Eliminating overlapping of pattern matching when verifying Erlang programs in μCRL

Qiang Guo and John Derrick
Department of Computer Science,
The University of Sheffield,
Regent Court, 211 Portobello Street, S1 4DP, UK
{Q.Guo, J.Derrick}@dcs.shef.ac.uk

September 4, 2006

Abstract

When verifying Erlang programs in the process algebra μCRL specification, if there exists overlapping between patterns in the Erlang source codes, the problem of overlapping in pattern matching occurs when translating the Erlang codes into the μCRL specification. This paper investigates the problem and proposes an approach to overcome it. The proposed method rewrites an Erlang program with overlapping patterns into a counterpart program that has no overlapping patterns. Structure Splitting Trees (SSTs) are defined and applied for pattern evaluation. The use of SSTs guarantees that no overlapping patterns will be introduced into the rewritten Erlang code.

Keywords: Erlang language, μCRL specification, Verification, Translation, Pattern matching, Overlapping, SSTs.

1 Introduction

Formal methods are often used for system design and verification. Formal methods are mathematically based techniques. Their mathematical underpinning allows formal methods to specify systems in a more precise, more consistent and non-ambiguous fashion. Model checking [12] is an automatic formal verification technique that has been widely used in verifying requirements and design for a variety of real-time embedded and safety-critical systems.

When verifying systems using model checking based techniques, specification of the system under development is often modelled by a formal specification language such as the process algebra. A model checker is applied to examine the properties that should hold for the system over a finite state system. If the model fails to satisfy some desired properties, faults are determined to exist in the design.

The advantage of using model checking based techniques for system verification is that, when a fault is detected, the model checker can generate a counter example. These faulty traces help system designers to understand the reasons that cause the occurrence of failures and provide clues for fixing the problem.

Two ways might be considered when using model checking based techniques for system verification. In one way, one can use a specification language in combination with a model checker to obtain a correct specification that is used to write an implementation in a programming language; in the other way, one may take the program code as a starting point and abstracts that into a model that can be checked by a model checker. In the second situation, an interpretation mechanism needs to be defined in order that the source code of a programming language can be translated into the formal specification language used for describing the system under development.

Recently, verification of Erlang programs in the process algebra μCRL specification has been studied [10, 7, 8, 15]. The programming language Erlang [1] is a concurrent functional programming language with explicit support for real-time and fault-tolerant distributed systems. The process

algebra μCRL (micro Common Representation Language) [14] is a formal specification language. It is extended from the process algebra ACP [4] where equational abstract data types [14] are integrated into process specification. When an Erlang program is translated into a μCRL specification, a Labelled Transition System (LTS) can be obtained by using some existing tools such as the CÆSAR/ALDEBARAN Development Package (CADP) [11]. The LTS is used to check the properties that should hold for the system under development.

Benac Earle *et al.* [3, 6] studied the verification of Erlang programs in the process algebra μCRL specification and defined a set of rules for the translation of an Erlang code into μCRL. In their work, translation rules for communication, generic server, supervision tree, functions with side-effects, higher-order functions and pattern matching are defined respectively. They also developed a tool set, *etomcrl*, which automatically translates Erlang codes into a μCRL specification.

However, in the tool set *etomcrl*, pattern matching in an Erlang code are translated in a way where overlapping is not considered. This, however, could cause misinterpretation when translating an Erlang program into the μCRL specification.

In Erlang, evaluation of pattern matching works from top to bottom and from left to right. When a pattern is matched, evaluation is terminated after the corresponding clauses are executed. However, in μCRL, the tool set instantiator does not evaluate rewriting rules in a fixed order. If there exists overlapping between patterns, the problem of overlapping in pattern matching occurs, which could lead to the system being represented by a faulty model. More details about the problem are explained in Section 4.2.

This paper investigated the problem and proposed an approach to overcome it. The proposed method rewrites an Erlang program with overlapping patterns into a counterpart program that has no overlapping patterns. In the counterpart program, functionalities defined in the original program remain unchanged. Structure Splitting Trees (SSTs) are defined and applied for pattern evaluation. The use of SSTs guarantees that no overlapping patterns will be introduced into the rewritten code.

The rest of this paper is organized as follows: Section 2 introduces the Erlang programming language; Section 3 describes the process algebra μCRL; Section 4 discusses the translation of Erlang programs into the process algebra μCRL specification and the problem of overlapping in pattern matching when translating the Erlang programs into μCRL; Section 5 looks at ways to eliminate the problem of overlapping in pattern matching; Section 6 explains the model checking Erlang programs in the μCRL specification with a case study; Conclusions are drawn in Section 7.

2 The Erlang language

The programming language Erlang [1] is a concurrent functional programming language with explicit support for real-time and fault-tolerant distributed systems. Since being developed, it has been used to implement some substantial business critical applications such as the Ericsson AXD 301 high capacity ATM switch [9].

An Erlang program consists of a set of modules, each of which defines a number of functions. A module is uniquely identified by its name as an atom. A function is uniquely identified by the module name, function name and arity (the number of arguments). Two functions with the same name and in the same module, but with different arities are two completely different functions. Functions that are accessible from other modules need to be explicitly declared as *export*. A function named *f_name* in the module *module* and with arity N is often denoted as *module:f_name/N*.

Erlang is a language with light-weight processes. Several concurrent processes can run in the same virtual machine, each of which being called a *node*. Each process has a unique identifier to address the process and a message queue to store the incoming messages. Communication between processes is handled by asynchronous message passing. The receiving process reads the message buffer by a *receive* statement. When reading a message, a process is suspended until a matching message arrives or timeout occurs. A distributed system can be constructed by connecting a number of virtual machines.

