T U M

I N S T I T U T F Ü R I N F O R M A T I K

A Verification Environment for I/O Automata – Part II: Theorem Proving and Model Checking –

Olaf Müller

TUM-I9912
Juni 99

T E C H N I S C H E U N I V E R S I T Ä T M Ü N C H E N

TUM-INFO-06-I9912-50/1.-FI

Druck: Institut für Informatik der
Technischen Universität München

A Verification Environment for I/O Automata

– Part II: Theorem Proving and Model Checking –

Olaf Müller*
Institut für Informatik, Technische Universität München, Germany.
Email: mueller@in.tum.de

Abstract

We describe a verification framework for I/O automata in Isabelle. It includes a temporal logic, proof support for showing implementation relations between live I/O automata, and a combination of Isabelle with model checking via a verified abstraction theory. The underlying domain-theoretic sequence model turned out to be especially adequate for these purposes. Furthermore, using a tailored combination of Isabelle's logics HOL and HOLCF we achieve two complementary goals: expressiveness for proving meta theory (HOLCF) and simplicity and efficiency for system verification (HOL).

1 Introduction

I/O automata [26, 15] are used to model reactive, distributed systems. In this paper we present an extensive framework for the verification of I/O automata in Isabelle, combined with model checking tools. This framework is based upon several extensions to the standard theory of I/O automata which are described in part I of this paper [31].

These extensions comprise first of all a linear-time temporal logic, called Temporal Logic of Steps (TLS), which is similar to TLA [23], but evaluates formulas over sequences of alternating states and actions, which in addition may be finite. The applications of TLS are twofold. First, it can be used to define and reason about live I/O automata by establishing live implementation relations. Second, TLS can be employed as a property specification language for I/O automata. Furthermore, in [31] for both TLS applications abstraction rules have been developed which allow to reduce reasoning about a large or even infinite automaton to a finite and smaller automaton. Together with translations to appropriate model checkers this forms the basis for an effective combination of Isabelle with a model checker as external oracle.

In this paper we describe how all these notions have been embedded in Isabelle. The aim is to build a setting, in which the theory of I/O automata itself is verified (we only use definitional theory extensions), but that at the same time enables efficient system verification for the user. This is accomplished by combining Isabelle's object logics HOL and HOLCF in such a way, that the user employs only the simpler logic HOL, whereas the use of the more expressive, but difficult HOLCF is restricted to meta-theoretic arguments.

The framework is based upon a semantic model of lazy lists using Scott's domain theory, as provided by Isabelle/HOLCF [32]. In a comparison [11] to other sequence formalizations,

*Research supported by BMBF, *KorSys*

which all incorporated functions on natural numbers in some way, this sequence model turned out to be the most adequate.

TLS is not encoded directly, but as an instance of a generic temporal logic TL, which is evaluated over state sequences. This enables us to study the adequateness of our domain-theoretic sequence model in a more general setting, and, furthermore, reveals the connection to standard temporal logics over state sequences [28, 21].

The abstraction theory generates proof obligations which are delegated to model checking tools. for this purpose, translations are given to the STeP model checker [5] and to μcke [4]. Both translations are illustrated by means of a simple example.

Due to lack of space we only present the main definitions and theorems and omit proofs. For full details concerning the entire developments in Isabelle the interested reader is referred to the author's PhD thesis [30]. Furthermore, the theories are part of the Isabelle distribution which is available at the following www-page:
`http://www.in.tum.de/~isabelle/library/HOLCF/IOA`.

1.1 Structure of the Paper

The paper is organized as follows. In §2 we briefly introduce the tools used in our framework. In §3 and §4 the models for sequences and basic I/O automata are presented. §5 describes the generic temporal logic TL, §6 its instance to the temporal logic of steps TLS. In §7 TLS is used to describe live I/O automata. Abstraction rules are derived in §8, which are applied to the combination of Isabelle with model checking in §9. In §10 and §11 we present some related work and conclude.

2 Preliminaries

In this section we briefly introduce the components of our toolbox.

2.1 The Theorem Proving System

Isabelle [36] is a generic theorem proving environment that supports a number of object logics. We only use Isabelle's instantiation of higher-order logic (HOL) and its extension to domain theory (HOLCF) [32].

HOL is based on Church's formulation of simple type theory [9], which has been augmented by polymorphism, type classes like in Haskell, and extension mechanisms for defining new constants and types. The syntax is that of simply typed λ-calculus with an ML-style first order language of types. In this paper we employ standard mathematical notation, which, however, differs only slightly from the syntax in Isabelle. Type abbreviations, constant declarations, definitions and theorems are introduced by the keywords **types**, **consts**, **defs**, and **thms**, respectively.

HOLCF conservatively extends HOL with concepts of domain theory such as complete partial orders, continuous functions and a fixpoint operator. Whereas HOL is restricted to total functions, HOLCF allows arbitrary recursive function definitions and is therefore especially useful for handling infinite or partial objects. In HOLCF Isabelle's type classes are used to distinguish between HOL types and domains. We write τ_C if a type τ is of class C. The default type class of HOL is *term*, the default type class of HOLCF is *pcpo*. The latter is equipped with a complete partial order $\sqsubseteq$ and a least element $\perp$. There is a special type for continuous function between *pcpos*, the type constructor is denoted by $\to_c$ in contrast to the standard HOL constructor $\to$. Abstraction and application of continuous functions is denoted by Λ (instead of λ) and $f\,`t$ (instead of $f\ t$). There is a tailored tactic that discharges admissibility obligations. HOLCF includes a datatype package that allows the convenient definition of recursive domains.

2.2 The Model Checking Tools

STeP [5] is a theorem proving environment, comprising several decision procedures and a LTL model checker. In comparison to Isabelle, it is tailored to the specific needs of the logic of Manna/Pnueli [28] and thus offers much less flexibility and extensibility. In our tool box we make only use of STeP's model checker in order to verify temporal formulas.

μcke [4] is a model checker for Park's μ-calculus[1], based on BDD technology. We use it in order to verify implementation relations between I/O automata.

3 Finite and Infinite Sequences

Using the HOLCF domain package, possibly infinite sequences are defined by the simple recursive domain equation

$$\textbf{domain}\quad (\alpha)\,seq = nil \ \mid\ HD\ \alpha \sharp (\textbf{lazy}\ TL\ (\alpha)\,seq)$$

where *nil* and the right-associative "cons"-operator $\sharp$ are the constructors and *HD* and *TL* the selectors of the datatype. As $\sharp$ is strict in its first argument and lazy in the second, sequences of type $(\alpha)\,seq$ come in three flavors: finite total (ending with *nil*), finite partial (ending with $\perp$), and infinite.

Due to the domain constructions performed by the domain package, the definition of $(\alpha)seq$ requires the argument type to be in type class *pcpo*. However, it will turn out to be crucial that elements of sequences can be handled in plain HOL. Therefore types of class *term* are lifted to flat domains by introducing a type constructor $(\alpha)\,lift$ using the HOL datatype package

$$\textbf{datatype}\quad (\alpha_{term})\,lift = Undef \mid Def(\alpha_{term})$$

and defining the least element and the approximation ordering as

$$\begin{array}{lrcl} \textbf{defs} & \perp & \equiv & Undef \\ & x \sqsubseteq y & \equiv & (x = Undef \lor x = y) \end{array}$$

Note that $\perp$ and $\sqsubseteq$ are overloaded and this definition only fixes their meaning at type $(\alpha_{term})lift$. We define an unpacking function $the :: (\alpha)\,lift \to \alpha$ such that $the\,(Def\ x) = x$ and $the(Undef) = arbitrary$ where *arbitrary* is a fixed, but unknown value.

Now we can define a type of sequences that permits elements of type class *term* together with a corresponding "cons"-operator:

$$\begin{array}{ll} \textbf{types} & (\alpha_{term})\,sequence = ((\alpha_{term})\,lift)\,seq \\ \textbf{consts} & Cons :: \alpha_{term} \to (\alpha_{term})\,sequence \to_c (\alpha_{term})\,sequence \\ \textbf{defs} & Cons \equiv \lambda x.\,\Lambda xs.\,(Def\ x) \sharp xs \end{array}$$

Isabelle's syntax mechanism is used to write $x\hat{\ }xs$ instead of $Cons\,x\,`xs$. Finite sequences $a_1\hat{\ }\ldots\hat{\ }a_n\hat{\ }nil$ are abbreviated by $[a_1,\ldots,a_n!]$ and partial sequences $a_1\hat{\ }\ldots\hat{\ }a_n\hat{\ }\perp$ by $[a_1,\ldots,a_n?]$. The corresponding

[1]For an introduction into syntax and semantics of Park's μ-calculus see [34, 7]. We assume a typed version with finite carrier sets (see [3]). Park's μ-calculus can express any property of the modal μ-calculus [20].

sequence flavors are characterized by the predicates *Finite* and *Partial*. From now on the default type class is assumed to be *term*, thus we will omit the explicit typing subscripts.

Recursive functions on sequences are defined as fixpoints, from which the characterizing recursive equations are derived automatically by a tactic. For example, *Map* has type

$$\textbf{consts}\quad Map \ :: \ (\alpha \to \beta) \to (\alpha)\,sequence \to_c (\beta)\,sequence$$

and the following rewrite rules

$$\begin{array}{llcl}\textbf{thms} & Map\,f\,`\bot & = & \bot \\ & Map\,f\,`nil & = & nil \\ & Map\,f\,`(x\hat{\ }xs) & = & f(x)\hat{\ }Map\,f\,`xs\end{array}$$

are automatically derived from the definition

$$\begin{array}{lll}\textbf{defs}\quad Map\,f = fix`(\Lambda h.\Lambda s\,.\,\textbf{case}\ s\ \textbf{of} & nil & \Rightarrow\ nil \\ & \mid (x\hat{\ }xs) & \Rightarrow\ f(x)\hat{\ }(h`xs))\end{array}$$

According to domain theory, the argument of *fix* in this definition has to be a continuous function in order to guarantee the existence of the least fixed point. This continuity requirement is handled automatically by type checking, as every occuring function is constructed using the continuous function type $\to_c$.

As the characterizing equations are derived automatically, we will omit the fixpoint definition from now on. The equations for $\oplus$ (concatenation) and *Filter* are given below.

$$\begin{array}{llcl}\textbf{consts} & \oplus & :: & (\alpha)\,sequence \to_c (\alpha)\,sequence \to_c (\alpha)\,sequence \\ & Filter & :: & (\alpha \to bool) \to (\alpha)\,sequence \to_c (\alpha)\,sequence \\ \textbf{defs} & \bot \oplus y & = & \bot \\ & nil \oplus y & = & y \\ & (x\hat{\ }xs) \oplus y & = & x\hat{\ }(xs \oplus y) \\ \\ & Filter\ P\,`\bot & = & \bot \\ & Filter\ P\,`nil & = & nil \\ & Filter\ P\,`(x\hat{\ }xs) & = & \textbf{if}\ P(x)\ \textbf{then}\ x\hat{\ }Filter\ P\,`xs \\ & & & \qquad\qquad \textbf{else}\quad Filter\ P\,`xs\end{array}$$

Boolean predicates on sequences can be defined by means of an auxiliary continuous predicate, which yields one of the truth values $\bot$, TT, or FF of the domain *tr* of truthvalues. As an example we present the *Forall* predicate, which uses an auxiliary $Forall_c$ predicate (and the conjunction *andalso* on *tr*):

$$\begin{array}{llcl}\textbf{consts} & Forall & :: & (\alpha \to bool) \to (\alpha)\,sequence \to bool \\ & Forall_c & :: & (\alpha \to bool) \to (\alpha)\,sequence \to_c tr \\ \textbf{defs} & Forall_c\ P\,`\bot & = & \bot \\ & Forall_c\ P\,`nil & = & TT \\ & Forall_c\ P\,`(x\hat{\ }xs) & = & Def(P\ x)\ andalso\ Forall_c\ P\,`xs \\ \\ \textbf{thms} & Forall\ P\ xs & \equiv & Forall_c\ P\,`xs \neq FF \\ & Forall\ P\ \bot & = & True \\ & Forall\ P\ nil & = & True \\ & Forall\ P\ (x\hat{\ }xs) & = & P(x)\ \wedge\ Forall\ P\ xs\end{array}$$

