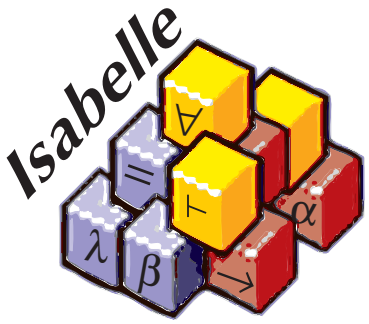


Integrating Theorem Proving and Model Checking in Isabelle/IOA

Tobias Hamberger
Institut für Informatik
Technische Universität München
hambergt@in.tum.de

August 11, 1999

Abstract

Isabelle is a generic theorem proving environment which was used successfully in recent years for many practical applications. In particular, the model of Input/Output-automata in combination with its formal meta-theory, abstraction theory and temporal logic has been embedded in Isabelle/HOLCF. An advantage of Isabelle is, that it can cope with arbitrary large systems, even infinite in the number of states. A disadvantage is the necessary interaction of the user during verification process. On the other hand, model checkers enable automatical verification, but their use is usually restricted to small, finite systems. In this paper, we present an integrated environment for specification and verification of distributed systems in Isabelle. Beginning with the Isabelle formulation of I/O-automata, some additional components were created by the author and assembled to a consistent tool. Here, we take a closer look on the integration of the model checker μcke as an external verification tool for Isabelle.
We also present a direct support of the specification language for I/O-automata known from literature and we present a proof tactic for implementation relations of I/O-automata in Isabelle. The model checker μcke, which is used in this tactic, is integrated in Isabelle as an oracle. The usage of the resulting verification environment for reactive systems is presented by two examples.

Contents

1 Introduction

1.1 Motivation

Reactive systems play an important role in application of information systems. They have the property that they can observe events in their environment and may then react by executing some actions. Such systems may be *distributed*, which means that they consist of several components which may communicate with each other (e.g. via actions).

A formal model for such systems is that of *Input/Output automata* ([9], shortly written as *I/O-automata*). An I/O-automaton may be partitioned into several other automata, where for each automaton, a set of actions is specified, which are distinguished between input actions and output actions. Here, communication takes place through execution of actions (which may contain parameters) that is an output action for the sending component and is an input action for each receiving component.

By describing a system via I/O-automata one can make *formal verification* of system properties. This is, for example, possible with the theorem prover *Isabelle* ([15] and [8]). Theorem provers like Isabelle make it possible to prove properties interactively. This means that a user must generally think about the proof step to apply and the theorem prover then executes this step and gives the new proof state as output and awaits the next proof steps.
On the other hand, *model checkers* are tools which may in principle automatically verify certain restricted formulas. This means that a user produces as input only the formula to prove and the model checker tests whether this formula is valid for each possible state. A disadvantage of model checkers is that their area of application is restricted, typically to finite-state systems.

In this paper, a part of the verification environment of Isabelle/IOA will be described, where these two concepts of theorem proving and model checking are combined: for a certain property class on I/O-automata, proof obligations in Isabelle are delegated to a model checker whose results Isabelle uses to verify the property. This usage of the model checker was implemented in a generic way so that the model checker may be used in Isabelle also for non-I/O-automaton verification tasks.
Moreover, the comfortable input syntax for I/O-automata in the Isabelle system, which is similar to the classical I/O-automaton syntax described in [9], will be introduced.

1.2 Main Goals and Outline of the Paper

Four items are described in this paper, which are listed in a more precise way as following. As the structure of this paper corresponds strongly with these items (exactly one section per item), also the corresponding section is given at each item.

- The syntax of Isabelle theories was extended by a section `automaton`, which enables the user to give I/O-automata written in the syntax form for I/O-automata according to [9] as input for Isabelle. This syntax will be described in section 2.
- *Implementation relation properties* for I/O-automata can be proved in some cases by translating the problem in another logic (the *μ-calculus*) and then prove this formula by using the model checker *μcke* [2], which is also based on the *μ-calculus*. This translation will be described in section 3.
- The interface between Isabelle and *μcke* is implemented as an oracle. A proof tactic is provided in Isabelle where for proving formulas of the μ-calculus, one can use this oracle. This will be described in section 4.
- After introducing this verification environment, two applications are shown in section 5.

1.3 Related Work

This paper describes the results of the authors diploma thesis [7] and relies also on [12], where the theoretical basis for the here introduced practical concepts is given. For similar problems on I/O-automata with model checker use in Isabelle, there is a case study in [6]. There, the nice input syntax for I/O-automata was not available and the transfer from Isabelle to the model checker was done by hand, and not automatically. A further combination of Isabelle and model checker for I/O-automata is described in [10], but again without automatic coupling of these two tools.

2 The I/O-Automata Syntax for Isabelle

Firstly, the convenient syntax for I/O-automata according to Lynch/Tuttle [9] will be described. Then, its use in Isabelle will be demonstrated.

2.1 A Short Introduction to Safe I/O-Automata

I/O-automata provide a helpful means to describe reactive systems. An I/O-automaton consists of a set of possible states and a set of actions, which may be executed on the automaton and cause the automaton to change its state. Actions may have parameters and each action may be specified by a precondition (a predicate over the state and action parameters) and a post-condition (description of the automaton state after action execution). Each action is either an input, output or internal one. Input actions are *input enabled*, which means that they are always executable on this automaton and so, must not have any precondition.

Formally a *Safe* I/O-automaton can be described by subsequent definitions.

Definition: *(Signature)*
An *action signature* S is a triple of three mutually disjoint sets: $in(S)$, the set of input actions, $out(S)$, the set of output actions and $int(s)$, the set of internal actions. The union of $in(S)$ and $out(S)$ is $ext(S)$, the set of external actions. The union of all three sets is written as $acts(S)$.

Definition: *(Safe I/O-Automaton)*
A *Safe I/O-automaton* A consists of:

- an action signature *sig(A)*
- a state set *states(A)*
- *start(A)*, a non-empty set of initial states with $start(A) \subseteq states(A)$
- a transition relation *steps(A)*$\subseteq states(A) \times acts(sig(A)) \times states(A)$, where for each $s \in states(A)$ and $a \in in(sig(A))$ there exists some $(s, a, t) \in steps(A)$. For $(s, a, t) \in trans(A)$, we will write $s \rightarrow_A^a t$.

For I/O-automata, the operations *composition* (several automata are parallely composed to one automaton), *restriction* and *hiding* (the set of external and internal actions of an automaton can be modified) and *renaming*

(action names can be renamed) are avialable, too. However, such definitions will not be given here, as they do not play an important role in this presentation of the verification system. The interested reader should consult [9] or [12]. Whenever such operations are used in the examples, they will be further explained there.