An advantage of Erlang is that it uses design patterns (provided by OTP) where a number of generic components are encapsulated. The use of OTP helps to reduce the complexity of system development and testing, while increases the robustness. *Generic server* and *supervisor* are two commonly used generic components in system design. The following briefly reviews these two components.

2.1 Generic server component

The Erlang Open Telecom Platform (OTP) supports a generic implementation of a server by providing the *gen_server* module. The *gen_server* module provides a standard set of interface functions for synchronous and asynchronous communication, debugging support, error and timeout handling, and other administrative tasks. A generic server is implemented by providing a *callback module* where (*callback*) functions are defined specifying the concrete actions of the server such as server state handling and response to messages. When a client wants to synchronously communicate with the server, it calls the standard *gen_server:call* function with a certain message as an argument. If an asynchronous communication is required, the *gen_server:cast* is invoked where no response is expected after a request is sent to the server.

```
-module(client).
-export([start_link/3, init/3]).

start_link(Locker, Resources, Type)- >
    {ok, spawn_link(client, init
        [Locker, Resources, Type])}.

init(Locker, Resources, Type) - >
    loop(Locker, Resources, Type).

loop(Locker, Resources, Type) - >
    gen_server:call(Locker, {request,
            Resources, Type}),
    gen_server:call(Locker, release),
    loop(Locker, Resources, Type).
```

A: Source code of client

```
-module(locker).
-behaviour(gen_server).
-export([start_link/1, init/1]).

start_link(Request) - >
    gen_server:start_link({local, locker},
        locker, [Request], []).

init(Args) - >
    {ok, Args}.
handle_call(request, Client, Pending) - >
    case Pending of
        [ ] - > {reply, ok, [Client]};
        _ - > {noreply, Pending++[Client]}
    end;
handle_call(release, Client, [_| Pending]) - >
    case Pending of
        [ ]- > {reply, done, Pending};
        _ - > gen_server:reply(hd(Pending), ok),
            {reply, done, Pending}
    end;
handle_call(stop, Client, Requests) - >
    {ok, normal, ok, Request}.

terminate(Reason, Requests) - > {ok}.
```

B: Source code of generic server

Figure 1: The source code of Erlang generic server and client.

Figure 1 illustrates a simple server-client system where a client can acquire the lock by sending a *request* message and release it by sending a *release* message. In the example the server might be called with a *request* or a *release* message. If the message is *request* and *Pending* is an empty list, the server returns the client with *ok*, and the server comes to the new state [*Client*]; otherwise, the reply is postponed and the server goes to a new state where the requesting *Client* is added to the end of *Pending* list. If a *release* message is received, the server will send a *reply* to the first waiting caller in the *Pending* list.

A *terminate* function is defined in the call back module. This function is called by the server when it is about to terminate. It allows the server to do any necessary cleaning up. Its return value is ignored.

2.2 Supervisor component

When developing concurrent and distributed systems using Erlang language, a commonly accepted assumption is that any Erlang process may unexpectedly die due to hardware failure or software errors in the code being executed in the process. Erlang/OTP supports fault-tolerance by using the supervision tree design pattern.

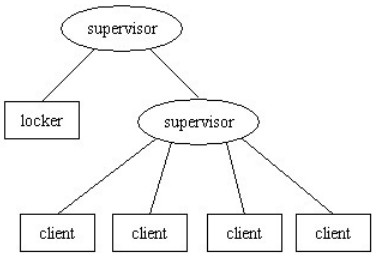

Figure 2: Supervisor tree for locker and clients.

Supervision tree is a structure where the processes in the internal nodes (supervisors) monitor the processes in the external leafs (workers). A supervisor is a process that starts a number of child processes, monitors them, handles termination and stops them on request. The children themselves can also be a supervisor, supervising its children in turn. Figure 2 demonstrates the structure of a supervision tree.

3 The process algebra μCRL

The process algebra μCRL (micro Common Representation Language) [14] is extended from the process algebra ACP [4] where equational *abstract data types* [14] are integrated into process specification.

sort
Bool, N
func
T,F: $\rightarrow$ Bool
0: $\rightarrow$ N
S: N $\rightarrow$ N
add, times : N $\times$ N $\rightarrow$ N
var
x,y : N
rew
add(x,0) = x
add(x,S(y)) = S(add(x,y))
times(x,0) = 0
times(x,S(y)) = add(x, times(x,y))
comm
in|out = com
proc
counter(x:N) = p
buffer = q

Figure 3: An example of a μCRL specification.

A μCRL specification is comprised of two parts: the data types and the processes. Processes are declared using the keyword *proc*. A process may contain actions representing elementary activities that can be performed. These actions must be explicitly declared using the keyword *act*.

Data types used in μCRL are specified as the standard abstract data types, using sorts, functions and axioms. Sorts are declared using the key work *sort*, functions are declared using the keyword *func* and *map* is reserved for additional functions. Axioms are declared using the keyword *rew*, referring to the possibility to use rewriting technology for evaluation of terms.

A number of process-algebraic operators are defined in μCRL, these being: sequential composition ($\cdot$), non-deterministic choice ($+$), parallelism ($\|$) and communication ($|$), encapsulation (∂), hiding

(τ), renaming (ρ) and recursive declarations. A conditional expression $true \lhd condition \rhd false$ enables that data elements influence the course of a process, and an alternative quantification operator ($\sum$) provides the possibly infinite choice over some sorts.