The usual proof principle for sequences is *structural induction*. In contrast to finite structural induction it contains an admissibility requirement (adm).

$$\textbf{thms}\quad \frac{adm(P) \quad P(\bot) \quad P(nil) \quad \forall x, xs.\,P(xs) \Rightarrow P(x\hat{\ }xs)}{\forall y.\,P(y)}\ (induct)$$

Of course, the finite version is available as well.

$$\textbf{thms}\qquad \frac{P(nil) \quad \forall x, xs.\, P(xs) \wedge Finite(xs) \Rightarrow P(x\hat{\ }xs)}{\forall y.\, Finite(y) \Rightarrow P(y)}\ (\mathit{fin\text{-}induct})$$

Finally, there are co-inductive proof principles, namely the *take-lemma*

$$\textbf{thms}\qquad \frac{\forall n.\, take\ n\,\text{`}s = take\ n\,\text{`}t}{s = t}\ (\mathit{take\text{-}lemma})$$

and the *bisimulation* rule, which follows easily from the take lemma.

$$\textbf{thms}\qquad \frac{bisim(R) \quad (s,t) \in R}{s = t}\ (\mathit{bisimulation})$$

Here, the predicate *bisim* expressing bisimularity is defined as follows:

$$\begin{array}{lll} \textbf{consts} & bisim & :: \quad ((\alpha)\, sequence \times (\alpha)\, sequence)\, set \rightarrow bool \\ \textbf{defs} & bisim(R) & \equiv \quad \forall s\ t.\, (s,t) \in R \Rightarrow \\ & & \qquad (s = \bot \Rightarrow t = \bot) \wedge \\ & & \qquad (s = nil \Rightarrow t = nil) \wedge \\ & & \qquad (\exists s\ s'.\, s = a\hat{\ }s' \Rightarrow \exists b\ t'.\, t = b\hat{\ }t' \wedge (s',t') \in R \wedge a = b) \end{array}$$

Thus, we get a sequence package, which allows powerful recursion like infinite concatenation. Furthermore, in contrast to approaches which model sequences as functions on the natural numbers, operations like *Filter* that relocate elements in an unhomogenous way are treated easily. See [11] for an elaborate comparison of our sequence model with other formalizations, where our approach turned out to be the most adequate.

Furthermore, note the advantages of *lifted* sequence elements: proof procedures tailored for two-valued logic may be employed (cf. the $\wedge$ in the last equation for *Forall*), and HOL theories and libraries may be reused. About 170 theorems have been derived in the Isabelle setting, most of them in one step by a tailored induction tactic.

4 Safe I/O Automata

In the sequel we merely sketch the embedding of safe I/O automata in Isabelle. For more details see the author's PhD thesis [30].

4.1 Basic I/O Automata

An *action signature* models different types of actions and is described as

$$\textbf{types}\quad (\alpha)\, signature = (\alpha)\, set \times (\alpha)\, set \times (\alpha)\, set$$

where the components may be extracted by the selector functions *inputs*, *outputs*, and *internals*, respectively. We collectively refer to *internals* and *outputs* as *locals*, and to *outputs* and *inputs* as *externals*. The union of all three action sets, which always have to be disjoint, is denoted by *actions*.

A *safe I/O automaton* is a triple of an action signature, a set of start states, and a set of transition triples (called *steps*) described by the type

$$\textbf{types}\quad (\alpha,\sigma)\, ioa \quad = \quad (\alpha)\, signature \times (\sigma)\, set \times (\sigma \times \alpha \times \sigma)\, set$$

where the components may be extracted by the functions *sig-of*, *starts-of*, and *trans-of*, respectively. We write $s \stackrel{a}{\rightarrow}_A t$ for $(s,a,t) \in \mathit{trans\text{-}of}(A)$. Furthermore the abbreviations *act*, *ext*, *int*, *in*, *out*, and *local* are introduced for *actions* $\circ$ *sig-of*, *externals* $\circ$ *sig-of*, *internals* $\circ$ *sig-of*, *inputs* $\circ$ *sig-of*, *outputs* $\circ$ *sig-of*, and *locals* $\circ$ *sig-of*, respectively.

There are several well-formedness requirements posed on safe I/O automata which are expressed by the predicate *is-safe-IOA*. It demands that the first component be an action signature, the second be an non-empty set of start states and the third be an input-enabled state transition relation, whose actions stem from the action signature:

$$
\begin{array}{llcl}
\textbf{defs} & \textit{is-sig-of}(A) & \equiv & \textit{is-sig}\ (\textit{sig-of}\ A) \\
& \textit{is-starts-of}(A) & \equiv & \textit{starts-of}(A) \neq \{\} \\
& \textit{is-trans-of}(A) & \equiv & \forall (s,a,t) \in \textit{trans-of}(A).\ a \in \textit{act}(A) \\
& \textit{input-enabled}(A) & \equiv & \forall a \in \textit{in}(A).\ \forall s.\ \exists t.\ s \stackrel{a}{\rightarrow}_A t \\
& \textit{is-safe-IOA}(A) & \equiv & \textit{is-sig-of}(A) \wedge \textit{is-starts-of}(A) \wedge \\
& & & \textit{is-trans-of}(A) \wedge \textit{input-enabled}(A)
\end{array}
$$

The set of reachable states of an I/O automaton A is defined inductively as the least set of states satisfying the following two rules:

$$
\textbf{inductive} \quad \frac{s \in \textit{starts-of}(A)}{s \in \textit{reachable}(A)}\ (\textit{reach}\text{-}0) \qquad \frac{s \in \textit{reachable}(A) \qquad s \stackrel{a}{\rightarrow}_A t}{t \in \textit{reachable}(A)}\ (\textit{reach}\text{-}n)
$$

Isabelle's syntax translation mechanism is used to write *reachable* $A\ s$ for $s \in \textit{reachable}\ A$.

A state predicate $P :: \sigma \rightarrow \textit{bool}$ is called an invariant of an I/O automaton A if it holds for all reachable states:

$$
\textbf{defs} \quad \textit{invariant}\ A\ P \ \equiv\ (\forall s.\ \textit{reachable}\ A\ s \Rightarrow P(s))
$$

For invariants the following associated proof rule has been derived.

$$
\textbf{thms} \quad \frac{(\forall s.\ s \in \textit{starts-of}(A) \Rightarrow P(s)) \qquad (\forall s\ a\ t.\ \textit{reachable}\ A\ s \wedge P(s) \wedge s \stackrel{a}{\rightarrow}_A t \Rightarrow P(t))}{\textit{invariant}\ A\ P}
$$

There are composition operators for parallel composition, hiding of internal actions, and renaming. We present merely the parallel composition operator $\|$:

$$
\begin{array}{llcl}
\textbf{consts} & \| & :: & (\alpha,\sigma)\,\textit{ioa} \rightarrow (\alpha,\tau)\,\textit{ioa} \rightarrow (\alpha,\sigma\times\tau)\,\textit{ioa} \\
\textbf{defs} & A \,\|\, B & \equiv & (\textit{sig-comp}\ (\textit{sig-of}\,A)\ (\textit{sig-of}\,B), \\
& & & \ \{(u,v) \mid u \in \textit{starts-of}(A) \wedge v \in \textit{starts-of}(B)\}, \\
& & & \ \{(s,a,t) \mid\ (a \in \textit{act}(A) \vee a \in \textit{act}(B)) \qquad\qquad \wedge \\
& & & \qquad\quad \textbf{if}\ a \in \textit{act}(A)\ \textbf{then}\ (\textit{fst}\ s) \stackrel{a}{\rightarrow}_A (\textit{fst}\ t) \\
& & & \qquad\qquad\qquad\qquad\quad \textbf{else}\ (\textit{fst}\ s) = (\textit{fst}\ t) \quad \wedge \\
& & & \qquad\quad \textbf{if}\ a \in \textit{act}(B)\ \textbf{then}\ (\textit{snd}\ s) \stackrel{a}{\rightarrow}_B (\textit{snd}\ t) \\
& & & \qquad\qquad\qquad\qquad\quad \textbf{else}\ (\textit{snd}\ s) = (\textit{snd}\ t)\})
\end{array}
$$

Here, *sig-comp* defines the composition of signatures as follows:

$$
\begin{array}{llcl}
\textbf{consts} & \textit{sig-comp} & :: & (\alpha)\,\textit{signature} \rightarrow (\alpha)\,\textit{signature} \rightarrow (\alpha)\,\textit{signature} \\
\textbf{defs} & \textit{sig-comp}\ s_1\ s_2 & \equiv & ((\textit{inputs}(s_1) \cup \textit{inputs}(s_2)) \setminus (\textit{outputs}(s_1) \cup \textit{outputs}(s_2)), \\
& & & \ \textit{outputs}(s_1) \cup \textit{outputs}(s_2), \\
& & & \ \textit{internals}(s_1) \cup \textit{internals}(s_2))
\end{array}
$$

For parallel composition compatibility is required which states that each action is an output action of at most one I/O automaton and that internal action names are unique.

$$
\begin{array}{llcl}
\textbf{consts} & \textit{compatible} & :: & (\alpha,\sigma)\,\textit{ioa} \rightarrow (\alpha,\tau)\,\textit{ioa} \rightarrow \textit{bool} \\
\textbf{defs} & \textit{compatible}\ A\ B & \equiv & (\textit{out}(A) \cap \textit{out}(B) = \{\}) \wedge \\
& & & (\textit{act}(A) \cap \textit{int}(B)\ = \{\}) \wedge \\
& & & (\textit{act}(B) \cap \textit{int}(A)\ = \{\})
\end{array}
$$

4.2 Isabelle Syntax of I/O Automata

So far, we defined I/O automata as parameterized tuples over arbitrary state and action types. In the sequel we will describe how concrete state spaces and actions are represented in Isabelle.

Actions are defined easily using Isabelle's **datatype** construct. States are not represented by variables, but as tuples, where each component represents a variable. Selector functions are introduced, whose names are identical to the variable names in the informal description. This approach conforms with [26].

Let us illustrate this format with the simple example of a buffer. The buffer *Buf* is modeled by a variable *queue* of type $(bool)list$, which is initially empty. Actions and transitions are given in the usual precondition/effect style as:

input $S(m), m \in bool$	**output** $R(m), m \in bool$
post: $queue := queue@[m]$	**pre**: $queue = m : rst$
	post: $queue := rst$

In Isabelle the action type of *Buf* is defined as **datatype** $action = S(bool) \mid R(bool)$. The automaton *Buf* is then internally defined as follows, where *queue* is the identity on the (trivial) tuple of type $(bool)list$, and transitions are represented in a set comprehension format.

$$\begin{array}{lcl}
Buf & \equiv & (Buf\text{-}sig, \{[]\}, Buf\text{-}trans, \{\}, \{\}) \\
Buf\text{-}sig & \equiv & (\bigcup_{m \in bool}\{S(m)\}, \bigcup_{m \in bool}\{R(m)\}, \{\}) \\
Buf\text{-}trans & \equiv & \{(s, a, s') \mid \textbf{case } a \textbf{ of} \\
& & \qquad\qquad\qquad S(m) \;\Rightarrow\; queue(s') = queue(s)@[m] \\
& & \qquad\qquad\quad \mid\; R(m) \;\Rightarrow\; queue(s) = m : rst \wedge queue(s') = rst\}
\end{array}$$

There is an automatic translation of the precondition/effect style into the set comprehension format. It is described in [18]. Therefore the user may always stay within his familiar specification format.

