Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 238 (2009) 139–154
www.elsevier.com/locate/entcs

Equational Abstractions for Reducing the State Space of Rewrite Theories

Lars Helge Haß[1] and Thomas Noll[2]

Software Modeling and Verification Group
RWTH Aachen University
52056 Aachen, Germany

Abstract

The combinatorial explosion of state spaces is the biggest problem in applying model checking methods to concurrent systems. In this paper we present a new state-space reduction technique that is tailored to system specifications in Rewriting Logic, a unified semantic framework for concurrency which is based on conditional term rewriting modulo equational theories. The idea is to hide "unimportant" details of the system's behavior (such as internal computations) in the equations, and to represent only "interesting" state changes (such as communication operations) by explicit transitions. We show how this optimization can be implemented by transforming the Rewriting Logic specification, avoiding the construction of the full state space. Moreover we establish the correctness of our technique by proving that the original and the reduced system are weakly bisimilar, and demonstrate its usability by applying it to the concurrent functional programming language Erlang.

Keywords: state-space reduction, equational abstraction, weak bisimilarity

1 Introduction

In this paper we address the issue of software verification, concentrating on the first part of the validation procedure, the construction of the (transition-system) model to be checked. This work is carried out in the context of the Rewriting Logic framework as introduced by J. Meseguer in [9] and [10], which has proven to be an adequate modeling formalism for many concrete specification and programming languages [8]. In this approach, the state of a system is represented by an equivalence class of terms modulo a given set of equations, and transitions correspond to rewriting operations on the representatives. Hence Rewriting Logic supports both the definition of programming formalisms and, by employing (equational) term rewriting methods, the execution or simulation of concrete systems.

[1] Email: hass@cs.rwth-aachen.de
[2] Email: noll@cs.rwth-aachen.de

doi:10.1016/j.entcs.2009.05.017

Here we will employ equations to define abstraction mappings which reduce the state space of the system. More concretely the idea is to turn certain transitions which represent "unimportant" details of the system's behavior into equations. Thus these transitions are "hidden" in the current state, generally reducing the number of states, which in turn enables large concurrent systems to become formally verifiable.

Technically the modification of the given Rewriting Logic specification is achieved by choosing a special action label τ, the *silent* or *unobservable* action (which is often used in calculi such as CCS [12] for denoting internal operations), for distinguishing internal computations of the system from those which should be represented by explicit transitions. We will show under which restrictions it is possible to modify the given Rewriting Logic specification by turning τ-transitions into equations in such a way that the resulting transition system is equivalent to the original one. Here equivalence has to be interpreted as weak bisimulation, meaning that each step of the original system can be simulated by a corresponding action of the reduced system and vice versa, where τ-transitions are ignored.

The consequence of this equivalence is that our abstraction is sound and complete with respect to specification formalisms which cannot distinguish weakly bisimilar systems. Examples of temporal logics with this property are $\mathrm{CTL}_{\setminus X}$ and $\mathrm{CTL}^*_{\setminus X}$ where the subscript "$\setminus X$" refers to the respective fragment without the "next" operator (see [3] for details). Here, for every formula of such a logic, the property holds true in the abstract system if and only if it can be guaranteed for the original system. Thus the efficiency of model checking can be improved by considering the abstract instead of the original system.

An important aspect of our work is that the abstraction transformation is not carried out on the actual transition system itself but on its "compact description", the Rewriting Logic specification. This is very different from settings where the complete original systems is required, such as the transformation of nondeterministic to deterministic finite automata by removal of ε-transitions, and generally reduces the peak memory requirements.

The remainder of this paper is organized as follows. Section 2 introduces the Rewriting Logic framework, which is then employed in Section 3 to formalize the operational semantics of the Erlang programming language. In Section 4 we present our abstraction technique, demonstrate its application to Erlang in Section 5, and establish its correctness in Section 6. Finally Section 7 concludes with some remarks.

2 The Rewriting Logic Approach

This section introduces our modeling formalism, Rewriting Logic, which is based on conditional term rewriting modulo equational theories. In order to support the efficient implementation of rewrite theories we will present a variant of the original definition in [10], which has been developed in [6,7].

2.1 Syntax of Labeled Rewrite Theories

Our goal is to model concurrent systems which are composed of (dynamically created) sequential processes. We therefore distinguish between terms that represent single processes and terms that represent concurrent process systems.

Definition 2.1 Let Σ be a signature and $X = X^P \cup X^S$ a set of (process and system) variables. The set of *process terms over* X, denoted by $T^P_\Sigma(X)$, is inductively defined by $X^P \subseteq T^P_\Sigma(X)$, and $f(t_1, \ldots, t_n) \in T^P_\Sigma(X)$ whenever $t_1, \ldots, t_n \in T^P_\Sigma(X)$ and $f \in \Sigma^{(n)}$. Let $\| \notin \Sigma$ moreover be a binary function symbol. The set of *system terms over* X, written $T^S_\Sigma(X)$, is inductively defined by $X^S \cup T^P_\Sigma(X) \subseteq T^S_\Sigma(X)$ and $t_1 \parallel t_2 \in T^S_\Sigma(X)$ whenever $t_1, t_2 \in T^S_\Sigma(X)$.

After these preliminaries we can introduce the syntax of labeled rewrite theories as our modeling formalism.

Definition 2.2 A *labeled rewrite theory (LRT)* is a quadruple $\mathcal{T} = (\Sigma, E, L, R)$ where Σ is a signature, $E \subseteq T^P_\Sigma \times T^P_\Sigma$ is a finite set of (process) equations, L is a finite set of labels, and $R \subseteq \left(T^P_\Sigma(X) \times L \times T^P_\Sigma(X)\right)^+$ is a finite set of (conditional) transition rules, each represented as

$$\frac{c_1 \xrightarrow{\alpha_1} d_1 \ \ldots \ c_k \xrightarrow{\alpha_k} d_k}{l \xrightarrow{\alpha} r}$$

Thus a transition rule expresses how the (sub-)system represented by term l can evolve to r provided that the component processes c_i have respective successor states d_i, for each $1 \leq i \leq k$.

2.2 Semantics of Labeled Rewrite Theories

Reduction systems are the abstract mathematical models by which we represent computations of arbitrary systems. We are interested in particular reduction systems in which certain labels are attached to the reductions to indicate the type of the reduction, and in which an element is distinguished as the initial state.