In μCRL, parallel processes communicate via synchronization of actions. The keyword *comm* is reserved for communication specification. The communication specification describes which actions may synchronize on the level of the labels of actions. For example, in *comm in|out*, each action $in(t_1, ..., t_k)$ can communicate with $out(t_1^{'}, ..., t_k^{'})$ provided $k = m$ and t_1 and $t_1^{'}$ denote the same element for $i = 1, ..., k$.

Figure 3 illustrates an example of a μCRL specification.

4 Translating Erlang into μCRL

In order that an Erlang program can be translated into μCRL, Benac Earle *et al.* [3, 6] defined a set of translation rules. In their work, translation rules for communication, generic server, supervision tree, functions with side-effects, higher-order functions and pattern matching are defined respectively. They also developed a tool set, *etomcrl*, which automatically translates Erlang codes into a μCRL specification.

4.1 Translation rules

The translation from Erlang to μCRL is performed in two stages. First, a source to source transformation is applied, resulting in Erlang code that is optimised for the verification, but has identical behaviour. Second, this code is translated to μCRL.

In μCRL, a data type $Term$ is defined where all data types defined in Erlang are embedded. The translation of the Erlang data types to μCRL is then basically a syntactic conversion of constructors as shown in Figure 4.

sort
    Term
func
    pid: Natural → Term
    int: Natural → Term
    nil: → Term
    cons: Term # Term → Term
    tuplenil: Term → Term
    tuple: Term # Term → Term
    true: → Term
    false: → Term

Figure 4: Translation of data types in Erlang to μCRL

Atoms in Erlang are translated to μCRL constructors; *true* and *false* represent the Erlang booleans; *int* is defined for integers; *nil* for the empty list; *cons* for a list with an element (the head) and a rest (the tail); *tuplenil* for a tuple with one element; *tuple* for a tuple with more than one element; and *pid* for process identifiers. For example, a list $[E_1, E_2, ..., E_n]$ is translated to μCRL as $cons(E_1, cons(E_2, cons(..., nil)...))$. A tuple $\{E_1, E_2, ..., E_n\}$ is translated to μCRL as $tuple(E_1, tuple(E_2, ..., tuplenil(E_n)...))$.

Variables in Erlang are mapped directly to variables in μCRL. Operators are also translated directly, specified in a μCRL library. For example, $A + B$ is mapped to *mcrl_plus(A,B)*, where *mcrl_plus(A,B) = int(plus(term_to_nat(A), term_to_nat(B)))*.

High-order functions in an Erlang code are flattened into first-order alternatives. These first-order alternatives are then translated into rewrite rules.

Program transformation is defined to cope with side-effect functions. With a source-to-source transformation, a function with side-effects is either determined as a pure computation or a call to another function with side-effects. *Stacks* are defined in μCRL where *push* and *pop* operations are defined as communication actions. The value of a pure computation is pushed into a stack and is popped when it is called by the function.

Communication between two Erlang processes are translated into two process algebra processes, one of which is defined as a buffer, while the other implements the logic. The synchronous communication is modelled by the synchronizing actions of process algebra. One action pair is defined to synchronize the sender with the buffer of the receiver, while another action pair to synchronize the active receive in the logic part with the buffer. Figure 5 illustrates the translation rules.

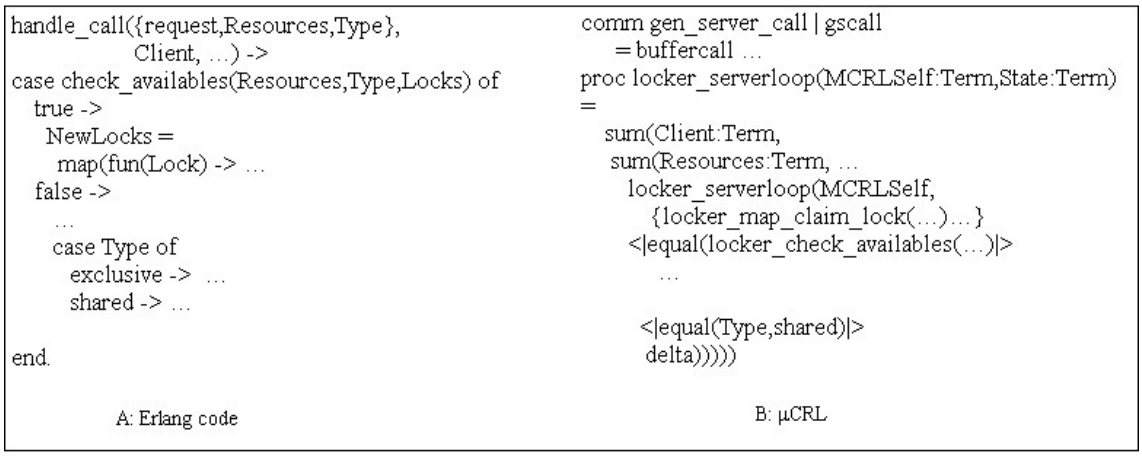

Figure 5: Translation of communication in Erlang to μCRL

4.2 The problem of overlapping in pattern matching

However, in the tool set *etomcrl*, pattern matching in an Erlang code is translated in a way where overlapping is not considered. This could cause misinterpretation when translating an Erlang program into μCRL.

```
-module(check_list).
-export([check/1]).
check(List)− >
    case List of
        [ ] − >
            empty_list;
        [1 | _] − >
            head_check;
        [_ | 2, 3] − >
            tail_check
    end.
```

Figure 6: An Erlang program with overlapping patterns.