2.2 The I/O-Automata Syntax

A syntax for I/O-Automata is described by the following grammar. A similar pattern has been used in [4] and was here modified to comply with the Isabelle syntax. The syntax is very intuitive and the connection to above I/O-automaton definition should be obvious by examining one of the following examples.

<automaton_def> ::= **automaton** <name>
signature
actions <datatype>
[**inputs** <action_list>]
[**outputs** <action_list>]
[**internals** <action_list>]
states <component_list>
[**initially** <predicate>]
transitions <transition_list>

<action_list> ::= <action> | <action> , <action_list>

<component_list> ::= <component> | <component> <component_list>
<component> ::= <name> **::** <type>

<transition_list> ::= <transition> | <transition> <transition_list>
<transition> ::= <action> (<transrel> | <pre1> | <post>)

<transrel> ::= **transrel** <predicate>

<pre1> ::= <pre> [<post>]
<pre> ::= **pre** <predicate>

<post> ::= **post** <assign_list>
<assign_list> ::= <assign> | <assign> , <assign_list>
<assign> ::= <name> **:=** <term>

The following conventions apply:

- If the initial condition (**initially**) is missing, any state is permitted as an initial one (equivalent to **initially** *True*).
- If at an action, a precondition is specified (**pre**), but the postcondition (**post**) is left out, all state components remain unchanged when this action is executed. Also, if in a postcondition, an assignment term for a component is missing, this component remains unchanged, too.
- If **transrel** is used for specifying an action, the component values *before* executing the action are referenced simply with their name and the values *after* execution are referenced with the primed component name (e.g. for component *var*, write *var*).

2.3 Example

Imagine a very trivial alarm manager of a helicopter alarm system. It consists of the actions *Alarm(a : Alarms)* indicating the arrival of alarm a (and storing it), *Info(i : Alarms)* indicating that an alarm i is in the store, and *Ack(a : Alarms)* meaning that an alarm a has been acknowledged by the pilot (and can be deleted from the store). Subsequent implementation focuses especially on the so-called Point-of-No-Return alarm and has therefore only a flag as memory, which indicates whether this alarm is stored.

automaton *Aut_C*
signature
inputs
$Alarm(a), a \in Alarms$
outputs
$Info(i), i \in Alarms$
internals
$Ack(a), a \in Alarms$
states
$flag : bool$
initially
$flag = False$
transitions
$Alarm(a)$
post $flag :=$ if $(a = Pon)$ then $True$ else $flag$
$Info(i)$
pre if $(i = Pon)$ then $flag$ else $True$
$Ack(a)$
transrel if $(a = Pon)$ then $(flag \wedge flag' = False)$ else $(flag' = flag)$

When using this kind of syntax in Isabelle, the following issues have to be considered.

- After the line **automaton** <name>, the Isabelle system requires an equality sign.
- The line **actions** <datatype> does not appear in standard literature about I/O-automata [9]. In Isabelle, it specifies the (finite) datatype from which the action sets are built.
- The syntax elements <type> and <datatype> must comply with the the type syntax in Isabelle.
- The syntax elements <action>, <predicate> and <term> must be written as <String>, which corresponds to the proper Isabelle term to be expressed.

A syntax diagram for the use of I/O-automata in Isabelle is given in the appendix. There, also the Isabelle syntax of certain operations on automata, which are also supported by this syntax, is given. These operations are *composition* of I/O-automata, *hiding*, *restriction* and *renaming* of actions in a I/O-automaton.

Example continued. To describe above automaton *AutC* in a Isabelle theory `SimEx`, the file `SimEx.thy` will look like this:

```
SimEx = IOA +

datatype alarm = Pon | Eng | Fue
datatype ('a)action = Info 'a | Ack 'a | Alarm 'a

automaton Aut_C =
signature
actions (alarm)action
inputs
        "Alarm a"
outputs
        "Info i"
internals
        "Ack a"
states
        flag :: bool
initially
```

```
        "flag = False"
transitions
"Alarm a"
        post flag := "(if (a=Pon) then True else flag)"
"Info i"
        pre "if (i=Pon) then flag else True"
"Ack a"
        transrel "if (a=Pon) then (flag=True & flag'=False)
                  else (flag'=flag)"
```

3 The Implementation Relation and the μ-Calculus

In this section, the implementation relation property between two automata and the μ-calculus will be introduced. Then, a way will be shown to translate implementation relation properties into the μ-calculus. This serves as a base to use the model checker for verification of implementation relations. The simple example of preceding section will be continued in this context.

3.1 The Implementation Relation

In this subsection the implementation relation between two Safe I/O-automata will be defined. For that end, we need some auxiliary definitions.

Definition: *(Execution Fragment and Execution Sequence)*
An *execution fragment* for an automaton A is a finite or infinite sequence $s_0\, a_1 s_1\, a_2 s_2 \ldots$ of alternating states and actions of A, where for all i holds: $s_i \rightarrow_A^{a_{i+1}} s_{i+1}$. An *execution sequence* for an automaton A is an execution fragment with $s_0 \in \mathit{start}(A)$.

Definition: *(Accessibility)*
A state $s \in \mathit{states}(A)$ is *accessible in* A, iff there exists in A a finite execution fragment which ends in s.

Definition: *(Trace and Trace Sequence)*
A *trace* of an execution fragment α, written as $\mathit{trace}(\alpha)$, is a sequence containing exclusively all external actions of α. γ is a *trace sequence* of an automaton A iff there exists an execution sequence α of A with $\gamma = \mathit{trace}(\alpha)$. The set of all trace sequences of A is written as $\mathit{traces}(A)$.

Definition: *(Step)*
Let γ be a finite sequence over $\mathit{ext}(\mathit{sig}(A))$ and $s, t \in \mathit{states}(A)$. Then, (s, γ, t) is a *step* in automaton A, written as $s \Rightarrow_A^{\gamma} t$, iff there exists in A a finite execution sequence α which starts in s, ends in t and with $\mathit{trace}(\alpha) = \gamma$.

The following definition will explain the implementation relation. Informally spoken, an automaton C implements an automaton A, when for each sequence of actions in C which is externally visible, the same sequence is also possible in A.

