Erlang Specification Method – A Tool for the Graphical Specification of Distributed Systems

Frank Huch
Institut für Informatik, Christian-Albrechts-Universität Kiel,
D-24098 Kiel, Germany
fhu@informatik.uni-kiel.de

ABSTRACT

We present a tool for the specification, documentation and verification of specifications for distributed systems. The used specification formalism is especially designed for the later implementation using the functional programming language Erlang. The main point of the specification formalism is using the same kind of communication and process concept in the specification as provided by Erlang. Processes are graphically specified as labeled transition systems with sequential behaviour, branching, sending or receiving of values, and the creation of new processes as possible actions. As in Erlang receiving values can be expressed with pattern matching against all messages in the mailbox of a process.

The tool provides an editor for a convenient development of specifications, a consistence checker, a simulator, a component for generating Erlang code from the specification and a model checker for proving properties expressed in LTL for abstractions of the specifications.

Keywords: specification, verification, Erlang, distributed system

1 INTRODUCTION

Growing requirements of industry and society impose greater complexity of software development. Consequently understandability, maintenance and reliability can not be warranted.

Things get even harder whenever leaving the sequential territory and developing distributed systems. In addition to the problem of designing large systems we get the problem that the behaviour of concurrent processes is even harder to understand than a sequential behaviour. In distributed systems many programs are executed concurrently and interact via communication. Hence for a simulation of what happens when a program is executed one has to consider many states in many processes, the system can be in. Considering hundred or more concurrent processes it is clear that this is impossible, just knowing the code of the program. The difference to sequential programs is the great number of possible execution paths of concurrent and distributed system in dependency of schedulers or the performance of system components. Different runs of a program can lead to different results, even deadlocks. Hence a program tested and running well on a plattform can deadlock on another one.

The same problems appear during the development of distributed systems. Therefore we need a formal method for the specification of distributed systems which can also be used as a good documentation of its implementation. The way from the specification to the implementation should be straightforward to provide fast development and should also be seconded by automatic generation of code or code skeletons. But for using the specifications as documentation it is also necessary that the implementation does not differ too much from the specification and that the way back from the implementation to the specification can also be taken. To provide this the specification formalism should be tailored especially to the programming language and use the same kind of process model and communication. Messages should have the same format and new processes should be created in the same way.

In this paper we present the Erlang Specification Method (ESM) for the specification of concurrent and distributed systems designed especially for the distributed functional language Erlang. This formalism was used for the first time for the specification of a communication system. This was done by groups of students in a practical course. In the development almost 70% of work had to be done in specification. But this resulted in very good structured systems which work very robustly and have an almost professional character. Time for the development was very short and we were surprised of the quality and complexity of the developed systems. After the practical course we started to develop a tool for the graphical specification of Erlang processes. It supports a comfortable graph editor, consistence checks of the specifications, a simulation component and a model checker for proving properties of the system expressed in LTL. Because ESM is very expressive, the verification component also provides abstraction.

In Section 2 we present our specification formalism in detail. For the simulation of a specified system a formal semantics is defined in Section 3. Section 4 shows how easy code skelettons can be generated from the specification. The verification component is described in Section 5 and in the conclusions in Section 6 we show the experiences made in the development of a communication system and discuss future work. For readers, who are not familiar with Erlang, we give short introduction in Appendix A.

2 THE SPECIFICATION FORMALISM ESM

In the development of large distributed systems it is necessary to specify the system design. A large part of the specification of a distributed system is the specification of protocols, which processes use for their communication. Other parts of the system specification are the partitioning in processes, their location in a network and the behaviour of the sequential parts. In this paper we will not discuss the specification of the partitioning in processes and their location. We will concentrate on the protocol specification.

In practice Erlang developers use specification formalisms like UML [Alh98] or SDL [ST87] for the specification of protocols. The main problem of the use of these formalisms is that communication between processes differs from communication in Erlang. But in the specification of a protocol the kind of communication is a major point. Implementing UML or SDL specifications in Erlang leads to a program which simulates the corresponding communication of the specification formalism. We lose the expressiveness of accessing the messages of the mailbox with pattern matching. Another point of implementing these specifications in Erlang is that we have to guarantee that our implementation of the UML or SDL communication is correct. This can especially be hard using UML with remote method invocation. Another problem of UML is that it uses an object oriented view which can be difficult to be implemented in Erlang considering inheritance.

On the first sight SDL seams to match communication in Erlang. And Erlang programmers are supposed to use it for the specification of Erlang programs [AWV93]. In SDL processes are also identified by a pid and every process has a mailbox in which incoming messages are stored. But the mailbox is just a queue and a process has to take the messages out, in the same chronological order, as they come in. So processes have to consider many possible interleaving sequences of incoming messages instead of picking them out in a supposed order. Implementing an SDL specification in Erlang, all receive statements are flat and all possible messages have to be considered in every `receive` statement. The program gets larger than it could be using pattern matching on the mailboxes.