In Erlang, evaluation of pattern matching works from top to bottom and from left to right. When a pattern is matched, evaluation terminates after the corresponding clauses are executed.

However, the μCRL tool set instantiator does not evaluate rewriting rules in a fixed order. If there exists overlapping between patterns, the problem of overlapping in pattern matching occurs, which could lead to the system being represented by a faulty model.

Figure 6 illustrates an example where a list is checked. If $List = [1, 2, 3]$, the program returns *head_check* when it is executed, although $List$ matches $[\_|2, 3]$ as well. However, when translating the code into the μCRL specification, the μCRL tool set instantiator does not evaluate rewriting rules in a fixed order. The return value from the μCRL model checker could be either *head_check* or *tail_check*. The final μCRL specification could represent the Erlang program in an incorrect pattern.

To overcome this problem, guards need to be defined and applied in order that rewriting rules are forced to be evaluated in a fixed order.

5 Eliminating overlapping in pattern matching

Two possible ways might be considered for the elimination of overlapping in pattern matching. In one way, one may introduce a set of guards in rewriting rules and force the μCRL tool set instantiator to evaluate rewriting rules in a fixed order, while, in the other way, one may consider to transform the Erlang source codes and rewrite the pattern matching clauses such as *case* into a series of *case_functions*.

5.1 Applying guards in rewriting rules

Benac Earle [6] proposed a method to overcome the problem of overlapping in pattern matching by introducing a set of guards into μCRL.

$$patterns\_match(P, V, \sigma)$$

$$= \begin{cases} \langle true, \sigma, \cup\{P \mapsto V\}, & var(P) \text{ and } P \notin dom(\sigma) \\ \langle equal(V, \sigma(P)), \sigma\rangle, & var(P) \text{ and } P \in dom(\sigma) \\ \langle is\_list(V) \wedge \phi \wedge \psi, \sigma_t\rangle, & P = [H|T] \\ & \langle \phi, \sigma_h\rangle = patterns\_match(H, hd(V), \sigma) \\ & \langle \psi, \sigma_t\rangle = patterns\_match(T, tl(V), \sigma_h) \\ \langle is\_tuple(V) \wedge \phi_1, \sigma_1\rangle & P = \{P1, ..., P_n\} \\ & \langle \phi_1, \sigma_1\rangle = patterns\_match(P_1, element(1, V), \sigma) \\ & \langle \phi_2, \sigma_2\rangle = patterns\_match(P_2, element(2, V), \sigma_1) \\ & \cdots \\ & \langle \phi_n, \sigma_n\rangle = patterns\_match(P_n, element(n, V), \sigma_{n-1}) \\ \langle equal(P, V), \sigma\rangle & otherwise \end{cases}$$

Figure 7: The definition of patterns_match function.

In the proposed method, a *patterns_match* function is defined (see Figure 7). This function has three arguments: a pattern, an expression and a mapping from variables to expressions, and returns a condition and a new mapping. Inside the function, an auxiliary function $var(P)$ is defined to returns the logic value *true* if P is a variable and *false* otherwise. A guard can then be defined in the μCRL specification.

Figure 8 demonstrates the translation rules where function $cond(P, V, \sigma)$ is the projection of $patterns\_match(P, V, \sigma)$ and $\{V_i \mapsto Vi\}$ represents the mapping from $\{V_1, ..., V_n\}$ to $\{V_1, ..., V_n\}$.

Note that in the μCRL specification (Figure 8-B), a *case* function $case_1$ is invoked when the evaluation of pattern matching starts. Here, $case_1$ is functionally equivalent to the first *case* clause in the Erlang code (Figure 8-A). Function $case_1$ calls another function $case_2$ where $cond(Q_1, E)$ is evaluated. If $cond(Q_1, E)$ returns *true*, it indicates that the first pattern is matched. Clause B_1 is executed and the evaluation terminates; otherwise, if $cond(Q_1, E)$ returns $false$, function $case_3$ is called where pattern Q_2 is evaluated. The evaluation continues in such an order until all patterns have been examined. It can be seen that, by introducing a guard *cond*, the μCRL instantiator evaluates the rewriting rules from top to bottom, which is identical to the order by which the patterns are examined in the Erlang code.

However, the proposed method is not applied in the tool set *etomcrl*.

5.2 Rewriting Erlang source code

The other way to overcome the problem is to rewrite the Erlang code before the translation starts. The rewriting operation rewrites all pattern matching clauses in the original code into some calling functions. A calling function is activated by a guard that is determined by function *patterns_match*.

A: Erlang source code.	B: μCRL specification.
$f\_name(V_1, ..., V_n) - >$ case E of $Q_1 - > B_1;$ $\cdots$ $Q_m - > B_m$ end.	$f\_name(V_1, ..., V_n) =$ case1$(V_1, ..., V_n, E)$. $case_1(V_1, ..., V_n, V_{n+1}) =$ $case_2(V_1, ..., V_n, cond(Q_1, V_{n+1}, \{V_i \mapsto V_i\}), V_{n+1})$. $case_2(V_1, ..., V_n, true, Q_1) = B_1;$ $case_2(V_1, ..., V_n, false, V_{n+1}) =$ $case_3(V_1, ..., V_n, cond(Q_2, V_{n+1}, \{V_i \mapsto V_i\}), V_{n+1})$. $\cdots$ $case_N(V_1, ..., V_n, true, Q_m) = B_m$.

