# Traces of I/O-Automata in Isabelle/HOLCF

Olaf Müller* and Tobias Nipkow**

Institut für Informatik, Technische Universität München
D-80290 München, Fax +49-89-2892-8183
{mueller,nipkow}@informatik.tu-muenchen.de

**Abstract.** This paper presents a formalization of finite and infinite sequences in domain theory carried out in the theorem prover Isabelle. The results are used to model the metatheory of I/O automata; they are, however, applicable to any trace based model of parallelism which distinguishes internal and external actions. We make use of the logic HOLCF, an extension of HOL with domain theory and show how to move between HOL and HOLCF. This allows us to restrict the use of HOLCF to metatheoretic arguments while actual refinement proofs between I/O automata are carried out within the simpler logic HOL. In order to evaluate the formalization we prove the correctness of a generalized refinement concept in I/O automata.

## 1 Introduction

This paper is concerned with formal models of finite and infinite behaviours of concurrent automata in a theorem prover. The aim of this work is to provide the formal basis for the verification of distributed systems. We believe that it is not sufficient to merely use a theorem prover to discharge externally generated proof obligations but that the metatheory of the underlying formal model should also be supported by the theorem prover. This does not only rule out potential sources of unsoundness (like external verification condition generators). It also provides a greater degree of flexibility because we do not need to hardwire certain proof methods but can derive new ones from the metatheory at any point.

This work is carried out in the context of I/O automata (IOA), a popular model of distributed systems which has been used for a number of non-trivial applications, e.g. in the area of communication protocols [9, 4]. The results, however, apply to any trace based model of parallelism which distinguishes internal and external actions.

The starting point for our work is an existing formalization of I/O automata in Isabelle/HOL, the higher order logic of the theorem prover Isabelle [17]. (Unless noted otherwise, HOL will refer to Isabelle/HOL rather than Gordon's HOL system [7].) The capabilities of this formalization have been illustrated with two protocol verifications [13, 12] where Isabelle was also combined with a model

* Research supported by BMBF, *KorSys*.
** Research supported by ESPRIT BRA 6453, *Types*.

checker. However, moving to more sophisticated examples we realized some inadequacies of our formalization which are caused by the fact that we model traces as functions from time to actions. In particular, this formalization was restricted to a rather limited refinement notion.

The purpose of this paper is to provide I/O automata with a new and more powerful model of traces based on lazy lists as in functional programming. Logically this means we leave the HOL world of total functions and enter into domain theory, i.e. the world of partial functions and undefined and infinite objects. This step should not be taken lightly because partiality complicates the logic and the proofs. Fortunately, Isabelle also supports HOLCF, an extension of HOL with the notions of domain theory. Hence we can work in HOL as long as possible and only move into HOLCF if really required. Part of this paper provides a methodology for moving between the two levels. This allows to use HOLCF for the more sophisticated metatheory, whereas normal refinement proofs can still be done in the simpler logic HOL. The main benefit of our new model of traces is a generalized refinement concept which the simpler HOL model does not permit.

The overall aim of our work is to provide a tool environment for the analysis of I/O automata, including the Isabelle formalization described in this paper, a model checker and an appropiate abstraction methodology.

The structure of the paper is as follows. After a brief introduction to the existing model of I/O automata in HOL (Section 2), we point out the problem with its weak refinement concept (Section 3). Then we introduce HOLCF and the means for moving from HOL to HOLCF (Section 4). Finally we recast trace theory in HOLCF (Section 5) and generalize the refinement concept (Section 6).

### 1.1 Related Work

Infinite sequences are part of many trace based specification formalisms. Nevertheless there is not as much related work as one might expect, as the underlying metatheory is not always formalized. Often the theorem prover is only used to prove refinements, but the refinement notion itself is not semantically embedded. This is particularly true for a couple of case studies within the I/O automata model – for example Fischer's protocol [9] and an audio control protocol provided by Philips Laboratories [4] – carried out in the Larch prover and Coq.

Closely related to our work are the papers of Chou and Peled [3] and Loewenstein [8]. Chou and Peled model infinite and finite sequences as a prerequisite for the formal verification of a partial-order reduction technique in the theorem prover HOL [7]. Their formalization models sequences as the disjoint union of finite lists and the type $\mathsf{nat}\Rightarrow\alpha$ which represents infinite sequences. Whereas we can build on top of a logic describing domain theory in general, they provide such concepts as prefix ordering or limits of ascending chains in a more ad hoc fashion tailored for their specific dataypes. Loewenstein develops a formal theory of simulations between infinite automata in the theorem prover HOL. His sequences are functions of type $\mathsf{nat}\Rightarrow\alpha$. Finite sequences are just seen as prefixes of infinite sequences; they are not explicitly used to describe system behaviour,

but to facilitate the proofs, and therefore less requirements than in our setting are imposed on them.