We see that the model of communication is a major point for the choice of a specification formalism. Many other formalisms like LOTOS [vEVD89] and Estelle [ISO89] exist for the specification of protocols. But there is no formalism defined yet, which uses the same communication as Erlang does. But for developing distributed systems in Erlang with all the opportunities of Erlang such a formalism has to be defined. The result is the Erlang Specification Method (ESM):

For every kind of process $p \in \mathcal{P}$ a labeled transition system $\mathcal{T}_p = (S_p, I_p, \longrightarrow_p, vars_p)$ is defined which specifies its behaviour. The states $s \in S_p$ are labeled with variables representing the knowledge of the process in this state. $Vars$ is the set of all variables and $vars_p : S_p \longrightarrow Vars^*$, yields the sequence[1] of variables the states are labeled with. The initial states $i \in I_p$ are labeled with the variables the process binds its knowledge in, when it is started. Multiple initial states are possible because a process can be started in different ways.

The labels of the transitions will contain Erlang terms over variables of the predecessor nodes for the specification of calls to sequential parts. Therefore we assume Σ to be set of function symbols with arity. As an example $+/2 \in \Sigma$. Now we can define Erlang terms over a set S, which will later be instantiated with variables or pids:

$$S \subseteq T_\Sigma(S)$$
$$v_1, \ldots, v_n \in T_\Sigma(S) \text{ and } f/n \in \Sigma \Longrightarrow f(v_1, \ldots, v_n) \in T_\Sigma(S)$$

Let $C = \{\texttt{[ ]}/0, \texttt{[.|.]}/2\} \cup \{\texttt{\{...\}}/n \mid n \in \mathbb{N}\} \cup \{a/0 | a \in Atoms\}$ be the set of constructor functions, where $Atoms$ is the set of all allowed Erlang atoms, `[ ]`and `[.|.]`are the constructors for building lists and `{...}` is the constructor for tuples of any arity. For the set of constructor terms, which will later be the values we calculate on or the possible patterns we then use $T_C(S)$.

The following transitions are possible:

- Sequential evaluation which is specified with an Erlang expression. Its behaviour is specified with natural language. For access to the result we allow pattern matching. The variables which are bound in pattern matching can be part of the knowledge of the destination state.
  Notation: $(s, pat \texttt{ <- } e, t) \in \longrightarrow_p$, where $s, t \in S_p$, $e \in T_\Sigma(Vars(s))$, $pat \in T_C(Vars)$ and $vars_p(t) \subseteq vars_p(s) \cup vars(pat)$

- Branching in dependence of an the value of an Erlang expression. Again the behaviour of the expression is specified with natural language, but the result is matched against multiple patterns. All matching patterns can be performed. The difference to the sequential behaviour is that branching is allowed with the same expression.
  Notation: $(s, \texttt{case } e \texttt{ of } pat, t) \in \longrightarrow_p$, where $s, t \in S_p$, $e \in T_\Sigma(Vars(s))$, $pat \in T_C(Vars)$ and $vars_p(t) \subseteq vars_p(s) \cup vars(pat)$

- Sending a value to another process. The pid of the other process must be known in this state. We use the Erlang syntax for sending actions.
  Notation: $(s, X \,!\, v, t) \in \longrightarrow_p$, where $s, t \in S_p$, $X \in vars_p(s)$, $v \in T_C(Vars(s))$ and $vars_p(t) \subseteq vars_p(s)$

- Receiving values and binding variables to (parts of) these values with pattern matching. The bound variables can be knowledge of the destination state.

[1] In most cases we will handle this sequence as a set of variables. The order is only relevant in the creation of new processes, when values are passed.

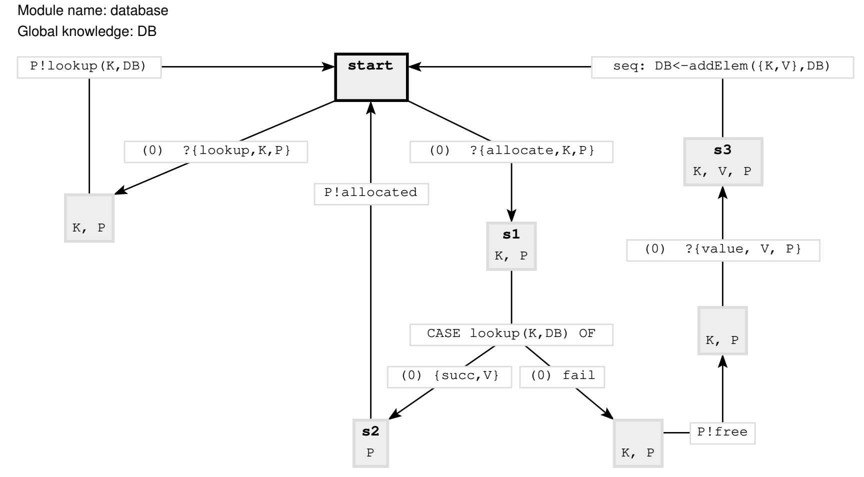

Figure 1
The specification of a database process

Notation: $(s, ?pat, t) \in \longrightarrow_p$, where $s, t \in S_p$, $pat \in T_C(Vars)$ and $vars_p(t) \subseteq vars_p(s) \cup vars(pat)$