Definition 2.3 A *labeled transition system (LTS)* is a quadruple $(S, s_0, L, \rightarrow)$ where S is a set of objects called states, $s_0 \in S$ is the initial state, L is a finite set of labels, and $\rightarrow = \bigcup_{\alpha \in L} \xrightarrow{\alpha}$ is a transition relation such that $\xrightarrow{\alpha} \subseteq S \times S$ for every $\alpha \in L$.

Here we assume that the set of labels, L, contains a distinguished element τ. This label indicates a local evaluation step without side effects, that is, a transition which does not "essentially" modify the current state. From now on we will use the symbol a to denote a transition which (possibly) involves side effects, that is, $a \in L \setminus \{\tau\}$.

Based on this definition we can now give the (operational) semantics of labeled rewrite theories in terms of labeled transition systems. It formalizes the understanding that a concurrent system whose current state is represented by the term s (or some E-equivalent thereof) can evolve to the state t provided that there exists

a transition rule whose left-hand side matches a subterm of s modulo E and whose conditions are fulfilled. This intuitive notion can be formally described as follows.

Definition 2.4 The *transition relation of a labeled rewrite theory* $\mathcal{T}$,

$$\rightarrow_{\mathcal{T}} = \bigcup_{\alpha\in L, n\in\mathbb{N}} \xrightarrow{\alpha}_{\mathcal{T}_n} \subseteq T_\Sigma^S(X)/E \times L \times T_\Sigma^S(X)/E,$$

is inductively defined by $\xrightarrow{\alpha}_{\mathcal{T}_0} := \emptyset$ and

$$\begin{aligned}\xrightarrow{\alpha}_{\mathcal{T}_{n+1}} := \{([s]_E, \alpha, [t]_E) \mid \exists \frac{c_1 \xrightarrow{\alpha_1} d_1 \ \ldots \ c_k \xrightarrow{\alpha_k} d_k}{l \xrightarrow{\alpha} r} \in R, w \in Pos(s), \sigma \in Sub \\ \text{such that } s|_w =_E l\sigma, t =_E s[w \leftarrow r\sigma], \\ \text{and } [c_i\sigma]_E \xrightarrow{\alpha_i}_{\mathcal{T}_n} [d_i\sigma]_E \text{ for every } 1 \le i \le k\}.\end{aligned}$$

Here

- $T_\Sigma^S(X)/E := \{[t]_E \mid t \in T_\Sigma^S(X)\}$ is the *quotient* of $T_\Sigma^S(X)$ w.r.t. E,
- $[t]_E := \{s \in T_\Sigma^S(X) \mid s =_E t\}$ denotes the *E-equivalence class* of t,
- $Pos(s)$ is the set of *positions* of s,
- $s|_w$ denotes the *subterm* of s at position $w \in Pos(s)$,
- $s[w \leftarrow t]$ is obtained from s by replacing $s|_w$ by t, and
- $Sub := \{\sigma \mid \sigma : X \rightarrow T_\Sigma^S(X)\}$ is the set of *substitutions*.

Moreover n is called the *depth* of the respective transition.

Note that the transition relation induced by an LRT operates on system terms, and that it is closed under substitutions and contexts.

2.3 Weak Bisimulation

Our final goal is to reduce the size of the transition system which is induced by a given rewrite theory. To guarantee the correctness of the abstraction, the reduced system, in some sense, has to be equivalent to the original one. Here we will require that both systems are weakly bisimilar, meaning that each step of the original system can be simulated by a corresponding action of the reduced system and vice versa, where τ-transitions are ignored.

Definition 2.5 Let $(S, s_0, L, \rightarrow)$ be a LTS and $s, t \in S$. We write $s \overset{\varepsilon}{\Rightarrow} t$ if $s \xrightarrow{\tau}{}^* t$, i.e., if there is a (possibly empty) sequence of τ-labeled transitions leading from s to t. Moreover, for each $\alpha \in L$, we write $s \overset{\alpha}{\Rightarrow} t$ if there exist $s', t' \in S$ such that $s \overset{\varepsilon}{\Rightarrow} s' \xrightarrow{\alpha} t' \overset{\varepsilon}{\Rightarrow} t$. For each label $\alpha \in L$, we mean $\hat{\alpha}$ to stand for ε if $\alpha = \tau$, and for α otherwise.

A binary relation $\mathcal{R} \subseteq S_1 \times S_2$ over the sets of states of two LTSs $A = (S_1, s_1, L, \rightarrow_1)$ and $B = (S_2, s_2, L, \rightarrow_2)$ is a *weak bisimulation* if, whenever $s \mathcal{R} t$ and $\alpha \in L$,

- if $s \xrightarrow{\alpha}_1 s'$, then there exists $t' \in S_2$ such that $t \overset{\hat{\alpha}}{\Rightarrow}_2 t'$ and $s' \mathcal{R} t'$, and
- if $t \xrightarrow{\alpha}_2 t'$, then there exists $s' \in S_1$ such that $s \overset{\hat{\alpha}}{\Rightarrow}_1 s'$ and $s' \mathcal{R} t'$.

Two states $s, t \in S$ are *weakly bisimilar*, written $s \approx t$, if there is a weak bisimulation $\mathcal{R}$ such that $s \mathcal{R} t$. The LTSs A and B with initial states s_1 and s_2, respectively, are *weakly bisimilar* if $s_1 \approx s_2$.

We note that weak bisimilarity is an equivalence relation [12].

3 Modeling the Erlang Programming Language

In this section we present the application of the former framework to the programming language Erlang, which is mainly employed in telecom applications.

3.1 The Erlang Programming Language

Erlang/OTP is a programming platform providing the functionality for programming open distributed systems: the language Erlang with support for concurrency, and the OTP (Open Telecom Platform) middleware providing ready-to-use components and services such as, e.g., a distributed data base manager, support for "hot code replacement", and design guidelines for using the components. For a complete description see [1].

In the following we consider a core fragment of the Erlang programming language which supports the implementation of dynamic networks of processes operating on data types such as atomic constants (atoms), integers, lists, tuples, and process identifiers (pids), using asynchronous, call-by-value communication via unbounded ordered message queues called mailboxes. Real Erlang has several additional features such as modules, distribution of processes (onto nodes), and support for robust programming and for interoperation with non-Erlang code written in, e.g., C or Java.