Besides domain theory there are other logical frameworks that apply to the modelling of finite and infinite sequences. Feferman [5] develops a generalized recursion theory which does not need continuity for fixed point recursion and applies it to potentially infinite sequences. This approach has not been formalized in a theorem prover until now. Coinduction [15] provides another computation scheme based on bisimulations, but deals only with infinite or finite terminating sequences, and it is not obvious how to extend this approach to deal with computation on finite nonterminating sequences.

## 2 I/O-Automata in HOL

**HOL notation.** All formulas have been taken directly from the Isabelle input and translated automatically into LaTeX, thanks to a version of Isabelle/HOL that allows the use of mathematical symbols like $\exists$ or $\forall$.

Set comprehension has the shape {e. P}, where e is an expression and P a predicate. The projection functions on pairs are called fst and snd. Tuples are pairs nested to the right, e.g. (s,a,t) represents (s,(a,t)). All functions in HOL are total and the type constructor is $\Rightarrow$. If f is a function of type $\rho \Rightarrow \sigma \Rightarrow \tau$, application is written f x y. If there is only one argument we sometimes write rather f(x) than f x. Function composition is defined as (f o g)(x) = f(g(x)). Conditional expressions are written if A then B else C.

### 2.1 I/O Automata

I/O automata are finite or infinite state automata with labelled transitions and were initially introduced by Lynch and Tuttle [10]. The formalization in HOL sketched in this section represents only a fragment of the theory one can find in recent papers [6]. For example, we do not deal with fairness or time constraints. The details of the formalization can be found in a previous paper [13]. Here we focus on how to model traces and the refinement concept.

In the HOL model, an action signature is described by the type

$\alpha$ signature = ($\alpha$ set $*$ $\alpha$ set $*$ $\alpha$ set)

The first, second and third component of an action signature S is extracted with inputs, outputs, and internals. Furthermore we have

actions(S) $\equiv$ inputs(S) $\cup$ outputs(S) $\cup$ internals(S)
externals(S) $\equiv$ inputs(S) $\cup$ outputs(S).

Action signatures have to satisfy the following disjointness condition:

$$\begin{aligned} \text{is\_asig(triple)} \equiv\ & (\text{inputs(triple)} \cap \text{outputs(triple)} = \{\}) \quad \wedge \\ & (\text{outputs(triple)} \cap \text{internals(triple)} = \{\}) \wedge \\ & (\text{inputs(triple)} \cap \text{internals(triple)} = \{\}) \end{aligned}$$

An IOA is a triple of type

$(\alpha,\sigma)$ioa $= \alpha$ signature $* \sigma$ set $* (\sigma * \alpha * \sigma)$ set

(where the parameters $\alpha$ and $\sigma$ represent the type of actions and states) subject to the following predicate:

IOA (asig,starts,trans) $\equiv$ is_asig(asig) $\wedge$ starts $\neq \{\}$ $\wedge$ state_trans asig trans

Predicate state_trans requires in particular that the transition relationship is input-enabled:

state_trans asig R $\equiv$ $(\forall$(s,a,t)$\in$R. a$\in$actions(asig)) $\wedge$
$(\forall$a$\in$inputs(asig). $\forall$s. $\exists$t. (s,a,t)$\in$R)

The components of an IOA are extracted by asig_of, starts_of, and trans_of. The actions of an IOA are defined acts $\equiv$ actions o asig_of.

### 2.2 Executions and Traces in HOL

An *execution-fragment* of an IOA A is a finite or infinite sequence that consists of alternating states and actions. In HOL it is represented as a pair of sequences: an infinite *state sequence* of type nat $\Rightarrow$ state and an *action sequence* of type nat $\Rightarrow$ (action)option where

datatype $(\alpha)$option = None | Some$(\alpha)$

using an ML-like notation. A finite sequence in this representation ends with an infinite number of consecutive Nones. Using this representation, a step of an execution-fragment (as,ss) is (ss(i),a,ss(i+1)) if as(i) = Some(a). Formally:

is_execution_fragment A (as,ss) $\equiv$
$\forall$n a. (as(n)=None $\longrightarrow$ ss(Suc(n))=ss(n)) $\wedge$
(as(n)=Some(a) $\longrightarrow$ (ss(n),a,ss(Suc(n)))$\in$trans_of(A))