- Creation of new processes with a list of arguments. For the creation of new processes we also use the Erlang syntax of `spawn` and add X `<- spawn(...)` if the pid of the spawned process is needed in future states and has to be knowledge of the destination state.
  Notation: $(s, X\texttt{<-spawn}(p', i, e), t) \in \longrightarrow_p$ or $(s, \texttt{spawn}\,(p', i, e), t) \in \longrightarrow_p$, where $s, t \in S_p$, $X \in Vars$, $e \in T_\Sigma(Vars(s))$, $i \in I_{p'}$ an initial state of the process specification $p' \in \mathcal{P}$ and $vars_p(t) \subseteq vars_p(s) \cup \{X\}$

To prevent too much nondeterministic behaviour of a process there may be only two kinds of nodes with more than one outgoing arc. First, a branch can be specified with case arcs, for example a lookup in a database succeeds or fails. All expressions of a branch have to be identical. The user should define non–overlapping patterns, but non–determinism in the protocol specification is possible as well. In the tool, this formal definition is implemented as a special branching pattern, which allows the user to define only one expression. Furthermore branching is allowed, if every outgoing arc of a state is labeled with a receive operation. Again the user should define non–overlapping patterns, to prevent non–determinism, but in an early part of the specification non–determinism can be helpful.

In Figure 1 we present the specification of a database or name server, in which clients can store values with unique keys and look them up with a key. All presented specifications are created with our tool.

In our tool the initial states are marked with a border to distinguish them from the other states. For every state the user can optionally define a name, which is displayed in its first line. Only the initial states must have a name, because with these names they can be spawned from other processes. The variables of the state follow in the next lines. Furthermore it is possible to declare some variables as global knowledge. This is an abbreviation for the occurrence of these variables in every state of the process specification. In the formal semantics we have omitted this. But for clearer specifications we have added the declaration of global knowledge in the tool.

The specification of the database has only one initial state, named `start`. In this state the database accepts only messages matching the patterns `{allocate,K,P}` and `{lookup,K,P}`. The variable `K` is bound to the requested key and `P` to the pid of the requesting process in both cases. In the `lookup`–case the database just answers with the lookup–value of the key in the database, where `lookup` is a sequential function. Its specification can colloquially be defined by: "`lookup` yields the corresponding value of a key stored in the database". This specification is stored in the tool and can be accessed or modified by clicking on the arc. When an `allocate`–message is received, we have to consider two cases. First, the key is already allocated and the database only notifies the requesting process and succeeds in its initial state. Second, the key is free. In the formal syntax this branch was modeled by two arcs labeled with case, the call of the lookup–function and the respective patterns. The numbers in front of the patterns in the specification in Figure 1 are an additional feature of the tool not formalized above. Here the user can defined priorities in which the patterns should be matched, if they overlap. This can sometimes be useful to make a specification more detailed. In our specification the patterns do not overlap and all priorities have no effect and can be by default set to zero. After a notification of the client, the database waits until the corresponding value is received and adds this new values to the database.

The main point in the specification of the database process is, that we guarantee mutual exclusion for multiple clients extending the database. Therefore we use two different states in which we wait for different messages. Furthermore, we check if the `allocate`–message and the `value`–message include the same pids in their third component. The variable `P` in the pattern `{value,V,P}` is already bound to the pid of the requesting process and is no introduction of a fresh unbound variable. Hence the second receive statement suspends until a value with the same pid is sent. We will later prove this property and need to refer to the states `s1`, `s2` and `s3`. Therefore we named them in the specification.

A specification of this database process in SDL would be more complicated. In the second receive statement we would have to list all possible other messages, which could be sent to the database from other clients. But the process may not

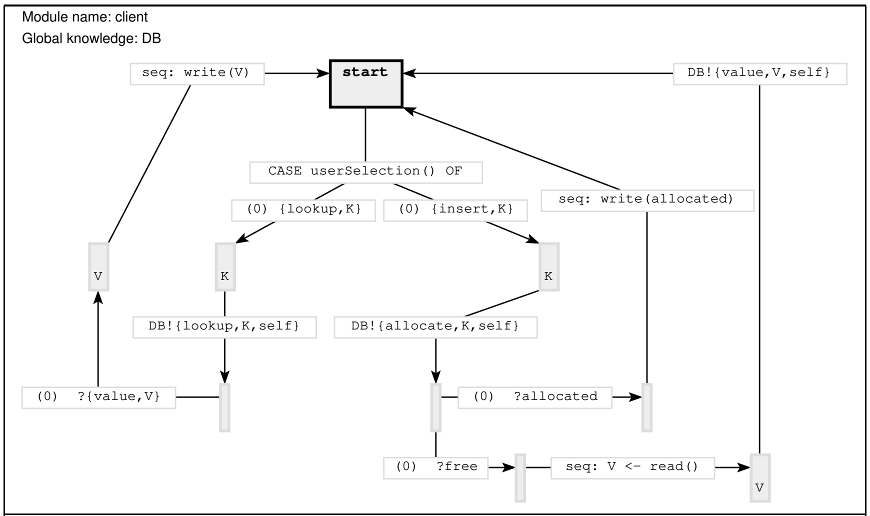

Figure 2

The specification of a client process

ignore these other messages and has to save them in a data structure, which has to be processed later. An alternative would be sending other messages, which do not match the pattern `{value,V,P}`, to the database process itself. But this means busy waiting, which should be impeded.