Definition: *(Implementation Relation)*
For automata C and A with $\mathit{in}(\mathit{sig}(C)) = \mathit{in}(\mathit{sig}(A))$ and $\mathit{out}(\mathit{sig}(C)) = \mathit{out}(\mathit{sig}(A))$ holds the *implementation relation* $C \preceq_S A$ iff:

- $traces(C) \subseteq traces(A)$

Strongly corresponding with the implementation relation is *forward simulation.*

Definition: *(Forward Simulation)*
For two automata C and A with $in(sig(C)) = in(sig(A))$ and $out(sig(C)) = out(sig(A))$, a relation R over $states(C) \times states(A)$ is a *forward simulation*, iff:

- If $s \in start(C)$, then $R[s] \cap start(A) \neq \{\}$.
- If s is accessible in C and $s' \in R[s]$ is accessible in A and $s \rightarrow_C^a t$, then there exists a state $t' \in R[t]$ with $s' \Rightarrow_A^\gamma t'$, where $\gamma = [a]$, if $a \in ext(sig(A))$ and $\gamma = []$ else.

If there exists a forward simulation for C and A, we write:

$$C \leq_F A.$$

Olaf Müller proved the following theorem in [12], which shows that for verifying the implementation relation, it suffices to verify the existence of a forward simulation. This will be also used by the verification framework presented later.

Theorem 3.1:
For two automata C and A with $in(sig(C)) = in(sig(A))$ and $out(sig(C)) = out(sig(A))$ holds:

$$C \leq_F A \Rightarrow C \preceq_S A$$

3.2 The μ-Calculus

The μ-calculus is the underlying logic of the model checker μcke. So, a few words will be used to describe this logic. For a more detailed description, see [2].

The μ-calculus is an extension of the first order predicate logic by introducing the fix-point operators μ and ν. We now describe the syntax and semantics:

Definition: *(Types and Variables)*
Let T be a finite set of finite sets with $bool \in T$ and $bool = \{true, false\}$. Let X be a finite set of elements, which has a typing function $\tau_X : X \to T$. Let P be a finite set of elements, which has a typing function $\tau_P : P \to \cup_{n=1}^{\infty} T^n$. All $t \in T$, X and P should be mutually disjoint. Based on the sets T, X and P one can define a μ-calculus, where the *types* are the elements of T, the so-called *simple variables* are the elements of X and the so called *predicate variables* are the elements of P. For $p \in P$, $\tau_P(p)$ is the argument type of the represented predicate.

Definition: *(Ground Term)*
A *ground term* of type $t \in T$ is:

- a c with $c \in t$
- a x with $x \in \{x \in X | \tau_X(x) = t\}$

τ_X is extended on ground terms by defining for each $c \in t$: $\tau_X(c) = t$.

Definition: *(Formulas)*
Formulas are:

- boolean ground terms
- $x = y$ or $x \neq y$ with ground terms x and y, where $\tau_X(x) = \tau_X(y)$
- $\neg a$, $a \vee b$ and $\exists x.a$ with formulas a and b and $x \in X$
- $p(z_1, \ldots, z_n)$ with a *relational term* p and $(z_1, \ldots, z_n) \in \tau_P(p)$ and all z_i are ground terms of proper types.

Now it remains to define the relational terms:

Definition: *(Relational Terms)*
Relational Terms are:

- a p with $p \in P$
- $\lambda z_1 \ldots z_n.f$ with $z_i \in X$ and formula f
- $\mu z.p$ and $\nu z.p$ with $z \in P$, p a relational term, which is *formally monotone* (see below) in z, and $\tau_P(z) = \tau_P(p)$

τ_P is extended on relational terms, so that:

- $\tau_P(\lambda z_1 \ldots z_n.f) = \times_{i=1}^{n} \tau_X(z_i)$
- $\tau_P(\mu z.p) = \tau_P(z)$ or $\tau_P(\nu z.p) = \tau_P(z)$

Definition: *(Formal Monotonicity)*
A relational term P is *formally monotone* in a predicate variable z, iff each free occurrence of z in P is enclosed by an even number of negations.

In addition it shall be possible, to use $\forall$, $\wedge$ and $\rightarrow$ in formulas in their usual way.

The semantics of ground terms and relational terms will be defined by an assignment function ρ as follows:

Definition: *(Assignment)*
An *assignment* ρ assigns each variable $x \in X$ a $c \in \tau_X(x)$ and each $p \in P$ a predicate $I(p_\rho)$, which is the characteristic function of a set $p_\rho \subseteq \tau_P(p)$.
With ρ given, an assignment ρ' can be defined by:

$\rho' := \rho\{z_1 \rightarrow a_1, \ldots, z_n \rightarrow a_n\}$, with $z_i \neq z_j$ and $a_i \in \tau_X(z_i)$, if $z_i \in X$ or a_i as the characteristic function of a subset of $\tau_P(z_i)$, if $z_i \in P$

Here, ρ' has following property.

- $\rho'(a) = \rho(a)$, if $a \notin \{z_1, \ldots, z_n\}$
- $\rho'(a) = a_i$, if $a_i = z_i$

Definition: *(Evaluation of Ground Terms)*
Ground terms are evaluated relative to an assignment ρ in the following way:

- $[c]\rho = c$, if $c \in t \in T$
- $[x]\rho = \rho(x)$, if $x \in X$

The evaluation of formulas corresponding to an assignment ρ is defined canonically and will be omitted here. It remains only to define the evaluation of relational terms. They are evaluated to a predicate with appropriate argument type:

Definition: *(Evaluation of Relational Terms)*

- $[p]\rho = \rho(p)$, if $p \in P$
- $[\lambda z_1 \ldots z_n.f]\rho = I(p)$ with $p = \{(a_1, \ldots, a_n) \in \times_{i=1}^{n} \tau_X(z_i) |\ [f]\rho'\}$ and $\rho' = \rho\{z_i \rightarrow a_i | 1 \leq i \leq n\}$
- $[\mu z.p]\rho = lfp(Q)$ with an assignment Q between predicates of argument type $\tau_P(z)$, which is defined by $Q(f) := [p]\rho\{z \rightarrow f\}$
- $[\nu z.p]\rho = gfp(Q)$ with a mapping Q between predicates of argument type $\tau_P(z)$, which is defined by $Q(f) := [p]\rho\{z \rightarrow f\}$

Here, *lfp* is the least and *gfp* is the greatest fix-point of a monotonous mapping. For calculating these fix-points, we refer to [2] and [1]. There, it is also proved that for the terms and formulas defined here, these fix-points always exist (that was the reason for using the formal monotonicity here).

3.3 Translating Implementation Relations into the μ-Calculus

To prove a given implementation relation by a μ-calculus-based model checker, one has to translate the implementation relation, respectively the forward simulation, property (see Theorem 3.1), to a μ-calculus formula. The correctness of following translation process was again proved by Olaf Müller in [12] and will be described below.