Note that there is no requirement that None be followed only by None. Nones may occur at arbitrary points in the sequence, indicating that no action has been performed. In the trade this is known as "invariance under stuttering" [1]. An example execution-fragment is shown below.

| as: | Some($a_1$) | Some($a_2$) | None | Some($a_3$) | None | ... |
|---|---|---|---|---|---|---|
| ss: | $s_1$ | $s_2$ | $s_3$ | $s_3$ | $s_4$ | ... |

An *execution* of A is an execution-fragment of A beginning in a start state of A:

executions(A) $\equiv$ {(as,ss) . ss(0)$\in$starts_of(A) $\wedge$is_execution_fragment A (as,ss)}

If we filter the action sequence of an execution of A so that it has only external actions, we obtain a *trace* of A. The traces of A are defined by

traces(A) $\equiv$ {filter($\lambda$a.a$\in$externals(asig_of(A)) as . $\exists$ss. (as,ss)$\in$executions(A)}

where filter P replaces Some(a) by None if P(a) does not hold:

filter P as $\equiv \lambda$i. case as(i) of
None $\Rightarrow$ None
| Some(a) $\Rightarrow$ if P(a) then Some(a) else None

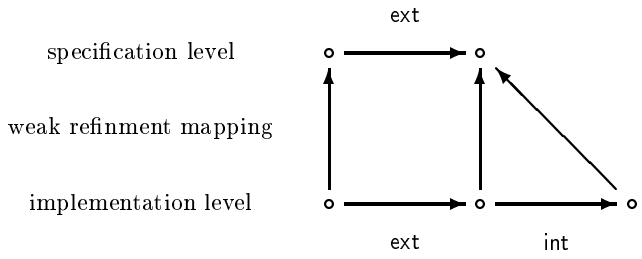


**Fig. 1.** Simulation by a weak refinement: ext external action, int internal action

### 2.3 Refinement Mappings in HOL

A refinement mapping f maps the states of a concrete automaton C (the implementation) to those of an abstract automaton A (the specification). The IOA formalization in HOL supports a weak concept of refinement mappings defined as follows (see also Fig. 1):

```
is_weak_refmap f C A =
      (∀ s∈starts_of(C). f(s)∈starts_of(A)) ∧
      (∀ s t a. reachable C s ∧ (s,a,t)∈trans_of(C)
       ⟶ if a∈externals(asig_of(A)) then (f(s), a,f(t))∈trans_of(A)
                                    else f(s) = f(t))
```

The following theorem proved in HOL states that the existence of a weak refinement mapping implies that the traces of C are contained in those of A:

```
IOA(C) ∧ IOA(A) ∧
externals(asig_of(C)) = externals(asig_of(A)) ∧
is_weak_refmap f C A
⟶ traces(C) ⊆ traces(A)
```

This notion of a refinement mapping is weaker than the ones usually used in the literature [11] because it does not allow internal actions in the abstract automaton. In particular, is_weak_refmap (λx.x) C C does not hold for all C.

## 3 Problems with the HOL Model

### 3.1 Example for Necessity of Normal Forms

Unfortunately the I/O automata model using the datatype option has some drawbacks. Informally speaking, None stands for nothing, but it is not really nothing. Therefore traces differ only because of a different number of Nones in them, although they are semantically equivalent. This leads to an inadequate representation of the notion of refinement, as the following example shows.

Let A and C be the two automata in Fig. 2, where act and int are an external and internal action respectively. In HOL this becomes


**Fig. 2.** Observably equal I/O-Automata

$A \equiv (\{\},\{act\},\{int\}),\{s\},\{(s,act,t),(t,int,s)\})$
$C \equiv (\{\},\{act\},\{\}),\{s'\},\{(s',act,s')\})$

These are observably identical automata, as int is internal. Therefore we would expect $traces(C) \subseteq traces(A)$. Now consider the action sequence $as \equiv \lambda i.Some(act)$. We have $as \in traces(C)$ but $as \notin traces(A)$. In our representation as is not a legal trace of A, because every infinite execution of A has also infinitely many internal actions int and filtering internal actions yields Nones, which cannot be eliminated further. Therefore A cannot produce as but only some sequence like

$as' \equiv \lambda i.$if even(i) then Some(act) else None

Notice that as' is also a possible trace of C, because our formalization allows the insertion of a finite number of Nones: our automata allow "stuttering", but they do not allow "mumbling" [2], i.e. the removal of None-steps which should not be observable.