In Figure 2 we also present the specification of a client process. A client can enroll new entries in the database or lookup keys, in dependence of a users choice. The here presented example is very easy and small and it is no problem to program this directly in Erlang. But programming larger systems with multiple different kinds of processes it is difficult to have an overview of all processes taking part in a protocol. The system helps the developer, because she is restriced to only the communication parts of the protocol and can define the protocol stepwise. She starts with the definition of the sending action starting a special protocol, for example the userSelection in Figure 2 and the resulting `lookup`–message sent to the database. Then she switches to the database and specifies how it reacts. It sends back the corresponding value and succeeds in the initial state. Again the developer switches to the process which will receive the message, the client. After finishing this part of the specification she backtracks and succeeds with the non–finished states. This track through the specification is seconded by a stack, where unfinished states are put and can later be retract and finished.

3 SEMANTICS OF ESM

In the operational semantics multiple processes will be created and every new process will be identified by its own pid. To model these pids we define $Pid := \{@n \mid n \in \mathbb{N}\}$. A state of the operational semantics of an ESM specification is a finite set of processes: $State := \mathcal{P}_{fin}(Proc)$, where a process consist of a pid, a state, the knowledge bound to the variables, and a mailbox: $Proc := Pid \times \bigcup_{p \in \mathcal{P}} S_p \times Know \times MBox$. Mailboxes $MBox := T_C(Pid)^*$ are words over ground terms of an execution. The knowledge of a state is a binding of its variables $Know : Vars \longrightarrow T_C(Pid)$. This knowledge will be updated when a process executes some actions. Therefore we use the function

$$\uplus : Know \times Know \longrightarrow Know$$
$$(\rho_1 \uplus \rho_2)(X) = \begin{cases} \rho_1(X), & \text{if } \rho_1(X) \text{ is defined} \\ \rho_2(X), & \text{otherwise} \end{cases}$$

Finite knowledges, to which we can restrict in the semantics will also be represented as the graph notation of a substitution ($[X_1/v_1, \ldots, X_n/v_n]$).

The operational semantics is denoted by the relation $\Longrightarrow \subset State \times Label \times State$, where $Label$ is the set of actions relevant for concurrency:

$$\begin{aligned} Label := \{\texttt{!}v \mid v \in T_C(Pid)\} \cup \{\texttt{?}v \mid v \in T_C(Pid)\} \cup \\ \{\texttt{spawn(}f\texttt{)} \mid f/n \in FS(p)\} \cup \{\varepsilon\} \end{aligned}$$

We write $s \overset{a}{\Longrightarrow} s'$ for $(s, a, s') \in \Longrightarrow$ and $s \Longrightarrow s'$ for $s \overset{\varepsilon}{\Longrightarrow} s'$.

The semantics of the sequential parts of the specification is defined by an interpretation $\mathcal{A} = (T_C(Pid), \iota)$, where $f_{\mathcal{A}} = \iota(f/n) : T_C(Pid)^n \longrightarrow T_C(Pid)$ interprets the used functions. The semantics of an Erlang term with respect to a local knowledge can then be defined algebraically:

$$\begin{aligned} &\mathcal{E} :: T_\Sigma(Vars) \times Pid \times Know \longrightarrow T_C(Pid) \\ &\mathcal{E}[\![X]\!]^p \rho = \rho(X) \\ &\mathcal{E}[\![\texttt{self}]\!]^p \rho = p \\ &\mathcal{E}[\![\phi(e_1, \ldots, e_n)]\!]^p \rho = \phi_{\mathcal{A}}(\mathcal{E}[\![e_1]\!]^p \rho, \ldots, \mathcal{E}[\![e_n]\!]^p \rho) \end{aligned}$$

The definition of the operational semantics $\Longrightarrow$ is presented in Figure 3.To keep the rules for assignment, case and receiving easily, we use functions match and mailmatch to model Erlang's mechanism of pattern matching. In the rule for assignment the expression is evaluated and matched against the pattern. If this is successful, the knowledge is updated with the corresponding substitution. Compared to other functional languages pattern matching in Erlang is more complicated, because of non–linear patterns with multiple occurrences of the same variable:

$$\begin{aligned} &\text{match}(X,t) = [X/t] \\ &\text{match}(c(pat_1, \ldots, pat_n), c(v_1, \ldots, v_n)) = \\ &\qquad\qquad \text{match}(pat_1, v_1) \oplus \ldots \oplus \text{match}(pat_n, v_n) \\ &\text{match}(\_, \_) = \text{Fail, otherwise} \end{aligned}$$