4.3 Executions and Traces

Execution fragments of an I/O automaton A are (1) finite or infinite sequences of alternating states s_i and actions a_i, where (2) triples $s_i\, a_i\, s_{i+1}$ represent steps of A. The first condition is encoded into the type of an execution fragment, which is modeled by a pair of a start state and a sequence of action/state pairs:

$$\textbf{types} \quad (\alpha, \sigma)\, exec = \sigma \times (\alpha \times \sigma)\, sequence$$

The second condition is captured by the predicate *is-exec-frag*, which checks recursively if all transitions are steps of A.

$$\begin{array}{llcl}
\textbf{consts} & is\text{-}exec\text{-}frag & :: & (\alpha,\sigma)\, ioa \to (\alpha,\sigma)\, exec \to bool \\
\textbf{defs} & is\text{-}exec\text{-}frag\ A\ (s, \bot) & = & True \\
& is\text{-}exec\text{-}frag\ A\ (s, nil) & = & True \\
& is\text{-}exec\text{-}frag\ A\ (s, (a,t)\hat{}\, ex) & = & s \xrightarrow{a}_A t \wedge is\text{-}exec\text{-}frag\ A\ (t, ex)
\end{array}$$

The derivation of the equations for *is-exec-frag* is analogous to that for *Forall*. *Executions* are execution fragments beginning with a start state.

$$\textbf{defs} \quad execs(A) \;\equiv\; \{(s, ex).\ s \in starts\text{-}of(A) \wedge is\text{-}exec\text{-}frag\ A\ (s, ex)\}$$

A *trace* of A is the subsequence of external actions of an execution of A. It describes the visible behaviour of A.

$$\begin{array}{llcl}
\textbf{consts} & mk\text{-}trace & :: & (\alpha,\sigma)\, ioa \to (\alpha \times \sigma)\, sequence \to_c \alpha\, sequence \\
\textbf{defs} & mk\text{-}trace\ A & \equiv & \Lambda ex.\ Filter\ (\lambda a.\ a \in ext\ A)\,`(Map\ fst\,`ex) \\
& traces(A) & \equiv & \{mk\text{-}trace\ A\,`ex \mid_{ex} \exists s.\ (s, ex) \in execs(A)\}
\end{array}$$

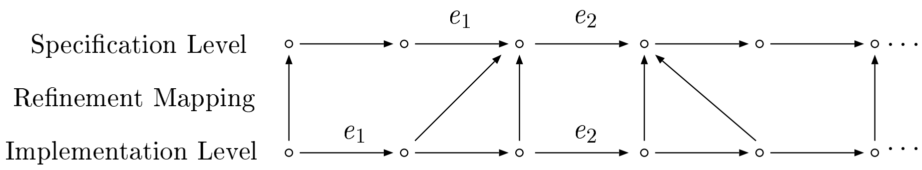

Figure 1: Refinement mapping: e_i are external actions, internal actions are omitted.

4.4 Refinement Notions and Compositionality

Safe implementation w.r.t. two I/O automata C and A is defined via trace inclusion. Furthermore, the external actions have to be the same.

$$\begin{array}{lll} \textbf{defs} \quad C \preceq_S A & \equiv & \mathit{in}(C) = \mathit{in}(A) \wedge \mathit{out}(C) = \mathit{out}(A) \wedge \\ & & \mathit{traces}(C) \subseteq \mathit{traces}(A) \end{array}$$

Such implementation relations between C and A are shown by *simulations*. Here we present only *forward simulations* and the simpler *refinement mappings* (see Fig. 1). A refinement mapping f is a function between the state spaces of C and A that maps every start state of C to a start state of A and guarantees for every step $s \stackrel{a}{\rightarrow}_C t$ of C the existence of a corresponding *move* of A, i.e. a finite execution fragment with first state $f(s)$, last state $f(t)$ and external behaviour a. A *move* is formalized by the predicate *is-move*.

$$\begin{array}{ll} \textbf{consts} & \mathit{is\text{-}move} :: (\alpha,\sigma)\,\mathit{ioa} \rightarrow (\alpha \times \sigma)\,\mathit{sequence} \rightarrow (\sigma \times \alpha \times \sigma) \rightarrow \mathit{bool} \\ \textbf{defs} & \mathit{is\text{-}move}\ A\ \mathit{ex}\ (s, a, t) \equiv \\ & \quad \mathit{is\text{-}exec\text{-}frag}\ A\ (s, \mathit{ex}) \wedge \mathit{Finite}(\mathit{ex}) \wedge \\ & \quad \mathit{last\text{-}state}\ (s, \mathit{ex}) = t \wedge \\ & \quad \mathit{mk\text{-}trace}\ A\ `\mathit{ex} = (\textbf{if}\ a \in \mathit{ext}(A)\ \textbf{then}\ [a!]\ \textbf{else}\ \mathit{nil}) \end{array}$$

where $\mathit{last\text{-}state} :: (\alpha,\sigma)\,\mathit{exec} \rightarrow \sigma$ denotes the final state of an execution, if it is finite, otherwise an unspecified value. The predicate *is-ref-map* characterizes refinement mappings.

$$\begin{array}{ll} \textbf{consts} & \mathit{is\text{-}ref\text{-}map} :: (\sigma_1 \rightarrow \sigma_2) \rightarrow (\alpha,\sigma_1)\,\mathit{ioa} \rightarrow (\alpha,\sigma_2)\,\mathit{ioa} \rightarrow \mathit{bool} \\ \textbf{defs} & \mathit{is\text{-}ref\text{-}map}\ f\ C\ A \equiv \\ & \quad (\forall s_0 \in \mathit{starts\text{-}of}(C).\ f(s_0) \in \mathit{starts\text{-}of}(A)) \wedge \\ & \quad (\forall s\ t\ a.\ \mathit{reachable}\ C\ s \wedge s \stackrel{a}{\rightarrow}_C t \\ & \qquad\qquad \Rightarrow \exists \mathit{ex}.\ \mathit{is\text{-}move}\ A\ \mathit{ex}\ (f\ s, a, f\ t)) \end{array}$$

Forward simulations are defined by the predicate *is-simulation*:

$$\begin{array}{llll} \textbf{consts} & \mathit{is\text{-}simulation} & :: & (\sigma_1 \times \sigma_2)\,\mathit{set} \rightarrow (\alpha,\sigma_1)\,\mathit{ioa} \rightarrow (\alpha,\sigma_2)\,\mathit{ioa} \rightarrow \mathit{bool} \\ \textbf{defs} & \mathit{is\text{-}simulation}\ R\ C\ A & \equiv & (\forall s_0 \in \mathit{starts\text{-}of}(C).\ R[s_0] \cap \mathit{starts\text{-}of}(A) \neq \{\}) \wedge \\ & & & (\forall s\ s'\ t\ a.\ \mathit{reachable}\ C\ s \wedge s \stackrel{a}{\rightarrow}_C t \wedge (s, s') \in R \\ & & & \qquad\qquad \Rightarrow \exists t'\ \mathit{ex}.\ (t, t') \in R \wedge \mathit{is\text{-}move}\ A\ \mathit{ex}\ (s', a, t')) \end{array}$$

Refinement mappings are specific forward simulations:

$$\textbf{thms} \qquad \frac{\mathit{is\text{-}ref\text{-}map}\ f\ C\ A}{\mathit{is\text{-}simulation}\ \{(i, o).\ f(i) = o\}\ C\ A}$$

The correctness of both concepts is therefore established by the following theorem:

$$\textbf{thms} \qquad \frac{\mathit{is\text{-}simulation}\ R\ C\ A \quad \mathit{in}(C) = \mathit{in}(A) \wedge \mathit{out}(C) = \mathit{out}(A)}{C \preceq_S A}$$

Note the following important methodological point: the correctness theorem above has been proved making heavy use of HOLCF because it involves recursively defined sequences. However, the predicate *is-simulation* can be shown in the simpler logic HOL. Therefore actual refinement proofs in applications can be done in HOL, whereas the more powerful but at the same time more complicated domain theory is only utilized for the meta theory of I/O automata. This is a remarkable advantage of the decision to use sequences with *lifted* elements.

In [30] further meta-theoretic proofs in Isabelle are described, like non-interference and compositionality. The latter states the following

$$\textbf{thms}\quad \frac{\begin{array}{c} \textit{is-trans-of}(A_1) \wedge \textit{is-trans-of}(A_2) \wedge \textit{is-trans-of}(B_1) \wedge \textit{is-trans-of}(B_2) \\ \textit{is-sig-of}(A_1) \wedge \textit{is-sig-of}(A_2) \wedge \textit{is-sig-of}(B_1) \wedge \textit{is-sig-of}(B_2) \\ \textit{compatible}\ A_1\ B_1 \wedge \textit{compatible}\ A_2\ B_2 \\ A_1 \preceq_S A_2 \\ B_1 \preceq_S B_2 \end{array}}{(A_1 \parallel B_1) \preceq_S (A_2 \parallel B_2)}$$

and required almost 1000 proof steps making heavy use of coinductive proof principles.

5 A Generic Temporal Logic

In this section we embed a generic temporal logic over finite and infinite sequences of states (called TL) into Isabelle. We use a shallow embedding, which means that we do not explicitly distinguish between syntax and semantics of temporal formulas. Instead, formulas are directly regarded as predicates and temporal operators as predicate transformers. However, Isabelle's syntax facilities permit these transformers to be denoted by the usual syntax.

We introduce a type for predicates and a corresponding notion of evaluation, which simply means function application.

$$\begin{array}{llcl} \textbf{types} & (\alpha)\textit{pred} & = & \alpha \to \textit{bool} \\ \textbf{consts} & \_\_ \models \_\_ & :: & \alpha \to (\alpha)\textit{pred} \to \textit{bool} \\ \textbf{defs} & x \models P & \equiv & P(x) \end{array}$$

The boolean connectives $\wedge$, $\vee$, $\neg$, $\Rightarrow$, and $=$ are lifted to predicates in a pointwise way[2]. As (α)*pred* is polymorphic, it is used to describe both state predicates and sequence predicates. The latter represent the temporal formulas of TL.

$$\begin{array}{llcl} \textbf{types} & (\alpha)\,\textit{temporal} & = & ((\alpha)\,\textit{sequence})\,\textit{pred} \end{array}$$

As lifted boolean connectives already exist for this type, it suffices to define $\Box$, $\bigcirc$, and $\langle - \rangle$, where the latter means lifting a state predicate to a temporal formula.

$$\begin{array}{llcl} \textbf{consts} & \langle - \rangle & :: & (\alpha)\,\textit{pred} \to (\alpha)\,\textit{temporal} \\ \textbf{defs} & \langle P \rangle & \equiv & \lambda s.\, P\ (\textit{the}\ (\textit{HD}\ \text{`}s)) \end{array}$$

$$\begin{array}{llcl} \textbf{consts} & \Box, \bigcirc & :: & (\alpha)\,\textit{temporal} \to (\alpha)\,\textit{temporal} \\ \textbf{defs} & \Box\, P & \equiv & \lambda s.\, \forall s_2.\, s_2 \leq^{+}_{\textit{suf}} s \Rightarrow P(s_2) \\ & \bigcirc\, P & \equiv & \lambda s.\, \textbf{if}\ \textit{TL}\ \text{`}s = \bot\ \textbf{then}\ P(s)\ \textbf{else}\ P\ (\textit{TL}\ \text{`}s) \end{array}$$

Here suffixes and non-empty suffixes are defined as follows:

$$\begin{array}{llcl} \textbf{consts} & \leq_{\textit{suf}},\ \leq^{+}_{\textit{suf}} & :: & (\alpha)\,\textit{sequence} \to (\alpha)\,\textit{sequence} \to \textit{bool} \\ \textbf{defs} & s_2 \leq_{\textit{suf}} s & \equiv & \exists s_1.\, \textit{Finite}(s_1) \wedge s = s_1 \oplus s_2 \\ & s_2 \leq^{+}_{\textit{suf}} s & \equiv & s_2 \neq \textit{nil} \wedge s_2 \neq \bot \wedge (s_2 \leq_{\textit{suf}} s) \end{array}$$

[2]Because it is always clear from the context whether a connective is lifted or not, we use the same symbols, although in Isabelle the syntax differs slightly.