- Define on Act the characteristic predicate for internal actions of each automaton:
  - $Internal_A(a) := a \in int(asig(A))$
  - $Internal_C(a) := a \in int(asig(C))$
- Define the characteristic predicate of the initial states:
  - $Start_A(s) := s \in start(A)$
  - $Start_C(s) := s \in start(C)$
- Define the characteristic predicate for transitions:
  - $Trans_A(s, a, t) := s \rightarrow_A^a t$
  - $Trans_C(s, a, t) := s \rightarrow_C^a t$
- The predicate $IntStepStar_A$ indicates whether a state t of A is accessible from s via a *finite* sequence of actions:

  * $IntStep_A(s,t) := \exists a.Internal_A(a) \wedge Trans_A(s,a,t)$
  * $IntStepStar_A := \mu P.\lambda\ s\ t\ .(s = t) \vee \exists u.IntStep_A(s,u) \wedge P(u,t)$

- Following predicate on transitions of A holds iff either the action a is internal in C and $IntStepStar_A(s,t)$ holds or $s \Rightarrow_A^{[s]} t$ holds:

  * $Move_A(s,a,t) := (Internal_C(a) \wedge IntStepStar_A(s,t)) \vee$
    $\exists u_1\ u_2.IntStepStar_A(s,u_1) \wedge Trans_A(u_1,a,u_2) \wedge IntStepStar_A(u_2,t)$

- Then, the existence of a forward simulation can be described with the aid of following predicate:

  * $isSim_{CA} := \nu P.\lambda s_1\ t_1.\forall a\ s_2.Trans_C(s_1,a,s_2)$
    $\rightarrow \exists t_2.Move_A(t_1,a,t_2) \wedge P(s_2,t_2)$

- Finally, the property expressing the forward simulation is:

  * $\forall s\ t\ .Start_C(s) \wedge Start_A(t) \rightarrow isSim_{CA}(s,t)$

Due to Theorem 3.1, the existence of a forward simulation also implies that the implementation relation holds. So, the implementation relation holds if latter formula holds.

3.4 Example

Analogously to the simple helicopter alarm system `Aut_C` in section 2.3 one can define an automaton `Aut_A` with the same input and output actions as `Aut_C`, which does also have the same preconditions and effects, but where the internal action *Ack* is missing. Let's assume that both automata were defined in a Isabelle theory file `SimEx.thy`. Additionally to the example of the latter section, the theory used by `Simex` should not be `IOA` (see first line "`SimEx = IOA +`" of theory file there), but `MuIOAOracle`, where `MuIOAOracle` is a theory where the verification environment explained in this paper is defined.
So, to execute this example proof, one has to load the theories `MuIOAOracle` and `SimEx`. To prove that `Aut_C` implements `Aut_A`, we simply type into a running Isabelle session:

```
Goal "Aut_C =<| Aut_A";
```

In the ideal case, the tactic `is_sim_tac` of theory `MuIOAOracle` manages all proof work:

```
by (is_sim_tac aut_simps 1);
```

Here, `aut_simps` is a theorem list which contains all theorems which were generated[1] at the definition of `Aut_A` and `Aut_C`. These theorems define the automata and the start states, signature and transitions of the automata. For this example, `aut_simps` is defined in file `SimEx.ML` as follows:

```
val aut_simps = [Aut_A_def,Aut_A_asig_def,
                 Aut_A_start_def,Aut_A_trans_def,
                 Aut_C_def,Aut_C_asig_def,
                 Aut_C_start_def,Aut_C_trans_def];
```

In the first part of the proof, this tactic verifies the equality of the sets of input actions between these two automata and also the equality of the sets of output actions. Then for proving the implementation relation, it only remains to verify:

$$Traces(Aut\_C) \subseteq Traces(Aut\_A)$$

This is proved by translation to the μ-Calculus in the way described above. The intermediate result of the tactic is a subgoal formula in the Isabelle theory `MuCalculus`:

```
1. [|Internal_of_A == λ a.  a ∈ int Aut_A;
    Internal_of_C == λ a.  a ∈ int Aut_C;
    Start_of_A == λ s0.  s0 ∈ starts_of Aut_A;
    Start_of_C == λ s0.  s0 ∈ starts_of Aut_C;
    Trans_of_A == λ (s0,t0) a.  s0 -a--Aut_A-> t0;
    Trans_of_C == λ (cs0,ct0) a.  cs0 -a--Aut_C-> ct0;
    IntStep_of_A == λ (s0,t0).  ∃ a.  Internal_of_A a ∧
     Trans_of_A (s0,t0) a;
    IntStepStar_of_A == Mu P. (λ (s0,t0).  s0 = t0 ∨
     (∃ u0.  IntStep_of_A (s0,t0) ∧ P (u0,t0)));
    Move_of_A == λ(s0,t0) a.  Internal_of_C a ∧ IntStepStar_of_A (s0,t0)
     ∨ (∃ u0 v0.  IntStepStar_of_A (s0,u0) ∧
      Trans_of_A (u0,v0) a ∧ IntStepStar_of_A (v0,t0));
    isSimCA == Nu P. (λ (cs0,s0).  ∀ a ct0.
     Trans_of_C (cs0,ct0) a → (∃ t0.  Move_of_A (s0,t0) a ∧ P (ct0,t0)))|]
==> ∀ cs0 s0.  Start_of_C cs0 ∧ Start_of_A s0 → isSimCA(cs0,s0)
```

For completion of the proof the tactic still has to use the model checker μcke, which will be described in the following section.

[1] In case of describing the automata in Isabelle with the syntax introduced in the preceding chapter.

4 From μ-Calculus to μcke

To make the final step when proving implementation relations of I/O-automata in Isabelle, the model checker μcke will be used to prove the remaining formula of the μ-calculus. In this section the interface of this model checker for Isabelle will be introduced. However, this interface is not only useful for proving implementation relations, but can be used for verifying any given formula of the μ-calculus which is conform to the interface syntax.

4.1 Description of the Interface and its usage

To prove a formula of the Isabelle theory `MuCalculus` (where the μ-calculus is defined, see [17]), the formula has to have following structure:

1. The formula may have premises. Then, when translating this formula to μcke, only the conclusion of the formula will be considered as proof goal and each premise will be considered as equation which defines a predicate (which may then occur in the conclusion or in other premises). Hence, each premise has to observe the proper type and structure constraints. A correct example, where `f` is a predicate defined in the premise, is:

   ```
   f == % a b. a & b ==> ? a b. f a b
   ```

2. The predicate definitions occuring in the premises have to be *meta-level equations* of the form $f \equiv \lambda x.<term>$. In particular, all arguments of this predicate head have to be bound by η-conversion to the term at the right hand. Furthermore, if an argument is of tuple type, its structure must be fully instantiated; e.g. if x is a variable of type $bool \times bool$, a function definition $f \equiv fst(x)$ must be modified to $f \equiv \lambda(a, b).fst(a, b)$.