Figure 8: Translating Erlang code into μCRL with guards.

Function *patterns_match* takes the predicate of the pattern matching clauses and one pattern as arguments. If the predicate matches the pattern, function *patterns_match* returns *true*; otherwise, *false*.

Figure 9 shows an example where *case* clauses are considered. One can easily extend the method to other pattern matching statements such as if and *when*.

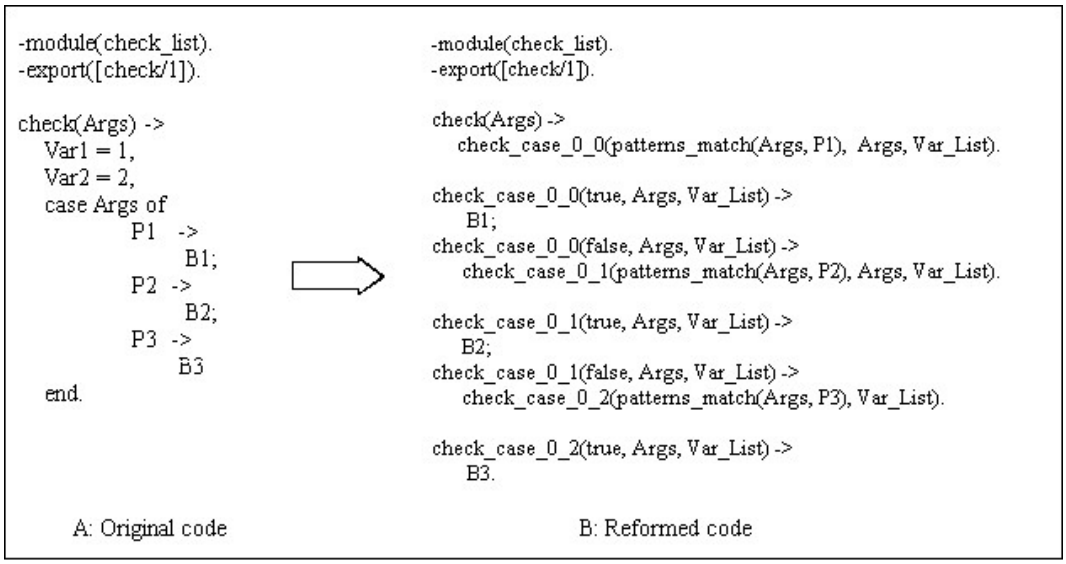

Figure 9: Rewriting the Erlang code.

Given an Erlang code with three patterns for matching, the program (shown in Figure 9-A) is rewritten into the format as shown in Figure 9-B. Function *check* calls function *check_case_0_0*. Function *check_case_0_0* has three arguments. The first argument is the matching result between the predicate $Args$ and the first pattern P_1; the second argument is the predicate $Args$; the last argument is a list of variables. It can be noted that, if $patterns\_match(Args, P_1)$ returns *true*, clauses defined in B_1 are executed; otherwise, function *check_case_0_1* is called where P_2 is evaluated. Var_List contains a list of variables that appears before the *case* clause and has been referred in B_1. Before constructing a case function, an analysis of variable dependency is required for the definition of Var_List. If a variable appears before the case clause and is referred in the clauses of a pattern, it should be added to Var_List. For example, if inside B_1, a clause like $K_1 = 2 \times Var_1$ is defined, one needs to add Var_1 to Var_List when constructing function *check_case_0_0*.

The problem now comes to define function *patterns_match*. The function cannot be simply defined as *case E of P − > true*, as it will either introduce new overlapping patterns or cause exception in the runtime.

```
patterns_match(E, P) − >
    case E of
        P− >
            true;
        _ − >
            false
    end.
```

A: Code with overlapping.

```
patterns_match(E, P)− >
    case E of
        P− >
            true
    end.
```

B: Code that causes exception.

Figure 10: Two faulty ways on defining patterns_match function in Erlang program.

Consider two examples shown in Figure 10, if *patterns_match* is defined as the one shown in Figure 10-A, new overlapping patterns will be introduced into the rewritten Erlang code (the question is, if "_" is not considered as a pattern, can we find a suitable expression of $\overline{P}$ such that $\overline{P} \cap P = \phi$ and $\overline{P} \cup P = \{all\ data\ sets\}$?).

If *patterns_match* is defined as the one shown in Figure 10-B, the code is syntactically correct and no overlapping will be introduced. However, the structure of the program will cause system exception if no pattern is matched, which reveals a semantic mistake of the program transformation.

In order to evaluate patterns effectively, we define a Structure Splitting Tree (SST).

Definition 1 *Let D be a datum of complex type. Let $D_{(1,i)}$ is a member of D and $D_{(2,j)}$ is a member of $D_{(1,i)}$. $D_{(1,i)}$ is called a first degree element of D and $D_{(2,j)}$ a second degree element of D. Let $D_{(n,k)}$ be a first degree element of $D_{(n-1,l)}$, $n \geq 2$, $D_{(n,k)}$ is called a n^{th} degree element of D.*

It can be noted that a datum might contain more than one N^{th} degree elements.

An SST is a dependent tree where a datum of complex type is graphically represented. In an SST, each node is labelled with an ID denoted by $N_{(i,j)}$ where i indicates the layer and j the number of the node. Each node contains a datum. The type of a datum is distinguished by a graphic shape. In this work, *atom* is represented by circle, *list* by square and *tuple* diamond. The tree starts with a root node that contains the complete set of data and terminates at a terminal node where the datum is either an atom or a list that contains a "_" character.