Further temporal operators are defined as usual:

$$\begin{array}{lrcl} \textbf{defs} & \Diamond P & \equiv & \neg\Box\neg P \\ & P \leadsto Q & \equiv & \Box(P \Rightarrow \Diamond Q) \end{array}$$

Validity of P means that it holds for all non-empty sequences[3]:

$$\textbf{defs} \quad \models P \quad \equiv \quad \forall s.\, s \neq nil \wedge s \neq \bot \Rightarrow s \models P$$

Treatment of Finite Sequences. Obviously, the empty sequence represents a pathological case in every temporal logic involving finite computations. In the definitions above this is reflected by the fact that $nil \models \langle P \rangle$ equals $P(arbitrary)$, where *arbitrary* is a fixed, but unknown value. Thus nothing reasonable can be concluded for this case. We solve the problem by circumventing the cases $s = \bot$ and $s = nil$ completely. They are excluded in the validity definition and the temporal operators are defined in such a way, that they do not introduce new statements of the form $nil \models P$ or $\bot \models P$. This is the reason why we define $\Box$ using only non-empty suffixes and use the *TL* operator for $\bigcirc$ only if the sequence consists of at least two elements.

Domain-Theoretic Sequence Model and Temporal Logics. The domain-theoretic sequence model in HOLCF turned out to be surprisingly adequate for defining a temporal logic. As the definition of the temporal operators shows, every theorem about TL boils down to sequence lemmas about *HD*, *TL*, and $\oplus$. Furthermore, admissibility obligations mostly cease to apply, as $\leq^{+}_{suf}$ needs $\oplus$ with a finite first argument only, so that finite structural induction can be applied. If admissibility obligations appear nevertheless, they can usually be discharged automatically, as *HD*, *TL*, and $\oplus$ are defined as continuous operators.

This simplicity is the result of a careful choice of the way the operators are formalized. In fact, a pointwise definition like in [28] would be infeasible in our sequence model (indexes are awkward in our setting, see [11, 30]), the same holds for a $\Box$ operator defined by some kind of drop operator motivated by the semantics of TLA [23]. A number of different attempts have already been made to definitionally embed temporal logics in higher-order logic (e.g. [24, 39, 8]). Up to our knowledge, only infinite sequences have been considered, which are represented by functions on natural numbers. It is not obvious how these approaches should be generalized to deal with finite sequences as well. Furthermore, operators that deal with stuttering are not considered there. Take, for example, the operator that eliminates stuttering by replacing all subsequences $s \cdots s$ by s. This would be some kind of filter operation which is easily dealt with in our setting. In a functional setting, however, we face an operator which relocates elements in an unhomogeneous way, which is extremely awkward to handle according to the results of [11]. Note, however, that adding infinite stuttering is not computable and can thus not be handled in our setting.

Comparison with standard LTL. We will now show that

$$\models_{LTL} P \qquad \text{iff} \qquad \models P$$

where $\models_{LTL}$ denotes validity in [28] or [21] and P is restricted to the operators of TL. There are two requirements to satisfy: first, the temporal operators must have the same semantics, and second, it must make no difference whether formulas are evaluated over finite and infinite sequences (for $\models$) or over infinite sequences only (for $\models_{LTL}$).

The first requirement is not completely trivial as we did not define the operators pointwise (as in [28]) in order to match our sequence model (see below), but it is easy to see. The second requirement is proved as follows. It is trivial that $\models P$ implies $\models_{LTL} P$, as $\models_{LTL}$ regards only the subset of infinite executions considered by $\models$. For the other direction define the operator ∇ that adds infinite stuttering as

$$\nabla\sigma \quad \equiv \quad \begin{cases} arbitrary & \text{if } \sigma = nil \vee \sigma = \bot \\ \sigma s_n s_n \ldots & \text{if } \sigma = s_0 s_1 \ldots s_n \text{ is finite or partial} \\ \sigma & \text{if } \sigma \text{ is infinite} \end{cases}$$

[3]In Isabelle different symbols are used for validity and evaluation to avoid ambiguities.

Now, suppose $\models_{LTL} P$, which means that P holds for all infinite sequences. We have to show that P holds for every non-empty sequence σ as well. As $\nabla\sigma \models P$ is infinite in this case, we directly get the result by the fact that $\sigma \models P$ equals $\nabla\sigma \models P$, which holds under the assumption $\sigma \notin \{nil, \perp\}$ and follows easily. This finishes the proof.

This relation can now be exploited to get a completeness result for TL simply by carrying it over from [21]. Below we prove some theorems in Isabelle, which represent a complete set of rules w.r.t. validity in TL according to [21][4].

thms

$$\frac{\models P \Rightarrow Q \qquad \models P}{\models Q}\ (mp) \qquad \frac{\models P}{\models \bigcirc P}\ (nex)$$

$$\frac{\models P \Rightarrow Q \quad \models P \Rightarrow \bigcirc P}{\models P \Rightarrow \Box Q}\ (ind)$$

$$\begin{array}{ll} \models \bigcirc\neg P = \neg\bigcirc P & (ax1) \\ \models \bigcirc(P \Rightarrow Q) \Rightarrow (\bigcirc P \Rightarrow \bigcirc Q) & (ax2) \\ \models \Box P \Rightarrow (P \wedge \bigcirc\Box P) & (ax3) \end{array}$$

6 A Temporal Logic of Steps

In this section we use the generic temporal logic over sequences of the previous section to embed TLS [31, 30], i.e. a temporal logic over executions of I/O automata. In terms of an informal sequence model, the idea is to encode executions $\alpha = s_0 a_1 s_1 \ldots$ into a sequence of triples, where every triple (s_i, a_{i+1}, s_{i+1}) represents one step $s_i \overset{a_{i+1}}{\rightarrow}_A s_{i+1}$ of the automaton A. As finite executions are asymmetric in the sense that they contain one more state than actions, a single stuttering triple $(s_n, \surd, s_n)$ is added for the final state s_n of finite executions, where $\surd$ denotes an action disjoint from all action signatures. Intuitively, this stuttering triple ensures that finite executions may possibly be continued to infinity. In fact, it has been shown in part I [31] that the evaluation of TLS formulas stays the same when extending executions by infinite stuttering.

6.1 Definitions

In Isabelle, the encoding consists of two parts. First, executions, which are represented as state/sequence pairs, are transformed into triple sequences. Note that hereby redundancy is introduced, as most of the states are represented twice. Second, the action type α is extended to $(\alpha)\,option$, which ensures that the stuttering action, represented by *None*, is not already an element of the action type α. Both transformation tasks are performed by the function *ex-to-seq* defined below.

consts $\quad ex\text{-}to\text{-}seq :: (\alpha, \sigma)\,exec \rightarrow (\sigma \times (\alpha)\,option \times \sigma)\,sequence$

defs

$$\begin{array}{lll} ex\text{-}to\text{-}seq\,(s, ex) & \equiv & ex\text{-}to\text{-}seq_c\ \text{`}(mk\text{-}total\ ex)\ s \\ ex\text{-}to\text{-}seq\,(s, \perp) & = & [(s, None, s)!] \\ ex\text{-}to\text{-}seq\,(s, nil) & = & [(s, None, s)!] \\ ex\text{-}to\text{-}seq\,(s, (a,t)\hat{\ }ex) & = & (s, Some(a), t)\hat{\ }ex\text{-}to\text{-}seq\,(t, ex) \end{array}$$

Note that *ex-to-seq* cannot be defined via a continuous auxiliary predicate $ex\text{-}to\text{-}seq_c$ immediately as done for *Forall*, as adding a further element to a partial sequence is not even monotone. Instead, we use a function *mk-total* which makes partial executions finite by substituting the final $\perp$ by *nil*, before

[4]Note that this completeness result cannot be proved within Isabelle as we use a shallow embedding.

the expected $ex\text{-}to\text{-}seq_c$ is applied.

$$
\begin{array}{lllll}
\textbf{consts} & mk\text{-}total & :: & (\alpha)\,sequence \rightarrow (\alpha)\,sequence \\
\textbf{defs} & mk\text{-}total(s) & \equiv & \textbf{if } Partial(s) \textbf{ then } \varepsilon\, t.\, Finite(t) \wedge s = t \oplus \bot \\
 & & & \qquad\qquad\qquad \textbf{else } s \\
\textbf{thms} & mk\text{-}total(\bot) & = & nil \\
 & mk\text{-}total(nil) & = & nil \\
 & mk\text{-}total(a\hat{\ }s) & = & a\hat{\ }(mk\text{-}total\ s)
\end{array}
$$

Note that *mk-total* reduces the occurring discontinuity to a generic function with the further advantage that it may be reused for other discontinuous definitions as well, e.g. for defining fair merge.

Formulas of TLS can now be defined as TL formulas, whose sequence elements are transition triples extended by an optional stuttering action *None*. Predicates over these triples are called *step predicates*.

$$
\begin{array}{llll}
\textbf{types} & (\alpha,\sigma)\,ioa\text{-}temporal & = & (\sigma \times (\alpha)\,option \times \sigma)\,temporal \\
 & (\alpha,\sigma)\,step\text{-}pred & = & (\sigma \times (\alpha)\,option \times \sigma)\,pred
\end{array}
$$

Evaluating formulas over executions boils down to evaluating formulas over sequences using *ex-to-seq*. The usual validity notions are defined accordingly[5].

$$
\begin{array}{llll}
\textbf{consts} & \_\_ \models_{ex} \_\_ & :: & (\alpha,\sigma)\,exec \rightarrow (\alpha,\sigma)\,ioa\text{-}temporal \rightarrow bool \\
 & \models_{ex} \_\_ & :: & (\alpha,\sigma)\,ioa\text{-}temporal \rightarrow bool \\
 & \_\_ \models_{A} \_\_ & :: & (\alpha,\sigma)\,ioa \rightarrow (\alpha,\sigma)\,ioa\text{-}temporal \rightarrow bool \\
\textbf{defs} & exec \models_{ex} P & \equiv & (ex\text{-}to\text{-}seq\ exec) \models P \\
 & \models_{ex} P & \equiv & \forall exec.\ exec \models_{ex} P \\
 & A \models_{A} P & \equiv & \forall exec \in execs(A).\ exec \models_{ex} P
\end{array}
$$

Note that for $\models_{ex}$ the boolean connectives have the same pointwise meaning as for $\models$, e.g. $exec \models_{ex} \neg P = exec \not\models_{ex} P$. This, however, does not hold for $\models_A$.

When talking about I/O automata it is often more convenient to use predicates on states and actions rather than step predicates, which always take the stuttering action into account. Thus, we introduce the functions ext_{s} and ext_{a} which lift state and action predicates to step predicates. Possibly occurring stuttering actions force the resulting step predicate to evaluate to *False*.

$$
\begin{array}{llll}
\textbf{consts} & ext_{\mathsf{s}} & :: & (\sigma)\,pred \rightarrow (\alpha,\sigma)\,step\text{-}pred \\
\textbf{defs} & ext_{\mathsf{s}}(P) & \equiv & \lambda(s,a,t).\ P(s)
\end{array}
$$

$$
\begin{array}{llll}
\textbf{consts} & ext_{\mathsf{a}} & :: & (\alpha)\,pred \rightarrow (\alpha,\sigma)\,step\text{-}pred \\
\textbf{defs} & ext_{\mathsf{a}}(P) & \equiv & \lambda(s,a',t).\ \ \textbf{case } a' \textbf{ of} \\
 & & & \qquad\qquad \begin{array}{lccc} & None & \Rightarrow & False \\ \mid & Some\ a & \Rightarrow & P(a) \end{array}
\end{array}
$$

We use the syntactical abbreviations $\langle P\rangle_{\mathsf{s}} = \langle ext_{\mathsf{s}}(P)\rangle$ and $\langle P\rangle_{\mathsf{a}} = \langle ext_{\mathsf{a}}(P)\rangle$.