3. The fixed point operators μ and ν may only occur at the outermost position of the right hand side of the definition equations. An arbitrary formula can be transformed to fulfill this requirement by splitting it into several auxiliary predicate definitions.

4. No operators or functions of non-boolean result type may be used. Before using the interface, each occurrence of such an item has to be removed, e.g. by rewriting or simplification.

5. Each type used in the formula must be boolean or a datatype with a finite number of elements.

6. Besides the abstraction used in the premises to bind predicate arguments to the right-hand-side definition term, abstraction may only be used for existential and universial quantification.

7. Free variables may not occur anywhere in the formula.

Not all the work to meet these requirements has to be done by the user. For transformations to fulfill requirements from number 1 to 3, solutions were already available through Robert Sandner's theory `MuckeSyn` [17] which is based on the theory `MuCalculus`. There, the tactic `mc_mucke_tac` was defined which will be shortly introduced in the following subsection. Note that for requirement 1, theorems to be proved have to fulfill a different syntax pattern (see there). This tactic is, however, not important for our I/O-automaton problem as its formula of the μ-calculus fulfills these requirements.

Theory `MuCalculus` was extended to meet also requirement 4 through applying some simplifications. They are integrated in a tactic `call_mucke_tac` which takes the `MuCalculus` formula as argument and whose usage will be explained in the following subsection. Note that `mc_tac` is based on this tactic as `call_mucke_tac` manages the model checker call from Isabelle. This tactic also manages the transformation from Isabelle datatypes to simple μcke enumeration types and, corresponding to requirement 5, a finiteness check for each datatype.

Therefore, the user only has to take care of the requirements 5 to 7. However, in the context of implementation relation problems treated in this paper, these requirements are generally fulfilled.

4.2 An example use of the μcke Interface

At first, we describe the syntax of the tactic `mc_mucke_tac`:

```
by (mc_mucke_tac  defs i );
```

Here, i indicates the number of the subgoal where the tactic is to apply and *defs* is a list of definitions formulas, which are taken as premises of the subgoal to prove. This syntax differs slightly from the requirement 1 above. There, these definitions were already premises of the subgoal whereas here, these definitions must be provided as a list parameter *defs*. Internally this tactic calls an auxiliary function `mk_lam_defs` for the premises to achieve requirement 1 and 2 and an auxiliary function `move_mus` to achieve requirement 3. Afterwards, the tactic `call_mucke_tac` is used to let the model checker prove the subgoal, which will be explained in following paragraph.

We continue the example of the preceding chapter and explain how `is_sim_tac` finishes the proof of the implementation relation. This is simply by calling tactic `call_mucke_tac` is called and the μ-calculus formula at the end of the preceding chapter is taken as its argument.

In general, the tactic `call_mucke_tac` is used in the following way:

```
by (call_mucke_tac  i );
```

Here, i is the number of the subgoal to which this tactic should be applied. The tactic executes the transformations necessary to use the model checker μcke (included are simplifications to achieve requirement 4) and produces an input file `tmp.mu`[2] which contains the description of the formula to prove in μcke syntax and which is then executed by the command[3]

```
mucke -nb -res tmp.mu
```

The model checker μcke then verified the given formula and then gave following output:

```
: value of
:   forall bool cs0. forall bool s0. Start_of_C(cs0) & Start_of_A(s0)
: -> isSimCA(cs0, s0)
: is
:   true
============================================================
... virtual memory size is 2.62 MB
... system + user time is 0.43 seconds (00:00.00)
```

Due to successful verification in μcke, the proof of the implementation relation in Isabelle was also considered as successful.

[2] If variable `trace_mc` in file `HOL/Modelcheck/mucke_oracle.ML` is set on `true`, this file, which is written to the Isabelle temporary directory, will not be deleted after use and can be inspected by an interested user.

[3] The ML-function `extract_result` in the oracle relies on the output syntax, which finishes with three lines about use of resources (see example here: the first line is a line containing equality sign, the second line describes the memory size and the third line describes the required time). Therefore, μcke is called with the option `-res`. Option `-nb` suppresses the output of a here useless banner where an ASCII-gram of a mosquito is displayed.

5 Case Studies

We will demonstrate the capabilities of the presented verification framework. We will continue a case study of [6] where properties of a helicopter alarm system were proven. The complexity will be significantly increased in the second case study where we investigate into a Leader Election Algorithm [4]. There, we we almost fully exploit the capabilities of this verification method.

5.1 The Helicopter Alarm System

We consider a helicopter alarm system, which stores and manages alarms which occuring during operation. Incoming alarms are modeled through an action *Alarm*. Here, an alarm is inserted in the store and the occurrence of the alarm is transmittable to the pilot through updating an information display which the pilot watches. This update is modelled by an action *Info*. Then, the pilot may handle an alarm of the store by executing an action *Ack*. All these actions have a parameter of an enumeration type *event*, which especially contains the elements *PonR* (corresponds to the Point-of-No-Return-Alarm) and *None* (with the meaning that no alarm has occurred or is treated). We focus on the alarm *PonR*, so the state of the automaton is modelled by a boolean variable $APonR\_incl$ and an information display *info* of type *event*. Together, we have following automaton description:

automaton *cockpit*
signature
  inputs $Alarm(a), a \in event$
  outputs
  $Info(i), i \in event$
  $Ack(a), a \in event$
states
  $APonR\_incl : bool$
  $info : event$
initially $APonR\_incl = False \wedge info = None$
transitions
$Alarm(a)$
  post $APonR\_incl :=$ if $(a = PonR)$ then $True$ else $APonR\_incl$
    $info :=$ if $(a = None)$ then $info$ else a
$Info(i)$ **pre** $i = info$
$Ack(a)$
  pre $(a = PonR \rightarrow APonR\_incl) \wedge (a = None \rightarrow \neg APonR\_incl)$
  post $info := None$
    $APonR\_incl :=$ if $(a = PonR)$ then $False$ else $APonR\_incl$

In this model, we now prove following three properties:

1. Before the action $Ack(PonR)$ occurs and between two occurrences of this action, the action $Alarm(PonR)$ must have occurred at least once.
2. Action $Info(PonR)$ may occur, iff action $Alarm(PonR)$ has occurred and since then not any other alarm (besides the trivial $None$) has occurred.
3. Action $Info(None)$ may occur, iff after the latest occurrence of action Ack not any other alarm has occurred.

As we want to show these properties by proving some implementation relation properties, we represent these properties by I/0-automata. These I/O-automata should have a simple structure and, in each case, meet one of the properties above.