Besides Erlang expressions e we operate with the syntactical categories of matching clauses cs, patterns p, and values v. The abstract syntax of Erlang expressions is summarized as follows:

$$\begin{aligned} e &::= e_1, e_2 \mid e(e_1, \ldots, e_n) \mid \textbf{case } e \textbf{ of } cs \textbf{ end} \mid p = e \mid e_1 ! e_2 \\ &\quad \mid \textbf{receive } cs \textbf{ end} \mid op(e_1, \ldots, e_n) \mid \textbf{spawn}(e_1, e_2) \mid \textbf{self}() \mid X \\ cs &::= p_1 \to e_1; \ldots; p_n \to e_n \\ p &::= op(p_1, \ldots, p_n) \mid X \\ v &::= op(v_1, \ldots, v_n) \end{aligned}$$

Here X ranges over Erlang variables, and op ranges over a set of primitive constants and operations including tupling $\{e_1, e_2\}$, list prefix $[e_1|e_2]$, the empty list $[]$, integers, pid constants, and atoms.

The functional sublanguage of Erlang is rather standard: atoms, integers, lists and tuples are value constructors; e_1, e_2 denotes sequential composition; and a function call is represented by $e(e_1, \ldots, e_n)$. An expression of the form **case** e **of** cs **end** involves matching: the value that e evaluates to is matched sequentially against the patterns p_i. If this succeeds, evaluation continues with e_i where the variables bound by p_i are correspondingly instantiated, otherwise a runtime error is raised. The same is true for the assignment $p = e$ where a runtime error is raised if the

```
-module locker
-export([start/0]).

locker() ->
    receive
        {request, Client} ->
            Client!ok,
            receive
                {release, Client} ->
                        locker()
            end
    end.

start() ->
    Locker = spawn(locker, []),
    spawn(client, [Locker]),
    spawn(client, [Locker]).

client(Locker) ->
    Locker!{request, self()},
        receive
            ok ->
                % critical section
                Locker!{release, self()},
                client(Locker)
        end.
```

Fig. 1. A Resource Locker in Erlang

value of e does not match p, and where this value is returned as the result otherwise.

The constructs involving non-functional behavior (i.e., side effects) are $e_1!e_2$ which denotes an output operation, sending the value of e_2 asynchronously to the process identified by e_1, whereas **receive** cs **end** inspects the mailbox q of the local process and retrieves (and removes) the first element in q that matches any pattern in cs. Once such an element v has been found, evaluation continues analogously to **case** v **of** cs **end**; otherwise, the process waits for a corresponding message to arrive. The expression **spawn**(e_1, e_2) dynamically creates a new process in which the function given by e_1 is applied to the arguments given by the list e_2 (and returns the pid of the new process), and **self**() returns the pid of the local process.

As an introductory example we consider a short Erlang program which implements a simple resource locker, i.e., an arbiter which, upon receiving corresponding requests from client processes, grants access to a single resource. It is given in Figure 1. An extended version of the algorithm is presented in [2].

Any Erlang program consists of a set of modules. Each module basically contains a list of function declarations. In our example the system is defined in one module. It is initialized using the start function, which, according to the export declaration, is the only function accessible from outside the locker module. By calling the **spawn** function, it generates three new processes: one locker and two clients.

The locker process runs the locker function in a non-terminating loop. It employs the **receive** construct to check whether a request message has arrived. The latter is expected to be a pair composed of a request tag and a client process identifier (which is matched by the variable Client). The client is then granted access to the resource by sending an ok flag. Finally, after receiving the release message from the respective client, the locker returns to its initial state.

A client process exhibits the complementary behavior. By issuing a request, it demands access to the resource. Here, the **self** builtin function returns the process identifier of the client process, which is then used by the locker process as a handle to the client. After receiving the ok message it accesses the resource, and releases

it afterwards.

3.2 A Formal Semantics of Erlang

In this section we present an overview of the formal semantics of Erlang. For details see [5,14].

The Erlang runtime system maintains a set of user processes. Any such process consists of three components: an Erlang expression which has to be evaluated, a process identifier which uniquely identifies the respective process, and which is internally determined by the system, and a mailbox for incoming messages, which is essentially a list of Erlang values. Moreover we will attribute a current evaluation environment to a process, which stores the bindings between the Erlang variables and the values assigned to them.

As before we will use certain standard denotations for the Rewriting Logic variables occurring in the rules, possibly in indexed or primed form. We let e denote an Erlang expression, p a pattern, X an Erlang variable, a an atom, v a value, f a function name, and c a clause. Moreover α refers to a transition label, cs to a list of clauses, i, j, k to pids, q to a mailbox, ρ to an environment, and s to a concurrent process system. Thus a single process is represented by a term of the form $\langle e \mid i \mid q \mid \rho \rangle$.

3.2.1 The Equational Theory

The next step involves the definition of the set of equations, E, of our rewrite theory. We only declare the parallel operator $\parallel$ to be associative and commutative: $E := \{s_1 \parallel (s_2 \parallel s_3) = (s_1 \parallel s_2) \parallel s_3 \ , \ s_1 \parallel s_2 = s_2 \parallel s_1\}$.

3.2.2 The Transition Rules

The most important part of our definition is the formalization of the operational behavior of Erlang process systems by conditional transition rules. To obtain a cleaner structure we decompose R into two disjoint subsets: $R := R^P \cup R^S$. Here, R^P contains the so-called *process-level rules* which operate on single processes while R^S, the set of *system-level rules*, deals with concurrent process systems. In the following we present some examples from both categories, again referring to [14] for the complete definition.

Process-Level Rules

The first rule describes the recursive evaluation of lists. Due to the leftmost-innermost evaluation strategy of Erlang we have to start the evaluation with the first expression in the list constructor.

$$(\text{list}_1)\ \frac{\langle e_1 \mid i \mid q \mid \rho \rangle \xrightarrow{\alpha} \langle e_1' \mid i \mid q' \mid \rho' \rangle}{\langle [e_1 \mid e_2] \mid i \mid q \mid \rho \rangle \xrightarrow{\alpha} \langle [e_1' | e_2] \mid i \mid q' \mid \rho' \rangle}$$

As soon as the first subexpression of the list constructor is irreducible (i.e., a value), evaluation proceeds with the second subexpression:

$$(\text{list}_2)\frac{\langle e \mid i \mid q \mid \rho\rangle \xrightarrow{\alpha} \langle e' \mid i \mid q' \mid \rho'\rangle}{\langle [v|e] \mid i \mid q \mid \rho\rangle \xrightarrow{\alpha} \langle [v|e'] \mid i \mid q' \mid \rho'\rangle}$$

The next rules formalize the behavior of the sequencing construct. The evaluation starts with the first subexpression and then proceeds to the second, ignoring the result of the first.