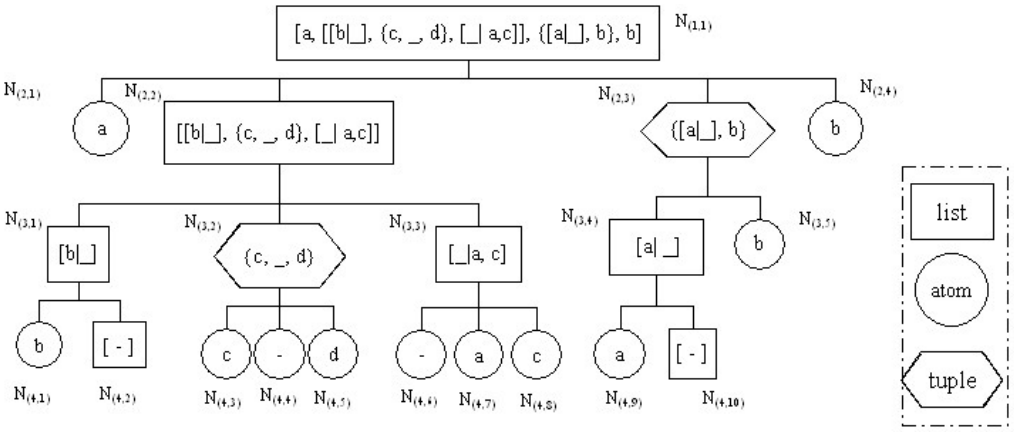

Figure 11: Structure splitting tree of a complex type datum.

Except the root node and the terminal nodes, every node $N_{(i,j)}$ has a parent node $N_{(i-1,g)}$

and several children nodes. The connection between $N_{(i,j)}$ and $N_{(i-1,g)}$ indicates that the datum contained in $N_{(i,j)}$ is a first degree element of $N_{(i-1,j)}$ and an i^{th} degree element of the root node.

Figure 11 illustrates an example where $[a, [[b|\_], \{c, \_, d\}, [\_|a, c]], \{[a|\_], b\}, b]$ is represented by an SST. Note that node $N_{(3,1)}$ contains a list that only the head element is cared. $N_{(3,1)}$ is split into two nodes $N_{(4,1)}$ (contains an atom b) and $N_{(3,2)}$ (contains a list whose elements are not cared), both being terminal nodes.

To check if P_1 matches P_2, one can build the SSTs of P_1 and P_2, and examines the SSTs from top to bottom and from left to right. When a node in one SST is compared with the corresponding node in the other SST, the type of datum is first compared. If two datum types do not match, the evaluation returns *false* and the process of evaluation terminates; otherwise, the two data are further checked. If the datum type is *atom*, and the values of two data are not equal, the evaluation returns *false* and the process of evaluation terminates; otherwise the check of this node is finished and the process of evaluation moves to another node.

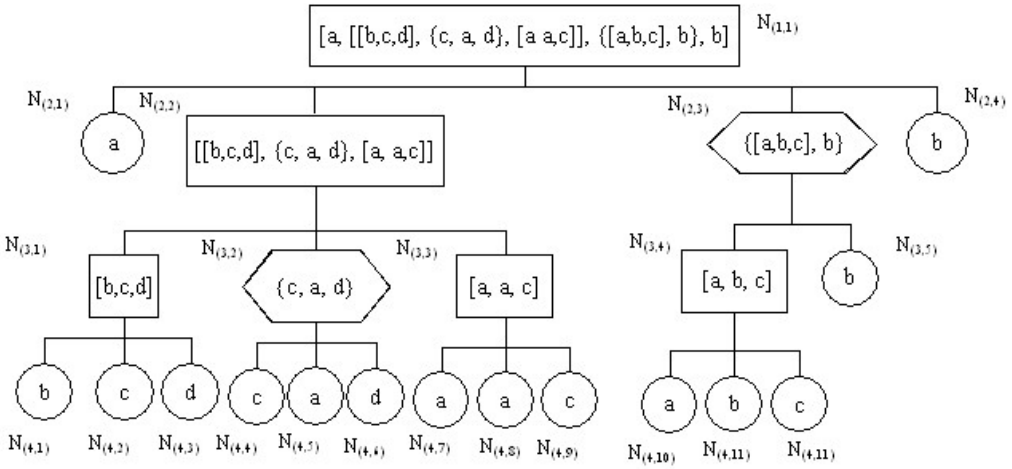

Figure 12: Structure splitting tree of [a,[[b, c, d], {c, a, d}, [a, a, c]], {[a, b, c], b}, b].

For example, to check if $P_1 = [a, [[b, c, d], \{c, a, d\}, [a, a, c]], \{[a, b, c], b\}, b]$ matches $P_2 = [a, [[b|\_], \{c, \_, d\}, [\_|a, c]], \{[a|\_], b\}, b]$, the SSTs of P_1 and P_2 are constructed, shown in Figure 12 and Figure 11 respectively. Nodes in Figure 12 are compared with the corresponding nodes in Figure 11 from top to bottom and from left to right. Root nodes of the SSTs are compared first. It can be seen that both root nodes contain a non-empty list, which indicates that the first layer comparison is matched. The evaluation moves on to the second layer where nodes $N_{(2,1)}$, $N_{(2,2)}$, $N_{(2,3)}$ and $N_{(2,4)}$ in Figure 12 are checked in sequence. $N_{(2,1)}$ contains an atom datum and its value needs to be compared with that of $N_{(2,1)}$ in Figure 11. Once the checking for the second layer is completed, the evaluation moves on to the next layer.