Let us begin with property 1:

automaton Al_before_Ack
signature
inputs
$Alarm(a), a \in event$
outputs
$Ack(a), a \in event$
states
$APonR\_incl : bool$
initially
$APonR\_incl = False$
transitions
$Alarm(a)$
post $APonR\_incl :=$ if $(a = PonR)$ then $True$ else $APonR\_incl$
$Ack(a)$
pre $(a = PonR \rightarrow APonR\_incl)$
post $APonR\_incl :=$ if $(a = PonR)$ then $False$ else $APonR\_incl$

Property 2 is described by following automaton:

automaton *Info_while_Al*
signature
inputs $Alarm(a), a \in event$
outputs
$Ack(a), a \in event$
$Info(i), i \in event$
states $info\_at\_Pon : bool$
initially $info\_at\_Pon = False$
transitions
$Alarm(a)$
post $info\_at\_Pon :=$ if $a = PonR$ then $True$
else (if $a = None$ then $info\_at\_Pon$ else $False$)
$Info(i)$
pre $(i = PonR \rightarrow info\_at\_Pon)$
$Ack(a)$
post $info\_at\_Pon := False$

For property 3, we choose following automaton:

automaton *Info_before_Al*
signature
inputs $Alarm(a), a \in event$
outputs
$Ack(a), a \in event$
$Info(i), i \in event$
states $info\_at\_None : bool$
initially $info\_at\_None = True$
transitions
$Alarm(a)$
post $info\_at\_None :=$ (if $a = None$ then $info\_at\_None$ else $False$)
$Info(i)$
pre $(i = None \rightarrow info\_at\_None)$
$Ack(a)$
post $info\_at\_None := True$

In Isabelle, all these automata were defined in the theory `Cockpit.thy` (see directory `HOLCF/IOA/Modelcheck`). Furthermore, in `Cockpit.ML`, the list `aut_simps` is defined, which contains all definitions of these automata. `Cockpit.ML` is automatically executed when loading theory `Cockpit` and there, also following three desired properties are verified:

1. *cockpit_hide* $\preceq_S$ *Al_before_Ack*
2. *cockpit* $\preceq_S$ *Info_while_Al*
3. *cockpit* $\preceq_S$ *Info_before_Al*

Their proofs are started by entering at each case:

1. `Goal "cockpit_hide =<| Al_before_Ack";`
2. `Goal "cockpit =<| Info_while_Al";`
3. `Goal "cockpit =<| Info_before_Al";`

At each case, we apply following proof step:

```
by (is_sim_tac aut_simps 1);
```

Here, *cockpit_hide* is defined by:

```
automaton cockpit_hide = hide "Info a" in cockpit
```

For explanations to the usage of this hiding operation in Isabelle, see the appendix. The tactic above was not enough for verifying property 1. But an application of following tactic has led to the solution:

```
by Auto_tac;
```

Here, `Auto_tac` is a tactic, which applies automatically all classical rules and simplifications of the logic HOL to the topical proof state.

In this rather simple model (*cockpit* possesses 16 states and 12 actions), the complexity for verification was still small. The average length of the produced μcke files was 200 lines, 4.5 MB of virtual memory was used and the model checker needed 0.85 seconds (see next subsection for the properties of the computing system).

5.2 The Leader Election Algorithm

We now take a look at the LeLann-Chang-Roberts algorithm (LCR) which was already modelled in [4] for I/O-automata. There, a finite number of processes is arranged to a ring structure and each process has an unique identifier. These processes must determine exactly one leader process and each process may only send messages to its right-hand neighbour.

We are modelling an algorithm, where at the beginning, each process may send its own identifier to its right neighbor. If a process receives an identifier higher than its own, it then sends the maximum of its own identifier and of all received identifiers. If a process receives its own identifier, it knows, that this identifier was sent through the whole ring and it may declare itself as leader.

We show by implementation relation that this algorithm meets the requirement that at least one process declares itself as leader. We model an I/O-automaton, where only three processed are arranged in the ring. The identifiers are of following Isabelle datatype:

```
datatype token = Leader | id0 | id1 | id2 | id3 | id4
```

We define a partial order on *token* which is similar to the canonical ordering of the natural numbers (excluding element `Leader`). For the three processes, we define automata *aut0*, *aut1* and *aut2*. The identifier of automaton *aut0* may be an `id0` or `id3`, of *aut1*, it may be `id1` or `id3` and for *aut2*, it is `id2`. For communication actions, we define following datatype:

```
datatype Comm = Zero_One token | One_Two token |
  Two_Zero token | Leader_Zero | Leader_One | Leader_Two
```

Here, `Zero_One` represents the sending operation of an identifier from *aut0* to *aut1*. `Leader_Zero` symbolizes the operation, where *aut0* declared itself to leader. The specifications of *aut0*, *aut1* and *aut2* are given in the file `Ring3.thy` in the directory `HOLCF/IOA/Modelcheck`. These automata were then assembled to a ring by composition:

```
automaton ring = compose aut0,aut1,aut2
```

Then we express the property that in automaton *ring*, at least one process declares itself to leader, by following automaton:

automaton *one_leader*
signature
outputs
$Zero\_One(t), t \in token$
$One\_Two(t), t \in token$
$Two\_Zero(t), t \in token$
$Leader\_Zero$
$Leader\_One$
$Leader\_Two$
states $leader : token$
initially $leader = Leader$
transitions
$Zero\_One(t)$ **pre** $True$
$One\_Two(t)$ **pre** $True$
$Two\_Zero(t)$ **pre** $True$
$Leader\_Zero$
pre $leader = id0 \vee leader = Leader$
post $leader := id0$
$Leader\_One$
pre $leader = id1 \vee leader = Leader$
post $leader := id1$
$Leader\_Two$
pre $leader = id2 \vee leader = Leader$
post $leader := id2$

When loading theory `Ring3`, a verification of following proof goal is executed in file `Ring3.ML`:

```
Goal "ring =<| one_leader";
```

By executing

```
by (is_sim_tac aut_simps 1);
```

with subsequent

```
by Auto_tac;
```

the verification of the proof goal was successful.

The examples were executed on a Sun ULTRA 2 with 1024 MB RAM and a 200 MHz processor. The ring size was varied and two slightly different

Ring size	#States	Isabelle preprocessing time	μcke Time	μcke Memory
3	1296	45 s	14.2 s	22.9 MB
4	7776	180 s	84.2 s	58.5 MB
3 + Semaphor	10368	45 s	179 s	203 MB
4 + Semaphor	124416	180 s	1128 s	762 MB
5 + Semaphor	1492992	390 s	∞	$>$ 1.2 GB

Table 1: Results for the LCR-algorithm

algorithms were investigated. Table 1 lists all data, where the first row corresponds to the variant chosen in theory `Ring3`.