$$(\text{seq}_1)\frac{\langle e_1 \mid i \mid q \mid \rho\rangle \xrightarrow{\alpha} \langle e_1' \mid i \mid q' \mid \rho'\rangle}{\langle e_1, e_2 \mid i \mid q \mid \rho\rangle \xrightarrow{\alpha} \langle e_1', e_2 \mid i \mid q' \mid \rho'\rangle}$$

$$(\text{seq}_2)\frac{}{\langle v, e \mid i \mid q \mid \rho\rangle \xrightarrow{\tau} \langle e \mid i \mid q \mid \rho\rangle}$$

The following rules deal with pattern-matching operations. Here we just formalize the **case** construct.

$$(\text{case}_1)\frac{\langle e \mid i \mid q \mid \rho\rangle \xrightarrow{\alpha} \langle e' \mid i \mid q' \mid \rho'\rangle}{\langle \textbf{case } e \textbf{ of } cs \textbf{ end} \mid i \mid q \mid \rho\rangle \xrightarrow{\alpha} \langle \textbf{case } e' \textbf{ of } cs \textbf{ end} \mid i \mid q' \mid \rho'\rangle}$$

$$(\text{case}_2)\frac{match(v, cs, \rho) \xrightarrow{\tau} (e, \rho')}{\langle \textbf{case } v \textbf{ of } cs \textbf{ end} \mid i \mid q \mid \rho\rangle \xrightarrow{\tau} \langle e \mid i \mid q \mid \rho'\rangle}$$

Here $match(v, cs, \rho)$ tests whether one of the clauses in cs matches the value v, and if successful returns both the instantiated right-hand side expression and the modified environment.

Next we consider one of the Erlang builtin functions which evoke side effects on the system level of the semantics:

$$(\text{spawn})\frac{}{\langle \textbf{spawn}(a, v) \mid i \mid q \mid \rho\rangle \overset{spawn(a,v,j)}{\longrightarrow} \langle j \mid i \mid q \mid \rho\rangle}$$

Here j denotes a fresh pid which uniquely identifies the new process, and which is returned as the result of the call of **spawn**. This can be formalized by introducing a "next pid" component in the system states; details can be found in [13,14].

Finally the following rule handles one of the central concepts of Erlang: asynchronous sending of messages. As we shall see the message will be appended to the mailbox of the target process. Note that a process can also send a message to itself.

$$(\text{send})\frac{}{\langle j!v \mid i \mid q \mid \rho\rangle \overset{msg(j,v)}{\longrightarrow} \langle v \mid i \mid q \mid \rho\rangle}$$

System-Level Rules

The first rule just expresses that if a single process in a concurrent system performs a computation step then so does the complete system.

$$(\text{Silent})\frac{\langle e \mid i \mid q \mid \rho\rangle \xrightarrow{\tau} \langle e' \mid i \mid q' \mid \rho'\rangle}{\langle e \mid i \mid q \mid \rho\rangle \parallel s \xrightarrow{\tau} \langle e' \mid i \mid q' \mid \rho'\rangle \parallel s}$$

Process generation is formalized as follows. As can be seen from rule (spawn), the **spawn** builtin function comes with two arguments: a function atom, and a list

of arguments. The new process will call this function with these arguments, starting with the empty mailbox and the empty environment.

$$\text{(Spawn)}\frac{\langle e \mid i \mid q \mid \rho\rangle \overset{spawn(a,v,j)}{\longrightarrow} \langle e' \mid i \mid q \mid \rho\rangle}{\langle e \mid i \mid q \mid \rho\rangle \parallel s \overset{\tau}{\to} \langle e' \mid i \mid q \mid \rho\rangle \parallel s \parallel \langle a(v) \mid j \mid \varepsilon \mid \varepsilon\rangle}$$

Next we specify how a message is stored in the mailbox of the receiving process.

$$\text{(Com)}\frac{\langle e_1 \mid i \mid q_1 \mid \rho_1\rangle \overset{msg(j,v)}{\longrightarrow} \langle e_1' \mid i \mid q_1' \mid \rho_1'\rangle}{\langle e_1 \mid i \mid q_1 \mid \rho_1\rangle \parallel \langle e_2 \mid j \mid q_2 \mid \rho_2\rangle \parallel s \overset{\tau}{\to} \langle e_1' \mid i \mid q_1' \mid \rho_1'\rangle \parallel \langle e_2 \mid j \mid q_2 \cdot v \mid \rho_2\rangle \parallel s}$$

Many more rules are required to complete the definition of our rewrite theory for Erlang. Together with an initial expression e_0, it induces the LTS $(S, s_0, L, \to)$ where s_0 is the initial state given by $s_0 = [\langle e_0 \mid i_0 \mid \varepsilon \mid \varepsilon\rangle]_E$ for some initial pid i_0.

4 Equational Abstraction

In the following we describe how to use equational abstractions to reduce the state space of the transition system. The idea is to "hide" computation steps which are defined by unconditional τ-rules because those do not involve side effects which could be interesting to observe. An example is rule (seq_2) from our Erlang specification in the previous section, which merely discards the first expression in a sequence as soon as it has been evaluated to normal form. Another example is the invocation of a function by replacing the call with its body.

Of course it would be possible to apply this abstraction on the original transition system, similar to transforming of nondeterministic to deterministic finite automata by removing ε-transitions. This, however, would imply that we have to compute the complete LTS before, meaning that the peak memory requirements are not improved.

Here we follow an alternative approach by modifying the Rewriting Logic specification instead such that the reduced LTS can be directly computed. Technically this will be achieved by moving unconditional τ-rules from the set of transition rules to the equational theory. This simple idea yields the following *ad hoc* definition.

Let $\mathcal{T} = (\Sigma, E, R, L)$ be a LRT. The *modified LRT* $\mathcal{T}' = (\Sigma, E', R', L)$ is obtained by shifting rules from R to E as follows:

$$R' := R \setminus \left\{ \overline{l \overset{\tau}{\to} r} \in R \mid l, r \in T_\Sigma^P(X) \right\}$$
$$E' := E \cup \left\{ (l, r) \mid \overline{l \overset{\tau}{\to} r} \in R, l, r \in T_\Sigma^P(X) \right\}$$

However this simple modification is not correct in the sense that the induced LTSs of the original and of the modified LRT are generally weakly bisimilar. This is due to the following two problems.

Nondeterminism: We assume an LRT $\mathcal{T} = (\Sigma, E, R, L)$ with $E = \emptyset$, and with the following transition rules: $R = \left\{ \overline{b \overset{\tau}{\to} c}\ , \overline{b \overset{\tau}{\to} d}\ , \overline{c \overset{a}{\to} d} \right\}$.