6.2 Some Theorems

Validity in TL, i.e. on sequences, is stronger than in TLS, i.e. on executions.

$$
\textbf{thms} \qquad \models P \ \Rightarrow\ \models_{ex} P \qquad (Val\text{-}Rel)
$$

[5] Once more the symbol $\models_{ex}$ is overloaded in a way not supported by Isabelle.

The proof is simple: (*Val-Rel*) postulates that $\textit{ex-to-seq}(\textit{exec}) \models_{ex} P$ holds for every *exec*, provided that $s \models_{ex} P$ holds for every s with $s \neq \bot \wedge s \neq \textit{nil}$. This is true as *ex-to-seq* produces only non-empty sequences[6].

Thus, the theorems $(ax1) - (ax3)$ carry over from $\models$ to $\models_{ex}$. Furthermore, the same holds for the theorems (mp), (nex), and (ind).

Note that the other direction of (*Val-Rel*) is not true, as not every transition sequence is an image under *ex-to-seq*: there may be *None* elements occurring not only after the final state of non-infinite executions, or non-identical successor states. Therefore, the completeness considerations for TL do not carry over to TLS. However, the specific form of sequences generated by *ex-to-seq* can be captured by a temporal formula for each automaton step. Having derived these step formulas, it is often sufficient to use further on only rules for $\models$ instead of $\models_{ex}$. Thus, they build some kind of interface between TL and TLS.

$$\textbf{thms} \qquad \frac{\forall s\ t.\ P(s) \wedge s \xrightarrow{a}_A t \Rightarrow Q(t)}{A \models_A \Box(\langle P \rangle_{\mathsf{s}} \wedge \langle \lambda x.\ x = a \rangle_{\mathsf{a}} \Rightarrow \bigcirc \langle Q \rangle_{\mathsf{s}})}$$

The proof of this theorem shows that these formulas indeed incorporate the fact that *ex-to-seq* adds stuttering steps at the end of finite executions only and produces always identical succeeding states. Thus, the application of these formulas yields temporal formulas, which can then be treated by standard LTL reasoning.

7 Live I/O Automata

Below, the I/O automaton model of §3 is extended to general liveness. Live I/O automata are represented by a pair of a safe I/O automaton and a TLS formula.

$$\textbf{types} \quad (\alpha, \sigma)\,\textit{live-ioa} \quad = \quad (\alpha,\sigma)\,\textit{ioa} \times (\alpha,\sigma)\,\textit{ioa-temporal}$$

A TLS formula P is said to be L-valid for a live I/O automaton (A, L) if it holds for all executions of A under the further assumption L:

$$\textbf{defs} \quad (A, L) \models_L P \quad \equiv \quad A \models_A (L \Rightarrow P)$$

This reflects the intuition that liveness conditions posed on a safe I/O automaton restrict its executions [15]. Live executions, traces, and implementations generalize the corresponding safe notions in a canonical way.

$$\begin{array}{lll} \textbf{defs} \quad \textit{live-execs}\ (A, L) & \equiv & \{\textit{exec}.\ \textit{exec} \in \textit{execs}(A) \wedge \textit{exec} \models_{ex} L\} \\ \quad\quad\ \ \textit{live-traces}\ (A, L) & \equiv & \{\textit{mk-trace}\ A\ \text{`}(\textit{snd}\ \textit{ex})\ |_{ex}\ \textit{ex} \in \textit{live-execs}\ (A, L)\} \\ \quad\quad\ \ C \preceq_L A & \equiv & \textit{in}(C) = \textit{in}(A) \wedge \textit{out}(C) = \textit{out}(A) \wedge \\ & & \textit{live-traces}(C) \subseteq \textit{live-traces}(A) \end{array}$$

Important cases of liveness are given by weak and strong fairness of an automaton A w.r.t. a set of actions *acts*, which are defined by the formulas *WF* and *SF*.

$$\begin{array}{llll} \textbf{consts} & \textit{Enabled} & :: & (\alpha,\sigma)\,\textit{ioa} \rightarrow (\alpha)\,\textit{set} \rightarrow \sigma \rightarrow \textit{bool} \\ & \textit{WF}, \textit{SF} & :: & (\alpha,\sigma)\,\textit{ioa} \rightarrow (\alpha)\,\textit{set} \rightarrow (\alpha,\sigma)\,\textit{ioa-temporal} \\ \textbf{defs} & \textit{Enabled}\ A\ \textit{acts}\ s & \equiv & \exists a \in \textit{acts}.\ \exists t.\ s \xrightarrow{a}_A t \\ & \textit{WF}\ A\ \textit{acts} & \equiv & \Diamond\Box\langle \textit{Enabled}\ A\ \textit{acts} \rangle_{\mathsf{s}} \Rightarrow \Box\Diamond\langle \lambda a.\ a \in \textit{acts} \rangle_{\mathsf{a}} \\ & \textit{SF}\ A\ \textit{acts} & \equiv & \Box\Diamond\langle \textit{Enabled}\ A\ \textit{acts} \rangle_{\mathsf{s}} \Rightarrow \Box\Diamond\langle \lambda a.\ a \in \textit{acts} \rangle_{\mathsf{a}} \end{array}$$

[6]Note that the cases $s = \bot$ and $s = \textit{nil}$ are the pathological cases of TL. As they are excluded by *ex-to-seq*, the stuttering action $\surd$ in TLS can, in a more abstract view, also be regarded as a remedy to rectify the insufficiency of general temporal logics involving non-infinite computations.

Note the particular meaning of *WF* and *SF* for finite executions. The formulas $\Box\Diamond P$ and $\Diamond\Box P$ express the same for finite executions, namely that P holds for the final stuttering step $(s_n, None, s_n)$. Thus, $\Box\Diamond\langle\lambda a.\ a \in acts\rangle_{\mathsf{a}}$ is false, as the lifting function ext_{a} maps the stuttering action *None* to *False*. Therefore, *WF* and *SF* express that the last state s_n is not enabled for any action in *acts*, which corresponds to the usual fairness definition using fairness sets [26].

Refinement mappings which transfer liveness from every execution of C to a corresponding one of A are called *live refinement mappings*

$$
\begin{array}{ll}
\textbf{consts} & \textit{is-live-refmap} :: (\sigma_1 \to \sigma_2) \to (\alpha, \sigma_1)\,\textit{live-ioa} \to (\alpha, \sigma_2)\,\textit{live-ioa} \to \textit{bool} \\
\textbf{defs} & \textit{is-live-refmap}\ f\ (C, L)\ (A, M) \equiv \\
 & \quad \textit{is-ref-map}\ f\ C\ A \wedge \\
 & \quad \forall \textit{exec} \in \textit{exec}(C).\ \textit{exec} \models_{ex} L \ \Rightarrow\ (\textit{cor}^{ref}\ A\ f\ \textit{exec}) \models_{ex} M
\end{array}
$$

where the corresponding execution $(cor^{ref}\ A\ f\ exec)$ is given by an infinite concatenation of possible moves that correspond to the single steps of *exec*.

$$
\begin{array}{llcl}
\textbf{consts} & \textit{cor}^{ref} :: (\alpha, \sigma_2)\,\textit{ioa} \to (\sigma_1 \to \sigma_2) \to (\alpha, \sigma_1)\,\textit{exec} \to (\alpha, \sigma_2)\,\textit{exec} & & \\
\textbf{defs} & \textit{cor}^{ref}\ A\ f\ (s, \bot) & = & (f\ s, \bot) \\
 & \textit{cor}^{ref}\ A\ f\ (s, \textit{nil}) & = & (f\ s, \textit{nil}) \\
 & \textit{cor}^{ref}\ A\ f\ (s, (a, t)\hat{\ }\textit{ex}_1) & = & (f\ s, (\varepsilon\ \textit{ex}_2.\ \textit{is-move}\ A\ \textit{ex}_2\ (f\ s, a, f\ t)) \\
 & & & \qquad \oplus\ \textit{snd}\ (\textit{cor}^{ref}\ A\ f\ (t, \textit{ex}_1))
\end{array}
$$

Live refinement mappings are correct, i.e. they induce live implementation.

$$
\textbf{thms} \quad \frac{\textit{is-live-refmap}\ f\ (C, L)\ (A, M) \quad \textit{in}(C) = \textit{in}(A) \wedge \textit{out}(C) = \textit{out}(A)}{(C, L) \preceq_L (A, M)}
$$

Similarly, but a bit more tricky, is the definition of *live forward simulations* and their soundness proof. See [30].

This theorem gives rise to the following proof method for showing that a live I/O automaton (C, L) implements another live I/O automaton (A, M) using temporal reasoning.

- First, show that f is a refinement mapping from C to A, i.e. prove the safety part.
- Then assume an execution $exec \in execs(C)$ with $exec \models_{ex} L$
- and prove $(cor^{ref}\ A\ f\ exec) \models_{ex} M$.

In this proof method, L is evaluated over executions of C, whereas M is evaluated over executions of A. Thus, it is not sufficient to merely apply temporal tautologies for the liveness proof. Rather there have to be means to switch between properties over an execution *exec* of C and those of the corresponding execution $exec' := (cor^{ref}\ A\ f\ exec)$ of A. For refinement proofs restricted to fairness, such a switch is needed only once at the beginning of the proof: assume $exec' \models_{ex} WF\ A\ acts$ to be false. This implies that $exec' \models_{ex} \Box\Diamond\langle Enabled\ A\ acts\rangle_{\mathsf{s}}$ and $exec' \models_{ex} \neg\Box\Diamond\langle\lambda a.\ a \in acts\rangle_{\mathsf{a}}$ hold. Now, both properties about $exec'$ can be reduced to properties about *exec* using the following theorems:

$$
\textbf{thms} \quad \frac{\textit{exec} \in \textit{execs}(C) \quad \textit{ext}(C) = \textit{ext}(A) \quad \Lambda \subseteq \textit{ext}(A)}{\textit{exec} \models_{ex} \Box\Diamond\langle a \in \Lambda\rangle_{\mathsf{a}} \quad \Rightarrow \quad (\textit{cor}^{ref}\ A\ f\ \textit{exec}) \models_{ex} \Box\Diamond\langle a \in \Lambda\rangle_{\mathsf{a}}}
$$

$$
\frac{\textit{exec} \in \textit{execs}(C) \quad \forall s\ t.\ \textit{reachable}\ C\ s \wedge \textit{reachable}\ A\ t \wedge t \models Q \Rightarrow s \models P}{(\textit{cor}^{ref}\ A\ f\ \textit{exec}) \models_{ex} \Diamond\Box\langle Q\rangle_{\mathsf{s}} \quad = \quad \textit{exec} \models_{ex} \Diamond\Box\langle P\rangle_{\mathsf{s}}}
$$

For the remainder of the proof only temporal tautologies are needed.

This approach has significant advantages for our Isabelle environment. For fairness the two theorems above permit hiding HOLCF from the user, who operates only within the simpler HOL or uses standard rules of temporal logic.

8 Abstraction Rules

In this section we derive abstraction rules, which permit reducing proof obligations about large or even infinite I/O automata to corresponding proof obligations about finite and significantly smaller automata. Rules are provided for properties expressed both as temporal formula (proof obligation $(C, L) \models_{ex} P$) and as I/O automata (proof obligation $(C, L_C) \preceq_L (P, L_P)$). In §9 we will show by means of an example how these rules can be used to combine Isabelle with model checking.

