, different kinds of data structures). Another problem will be the optimization of model checking by means of partial order reduction. For partial order reduction it is necessary to know which actions performed by the processes are independent.

In our semantics, only prop and send actions are dependent to other concurrent actions of the same kind (we use this information in our prototype). Defining a translation from Erlang to Promela which preserves this dependency information seems to be a another difficult and interesting task for future work.

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. Edmund Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.
4. 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.
5. María del Mar Gallardo, Jesús Martínez, Pedro Merino, and Ernesto Pimentel. αSPIN: Extending SPIN with abstraction. In *Proc. of the 9th Int. SPIN Workshop on Model Checking of Software*, volume 2318 of *LNCS*, pages 254–258, 2002.
6. Dov Gabbay, Amir Pnueli, Saharon Shelah, and Jonathan Stavi. On the temporal analysis of fairness. In *Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages*, pages 163–173. ACM SIGACT and SIGPLAN, ACM Press, 1980.
7. John Hatcliff, Matthew B. Dwyer, and Shawn Laubach. Staging static analyses using abstraction-based program specialization. *LNCS*, 1490:134–148, 1998.
8. Gerard J. Holzmann. The model checker SPIN. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997. Special issue on Formal Methods in Software Practice.
9. 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 Int. Conference on Functional Programming (ICFP '99).
10. Frank Huch. Verification of Erlang programs using abstract interpretation and model checking – extended version. Technical Report 99–02, RWTH Aachen, 1999.
11. Frank Huch. Model checking erlang programs - abstracting recursive function calls. In *Proceedings of WFLP'01, ENTCS*, volume 64. Elsevier, 2002.
12. Frank Huch. *Verification of Erlang Programs using Abstract Interpretation and Model Checking.* PhD thesis, RWTH Aachen, Aachen, 2002.
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. Yonit Kesten and Amir Pnueli. Modularization and abstraction: The keys to practical formal verification. In L. Brim, J. Gruska, and J. Zlatuska, editors, *The 23rd Int. Symposium on Mathematical Foundations of Computer Science (MFCS 1998)*, volume 1450 of *LNCS*, pages 54–71. Springer, 1998.
15. Orna Lichtenstein and Amir Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Conference Record of the 12th ACM Symposium on Principles of Programming Languages*, pages 97–107, New Orleans, Louisiana, 1985. ACM SIGACT-SIGPLAN, ACM Press.
16. Thomas Noll, Lars-åke Fredlund, and Dilian Gurov. The Erlang verification tool. In *Proceedings of TACAS'01*, volume 2031 of *LNCS*, pages 582–585. Springer, 2001.
17. David Schmidt and Bernhard Steffen. Program analysis as model checking of abstract interpretations. *LNCS*, 1503:351–380, 1998.
18. Moshe Y. Vardi. *An Automata-Theoretic Approach to Linear Temporal Logic*, volume 1043 of *LNCS*, pages 238–266. Springer, New York, NY, USA, 1996.