In the modified LRT $\mathcal{T}'$, $b =_{E'} c =_{E'} d$ and $\overline{c \overset{a}{\to} d} \in R'$. Hence there is

an infinite sequence of a-transitions of the form $[b]_{E'} \xrightarrow{a}_{\mathcal{T}'} [b]_{E'} \xrightarrow{a}_{\mathcal{T}'} \ldots$ In the original LRT, however, every computation is terminating: $[b]_E \xrightarrow{\tau}_{\mathcal{T}} [c]_E \xrightarrow{a}_{\mathcal{T}} [d]_E$ and $[b]_E \xrightarrow{\tau}_{\mathcal{T}} [d]_E$. Thus the two transition systems are not weakly bisimilar.

τ-rules in conditions: Let $\mathcal{T} = (\Sigma, E, R, L)$ with $E = \emptyset$, and with the following transition rules:

$$R = \left\{ \frac{x_1 \xrightarrow{\tau} y_1 \;\; x_2 \xrightarrow{a} y_2}{h(x_1, x_2) \xrightarrow{a} h(y_1, y_2)}, \frac{}{b \xrightarrow{\tau} c}, \frac{}{d \xrightarrow{a} e} \right\}.$$

Then $[h(b,d)]_E \xrightarrow{a}_{\mathcal{T}} [h(c,e)]_E$, but $[h(b,d)]_{E'}$ does not have an a-successor in $\mathcal{T}'$.

It is clear that the first problem, which is caused by the possible choice between an unconditional τ-transition and some other transition, cannot be solved by a simple construction. We therefore proceed by restricting the LRTs under consideration, assuming all unconditional τ-rules to be deterministic in the following sense.

Definition 4.1 A labeled rewrite theory $\mathcal{T} = (\Sigma, E, L, R)$ is called *restricted* if it satisfies the following conditions:

- if $\overline{l_1 \xrightarrow{\tau} r_1} \in R$, then there exist no $\dfrac{C}{l_2 \xrightarrow{\alpha} r_2} \in R$ and $w \in Pos(l_2)$ such that $l_1 =_E l_2|_w$, and
- all transition rules on system level are of the form $\dfrac{l \xrightarrow{\tau} r}{l \parallel s \xrightarrow{\tau} r \parallel s} \in R$ where $s \in T^S_\Sigma(X)$, or $\dfrac{l \xrightarrow{a} r}{l \parallel s \xrightarrow{a} r \parallel s'} \in R$ where $s, s' \in T^S_\Sigma(X)$.

We note that it is decidable whether a given LRT is restricted or not, provided that the equational theory is decidable.

The second problem, the presence of τ-rules in conditions, can be solved by changing the construction of the modified LRT. The idea is the following: whenever a condition can be satisfied by applying an unconditional τ-rule, we add a new rule which is obtained by dropping this condition under the appropriate instantiation. This can be formalized as follows.

Definition 4.2 Let $\mathcal{T} = (\Sigma, E, R, L)$ be a restricted labeled rewrite theory. Its *modification* $\mathcal{T}' = (\Sigma, E', R', L)$ is obtained by applying the following algorithm:

$R' := R; E' := E;$

for all $\overline{l_1 \xrightarrow{\tau} r_1} \in R$ with $l_1, r_1 \in T^P_\Sigma(X)$ **do**

$R' := R' \setminus \left\{ \overline{l_1 \xrightarrow{\tau} r_1} \right\};$

$E' := E' \cup \{(l_1, r_1)\};$

for all $\dfrac{c_1 \xrightarrow{\alpha_1} d_1 \ \ldots \ c_k \xrightarrow{\alpha_k} d_k}{l_2 \xrightarrow{\alpha} r_2} \in R', 1 \leq i \leq k$ and $\sigma \in Sub$

such that $l_1\sigma =_E c_i\sigma$ and $r_1\sigma =_E d_i\sigma$ **do**

$$R' := R' \cup \left\{ \frac{\ldots \ c_{i-1}\sigma \xrightarrow{\alpha_{i-1}} d_{i-1}\sigma \ c_{i+1}\sigma \xrightarrow{\alpha_{i+1}} d_{i+1}\sigma \ \ldots}{l_2\sigma \xrightarrow{\alpha} r_2\sigma} \right\}$$

end for

end for

Obviously the algorithm terminates for all LRTs as the number of unconditional τ-rules in R is finite. If this number is denoted by n and if m denotes the maximal number of conditions in R-rules, then the size of the modified set of transition rules, R', is quadratically bounded:

$$|R'| \leq |R| - n + (|R| - n)mn \leq m|R|^2.$$

Note that the transformation possibly creates new unconditional τ-rules in R', in which case it can be applied iteratively. The overall correctness is then guaranteed since, as we will see in Section 6, the LTSs of the modified rewrite theory is weakly bisimilar to the original one, and since weak bisimilarity is an equivalence (and thus transitive).

5 Application to Erlang

Before studying the effect of hiding unconditional τ-rules we have to show that the LRT for Erlang is restricted. But this is straightforward since unconditional τ-rules are deterministic and all rules on the system level have the required form.

We now demonstrate the construction of the modified LRT for Erlang, concentrating on the rules for lists and for sequential evaluation which were developed in Section 3:

$$(\text{list}_1)\ \frac{\langle e_1 \mid i \mid q \mid \rho\rangle \xrightarrow{\alpha} \langle e_1' \mid i \mid q' \mid \rho'\rangle}{\langle [e_1 \mid e_2] \mid i \mid q \mid \rho\rangle \xrightarrow{\alpha} \langle [e_1'|e_2] \mid i \mid q' \mid \rho'\rangle}$$

$$(\text{seq}_2)\ \frac{}{\langle v, e \mid i \mid q \mid \rho\rangle \xrightarrow{\tau} \langle e \mid i \mid q \mid \rho\rangle}$$

The construction in Def. 4.2 then removes (seq_2) from the set of transition rules, adds it to the equational theory, and yields the following modification of (list_1):

$$(\text{list}_1')\ \frac{}{\langle [(v, e_1)|e_2] \mid i \mid q \mid \rho\rangle \xrightarrow{\tau} \langle [e_1|e_2] \mid i \mid q \mid \rho\rangle}$$

Locker with	1 client	2 clients	3 clients
Original LTS	78 states	1014 states	13182 states
Abstract LTS	16 states	64 states	256 states

Table 1
Original vs. Abstract Locker System

Applying this modification procedure iteratively to the complete Erlang specification, we obtain a modified LRT which induces the abstract LTS. Adapting our prototype implementation of (Core) Erlang in Maude [13], we can now compare the size of the state space of the original and abstracted LTS for a varying number of client processes. The corresponding figures are given in Table 1.[3]

We see that the equational abstraction reduces the state space by orders of magnitude, implying that typical verification problems such as mutual exclusion, the absence of deadlocks, and starvation freedom can be analyzed more efficiently. The following section shows that this approach is correct: the original and the abstract system can be shown to be weakly bisimilar. This means, e.g., that for deciding the model-checking problem for logics which do not distinguish weakly bisimilar systems it suffices to consider the abstract system.