Central are *abstraction functions* which represent specific refinement mappings, namely homomorphisms w.r.t. the transition relation.

$$
\begin{array}{llll}
\textbf{consts} & \textit{is-abs} & :: & (\sigma_1 \to \sigma_2) \to (\alpha, \sigma_1)\,\textit{ioa} \to (\alpha, \sigma_2)\,\textit{ioa} \to \textit{bool} \\
\textbf{defs} & \textit{is-abs}\ h\ C\ A & \equiv & (\forall s_0 \in \textit{starts-of}\ C.\,(h\ s_0) \in \textit{starts-of}\ A) \wedge \\
& & & (\forall s\ \ a.\ \textit{reachable}\ C\ s \wedge s \xrightarrow{a}_C t \Rightarrow (h\ s) \xrightarrow{a}_A (h\ t))
\end{array}
$$

The key idea of abstraction functions is that they induce a neater correspondence between executions than refinement mappings do: the corresponding execution is given as a pointwise mapping, which means that both executions always proceed within the same time raster.

$$
\begin{array}{ll}
\textbf{consts} & \textit{cor}^{abs} :: (\sigma_1 \to \sigma_2) \to (\alpha, \sigma_1)\,\textit{exec} \to (\alpha, \sigma_2)\,\textit{exec} \\
\textbf{defs} & \textit{cor}^{abs}\ h\ (s, ex) \equiv (h\ s, \textit{Map}\ (\lambda(a,t).\,(a, h\ t))\,\text{`}ex)
\end{array}
$$

Thus, we get as a special case of the soundness of refinement mappings that abstraction functions induce safe implementation. This result is based on the fact, that $(\textit{cor}^{abs}\ h\ \textit{exec})$ is an execution of A provided that $\textit{exec}$ is an execution of C. We say that A is an *automaton weakening* of C.

$$
\begin{array}{ll}
\textbf{consts} & \textit{aut-weak} :: (\alpha, \sigma_2)\,\textit{ioa} \to (\alpha, \sigma_1)\,\textit{ioa} \to (\sigma_1 \to \sigma_2) \to \textit{bool} \\
\textbf{defs} & \textit{aut-weak}\ A\ C\ h \equiv \forall \textit{exec} \in \textit{execs}(C).\ \textit{cor}^{abs}\ h\ \textit{exec} \in \textit{execs}(A) \\
\textbf{thms} & \dfrac{\textit{is-abs}\ h\ C\ A}{\textit{aut-weak}\ A\ C\ h} \\
& \dfrac{\textit{is-abs}\ h\ C\ A \qquad \textit{in}(C) = \textit{in}(A) \wedge \textit{out}(C) = \textit{out}(A)}{C \preceq_S A}
\end{array}
$$

For the safety part, we considered until now, everything has been analogous to refinement mappings. For the liveness part, however, we get a significant improvement: whereas $\textit{cor}^{ref}$ permitted to transfer only specific patterns of fairness formulas from the corresponding execution $(\textit{cor}^{ref}\ A\ f\ \textit{exec})$ to $\textit{exec}$, the stronger $\textit{cor}^{abs}$ permits the analogous transfer for *any* kind of temporal formula, even in both directions. Such transfers are called *temporal weakenings* and *temporal strengthenings*, respectively.

$$
\begin{array}{ll}
\textbf{consts} & \textit{temp-strength}, \textit{temp-weak} :: \\
& (\alpha, \sigma_2)\,\textit{ioa-temporal} \to (\alpha, \sigma_1)\,\textit{ioa-temporal} \to (\sigma_1 \to \sigma_2) \to \textit{bool} \\
\textbf{defs} & \textit{temp-strength}\ Q\ P\ h \equiv \forall \textit{exec}.\,(\textit{cor}^{abs}\ h\ \textit{exec}) \models_{ex} Q) \Rightarrow \textit{exec} \models_{ex} P \\
& \textit{temp-weak}\ Q\ P\ h \equiv \textit{temp-strength}\ (\neg Q)\ (\neg P)\ h
\end{array}
$$

Our goal is to reduce them to associated *step weakenings/strengthenings*.

$$
\begin{array}{ll}
\textbf{consts} & \textit{step-strength}, \textit{step-weak} :: \\
& (\alpha, \sigma_2)\,\textit{step-pred} \to (\alpha, \sigma_1)\,\textit{step-pred} \to (\sigma_1 \to \sigma_2) \to \textit{bool} \\
\textbf{defs} & \textit{step-strength}\ Q\ P\ h \equiv \forall s\ a\ t.\ Q\ (h\ s, a, h\ t) \Rightarrow P\ (s, a, t) \\
& \textit{step-weak}\ Q\ P\ h \equiv \textit{step-strength}\ (\neg Q)\ (\neg P)\ h
\end{array}
$$

This is accomplished by the following set of theorems, which hold as well when interchanging *temp-strength* with *temp-weak*.

thms

$$\frac{\textit{temp-strength}\ P_1\ Q_1\ h \quad \textit{temp-strength}\ P_2\ Q_2\ h}{\textit{temp-strength}\ (P_1 \star P_2)\ (Q_1 \star Q_2)\ h}\ \star \in \{\wedge, \vee\}$$

$$\frac{\textit{temp-weak}\ P_1\ Q_1\ h \quad \textit{temp-strength}\ P_2\ Q_2\ h}{\textit{temp-strength}\ (P_1 \star P_2)\ (Q_1 \star Q_2)\ h}\ \star \in \{\Rightarrow, \leadsto\}$$

$$\frac{\textit{temp-strength}\ P\ Q\ h}{\textit{temp-strength}\ (\star P)\ (\star Q)\ h}\ \star \in \{\Box, \Diamond, \bigcirc\}$$

$$\frac{\textit{temp-weak}\ P\ Q\ h}{\textit{temp-strength}\ (\neg P)\ (\neg Q)\ h}$$

$$\frac{\textit{step-strength}\ P\ Q\ h}{\textit{temp-strength}\ (\langle P\rangle)\ (\langle Q\rangle)\ h}$$

These theorems form the basis for a tactic, called `abs_tac`, which automatically reduces temporal strengthenings/weakenings to step strengthenings/weakenings. This reduction represents a major advantage of our abstraction theory: the interactive theorem prover, verifying the abstraction's correctness, has to reason about steps only (which can be done in the simpler HOL), whereas reasoning about entire system runs may be left to the model checker.

Now we can define live abstractions and prove their correctness.

consts $\textit{is-live-abs} :: (\sigma_1 \to \sigma_2) \to (\alpha, \sigma_1)\,\textit{live-ioa} \to (\alpha, \sigma_2)\,\textit{live-ioa} \to \textit{bool}$

defs $\textit{is-live-abs}\ h\ (C, L)\ (A, M) \equiv \textit{is-abs}\ h\ C\ A \wedge \textit{temp-weak}\ M\ L\ h$

thms

$$\frac{\textit{is-live-abs}\ h\ (C, L)\ (A, M) \qquad \textit{in}(C) = \textit{in}(A) \wedge \textit{out}(C) = \textit{out}(A)}{(C, L) \preceq_L (A, M)}$$

Finally, we can present the abstraction rules. Rules on the *lhs* treat safe I/O automata, the *rhs* considers live I/O automata. Note that after applying these rules the tactic `abs_tac` has still to be invoked.

thms

$$\frac{\begin{array}{c}\textit{is-abs}\ h\ C\ A \\ \textit{temp-strength}\ Q\ P\ h \\ A \models_A Q\end{array}}{C \models_A P} \qquad \frac{\begin{array}{c}\textit{is-live-abs}\ h\ (C, L)\ (A, M) \\ \textit{temp-strength}\ Q\ P\ h \\ (A, M) \models_L Q\end{array}}{(C, L) \models_L P}$$

$$\frac{\begin{array}{c}\textit{in}(C) = \textit{in}(A) \wedge \textit{out}(C) = \textit{out}(A) \\ \textit{in}(Q) = \textit{in}(P) \wedge \textit{out}(Q) = \textit{out}(P) \\ \textit{is-abs}\ h_1\ C\ A \\ \textit{is-abs}\ h_2\ Q\ P \\ A \preceq_S Q\end{array}}{C \preceq_S P} \qquad \frac{\begin{array}{c}\textit{in}(C) = \textit{in}(A) \wedge \textit{out}(C) = \textit{out}(A) \\ \textit{in}(Q) = \textit{in}(P) \wedge \textit{out}(Q) = \textit{out}(P) \\ \textit{is-live-abs}\ h_1\ (C, L_C)\ (A, L_A) \\ \textit{is-live-abs}\ h_2\ (Q, L_Q)\ (P, L_P) \\ (A, L_A) \preceq_L (Q, L_Q)\end{array}}{(C, L_C) \preceq_L (P, L_P)}$$

There are two further rules, which allow to strengthen the abstract model if that appears to be too weak to prove the desired property, or to weaken the concrete model, if that contains unnecessary elements which hinder the intended abstraction. See [31, 30] for the underlying theory and the Isabelle distribution for their realization.

9 Combining Isabelle and Model Checking

The abstraction rules of the previous section generate proof obligations which should be discharged by a model checker. These obligations have either the form $(A, F_A) \models P$ (for proving temporal properties) or $(A, F_A) \leq_L (P, F_P)$ (for proving implementation relations). In the following we present translations to appropiate model checkers for each of them: the former is translated to STeP [5], the latter to μcke [4].

9.1 Temporal Properties: Translation to STeP

The tranlation to STeP is straight forward. I/O automata are transformed to transition systems [28], which represent the computational model of the STeP model checker [5]. TLS formulas are translated into Manna/Pnueli's LTL [28].

The idea of the translation is to encode the explicit actions of I/O automata into the state space and to add a further idling action. Liveness conditions are not coded into fairness sets, but are considered as a further assumption of the temporal property to be proved.

In the following we demonstarte the translation by the simple example of part I. This example represents a rough simplification of an industrial helicopter control system, which has also been verified using our toolbox [30].

Example 9.1
The alarm management of a cockpit control system can be described in a very abstract way by a stack. Alarms, which are initiated by the physical environment, are stored, then handled by the pilot and finally acknowledged, which means that the respective alarm is removed from the stack. When adding a new alarm to the stack, any older occurrences of this particular alarm are removed, such that only the most urgent instance of an alarm has to be treated by the pilot. There are 16 alarms, $\mathit{Alarms} = \{\mathit{PonR}, \mathit{Fuel}, \mathit{Eng}, \ldots\}$, which in the original specification made it impossible to verify the system via model checking. Actually, the original system could only be model checked for not more than four alarms. Note that the order of alarms in the stack has to be respected. However, for proving properties which concern merely a single alarm, for example the important alarm *PonR*, (Point of no Return), abstraction may be applied.

The I/O automaton $\mathit{Cockpit}_C$ is modeled by a list, called *stack*, which is initially empty. We present the transitions of $\mathit{Cockpit}_C$ in the usual informal format, which can easily be translated to our Isabelle setting.

input $\mathit{Alarm}(a), a \in \mathit{Alarms}$	**output** $\mathit{Ack}(a), a \in \mathit{Alarms}$
post: $\mathit{stack} := a : \mathit{filter}\ (\lambda x.\ x \neq a)\ \mathit{stack}$	**pre**: $\mathit{hd}(\mathit{stack}) = a \wedge \mathit{stack} \neq [\,]$
	post: $\mathit{stack} := \mathit{tl}(\mathit{stack})$

The properties we want to prove about $\mathit{Cockpit}_C$ are the following:

defs $P_1 \equiv \Box(\langle \lambda a.\ a = \mathit{Alarm}(\mathit{PonR})\rangle_{\mathsf{a}} \Rightarrow \bigcirc\langle \lambda \mathit{stack}.\ \mathit{PonR} \in \mathit{stack}\rangle_{\mathsf{s}})$
"Whenever PonR arrives, it is immediately stored in the stack"

$P_2 \equiv \Box\langle \lambda a.\ a \neq \mathit{Alarm}(\mathit{PonR})\rangle_{\mathsf{a}} \Rightarrow \Box\neg\langle \lambda \mathit{stack}.\ \mathit{PonR} \in \mathit{stack}\rangle_{\mathsf{s}}$
"If PonR never arrives, the system will never pretend this"

For both properties it is merely relevant whether *PonR* is in the stack or not. Thus, we construct the abstract I/O automaton $\mathit{Cockpit}_A$ which replaces the alarm stack by the boolean variable *PonR-in*, initially *false*, which indicates if $PonR$ is stored.

input $Alarm(a), a \in Alarms$
  post: **if** $a = PonR$ **then** $PonR\text{-}in := true$