[2] $\texttt{[}v_1, \ldots, v_n\texttt{]}$ is an abbreviation for $\texttt{[}v_1 \texttt{|} \texttt{[..|} \texttt{[}v_n \texttt{|} \texttt{[]]..]]}$

$$\frac{(s, pat\texttt{<-}e, t) \in \longrightarrow \quad \mathcal{E}[\![e]\!]^p\rho = v \quad match(pat, v) = \rho'}{\Pi, (p, s, \rho, q) \Longrightarrow \Pi, (p, t, \rho' \uplus \rho, q)}$$

$$\frac{(s, \texttt{case } e \texttt{ of } pat_1, t_1) \in \longrightarrow \quad \mathcal{E}[\![e]\!]^p\rho = v \quad \mathrm{match}(\rho(pat), v) = \rho'}{\Pi, (p, s, \rho, q) \Longrightarrow \Pi, (p, t_i, \rho' \uplus \rho, q)}$$

$$\frac{\begin{array}{c}\text{let } (s, \texttt{?}pat_1, t_1), \ldots, (s, \texttt{?}pat_m, t_m) \in \longrightarrow \text{ be all possible actions in } s \\ (i, j, \rho') \in \mathrm{mailboxmatch}((\rho(pat_1), \ldots, \rho(pat_m)), (v, \ldots, v_n))\end{array}}{\Pi, (p, s, \rho, (v_1, \ldots, v_j, \ldots, v_n)) \Longrightarrow \Pi, (p, t_i, \rho' \uplus \rho, (v_1, \ldots, v_{j-1}, v_{j+1}, \ldots, v_n))}$$

$$\frac{(s, X\texttt{!}e, t) \in \longrightarrow \quad \mathcal{E}[\![e]\!]^p\rho = v \quad \rho(X) = p'}{\Pi, (p, s, \rho, q), (p', u, \rho', q') \Longrightarrow \Pi, (p, t, \rho, q), (p', u, \rho', q' : v)}$$

$$\frac{\begin{array}{c}(s, X\texttt{=spawn(}m\texttt{,}i\texttt{,}e\texttt{)}, t) \in \longrightarrow \quad \mathcal{T}_m = (S_m, I_m, \longrightarrow_m) \\ i \in I_m \quad \mathcal{E}[\![e]\!]^p\rho = \texttt{[}v_1\texttt{,} \ldots \texttt{,}v_n\texttt{]}^2 \quad vars(i) = (X_1, \ldots, X_n)\end{array}}{\Pi, (p, s, \rho, q) \Longrightarrow \Pi, (p, t, \rho[X/p'], q), (p', i, [X_1/v_1, \ldots, X_n/v_n], q')}$$

Figure 3: *Operational Semantics*

Two derived substitutions can only be unioned if the overlapping parts are identical. This is tested in the union function $\oplus$, which yields Fail otherwise.

A branch can be chosen, if a pattern matches the semantics of the expression. In contrast to the assignment, the substitution in this rule is also applied to the patterns. Hence a comparison to local values can also be expressed with pattern matching.

The rule for receive actions is a little bit more difficult, but this cannot be impeded, because of the Erlang semantics. We want to allow convenient access to all elements of the mailbox with pattern matching. The elements of the mailbox are successively matched against the patterns. If a value matches one of the patterns it is removed from the mailbox. The sequencing messages do not have to be considered. But our specification can have overlapping patterns. It is possible that multiple patterns match this value. This is formalized by the function mailmatch, which yields a set of triples. Every element consists of the matching position in the patterns, the position of the matching value in the mailbox and the corresponding variable bindings. The position in the mailbox is identical for all yield triples.

$$\begin{aligned}&\mathrm{mailmatch}((pat_1, \ldots, pat_m), (v_1, \ldots, v_u)) \\ &\quad = \{(i, j, \rho) \mid \mathrm{match}(pat_i, v_j) = \rho \neq \mathrm{Fail} \text{ and} \\ &\qquad\qquad \mathrm{match}(pat_k, v_l) = \mathrm{Fail}\ \forall k < i, l < j\}\end{aligned}$$

A value can be sent to another process, if the variable for the destination is bound to a pid. The semantics of the expression, to be sent, is added to the corresponding mailbox. Finally, new processes can be spawned. Here we only present the rule, where the pid of the new process is stored in a variable. the anonymous process generation is defined analogously. In the here presented formal semantics runtime errors are missing. They can occur when a value should be sent to a non–pid, for example a natural number, or a matching fails. For shortness we do not formalize these rules here.

In the tool this semantics is used for the simulation component. The user can execute paths of the operational semantics. Therefore he has to decide two different kinds of non–determinism. First, which process performs a step in the interleaving semantics. Second, which action does a process perform if its behaviour is non–deterministic. The interpretation of the sequential functions cannot be completely given, because this would be their implementation. Hence, the user is asked for their semantics on the actual values, if this is needed. The chosen path is displayed as a message sequent chart and the actual state by coloring the actual states of the processes in their specifications and the corresponding bindings for the variables. The simulation can be used for debugging the protocol. This is better, than a debugging the implementation, because here also unfair paths can be tested. In the implementation the path which can be tested is influenced by the scheduler. A system working correct on one architecture can for example deadlock on another one. To support the tester, it is also possible to backtrack paths and chose other branches or perform random traces. Furthermore the specification is much smaller than its implementation. Finding protocol errors (e.g. deadlocks) is much easier on this abstraction, than in the code. In the specification only the relevant states for the concurrent behavior are considered.