Within this representation it is generally not possible to establish a refinement, if the abstract automaton has internal actions. In other words, the weak refinement mappings defined in Section 2.3 are already the most general refinement notion we could prove in this representation. This is a severe restriction we will now try to lift.

### 3.2 Requirements for a Datatype of Sequences

What we really need are normal forms of traces, where Nones are not allowed within a trace, but only at the end to indicate infinity. Such a normal form can be defined by demanding a monotone function f between traces that serves as an index transformation:

$$\begin{aligned} NF(tr) \equiv \varepsilon nf.\ \exists f.\ mono(f) \wedge (\forall i.\ nf(i)=tr(f(i))) \wedge \\ (\forall j.\ j \notin range(f) \longrightarrow tr(j)= None) \wedge \\ (\forall i.\ nf(i)=None \longrightarrow (nf(Suc\ i)) = None) \end{aligned}$$

Here $\varepsilon x.P(x)$ denotes Hilbert's description operator which stands for some a satisfying P(a). But the definition of NF shows already that such index transformations are very awkward to handle. Another complication is the definition

of infinite concatenation which will be necessary in a more general refinement proof.

Therefore we investigated different models of executions. The starting point was a collection of requirements for an abstract datatype of executions. These requirements are extracted from the proof outlines of IOA metatheory and will become clear in later sections when the proofs are described. Firstly, we need finite and infinite sequences. Secondly, operations on them should include hd, tl, map and filter. Thirdly, a predicate finite should exist and infinite concatenation must be expressible. All the above requirements are fulfilled very naturally by the well-known notion of "lazy lists" from functional programming. HOLCF directly supports the definition of lazy lists. Therefore we decided to model traces and executions in HOLCF.

# 4 HOLCF

## 4.1 Introduction

HOLCF [18] extends HOL with concepts of domain theory such as complete partial orders, continuous functions and a fixed point operator. As a result, the logic LCF [16] constitutes a proper sublanguage of HOLCF.

In HOLCF there is a special type for continuous functions. Elements of this type are called *operations*, the type constructor is denoted by $\rightarrow$ in contrast to the standard function type constructor $\Rightarrow$. For abstractions and applications of operations a specific syntax is introduced. The term $\Lambda$x.t denotes an abstraction of type $\sigma\rightarrow\tau$, and the term f`x denotes an application with f of type $\sigma\rightarrow\tau$.

HOLCF uses Isabelle's type classes to distinguish HOL and LCF types. More precisely, it introduces a type class pcpo of pointed complete partial orders, which becomes the default type class of HOLCF. It is a subclass of term, the default type class of HOL. The function space constructor $\rightarrow$ has arity (pcpo,pcpo)pcpo, i.e. $\sigma\rightarrow\tau$ is of class pcpo provided both $\sigma$ and $\tau$ are.

HOLCF comes with several standard domains. tr, the truth values, which are HOLCF's counterpart to HOL's bool, is a flat domain with the elements TT, FF and $\bot$. Operations on them include andalso, orelse and neg, which are strict extensions of the standard predicates $\wedge$,$\vee$ and $\neg$ on bool.

HOLCF also provides a datatype package [14] that allows to introduce pcpo datatypes as simple recursive domain equations. The package proves a number of theorems concerning the constructors, discriminators, and selectors of the datatype, as well as induction and co-induction principles. For example, the following equation

$$\text{domain } (\alpha)\text{sequence} = \text{nil} \mid (\alpha)\#(\text{lazy } (\alpha)\text{sequence}) \qquad (1)$$

defines the domain of finite and infinite sequences that are built by the constructors nil and #. The "cons"-operator # is strict in its first argument and lazy in the second.

### 4.2 Lifting

Such domain definitions as $(\alpha)$sequence above require that the argument type $\alpha$ has to be a domain type, too. However, for the application we have in mind — executions and traces of automata — this is rather inconvenient. Actions and states are more naturally modelled as HOL datatypes without dragging undefined elements and partial orders into it. In general we prefer to stay on the level of HOL types as long as possible and switch to pcpo types only if really required. In our context the advantage would be that metatheory (in HOLCF which offers more expressiveness and flexibility) can be hidden from the normal refinement proofs (in HOL which is easier to use).

To achieve this goal we introduce a type constructor lift of arity (term)pcpo which lifts every HOL-datatype to a pcpo type:

datatype $(\alpha)$lift $\equiv$ Undef | Def($\alpha$)

The least element and the approximation ordering are defined very easily:

$\bot \equiv$ Undef
x $\sqsubseteq$ y $\equiv$ (x=y) | x=Undef

This is known as a *flat* domain. Note that $\bot$ and $\sqsubseteq$ are overloaded and this definition only fixes their meaning at type $(\alpha)$lift.