---

output $Ack(a), a \in Alarms$
  pre: **if** $a = PonR$ **then** $PonR\text{-}in$
  post: **if** $a = PonR$ **then** $PonR\text{-}in := false$

The abstraction function h is obviously defined as follows:

$$
\begin{array}{llll}
\textbf{consts} & h & :: & (\alpha)list \to bool \\
\textbf{defs} & h(stack) & \equiv & PonR \in stack
\end{array}
$$

As $Cockpit_A$ has been designed already with h in mind, the property $is\text{-}abs\ h\ \ Cockpit_C\ \ Cockpit_A$ is easily established within Isabelle/HOL[7]. Thus, according to the first abstraction rule of the previous section, it remains to show that the corresponding abstract properties Q_i for $Cockpit_A$, defined as

$$
\begin{array}{llll}
\textbf{defs} & Q_1 & \equiv & \Box(\langle \lambda a.\ a = Alarm(PonR)\rangle_{\mathsf{a}} \Rightarrow \bigcirc \langle \lambda PonR\text{-}in.\ PonR\text{-}in\rangle_{\mathsf{s}}) \\
 & Q_2 & \equiv & \Box\langle \lambda a.\ a \neq Alarm(PonR)\rangle_{\mathsf{a}} \Rightarrow \Box\neg\langle \lambda PonR\text{-}in.\ PonR\text{-}in\rangle_{\mathsf{s}}
\end{array}
$$

are temporal strengthenings of the concrete P_i. These goals are reduced by `abs_tac` to the obligations $h(stack) \Rightarrow (PonR \in stack)$ and $(PonR \in stack) \Rightarrow h(stack)$, respectively, which both are trivial by definition. Therefore, the initial goals $Cockpit_C \models_A P_i$ have been reduced to the simpler goals $Cockpit_A \models_A Q_i$, which can now be verified by the STeP model checker. For this aim the I/O automaton $Cockpit_A$ is encoded into a STeP transition system. This is done by encoding explicit actions into the state space using a variable `Act`. Note that thereby the occurrence of an action can be observed only at the next state.

```
Transition System
   type actions = {AlarmP, AlarmNP, AckP, AckNP}
   local PonRin: bool
   local Act: actions
   Initially PonRin = false
   Transition AlarmP NoFairness:
     enable true
     assign PonRin := true, Act := AlarmP
   Transition AlarmNP NoFairness:
     enable true
     assign PonRin := PonRin, Act := AlarmNP
   Transition AckP NoFairness:
     enable PonRin
     assign PonRin := false, Act := AckP
   Transition AckNP NoFairness:
     enable true
     assign PonRin := PonRin, Act := AckNP
```

The following properties represent the Q_i which are easily verified by STeP.

```
SPEC
   PROPERTY P1:  [](()Act=AlarmP --> ()PonRin)
   PROPERTY P2:  []()Act!=AlarmP --> []!PonRin
```

---

[7]Note that the invariant that there are no duplicates in the stack is needed to show that $Ack(PonR)$ causes the transition from $PonR\text{-}in$ to $\neg PonR\text{-}in$.

9.2 Implementation Relations: Translation to the μ-Calculus

For the restricted case of fair trace inclusion [22] suggests already a translation of fair I/O automata into ω-automata, which can be tested w.r.t. language containment using the COSPAN model checker.

We, however, give a translation of forward simulations directly into the μ-calculus. Thus, a μ-calculus based model checker like μcke [4] can check forward simulations between I/O automata without the overhead of changing to another semantic model. A further, practical advantage of this approach is the fact that there is already a tactic in Isabelle which invokes the μ-calculus model checker μcke [4] as an external oracle. However, this approach only yields a complete decision procedure w.r.t. trace inclusion of two I/O automata A and P if P is deterministic and both A and P are safe.

The following translation of forward simulations into the μ-calculus has been completely automated [17]: there is a tactic in Isabelle, called *is_simulation* C A, which delegates the existence proof of a forward simulation to μcke.

Assume that a set of states $\mathcal{S}_A$ for each I/O automaton A and a global set of actions $\mathcal{A}$ is given. Furthermore assume that every basic I/O automaton is characterized by the following boolean predicates.

$$\begin{array}{l}
\mathit{In}_A : \mathcal{A} \to \mathit{bool} \\
\mathit{Out}_A : \mathcal{A} \to \mathit{bool} \\
\mathit{Int}_A : \mathcal{A} \to \mathit{bool} \\
\mathit{Start}_A : \mathcal{S}_A \to \mathit{bool} \\
\mathit{Trans}_A : \mathcal{S}_A \times \mathcal{A} \times \mathcal{S}_A \to \mathit{bool}
\end{array}$$

We skip the encoding of composition operators, they are straight forward. Internal steps, a finite sequence of internal steps, and a move are described by the following predicates:

$$\begin{array}{lcl}
\mathit{IntStep}_A(s,t) & \equiv & \exists a.\ \mathit{Int}_A(a) \wedge \mathit{Trans}_A(s,a,t) \\
\mathit{IntStep}_A^* & \equiv & \mu P.\ \lambda s,t.\ (s=t) \vee \exists u.\ \mathit{IntStep}_A(s,u) \wedge P(u,t) \\
\mathit{Move}_A(s,a,t) & \equiv & (\mathit{Int}_C(a) \wedge \mathit{IntStep}_A^*(s,t)) \vee \\
 & & \exists u_1,u_2.\ \mathit{IntStep}_A^*(s,u_1) \wedge \mathit{Trans}_A(u_1,a,u_2) \wedge \mathit{IntStep}_A^*(u_2,t)
\end{array}$$

Then, the existence of a forward simulation can be expressed by

$$\begin{array}{lcl}
\mathit{isSim}_{CA} & \equiv & \nu P.\ \lambda s_1,t_1.\ \forall a,s_2.\ \mathit{Trans}_C(s_1,a,s_2) \Rightarrow \exists t_2.\ \mathit{Move}_A(t_1,a,t_2) \wedge P(s_2,t_2) \\
\mathit{SimExists}_{CA} & \equiv & \forall a.\ \mathit{Inp}_C(a) \leftrightarrow \mathit{Inp}_A(a) \wedge \mathit{Out}_C(a) \leftrightarrow \mathit{Out}_A(a) \wedge \\
 & & \forall s,t.\ \mathit{Start}_C(s) \wedge \mathit{Start}_A(t) \Rightarrow \mathit{isSim}_{CA}(s,t)
\end{array}$$

Example 9.2
Consider once more the alarm system $\mathit{Cockpit}_C$ and its abstract counterpart $\mathit{Cockpit}_A$ under the abstraction h. We enhance the two automata by an output action *Info* which signals the presence of newly arrived alarms to the pilot.

Extension to $\mathit{Cockpit}_C$	Extension to $\mathit{Cockpit}_A$
output $\mathit{Info}(a), a \in \mathit{Alarms}$	**output** $\mathit{Info}(a), a \in \mathit{Alarms}$
pre: $\mathit{hd}(\mathit{stack}) = a$	**pre**: $a = \mathit{PonR} \Rightarrow \mathit{PonR\text{-}in}$

It is easily shown that *is_abs* h $\mathit{Cockpit}_C$ $\mathit{Cockpit}_A$ still holds with the additional *Info* action.

This abstraction can be used to prove property P_4, expressed by the following I/O automaton: the state is the same as for $\mathit{Cockpit}_A$, i.e. a boolean variable *PonR-in*, which is initially false. The actions of P_4 are:

input $\mathit{Alarm}(a), a \in \mathit{Alarms}$	**output** $\mathit{Info}(a), a \in \mathit{Alarms}$
post: **if** $a = \mathit{PonR}$ **then** $\mathit{PonR\text{-}in} := \mathit{true}$	**pre**: $a = \mathit{PonR} \Rightarrow \mathit{PonR\text{-}in}$

The I/O automaton P_4 expresses that every *PonR* alarm is not signaled to the pilot before it has arrived, i.e. *Alam* and *Info* actions appear always in the desired order. Using the abstraction rule for implementaion relations in the simpler fashion with $P = P^+$ we may conclude the desired refinement $Cockpit_C \preceq_S P_4$ from the simpler $Cockpit_A \preceq_S P_4$. As P_4 is a deterministic automaton, we may use the suggested translation to μcke to discharge this proof obligation. Actions and the automaton P_4 are encoded in μcke as follows.

```
enum Alarms {PonR, Fuel, Eng};
enum Actions {Alarm, Ack, Info};
class Action {Actions act; Alarms arg;};
class StateP4 {bool PonRin;};

bool IntP4(Action a) a.act = Ack;
bool StartP4(StateP4 s) s.PonRin = 0;
bool TransP4(StateP4 s, Action a, StateP4 t)
  (case
    a.act = Alarm :
         (if(a.arg = PonR)
            (t.PonRin = 1)
         else
            t.PonRin = s.PonRin);
    a.act = Ack : 0;
    a.act = Info :
         (a.arg = PonR -> s.PonRin = 1) &
         t.PonRin = s.PonRin;
   esac);
```

Encoding $Cockpit_A$ in an analogous way with state type `StateCo`, start condition `StartCo`, and transition relation `TransCo` we can encode the desired trace inclusion as follows.

```
bool IntStepP4(StateP4 s, StateP4 t)
  exists Action a. IntP4(a) & TransP4(s,a,t);

mu bool IntStepStarP4(StateP4 s, StateP4 t)
  s = t |  (exists StateP4 u.
              IntStepP4(s, u) & IntStepStarP4(u, t));

bool MoveP4(Action a, StateP4 x, StateP4 y)
  IntP4(a) & IntStepStarP4(x, y) |
  (exists StateP4 u1. IntStepStarP4(x, u1) &
    (exists StateP4 u2. TransP4(u1, a, u2) & IntStepStarP4(u2, y)));

nu bool isSim(StateCo s1, StateP4 t1)
 forall Action a, StateCo s2.
   TransCo(s1, a, s2) ->
     (exists StateP4 t2. MoveP4(a, t1, t2) & isSim(s2, t2));

bool SimExists
   forall StateCo s, StateP4 t. StartCo(s) & StartP4(t) -> isSim(s,t);
```

10 Related Work

There are several other attempts to support I/O automata verification by tools.

The MIT distributed systems group, which originally developed I/O automata, has done substantial efforts in verifying simulations between I/O automata using the Larch Prover (LP) [38]. A number of case studies have been performed, involving timing based systems as well (e.g. [25, 37]). Current work [14] aims at a formal language for I/O automata which allows to develop tools like static type checkers, simulators and code generators. The distinguishing feature to our work is is the fact that LP is a theorem prover for first-order logic. This means that the abstract notions of I/O automata incorporating composition operators and behaviours cannot be expressed within the logic. Instead, results on paper are used to extract a set of proof obligations from the formal description of the system. In particular, reasoning about meta-theory is impossible.

Archer and Heitmeyer [1, 2] verified several benchmark problems modeled as Lynch/-Vaandrager timed automata [27] in PVS [33]. Their goal is to build a customized prover on top of PVS, which is designed to process proof steps that resemble in style and size the typical steps in hand proofs. This is accomplished by tailored proof strategies, which resemble pretty much our specialized Isabelle tactics. However, their framework is restricted to invariant proofs, simulations are not taken into account until now. Furthermore, they do not consider meta-theory, although the logic of PVS would be powerful enough.

Further case studies have been performed with Coq [12] in the area of communication protocols [19, 6]. Again, they rely much more on unformalized meta-theory than we do.

Inspired by our work, Griffioen and Devillers [16] formalized the meta-theory of I/O automata in PVS [33] and proved the correctness of safe refinement mappings, but not of forward simulations. However, defining and reasoning about infinite concatenation turned out to be very awkward in their functional sequence model [10] (cf. the discussion in [11]). Thus, they left out some crucial theorems.

All approaches above consider neither temporal logics nor any issues of abstraction. Especially they do not cover a connection to model checking tools.