For future work we want to combine this part with the Erlang interpreter, which is able to calculate the expressions, if the occurring functions are already programmed in Erlang. This would lead to a comfortable specification and programming environment. If a function is not defined yet, the user is asked for its interpretation for the concrete values. This behavior is stored in an Erlang module. If the same calculation is needed once again, the function is already defined for these values. Furthermore, this function also yields a (finite) prototype definition for the real implementation of the function. For testing the behavior of this prototype definition can be compared with the implementation of the function.

4 CODE GENERATION FROM THE SPECIFICATION

In [DG86] Dowling and Gallier present a continuation semantics for flow graphs. This continuation semantics defines a system of equations for the semantics of a flow graph which can be interpreted as a tail recursive functional program. Consequently we can give an easy translation from a flow graph into a tail recursive functional program with the same semantics as the flow graph. We can use this translation for the generation of Erlang code (or a skeleton for an Erlang program) from the process specification. The states of the system are represented by functions. The parameters of the functions are determined by the knowledge of the corresponding states. Note that the global knowledge has to be included, because in functional languages global variables do not exist. In Erlang a global registration of processes is possible, but the use of global knowledge could yield conflicts and should be prevented. As a convention we use the first arguments of the functions for the global knowledge.

The body of a function implements the actions of the outgoing arcs and then the function of the destination state is called. If the process graph contains a deterministic sequence this is translate into a sequence of actions instead of defining a function for every state. A state with multiple outgoing receive actions is implemented as one receive statement. The code for the right hand–side of a pattern is the implementation of the destination state of the corresponding receive action or a call to the function of this implementation. Only states which are reached by multiple arcs have to be represented by their own function.

The result is a tail recursive Erlang skeleton in which the sequential parts or parts that have not yet been specified have to be added. Since Erlang optimizes tail recursion, the execution is efficient and there is no stack overflow after some recursive calls as it would happen in imperative languages.

As an example we present the Erlang code for the database. Only the initial state (multiple arcs lead to the initial state) gets a special function to prevent a blow up of the program size.

```
-module(database).
-export([start/1]).

start(DB) ->
   receive
     {allocate,K,P} ->
       case lookup(K,DB) of
         {succ,V} -> P!allocated,
                     start(DB);
         fail -> P!free,
                 receive
                   {value,V,P} ->
                     start(addElem({K,V},DB))
                 end
       end;
     {lookup,K,P} -> P!lookup(K,DB),
                     start(DB)
   end.
```

The code can automatically be generated by the tool and be executed by an Erlang interpreter after adding the called functions for sequential calculations.

5 VERIFICATION

In [Huc99] we presented an approach for the formal verification of Erlang programs using abstract interpretation and model checking. In general it is undecidable if a given program satisfies a property expressed in a temporal logic like linear time logic (LTL) [LP85, Var96], because for example termination can be expressed and an automatic verification contradicts the halting problem.

Although we present a specification formalism in this paper and not a programming language, the same problem appears. As other specification formalisms like SDL and UML our approach is very expressive and model checking for temporal properties in general is undecidable. We have data structures and numbers and can calculate on them.

In [Huc99] we defined a framework for abstract interpretation of Erlang programs, which is based on the standard operational semantics with respect to an abstract interpretation of values, predefined functions and constructor functions. The abstraction is defined in such a manner, that the abstract operational semantics includes all paths of the standard operational semantics, but is under some assumption finite state. A property expressed as an LTL–formula is fulfilled, if all paths of a system fulfill the property. Hence, if the abstract operational semantics fulfills such a property, the property is also proven for the real operational semantics, in other words for the system. Because the abstract operational semantics of the system is finite state under some assumptions, we can use model checking as an automatic proof technique.

We transfered this verification technique to ESM and added it as a component to our tool. In this component the user can define a finite domain abstract interpretation and the operational semantics is interpreted with respect to this abstract interpretation. To understand the abstraction, the user can run the simulation on this abstract domain and she can use it to prove properties of the specified system. If a given specification creates only a finite set of processes at runtime[3] the abstract operational semantics generates a finite transition system and LTL properties can automatically be proven with model checking. If the abstract operational semantics fulfills the property also the system fulfills it. Otherwise a counter example is generated. This path, can either be a counter example for the specified system or an additional path yield because of the additional nondeterminism in the abstraction. The user can inspect this with the simulation and try to refine the abstraction for more determinism.

[3] Also the mailboxes have to be restricted in size. See [Huc99] for details.

A basic abstract interpretation can automatically be generated from a given specification in the tool. Its domain is given by values occurring as patterns in the program. All values that match a pattern are identified by the same value. The parts that differ are represented by ?. For the communication the different pids are essential. Hence pids are also allowed in values of the abstract domain. Values with different pids are distinguished. Additionally an ordering $\preceq$ on this abstract domain is predefined which expresses that an abstract value is more precise than another one. The abstract domain is downwards closed with respect to $\preceq$.