6 Correctness Proof

We can now show that the LTS which is induced by a (restricted) LRT and the LTS which is induced by the modification of the former are weakly bisimilar. To this aim we first prove two intermediate results, which only deal with single processes. The first one expresses that every transition in the original system which is not obtained via an unconditional τ-rule is also present in the abstract system.

Lemma 6.1 *Let $\mathcal{T} = (\Sigma, E, R, L)$ be a restricted LRT, $\mathcal{T}' = (\Sigma, E', R', L)$ the corresponding modified LRT, $s, t \in T_\Sigma^P(X)$, and $\alpha \in L$. If $[s]_E \xrightarrow{\alpha}_{\mathcal{T}} [t]_E$ and $s \neq_{E' \setminus E} t$, then $[s]_{E'} \xrightarrow{\alpha}_{\mathcal{T}'} [t]_{E'}$.*

Proof. We prove the claim by induction over the depth of the transition in the original LRT $\mathcal{T}$ according to Def. 2.4.

$n = 0$**:** clear since $\rightarrow_{\mathcal{T}_0} = \emptyset$

$n \rightarrow n+1$**:** let $[s]_E \xrightarrow{\alpha}_{\mathcal{T}_{n+1}} [t]_E$ and $s \neq_{E' \setminus E} t$. According to Def. 2.4, there exist $\dfrac{c_1 \xrightarrow{\alpha_1} d_1 \ \ldots \ c_k \xrightarrow{\alpha_k} d_k}{l \xrightarrow{\alpha} r} \in R$, $w \in Pos(s)$ and $\sigma \in Sub$ such that $s|_w =_E l\sigma$, $t =_E s[w \leftarrow r\sigma]$, and $[c_i]_E \xrightarrow{\alpha_i}_{\mathcal{T}_n} [d_i]_E$ for every $1 \leq i \leq k$.

We distinguish the following cases:

(i) $\alpha = \tau$: here $k \geq 1$ since otherwise $s =_{E' \setminus E} t$. Now the analysis of the

[3] The numbers differ from those obtained in [15] since the latter does not consider the core language, and since it employs the ELAN rewriting tool.

conditions $[c_i]_E \xrightarrow{\alpha_i}_{\mathcal{T}_n} [d_i]_E$ for $1 \leq i \leq k$ yields:

- if $c_i \neq_{E'\setminus E} d_i$, then $[c_i]_E \xrightarrow{\alpha}_{\mathcal{T}'_n} [d_i]_E$ by induction hypothesis.
- if $c_i =_{E'\setminus E} d_i$, then by construction of the modified LRT according to Def. 4.2 there exists $\dfrac{C}{l\sigma \xrightarrow{\alpha} r\sigma} \in R'$ where C is obtained by dropping $c_i \xrightarrow{\tau} d_i$, which is therefore satisfied. Thus $[s]_E \xrightarrow{\alpha}_{\mathcal{T}'_{n+1}} [t]_E$.

(ii) $\alpha \neq \tau$: here $s \neq_{E'\setminus E} t$ since otherwise there exists $\overline{l' \xrightarrow{\tau} r'} \in R$, $w \in Pos(s)$ and $\sigma \in Sub$ such that $s|_w =_E l'\sigma$ and $t =_E s[w \leftarrow r'\sigma]$. This, however, contradicts the first restriction in Def. 4.1. Thus $[s]_E \xrightarrow{\alpha}_{\mathcal{T}'_{n+1}} [t]_E$ by induction hypothesis.

□

The next lemma claims that every transition in the modified LRT corresponds to a transition in the original LRT.

Lemma 6.2 *Let $\mathcal{T} = (\Sigma, E, R, L)$ be a restricted LRT, $\mathcal{T}' = (\Sigma, E', R', L)$ the corresponding modified LRT, $s, t \in T^P_\Sigma(X)$, and $\alpha \in L$. If $[s]_{E'} \xrightarrow{\alpha}_{\mathcal{T}'} [t]_{E'}$, then $[s]_E \xrightarrow{\alpha}_{\mathcal{T}} [t]_E$.*

Proof. Again we prove the claim by induction over the depth of the transition, now in the modified LRT.

$n = 0$**:** clear

$n \to n+1$**:** let $[s]_{E'} \xrightarrow{\alpha}_{\mathcal{T}'_{n+1}} [t]_{E'}$. Then there exist $\dfrac{c_1 \xrightarrow{\alpha_1} d_1 \ \ldots \ c_k \xrightarrow{\alpha_k} d_k}{l \xrightarrow{\alpha} r} \in R'$, $w \in Pos(s)$ and $\sigma \in Sub$ such that $s|_w =_{E'} l\sigma$, $t =_{E'} s[w \leftarrow r\sigma]$, and $[c_i]_{E'} \xrightarrow{\alpha_i}_{\mathcal{T}'_n} [d_i]_{E'}$ for every $1 \leq i \leq k$. We distinguish the following cases:

(i) If $\dfrac{c_1 \xrightarrow{\alpha_1} d_1 \ \ldots \ c_k \xrightarrow{\alpha_k} d_k}{l \xrightarrow{\alpha} r} \in R$, then, by induction hypothesis, $[c_i]_E \xrightarrow{\alpha_i}_{\mathcal{T}_n} [d_i]_E$ for every $1 \leq i \leq k$, and hence $[s]_E \xrightarrow{\alpha}_{\mathcal{T}_{n+1}} [t]_E$.