Note that, when evaluation comes to the fourth layer, the checking upon nodes $N_{(4,2)}$ and $N_{(4,3)}$ should be ignored since node $N_{(4,2)}$ in Figure 11 contains a list whose elements match any possible data.

The process of evaluation continues until all nodes in Figure 12 have been examined. It can be seen that, since the evaluation only check datum type (two types are the same or not) and the values of atoms (the values are equal or not), there should be no overlapping in the pattern matching and no exception will be caused during the runtime.

It is easy to see that the problem of evaluating the pattern of an SST is equivalent to that of searching nodes in a tree. A breadth-first search algorithm [5] or a depth-first search algorithm [5] can therefore be applied to solve the problem.

5.3 Comparison between two methods

It is easy to see that the two methods proposed above are functionally equivalent but realize the elimination of overlapping in pattern matching at different stages.

In the tool set *etomcrl*, before translation starts, some pre-processes are made where an Erlang program is transformed into a μErlang program. The structure of the μErlang program is closer to that of μCRL specification, which makes the translation easier.

The first method discussed in Section 5.1 considers the use of guards in the translation rules. The use of guards forces the rewriting rules to be evaluated in a fixed order. The first method copes with the problem at the stage of translation

The second method discussed in Section 5.2 considers the transformation of an Erlang program with overlapping patterns into one without overlapping patterns. Pattern matching clauses in the original code are replaced by a series of case functions. These functions are guarded by a *patterns_match* function. The second method tackles the problem at the stage of pre-process.

In this work, the second method is used as it involves in less effort in modifying the source codes of *etomcrl*.

6 Model checking Erlang in μCRL

Once Erlang programs are translated into a μCRL specification, an LTS can be derived by using some existing tool sets such as CADP. The properties of the system can then be examined through checking all transitions in the LTS.

We took a case study where a simplified version of resource manager is used. The resource manager is based on a real implementation in the control software of the AXD 301 ATM switch. It contains a *locker* and a number of *clients*. Locker provides access to an arbitrary number of resources for an arbitrary number of client processes. The clients may ask access to the resources either in a *shared* way or an *exclusive* way. For more about the resource manager, see [9].

Before translating the Erlang programs into the μCRL specification, some pre-processes are made. All functions that contain overlapping patterns are rewritten as discussed in Section 5.2. We also added an additional function (shown in Figure 6) to the original code. This is intended to evaluate the function *patterns_match* defined in Section 5.

After applying the *etomcrl* tool set to the rewritten codes, a μCRL specification file is obtained. An LTS is then generated by using CADP. Total 120 states and 193 transitions are explored by CADP. Figure 13 shows the LTS derived from the μCRL specification. The checking result implies that the model is correct, which suggests that the method proposed in this paper is capable of coping with the problem of overlapping in pattern matching.

7 Conclusions and future work

When verifying Erlang programs in the process algebra μCRL specification, if there exist overlapping patterns in the Erlang source codes, the problem of overlapping in pattern matching occurs when translating Erlang codes into the process algebra μCRL. The problem is caused due to the fact that the μCRL instantiator does not evaluate rewriting rules in a fixed order. This problem could lead to the Erlang programs being represented by a faulty μCRL model.

This paper investigated the problem and proposed an approach to overcome the problem. The proposed method rewrites an Erlang program with overlapping patterns into a counterpart program that has no overlapping patterns. The functionalities defined in the original program remain unchanged in the counterpart program. SSTs are defined and applied for pattern evaluation. An SST graphically represents a complex datum in a dependent tree.

When evaluating whether pattern P_a matches pattern P_b, the SSTs of P_a and P_b are constructed and compared. If the pattern of P_a's SST is identical to that of P_b' SST, the pattern matching evaluation returns *true;* otherwise, *false*. During the comparison of two SSTs, only the types of complex data and the values of atom data are evaluated. This guarantees that no overlapping pattern will be introduced into the rewritten Erlang codes.

A case study was carried out to evaluate the effectiveness of the proposed method. The evaluation result suggests that the proposed method is capable of coping with the problem of overlapping in pattern matching.

The evaluation of the proposed method in this paper considered the use of a comparatively simple example. More complicated systems are required to experimentally evaluate this method. This, however, remains a research topic in the future work.