As an example the pattern `{allocate,K,P}` occurs in the specification of the database. In the predefined abstract interpretation the value `{allocate,?,@4}` represents the value `{allocate,[],@4}` as well as `{allocate,4,@4}`. A less precise representation of these values is `{allocate,?,?}` and the worst representation is ? (? $\preceq$ `{allocate,?,?}` $\preceq$ `{allocate,?,@4}`). But during the computation of the operational semantics over the abstract domain, we may lose information of the concrete values, for example when it is stored in a data–structure and took out again. We have to calculate on worth representations.

The functions specifying sequential behaviour yield in this basic abstract interpretation ? for any argument. No information of there behaviour is considered. This basic abstract interpretation can then stepwise be refined. Therefore the user can define the result for relevant arguments.

The logic LTL is defined for unlabeled transition systems with state propositions. The operational semantics defines a labeled transition system. For the verification the user can add propositions to the states, which hold if one of the processes is in this state at runtime. For the propositions we also use Erlang terms, because it is easy to use the known expressions and we can also integrate pids into the propositions.

As an example we can add the following propositions to the specification of the database process:

STATE	PROPOSITION
s1	{allocate,P}
s2	allocated
s3	{set,P}

Then we can prove that a system consisting of a number of clients and a database process guarantees mutual exclusion for writing accesses to the database. This property can be expressed with the following extended LTL formula:

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

This formula says, that if a client p allocates a key, no other client sets a value in state `s3` until p sets it or the key is allocated. For a finite number of processes only a finite number of pids exist. Hence we can translate this extended LTL formula into an LTL formula. With the basic built in abstract interpretation and the LTL model checker this property can automatically be proven. Because the property holds for the abstraction, it also holds for the specified system and also for the implementation, that can be automatically generated. Hence mutual exclusion is also proven for the implemented system.

6 EXPERIENCES AND FUTURE WORK

We presented a specification formalism especially tailored for Erlang. Processes are specified with labeled transition systems. The states are labeled with variables representing the knowledge a process has in these states. The transitions are labeled with actions similar to the concurrency constructs of Erlang or function calls representing sequential behaviour. A main advantage of ESM is that the developed systems are clearly structured. The protocol/communication part is specified with ESM and the purely sequential parts are programmed in a pure functional language. We showed how ESM can be used in practice and how Erlang code can automatically be created. We developed a tool which provides the following components:

- A graphical editor for the development of specifications. The defined specifications are automatically checked for consistency, for example "Is a variable sent to another process known in the previous state?"
- A simulator for testing the behaviour of a specification. There are often some paths in the specification you do not consider in development. The simulation offers all possible steps and the user can find wrong paths, for example a process has terminated, but another one still sends messages to it.
- A code generator, which produces executable Erlang code for the specification. Only the sequential parts of the program have to be programmed, with respect to their textual specification.
- A verification component. The user can define abstract interpretations, which yield finite state transition systems. She can also specify properties in the expressive logic LTL and verify them automatically for the resulting transition system with model checking. If the abstraction fulfills the property also the system and its automatically generated implementation does.

We have not used the tool for the development of larger systems yet. But in two practical course, we saw that the use of ESM helps structuring the developed distributed systems. Two Groups of four students had to develop a communication system where a server provides services like talk, chat, mail and two graphical distributed real–time game. They specified the systems on a blackboard with ESM and did the simulation in a group, where every student behaved as a process. In this simulation many mistakes occurred, because every student tried to damage the behaviour of the whole system. The systems, the students designed and implemented, had an almost professional character although the time they needed for the development was very short. We also developed two distributed games:

- Atomic Bomberman: Up to four players run through a maze and use bombs to get through or fight against the other players.
- Swinx: An arbitrary number of players is connected in a ring. Each player has four seesaws. From the roof different weighted balls can be thrown down with a wagon.

> In dependence of the different weight of the two sides of the seesaws, the balls fly over the board (hopefully to the neighbors). If a column of balls reaches up to the roof, this player has lost. He is eliminated from the ring. To avoid this the balls have different colors. More than three balls of the same color in one row resolve.

To guarantee realtime–requirements, very complex protocols are needed in the implementation of the games. Again we got a clear structure using ESM.

In this practical course we also saw that the ESM specifications are also a good documentation of the systems. The students had to extend older protocolls when new features were added to the communication system. Especially the graphical notation helped to understand the behaviour of the processes and extensions could easily be made.

In the actual implementation the verification component is not efficient enough to be used in the context of larger systems. But it can be used for the verification of smaller subprocesses like the database process. For the verification of larger systems we have to implement partial order reduction [Pel94], as done in SPIN [Hol97]. Another problem with the model checker is that it excepts only LTL formulas. An extension to extend formulas would be interesting. A first approach would be a stupid translation, which yields a combinatorial blow up of the formula. Hence, we want to estimate if the model checking algorithm can profit of the compact structure and verify the property expressed in extended LTL more efficiently.

Another point missing in the tool is the specification of distribution. The user can already specify sending to remote nodes and spawning processes on remote nodes, but Erlang's mechanisms for programming robust applications, like process linking, error handling or monitoring are not provided yet. But for the development of real applications this is needed.