(ii) Otherwise, the transition rule has been added in the construction of the modified LRT as described in Def. 4.2. Corresponingly, there exist $m > k$, $c_{k+1}, \ldots, c_m, d_{k+1}, \ldots, d_m \in T^P_\Sigma(X)$ and $\sigma \in Sub$ such that

$$\frac{c_1 \xrightarrow{\alpha_1} d_1 \ \ldots \ c_k \xrightarrow{\alpha_k} d_k \ \ c_{k+1} \xrightarrow{\tau} d_{k+1} \ \ldots \ c_m \xrightarrow{\tau} d_m}{l' \xrightarrow{\alpha} r'} \in R,$$

$l =_E l'\sigma$, $r =_E r'\sigma$, $[c_i]_E \xrightarrow{\alpha_i}_{\mathcal{T}_n} [d_i]_E$ for $1 \leq i \leq k$, and $[c_j\sigma]_E \xrightarrow{\tau}_{\mathcal{T}_n} [d_j\sigma]_E$ for $k+1 \leq j \leq m$. Altogether this again implies $[s]_E \xrightarrow{\alpha}_{\mathcal{T}_{n+1}} [t]_E$

□

Both results now enable us to give the full bisimulation proof, showing that our construction of the modified LRT yields an abstract system which is weakly bisimilar to the original one.

Theorem 6.3 *Let $\mathcal{T} = (\Sigma, E, R, L)$ be a restricted LRT, $\mathcal{T}' = (\Sigma, E', R', L)$ the corresponding modified LRT, and $s_0 \in T^S_\Sigma(X)$ be some initial term. Then the induced LTSs $(T^S_\Sigma(X)/E, [s_0]_E, \to_{\mathcal{T}}, L)$ and $(T^S_\Sigma(X)/E', [s_0]_{E'}, \to_{\mathcal{T}'}, L)$ are weakly bisimilar.*

Proof. We show that the following relation is a weak bisimulation:

$$\mathcal{R} := \{([s]_E, [s']_{E'}) \mid s =_{E'} s'\}.$$

To this aim we have to prove that both transition systems weakly simulate each other w.r.t. $\mathcal{R}$.

(i) $(T_\Sigma^S(X)/E', [s_0]_{E'}, \to_{\mathcal{T}'}, L)$ simulates $(T_\Sigma^S(X)/E, [s_0]_E, \to_{\mathcal{T}}, L)$:
let $[s]_E \xrightarrow{\alpha}_{\mathcal{T}} [t]_E$. We distinguish the following cases:

- $s \in T_\Sigma^P$: if $s \neq_{E'\setminus E} t$, then $[s]_{E'} \xrightarrow{\alpha}_{\mathcal{T}'} [t]_{E'}$ by Lemma 6.1. Hence $[s]_{E'} \overset{\hat{\alpha}}{\Rightarrow}_{\mathcal{T}'} [t]_{E'}$. Otherwise, $\alpha = \tau$ and thus $[s]_{E'} \overset{\hat{\alpha}}{\Rightarrow}_{\mathcal{T}'} [t]_{E'}$.
- $s \in T_\Sigma^S \setminus T_\Sigma^P$: here s can be represented as $s = p \parallel ps$ where $p \in T_\Sigma^P(X)$ and $ps \in T_\Sigma^S(X)$. Moreover $[s]_E \xrightarrow{\alpha}_{\mathcal{T}} [t]_E$ implies that there exist

$$\frac{c_1 \xrightarrow{\alpha_1} d_1 \ \ldots \ c_k \xrightarrow{\alpha_k} d_k}{l \xrightarrow{\alpha} r} \in R,$$

$w \in Pos(s)$ and $\sigma \in Sub$ such that $s|_w =_E l\sigma$, $t =_E s[w \leftarrow r\sigma]$ and $[c_i\sigma]_E \xrightarrow{\alpha_i}_{\mathcal{T}} [d_i\sigma]_E$ for every $1 \leq i \leq k$. According to Def. 4.1, two cases are possible:
  - $\dfrac{l \xrightarrow{\tau} r}{l \parallel u \xrightarrow{\tau} r \parallel u} \in R$: if $s \neq_{E'\setminus E} t$, then $[s]_{E'} \xrightarrow{\alpha}_{\mathcal{T}'} [t]_{E'}$ by Lemma 6.1, and thus $[s]_{E'} \overset{\hat{\alpha}}{\Rightarrow}_{\mathcal{T}'} [t]_{E'}$. Otherwise, $[s]_{E'} \overset{\hat{\alpha}}{\Rightarrow}_{\mathcal{T}'} [t]_{E'}$ since the transition can be simulated by equational transformations.
  - $\dfrac{l \xrightarrow{a} r}{l \parallel u \xrightarrow{a} r \parallel u'} \in R$: here $s \neq_{E'\setminus E} t$ and $t = p' \parallel ps'$ since otherwise $p =_{E'\setminus E} p'$, and thus there would exist $\overline{l \xrightarrow{\tau} r} \in R$, $w \in Pos(p)$ and $\sigma \in Sub$ such that $p|_w =_E l'\sigma$ and $t =_E p'[w \leftarrow r'\sigma]$, in contradiction to Def. 4.1. Hence $[s]_{E'} \xrightarrow{\alpha}_{\mathcal{T}'} [t]_{E'}$ by Lemma 6.1, and thus $[s]_{E'} \overset{\hat{\alpha}}{\Rightarrow}_{\mathcal{T}'} [t]_{E'}$.

(ii) $(T_\Sigma^S(X)/E, [s_0]_E, \to_{\mathcal{T}}, L)$ simulates $(T_\Sigma^S(X)/E', [s_0]_{E'}, \to_{\mathcal{T}'}, L)$:
let $[s]_{E'} \xrightarrow{\alpha}_{\mathcal{T}'} [t]_{E'}$. Again we distinguish two cases:

- $s \in T_\Sigma^P$: here the transition can be divided into the following three parts.
  (a) equational transformations: $s =_{E'\setminus E} s'$
  (b) actual transition: $[s']_{E'} \xrightarrow{\alpha}_{\mathcal{T}'} [t']_{E'}$
  (c) equational transformations: $t' =_{E'\setminus E} t$

  Each of these parts can be simulated in $(T_\Sigma^S(X)/E, [s_0]_E, \to_{\mathcal{T}}, L)$ as follows.
  (a) Since E' and E only differ in the equations added by unconditional τ-rules, there exist $[t_1]_E, \ldots, [t_n]_E$ with $[t_i]_E \xrightarrow{\tau}_{\mathcal{T}} [t_{i+1}]_E$ for $1 \leq i \leq n-1$, $[t']_E \xrightarrow{\tau} [t_1]_E$, and $[t_n]_E \xrightarrow{\tau} [t]_E$.
  (b) Here $[s']_E \xrightarrow{\alpha}_{\mathcal{T}} [t']_E$ by Lemma 6.2.
  (c) just as in (a)

  Altogether we conclude that $[s]_E \overset{\hat{\alpha}}{\Rightarrow}_{\mathcal{T}} [t]_E$.
- $s \in T_\Sigma^S \setminus T_\Sigma^P$: here s and t can be represented as $s = p \parallel ps$ and $t = q \parallel ps'$, respectively. Correspondingly there exists $\dfrac{l \xrightarrow{\alpha} r}{l \parallel x \xrightarrow{\alpha} r \parallel x'} \in R$, and the transition can be divided into the following four parts.
  (a) equational transformations: $p =_{E'\setminus E} p'$

(b) actual transition: $[p' \parallel ps]_{E'} \xrightarrow{\alpha}_{\mathcal{T}'} [q' \parallel ps'']_{E'}$
(c) equational transformations: $q' =_{E' \setminus E} q$
(d) equational transformations: $ps'' =_{E' \setminus E} ps'$

Again each of these parts can be simulated in $(T_\Sigma^S(X)/E, [s_0]_E, \rightarrow_{\mathcal{T}}, L)$ as follows.