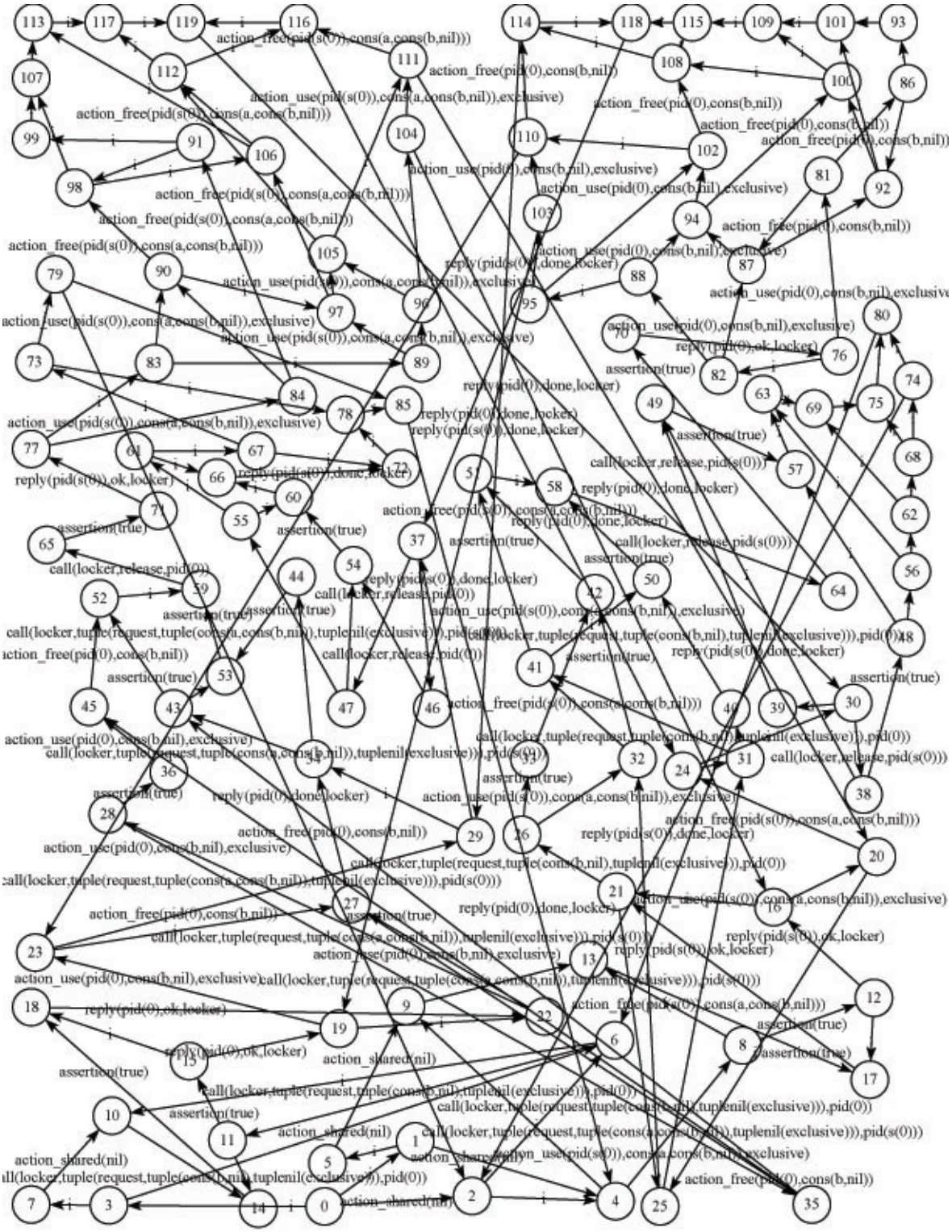

Figure 13: Labelled transition system generated from the locker system

Acknowledgements

We would like to thank Clara Benac Earle for her generous help throughout this work. We would also like to thank the developers of the tool sets of μCRL and CADP for allowing us to use the tool sets for system verification. This work is funded by the Engineering and Physical Sciences Research Council (EPSRC) under grant number EP/C525000/1.

References

[1] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang*. Prentice-Hall, second edition, 1996.

[2] T. Arts, C. Benac Earle, and J. Derrick. Verifying erlang code: A resource locker case-study. In *FME*, pages 184–203, 2002.

[3] T. Arts, C. Benac Earle, and Juan José Sánchez Penas. Translating erlang to μcrl. *Proceedings of the Fourth International Conference on Application of Concurrency to System Design (ACSD'04)*, pages 135–144, 2004.

[4] J.C.M. Baeten and J.A. Bergstra. Process algebra with signals and conditions. Report P9008, University of Amsterdam, 1990.

[5] J. Bang-Jensen and G. Gutin. *Digraphs: Theory Algorithms and Applications*. Springer-Verlag, London, 2001.

[6] C. Benac Earle. *Model check the interaction of Erlang components*. PhD thesis, The University of Kent, Canterbury, Department of Computer Science, 2006.

[7] C. Benac Earle and L. Å. Fredlund. Verification of language based fault-tolerance. In *EUROCAST*, pages 140–149, 2005.

[8] C. Benac Earle, L. Å. Fredlund, and J. Derrick. Verifying fault-tolerant erlang programs. In *Erlang Workshop*, pages 26–34, 2005.

[9] J. Blau, J. Rooth, J. Axell, F. Hellstrand, M. Buhrgard, T. Westin, and G. Wicklund. Axd 301: A new generation atm switching system. *Computer Networks*, 31:559–582, 1999.

[10] J. Blom and B. Jonsson. Automated test generation for industrial erlang applications. In *Erlang Workshop*, pages 8–14, 2003.

[11] CADP. *http://www.inrialpes.fr/vasy/cadp/*.

[12] E. Clarke, O. Grumberg, and D. Long. *Model Checking*. MIT Press, 1999.

[13] L. Å. Fredlund, D. Gurov, T. Noll, M. Dam, T. Arts, and G. Chugunov. A verification tool for erlang. *International Journal on Software Tools for Technology Transfer.*, 4:405–420, 2003.

[14] J. F. Groote and A. Ponse. The syntax and sematics of μcrl. In *Algebra of Communicating Processes 1994, Workshop in Computing*, pages 26–62, 1995.

[15] F. Huch. Verification of erlang programs using abstract interpretation and model checking. *ACM SIGPLAN Notices*, 34(9):261–272, 1999.