We see, that with increasing number of states in the automaton (second column), the work to be done by μcke increases (fourth column). The simplification work done by Isabelle (third column) before calling μcke corresponds to the number of single automata being composed to a ring (first column). Whereas in the case of five ring components, Isabelle can produce an input file for μcke, μcke cannot finish the verification as the main memory lacks which results into getting a core-dump.

Here, we have reached the limits of model checker complexity. Another more fundamenatal shortcoming of this implementation relation technique is of logical kind. For proving the effectiveness of this algorithm, one should also prove, that in fact, one automaton declares itself to leader. In our automaton model we would have to introduce fairness conditions for actions and the problem remains that this property has to be expressed by a Safe I/O-automaton, where no solution exists, as this property is a liveness property [5]

6 Conclusion

We have presented an Isabelle-based verification environment for I/O-automata consisting of a three layered architecture.

1. The uppermost layer provides the user a comfortable possibility to input I/O-automata in Isabelle.
2. The middle layer translates implementation relations between automata into a formula of the μ-calculus.
3. In the bottom layer, formulas of the μ-calculus, as described in the Isabelle theory `MuCalculus`, are transformed into an input for the model checker μcke and then verified. This interface from Isabelle to μcke can be used for any formula of the logic `MuCalculus`, when observing the constraints given in section 4.1.

This verification environment is applicable for proving implementation relations between automata with finite state spaces. The implementation relation means that an automaton fulfills some properties of another automaton and therefore implements this automaton. We have also given examples with a certain complexity.

The verification environment described here may be extended further as follows. Besides the comfortable input syntax, the Isabelle system could automatically prove some basic properties about well-formedness of I/O-automata, like the disjunctness of input and output actions for each automaton or the input-enabledness for each input action. Though some constraints can be guaranteed by a simple syntax check (e.g. disjointness), one generally has to prove that in another way. Implementing proof tactics for that task, which are automatically called at an automaton input, could be a task for future works.

Furthermore, with the implementation relation, it is only possible to describe properties of sequences of actions. It is primarily suited to describe relations between two given automaton. To describe a property of a single automaton by this means, on has to describe the desired property through another automaton and then prove that the automaton implements this property automaton. As in this work only *Safe I/O-automata* were used, one is somehow restricted in formulating properties via I/O-automata.

One possibility to overcome with this restriction is to extend the concept of implementation relation on *Fair I/O-automata*. The other possibility is, to formulate properties of I/O-automata directly in a temporal logic, e.g `TLS` [12], which has already been formalized in Isabelle. To extend this temporal logic to a new verification environment one has again to choose a model checker that can verify properties of that temporal logic and couple it with Isabelle.

Together with the here introduced components, one would then have a verification environment for I/O-automata in Isabelle with complementing techniques. On the one hand, abstraction proofs[4] on I/O-automata and preprocessig a model checker input are done in Isabelle, on the other hand, verification of abstract systems is delegated to model checkers. There, we have model checkers like μcke for primarily proving relations between automata and other model checkers for directly proving properties of automata.

[4] Abstaction proofs were not discussed here. It means the task to verify that instead of observing directly a given automaton, a more suitable, abstract version (which is for example small enough for model checkers) can be observed. For example, the helicopter alarm system in the here described case study is an abstraction of the real system (see [6]). Also this process of abstraction proofs can be supported by model checkers (again [6]). For a detailed formal description, see [12].

A Appendix

A.1 How to run the verification environment

Make sure that you have a running Isabelle session with the `IOA` logic loaded. It is important, that you use an Isabelle version which contains the files `ioa_syn.ML` and `ioa_package.ML` in the directory `HOLCF/IOA/meta_theory` of its sources. These files are necessary for using the IOA syntax. For using the model checker for verifying implementation relations, you need the theory `MuIOAOracle`, whose sources are in the subdirectory `HOLCF/IOA/Modelcheck`.

For using the model checker μcke *without* I/O-automata, you only need to have a running Isabelle session with the theories of `HOL/Modelcheck` loaded. It is important, that you use an Isabelle version which contains the file `mucke_oracle.ML` in this directory. Besides, you must have the model checker μcke installed on your system (see Armin Biere's home page [3]). To make it useable in Isabelle, you have to set an environment variable `MUCKE_HOME` containing the path of the mucke binary, which you can, for example, do in the file `etc/settings` of your Isabelle version.

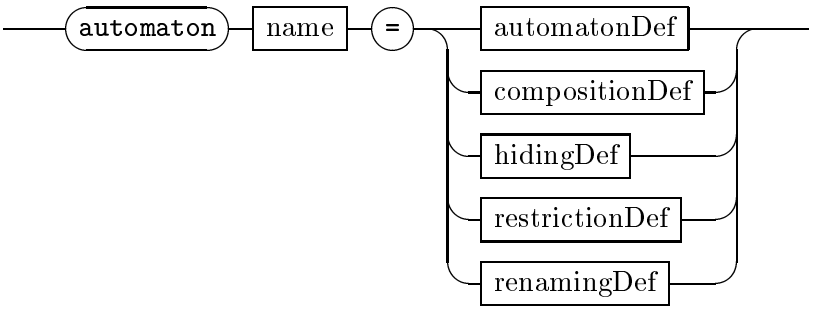
A.2 The Syntax Diagram for I/O-Automata in Isabelle

The following diagrams present the syntax for I/O-automata in Isabelle. In section 2, already a part was shown in a more abstract way. You can find its counterpart in Isabelle, if you follow the pattern *automatonDef* after beginning at *ioaDef*. Besides, if you begin in *ioaDef*, you can see also the syntax definitions for compositions of automata and hiding, restriction and renaming in an automaton. Following patterns were not specified further, but we will explain what they mean and which syntax to use for them if they do not correspond directly to an Isabelle syntax element (see [15]). For better understanding, see also the example in section 2.

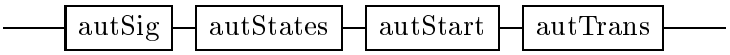
- At *automatonName*, you must write a name of an already defined Isabelle I/O-automaton.
- At *datatype*, you must write a name which corresponds to a datatype already defined in Isabelle. At *action*, you must write a term in Isabelle syntax, which is element of its proper datatype, and which is embraced by quotation marks.
- At *predicate*, you have to write a boolean term in quotes. At *term*, you have to write a term of proper type as quoted string.

- *function* has to be an Isabelle function term as quoted string. Here, it occurs at the definition of an automaton renaming. Please be careful about its semantics[5]!