Concerning temporal logics and abstraction, the only existing works merely deal with models different from I/O automata. Formal embeddings of temporal logic [24, 39, 8], however, are always based on functional sequence models in contrast to our algebraic sequence model. This results in disadvantages discussed in §11. An abstraction framework for TLA in Isabelle [29], even though remarkable for its practical usability, distinguishes itself from ours by being based on an axiomatization.

11 Discussion and Conclusion

We gave an overview of the verification framework for I/O automata in Isabelle, including temporal logics, proof support for live implementation relations, and the combination of Isabelle with model checking via abstraction. Some aspects had to be neglected or only glimpsed at, e.g. the coalgebraic proof support, improved abstraction rules, or the translations to model checkers. The entire I/O automata framework consists of 465 theorems, proved in 2332 proof steps. Its practical usability has been proven by an industrial case study of a cockpit alarm system [30].

In the following we discuss the advantages of our environment. First, we mention the more general benefits of tool supported verification, which gain, however, significantly by our particular choice of I/O automata, Isabelle, and higher-order logic.

Confidence in Specification The expressiveness of higher-order logic and Isabelle's facilities for structuring specifications and proofs (theories, lemmas, hierarchic simplification sets, etc.) guarantee a neat correspondence between the formal description and the actual specification on the one hand (cf. the direct translation of I/O automata into Isabelle), and the proof scripts and the informal, intuitive arguments on the other hand. This increases the confidence in really specifying and proving what one actually had in mind.

Confidence in Verification Computer-assisted proofs are usually more reliable than manual proofs, as they force the user to supply details for all cases. There are two reasons why this confidence in machine-checked proofs is even higher in our case than e.g. for proofs performed in PVS [33] or LP [13]. First, Isabelle itself is built according to the LCF system approach [35], which means that every proof is broken down to a small and clear set of primitive inferences. Second, we introduce new theories only in a definitional way, which ensures that no inconsistencies, for example caused by contradictory axioms, can occur.

Scalability Interactive provers like Isabelle in general scale up better than fully automatic proof tools like model checkers. In addition, we believe that our I/O automata formalization is in particular qualified for full-scale applications, because of the following reason. First, abstraction allows to significantly reduce the complexity of proof tasks. Second, proof of both simulations and invariants are essentially performed by case-splitting on actions and states, which increase only linear in the size of the components.

Readability Isabelle provides syntactical facilities like mixfix operators and powerful translations. Furthermore, most of the standard mathematical symbols like $\forall$, $\wedge$, $\in$ are supported[8]. Together with the expressiveness of higher-order logic this increases readability and thus considerably speeds up interactive proofs and the search for errors in specifications.

Rechecking During the course of carrying out a large proof, it is likely that definitions or lemmas have to be modified. Furthermore, one likes to polish already completed proofs. This can much safer and easier be done with computer-assistance.

There are further aspects which in particular profit from our methodology which combines HOL and HOLCF in such a way that both meta-theory and system verification are handled in the adequate logic, respectively. First, we discuss the rôle of HOL, i.e. the benefits for the user.

Automation Isabelle's automatic proof procedures permit to discharge the large amount of trivial, intermediate steps or simple cases which usually appear in refinement proofs. This applies in particular to proof tools which are tailored for the use in HOL.

Reuse HOL provides large data type libraries, which can be reused and need not be redone for each application.

Simplicity The conceptually simple HOL is much easier to use than HOLCF, which incorporates the entire complexity of Scott's domain theory.

Second, we discuss the rôle of HOLCF, i.e. the advantages for establishing meta-theory.

Expressiveness HOLCF provides infinite datatypes and arbitrary recursion. This allows to define runs of automata and powerful recursive functions like infinite concatenation or sophisticated merge functions, which turned out to be crucial for proving meta-theoretic results.

Extensibility Having the meta-theory at our disposal, we have a greater degree of flexibility because we do not need to hardwire certain proof methods but can derive new ones at any point.

Besides our HOL/HOLCF methodology there is a further specific advantage of our tool environment. It concerns the domain-theoretic sequence model, which turned out to be especially adequate for extending meta theory to temporal logic and live I/O automata. In particular, in contrast to existing TLA embeddings [24, 39, 8] it allows to incorporate finite sequences and to deal with certain operators that deal with stuttering.

References

[1] M. Archer and C. Heitmeyer. Human-style theorem proving using PVS. In E. Gunter, editor, *Proc. 10th Int. Conf. on Theorem Proving in Higher Order Logics (TPHOL'97)*, volume 1275 of *Lecture Notes in Computer Science*, pages 33–48. Springer-Verlag, 1997.

[8] Actually, the syntax employed in this paper represents only a slight modification, concerning subscripts and further special symbols.

[2] M. Archer and C. Heitmeyer. Verifying hybrid systems modeled as timed automata: A case study. In *Proc. Int. Workshop on Hybrid and Real-Time Systems (HART'97)*, volume 1201 of *Lecture Notes in Computer Science*, pages 171–185. Springer-Verlag, 1997.

[3] A. Biere. *Effiziente Modellprüfung des μ-Kalküls mit binären Entscheidungsdiagrammen.* PhD thesis, Institut für Informatik, Universität Karlsruhe, Germany, 1997.

[4] A. Biere. μcke – Efficient μ-calculus model checking. In O. Grumberg, editor, *Proc. 9th Int. Conf. on Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 468–471. Springer-Verlag, 1997.

[5] N. Bjørner, A. Browne, E. Chang, M. Colón, A. Kapur, Z. Manna, H. B. Sipma, and T. E. Uribe. STeP: deductive-algorithmic verification of reactive and real-time systems. In R. Alur and T. A. Henzinger, editors, *Computer Aided Verification: 8th International Conference*, volume 1102 of *Lecture Notes in Computer Science*, pages 415–418. Springer-Verlag, 1996.

[6] D. Bosscher, I. Polak, and F. Vaandrager. Verification of an audio control protocol. In W. d. R. H. Langmaack and J. Vytopil, editors, *Proc. 3rd Int. School and Symposium on Formal Techniques in Real Time and Fault Tolerant Systems (FTRTFT'94)*, volume 863 of *Lecture Notes in Computer Science*, pages 170–192. Springer, 1994.

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

[8] C.-T. Chou. Predicates, temporal logic, and simulations. In J. Joyce and J. Seger, editors, *Proc. 6th Int. Workshop on Higher Order Logic Theorem Provers and Applications*, volume 780 of *Lecture Notes in Computer Science*, pages 310–323. Springer-Verlag, 1993.

[9] A. Church. A formulation of the simple theory of types. *J. Symbolic Logic*, 5:56–68, 1940.

[10] M. Devillers and D. Griffioen. A formalization of finite and infinite sequences in PVS. Technical Report CSI-R9702, Computing Science Institute, University of Nijmegen, 1997.

[11] M. Devillers, D. Griffioen, and O. Müller. Possibly infinite sequences in theorem provers: A comparative study. In E. Gunter, editor, *Proceedings of the 10th International Conference on Theorem Proving in Higher Order Logics (TPHOL'97)*, volume 1275 of *Lecture Notes in Computer Science*, pages 89–104. Springer-Verlag, 1997.

[12] G. Dowek, A. Felty, H. Herbelin, G. Huet, C. Murthy, and C. P. et al. The Coq proof assistant user's guide version 5.8. Technical Report 154, INRIA, May 1993.

[13] S. Garland and J. Guttag. A guide to LP, the Larch Prover. Technical Report TR-82, DEC Systems Research Center, 1991.

[14] S. Garland, N. Lynch, and M. Vaziri. IOA: A language for specifying, programming, and validating distributed systems. Technical report, Laboratory for Computer Science, MIT, Cambridge, MA., September 1997.

[15] R. Gawlick, R. Segala, J. Sogaard-Andersen, and N. Lynch. Liveness in timed and untimed systems. Technical Report MIT/LCS/TR-587, Laboratory for Computer Science, MIT, Cambridge, MA., December 1993.

[16] D. Griffioen and M. Devillers. *Personal Communication.* 1998.

[17] T. Hamberger. Verifikation einer Hubschrauberüberwachungskomponente mit Isabelle und STeP. 1998. Student software development project, Technische Universität München, Germany.

[18] T. Hamberger. Kombination von Theorembeweisen und Model Checking für I/O Automaten. Master's thesis, Computer Science Department, Technical University Munich, 1999.

[19] L. Helmink, M. Sellink, and F. Vaandrager. Proof-checking a data link protocol. In H. Barendregt and T. Nipkow, editors, *Proceedings Workshop Esprit BRA "Types for Proofs and Programs"*, Nijmegen, The Netherlands, May 1993, number 806 in Lecture Notes in Computer Science. Springer-Verlag, 1994. Full version available as Report CS-R9420, CWI, Amsterdam, March 1994.

[20] D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27(1):333–354, 1983.

[21] F. Kröger. *Temporal Logic of Programs*. Springer-Verlag, 1987.

[22] R. Kurshan, M. Merritt, A. Orda, and S. Sachs. Modelling asynchrony with a synchronous model. In P. Wolper, editor, *Proc. 7th Int. Conf. Computer Aided Verification*, volume 939 of *Lecture Notes in Computer Science*, pages 339–352. Springer-Verlag, 1995.

[23] L. Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[24] T. Långbacka. A HOL formalisation of the Temporal Logic of Actions. In T. Melham and J. Camilleri, editors, *Proc. 7th Int. Workshop on Higher Order Logic Theorem Provers and Applications*, volume 859 of *Lecture Notes in Computer Science*, pages 332–347. Springer-Verlag, 1994.

[25] V. Luchangco, E. Söylemez, S. Garland, and N. Lynch. Verifying timing properties of concurrent algorithms. In *Proc. 7th Int. Conf. Formal Description Techniques (FORTE'94)*, pages 259–273. Chapman & Hall, 1994.

[26] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, 1996.

[27] N. Lynch and F. Vaandrager. Forward and backward simulations – part II: Timing-based systems. *Information and Computation*, 128(1):1–25, 1996.

[28] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, New York, NY, 1995.

[29] S. Merz. Mechanizing TLA in Isabelle. In *Workshop Verification in New Orientations*, 1995. Technical Report, University Maribor.

[30] O. Müller. *A Verification Environment for I/O Automata Based on Formalized Meta-Theory*. PhD thesis, Institut für Informatik, Technische Universität München, 1998.

[31] O. Müller. A verification environment for I/O automata — part I: Temporal logic and abstraction. 1999. submitted.

[32] O. Müller, T. Nipkow, D. Oheimb, and O. Slotosch. HOLCF = HOL + LCF. *J. Functional Programming*, 1999. to appear.

[33] S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. Srivas. PVS: combining specification, proof checking, and model checking. In R. Alur and T. A. Henzinger, editors, *Computer Aided Verification*, volume 1102 of *LNCS*. Springer, 1996.

[34] D. Park. Finiteness is mu-ineffable. *Theoretical Computer Science*, 1(3):173–181, 1976.

[35] L. C. Paulson. *Logic and Computation*. Cambridge University Press, 1987.

[36] L. C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[37] T. P. Petrov, A. Pogosyants, S. J. Garland, V. Luchangco, and N. A. Lynch. Computer-assisted verification of an algorithm for concurrent timestamps. In *Proc. 9th Int. Conf. on Formal Description Techniques and Protocol Specification, Testing, and Verification (FORTE/PSTV'96)*, pages 29–44. Chapman & Hall, 1996.

[38] J. F. Soegaard-Andersen, S. J. Garland, J. V. Guttag, N. A. Lynch, and A. Pogosyants. Computer-assisted simulation proofs. In C. Courcoubetis, editor, *Proc. 5th Int. Conf. Computer-Aided Verificatio (CAV'93)*, volume 697 of *Lecture Notes in Computer Science*, pages 305–319. Springer-Verlag, 1993.

[39] J. Wright. Mechanising the Temporal Logic of Actions in HOL. In M. Archer, J. Joyce, K. Levitt, and P. Windley, editors, *Proc. 1991 Int. Workshop on the HOL Theorem Proving System and its Applications*, pages 155–159. IEEE Computer Society Press, 1992.