Finally, it would also be interesting to extend our tool to a real development platform for Erlang systems, where parts of the sequential behaviour are only specified and the rest is already implemented. In the simulation this should automatically be considered and the user is only asked for the semantics of a function call, if the corresponding code does not exist. In this environment it would also be possible to change the protocols of a developed system and leave the sequential parts unchanged. Hence, systems could be maintained or extended very easily.

At the moment we are also investigating, if it is possible to generate ESM–specifications from existing Erlang code. This would help to understand the communication structure of existing software. The process specifications are much more compact, than the corresponding source code.

REFERENCES

[Alh98] Sinan Si Alhir. *UML in a Nutshell*. O'Reilly, Sebastapol, CA, 1998.

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

[DG86] W. F. Dowling and J. H. Gallier. Continuation semantics for flowgraph equations. *Theoretical Computer Science*, 44(3):307–331, 1986.

[Hol97] G. 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.

[Huc99] 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).

[ISO89] ISO. ESTELLE: A formal description technique based on an extended state transition model. Technical Report 9074, ISO, 1989. 1989.

[LP85] Orna Lichtenstein and Amir Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 97–107, New Orleans, Louisiana, January 13–16, 1985. ACM SIGACT-SIGPLAN, ACM Press.

[Pel94] D. Peled. Combining partial order reductions with on-the-fly model-checking. *Lecture Notes in Computer Science*, 818:377–390, 1994.

[ST87] R. Saracco and P. A. J. Tilanus. CCITT SDL: Overview of the language and its applications. *Computer Networks and ISDN Systems*, 13:65–74, 1987.

[Var96] M. Y. Vardi. An automata-theoretic approach to linear temporal logic. *Lecture Notes in Computer Science*, 1043:238ff., 1996.

[vEVD89] Peter H. J. van Eijk, Chris A. Vissers, and Michel Diaz. *The Formal Description Technique LOTOS: Results of the ESPRIT/SEDOS Project*. North-Holland, New York, 1989.

A INTRODUCTION TO ERLANG

Erlang [AWV93] is a functional programming language developed by Ericsson. The reduction strategy is call–by–value and Erlang is untyped. Erlang has additional features for concurrent and distributed programming. During the execution of an Erlang program many processes run concurrently. Processes can communicate with each other via asynchronous message passing. Therefore every process is identified by a unique process identifier (*pid*), to which messages can be sent: $pid!v$. Every process has a mailbox, in which all incoming messages are stored. Erlang provides a convenient access for a process to the elements of its mailbox with pattern matching. With the statement

$$\texttt{receive } pat_1 \texttt{->} e_1 ; \ldots ; pat_n \texttt{->} e_n \texttt{end}$$

the elements of the mailbox of a process are stepwise matched against all the patterns $pat_1, \ldots, pat_n$. The first element of the mailbox that matches a pattern pat_i with a substitution ρ is removed from the mailbox. The whole receive statement reduces to $\rho(e_i)$.

This access to the elements of the mailbox with pattern matching makes concurrent programming very easy in Erlang. You can pick out the messages of the mailbox you are interested in. All other messages are stored in the mailbox and can be handled later on. Hence programmers in Erlang do not have to consider all interleaving sequences of incoming messages. Programming gets easier and programs shorter.

New processes can be created with the statement

`spawn(`m`,`f`,[`$v_1, \ldots, v_n$`])`

and the reduction of the newly created process starts with the application $m : f(v_1, \ldots, v_n)$, where f is an exported function of the module m and the arguments $v_1, \ldots, v_n$ are reduced values. The functional result of the `spawn`–statement is the pid of the created process. This pid can be stored in datastructures or sent to other processes. Every process can also access his own pid, with the function `self()`.

For distributed programming Erlang uses the same process concept as for concurrent programming. In a distributed Erlang system multiple nodes exist on maybe multiple computers in a network. On an Erlang node processes can be executed concurrently. To distinguish nodes located on the same computer, every node gets a name when it is started. Communication between processes executed on different nodes stays the same as before. Therefore the pid of a process also contains the node on which it is executed.

The only difference in distributed programming are the creation of processes on other nodes and the communication between two independently started Erlang processes. Erlang has an extended `spawn`–statement

`spawn(`n`,`m`,`f`,[`$v_1, \ldots, v_n$`])`

with the same behaviour as `spawn`, except that the new process is started on the node n, which is the name of the node followed by the symbol @ and the hostname of the computer the node is located on. With this extension it is very easy to distribute processes developed in a concurrent environment, for example to optimize speed of an application or to guarantee fault tolerance to the failure of one node.

Creating processes on different nodes is an expressive mechanism for distributed programming, but some applications are inherent distributed and a hierarchical distribution is not possible. Examples for this are telephony or talk, name–servers or cash dispensers. Here processes are started independently and communication between these processes has to be initiated at runtime. In Erlang this can be realized with global registration of processes on a node. With the statement `register(`*name*`,`*pid*`)` the *pid* is globally registered as *name* on the node where the registration is executed. For the communication between two independently started processes on different nodes we can use the extended send operation `{`*name*`,`*node*`}!`v. The message v is sent to the process, which is registered on *node* as *name*. Usually this kind of communication is only chosen for the first contact. Then the pids are exchanged and all further messages are sent to pids, as in concurrent programming.