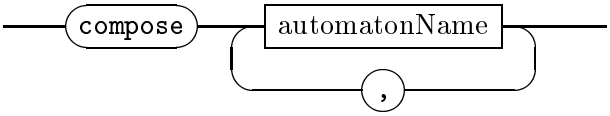
ioaDef



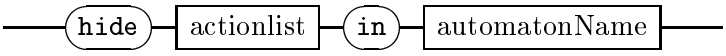
automatonDef



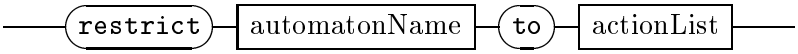
compositionDef



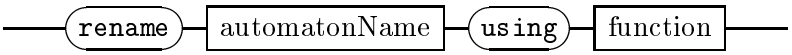
hidingDef



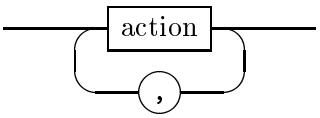
restrictionDef



renamingDef

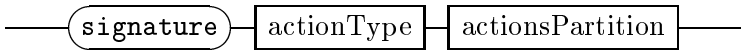


actionList

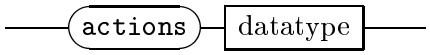


[5] In contrast to the usual understanding of a renaming function, we define as renaming function a *partial* function of elements of the action type of the result automaton to elements of the action type of the source automaton. It is the inverse of the usual renaming operation.

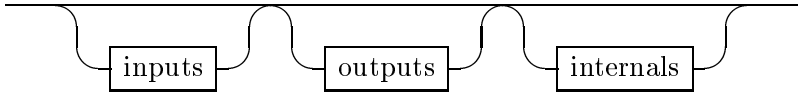
autSig



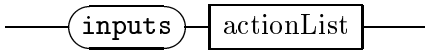
actionType



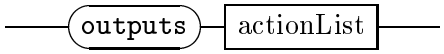
actionsPartition



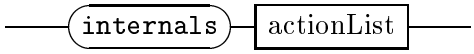
inputs



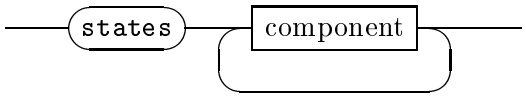
outputs



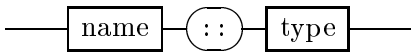
internals



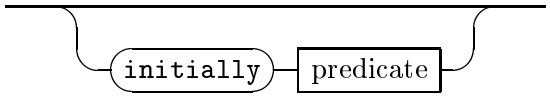
autStates



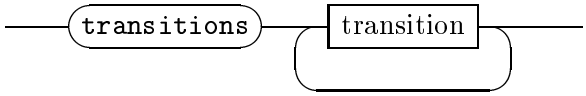
component



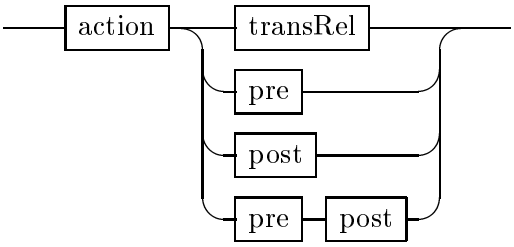
autStart



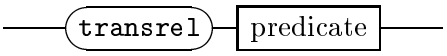
autTrans



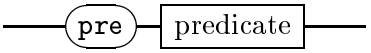
transition



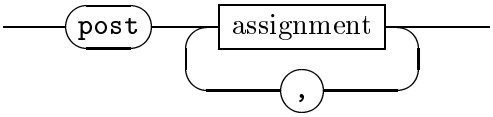
transRel



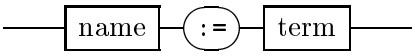
pre



post



assignment



References

[1] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, L.J. Hwang: *Symbolic model checking:* 10^{20} *states and beyond,* in Information and Computation, 98(2):142-170 (1992)

[2] Armin Biere: *Effiziente Modellprüfung des μ-Kalküls mit binären Entscheidungsdiagrammen,* PhD Thesis, Institut für Informatik, Universität Karlsruhe (1997)

[3] Sources of the Model Checker μcke: `http://www.iseran.ira.uka.de/~armin/mucke/`

[4] Stephen Garland, Nancy Lynch, Mandana Vaziri: *IOA: A language for specifying, programming, and validating distributed systems,* Technical report, Laboratory for Computer Science, MIT, Cambridge, MA (1997)

[5] R. Gawlick, R. Segals, J.F. Sogaard-Andersen, N.A. Lynch: *Liveness in timed and untimed systems,* Technical Report MIT/LCS/TR-587, Laboratory for Computer Science, MIT, Cambridge, MA (1993)

[6] Tobias Hamberger: *Verifikation einer Hubschrauberüberwachungskomponente mit Isabelle und STeP,* Systementwicklungsprojekt, Technische Universität München (1998)

[7] Tobias Hamberger: *Integration von Theorembeweisen und Model Checking für I/O-Automaten,* Diplomarbeit, Technische Universität München (1999)

[8] The Isabelle Theory Library: `http://isabelle.in.tum.de/library/`

[9] Nancy Lynch, Mark Tuttle: *An introduction to Input/Output automata,* in CWI Quarterly, 2(3):219-246 (1989)

[10] Olaf Müller and Tobias Nipkow: *Combining Model Checking and Deduction for I/O-Automata,* in TACAS'95, Proc. of the 1st International Workshop on Tools and Algorithms for the Construction and Analysis of Systems, LNCS vol. 1019, Springer Verlag (1995)

[11] Olaf Müller and Tobias Nipkow: *Traces of I/O-Automata in Isabelle/HOLCF,* in TAPSOFT'97, Proc. 7th Int. Joint Conf. on Theory and Practice of Software Development, LNCS vol. 1214, Springer Verlag (1997)

[12] Olaf Müller: *A Verification Environment for I/O Automata based on Formalized Meta-Theory,* PhD Thesis, Institut für Informatik, Technische Universität München (1998)

[13] Tobias Nipkow and Konrad Slind: *I/O automata in Isabelle/HOL,* LNCS vol. 996, Springer Verlag (1995)

[14] S. Owre, S. Rajan, J.M. Rushby, N. Shankar, M. Srivas: *PVS: combining specification, proof checking and model checking,* LNCS vol. 1102, Springer Verlag (1996)

[15] Lawrence C. Paulson: *Isabelle: A Generic Theorem Prover,* LNCS vol. 828, Springer Verlag (1994)

[16] Jan Philipps and Oscar Slotosch: *Case Study Avionics: Verification in Spin,* Internal Report, Technische Universität München (1997)

[17] Rober Sandner, Link to the Interface between the Isabelle Theory `MuCalculus` and μcke: `http://isabelle.in.tum.de/library/HOL/Modelcheck/`