(a) Since E' and E only differ in the equations added by unconditional τ-rules, there exist $[p_1 \parallel ps]_E, \ldots, [p_n \parallel ps]_E$ with $[p_i \parallel ps]_E \xrightarrow{\tau}_{\mathcal{T}} [p_{i+1} \parallel ps]_E$ for every $1 \leq i \leq n-1$), $[p \parallel ps]_E \xrightarrow{\tau} [p_1 \parallel ps]_E$, and $[p_n \parallel ps]_E \xrightarrow{\tau} [p' \parallel ps]_E$.
(b) Since $[p']_E \xrightarrow{\alpha}_{\mathcal{T}} [q']_E$ by Lemma 6.2, $[p' \parallel ps]_E \xrightarrow{\alpha}_{\mathcal{T}} [q' \parallel ps'']_E$.
(c) just as in (a)
(d) By applying equational transformations we can only stay in the same equivalence class.

Altogether we again conclude that $[s]_E \stackrel{\hat{\alpha}}{\Rightarrow}_{\mathcal{T}} [t]_E$.

□

7 Conclusions and Related Work

In this paper we have shown how equational abstractions can be used in the Rewriting Logic framework to reduce the state space of concurrent systems. In particular we have seen how to construct for a given restricted LRT a new one whose induced LTS is weakly bisimilar to the LTS induced by the original LRT, avoiding the construction of the (big) original LTS. Thus our abstraction can be employed to obtain efficient model-checking algorithms for logics such as $\mathrm{CTL}_{\setminus X}$ which cannot distinguish weakly bisimilar systems. We have demonstrated the use of the proposed method with the concurrent functional programming language Erlang. For the locker example we have seen that equational abstraction leads to a significant reduction of the state space.

The work described in the present paper builds on results in [15] where the idea of moving unconditional τ-transitions to the equational theory has been introduced, again in the context of the Erlang programming language, but no correctness proof has been given. In fact, as the discussion in Section 4 shows, the simple idea of turning unconditional τ-rules to equations does not work out; requiring a more elaborate construction.

Another publication which employs equations to define abstraction mappings on the state space is [11]. Here, in contrast to our work, the abstraction is expected to be correct only w.r.t. linear-time properties such that the resulting abstract systems are generally not weakly bisimilar to the original one. Moreover [4] studies the application of partial-order reduction techniques in a Rewriting Logic setting, again employing a notion of equivalence (stuttering) which is different from ours.

References

[1] Armstrong, J., S. Virding, M. Williams and C. Wikström, "Concurrent Programming in Erlang," Prentice Hall International, 1996, 2nd edition.

[2] Arts, T., C. Earle and J. Derrick, *Verifying Erlang code: a resource locker case study*, in: *Formal Methods – Getting IT Right*, Lecture Notes in Computer Science **2391** (2002), pp. 184–203.

[3] Emerson, E., "Temporal and Modal Logics," Handbook of Theoretical Computer Science **B**, Elsevier, 1990 pp. 996–1072.

[4] Farzan, A. and J. Meseguer, *State space reduction of rewrite theories using invisible transitions*, in: *11th Int. Conference on Algebraic Methodology and Software Technology (AMAST'06)*, Lecture Notes in Computer Science **4019** (2006), pp. 142–157.

[5] Fredlund, L.-A. and H. Svensson, *McErlang: a model checker for a distributed functional programming language*, SIGPLAN Not. **42** (2007), pp. 125–136.

[6] Leucker, M. and T. Noll, *Rapid prototyping of specification language implementations*, in: *Proceedings of the 10th IEEE International Workshop on Rapid System Prototyping (RSP'99)* (1999), pp. 60–65.

[7] Leucker, M. and T. Noll, *Rewriting logic as a framework for generic verification tools*, in: *Proceedings of Third International Workshop on Rewriting Logic and Its Applications (WRLA'00)*, Electronic Notes in Theoretical Computer Science **36** (2001), pp. 121–137.

[8] Martí-Oliet, N. and J. Meseguer, *Rewriting logic: Roadmap and bibliography*, Theoretical Computer Science **285** (2002), pp. 121–154.

[9] Meseguer, J., *Rewriting as a unified model of concurrency*, in: *Proceedings Int. Conference on Concurrency Theory (CONCUR'90)*, number 458 in Lecture Notes in Computer Science (1990), pp. 384–400.

[10] Meseguer, J., *Conditional rewriting logic as a unified model of concurrency*, Theoretical Computer Science **96** (1992), pp. 73–155.

[11] Meseguer, J., M. Palomino and N. Martí-Oliet, *Equational abstractions*, in: *19th Int. Conference on Automated Deduction (CADE-19)*, Lecture Notes in Computer Science **2741** (2003), pp. 2–16.

[12] Milner, R., "Communication and Concurrency," International Series in Computer Science, Prentice–Hall, 1989.

[13] Neuhäußer, M. and T. Noll, *Abstraction and model checking of Core Erlang programs in Maude*, in: *Proceedings of the 6th International Workshop on Rewriting Logic and its Applications (WRLA'06)*, Electronic Notes in Theoretical Computer Science **176** (2007), pp. 147–163.

[14] Noll, T., *A rewriting logic implementation of Erlang*, in: *Proceedings of First Workshop on Language Descriptions, Tools and Applications (ETAPS/LDTA'01)*, Electronic Notes in Theoretical Computer Science **44(2)** (2001), pp. 206–224.

[15] Noll, T., *Equational abstractions for model checking Erlang programs*, in: *Proceedings of the International Workshop on Software Verification and Validation (SVV'03)*, Electronic Notes in Theoretical Computer Science **118** (2005), pp. 145–162.