If in an operation on a lifted datatype $(\alpha)$lift a total function on $\alpha$ is involved, it is necessary to lift also this total function to a partial operation. Therefore we introduce a number of functionals that transform HOL functions to HOLCF operations using lift. The type variables $\alpha$,$\alpha_1$ and $\alpha_2$ are of class term, whereas $\beta$ is of class pcpo.

| | | |
|---|---|---|
| bool_lift | bool | $\Rightarrow$ tr |
| pred_lift | $(\alpha \Rightarrow$ bool$)$ | $\Rightarrow ((\alpha)$lift $\rightarrow$ tr$)$ |
| fun_lift_1 | $(\alpha \Rightarrow \beta)$ | $\Rightarrow ((\alpha)$lift $\rightarrow \beta)$ |
| fun_lift_2 | $(\alpha_1 \Rightarrow \alpha_2)$ | $\Rightarrow((\alpha_1)$lift $\rightarrow (\alpha_2)$lift$)$ |

The functional bool_lift lifts booleans to truth values, pred_lift lifts predicates, and fun_lift_1 resp. fun_lift_2 lift functions, the first only the argument type, the second also the result type. Formally:

```
bool_lift b ≡ if b then TT else FF
fun_lift_1 f ≡ Λx. case x of
                     Undef  ⇒ ⊥
                   | Def(y) ⇒ f(y)
fun_lift_2 f ≡ Λx. case x of
                     Undef  ⇒ ⊥
                   | Def(y) ⇒ Def(f(y))
pred_lift p  ≡ Λx. fun_lift_1 (λb. bool_lift (p b)) x
```

Had tr been defined as (bool)lift, which, for historical reasons, it has not been, then bool_lift would be superfluous and pred_lift would reduce to a special case of fun_lift_2. This shows that in principle two functionals would suffice.

Using the above lifting functions has the following advantages: Firstly, these concepts are frequently used, and abbreviating them increases readability. Secondly, continuity proofs are facilitated and automated. In HOLCF the $\beta$-reduction on domains is subject to the continuity restriction $\mathsf{cont(t)} \longrightarrow (\Lambda \mathsf{x.t(x)})\text{`}\mathsf{u} = \mathsf{t(u)}$ where $\mathsf{cont(t)}$ means that $\mathsf{t}$ is continuous. These continuity proof obligations are solved automatically for all terms of the LCF sublanguage ($\Lambda$-abstractions and `-applications). But for normal HOL terms these proof obligations have to be discharged manually. Therefore the lifting functionals can serve as a "continuity interface" to HOL. By proving them to be continuous and adding these theorems to the automatic proof tactic, we get automatic continuity proofs also for the combination of HOL and LCF terms.

## 5 IOA in HOLCF

Most parts of the I/O automata model remain unchanged. Only the notions of executions and traces are modelled in HOLCF domains. Therefore we restrict the description of the HOLCF automata model to them. The last section laid the foundation for such a hybrid description, as the type $(\alpha)\mathsf{lift}$ allows sequences to contain elements of HOL datatypes.

### 5.1 Appropriate Modelling of Sequences

Executions and traces are finite or infinite sequences that we decided to model by the domain equation (1) of section 4. This means that elements of type sequence come in 3 flavours:

- Finite total sequences: $\mathsf{a}_1\#\ldots\#\mathsf{a}_n\#\mathsf{nil}$. They are generated by processes which terminate after a finite number of output actions.
- Finite partial sequences: $\mathsf{a}_1\#\ldots\#\mathsf{a}_n\#\bot$. They are generated by processes which do not terminate but produce no more output after some point, e.g. by filter. Having this type of sequences at hand allows us to distinguish between automata that terminate and those that do not terminate but go on producing only internal steps.
- Infinite sequences: $\mathsf{a}_1\#\ldots\#\mathsf{a}_n\#\ldots$. They are generated by processes which do not terminate but keep on producing output.

All the operations known from functional programming with lazy lists, e.g. hd, tl, map, filter and the concatenation operator @, are easily defined.[3]

### 5.2 Appropriate Modelling of Executions

There are several ways to model executions by the sequences described above. Indeed, we spent a lot of time to find the most appropriate one.

[3] The actual implementation uses different names for these operations because the above ones are already used in HOL's theory of finite lists.

– First, it is inconvenient to use a pair of sequences, one for actions and one for states

  (action,state)execution = ((state)lift)sequence ∗ ((action)lift)sequence

  because this allows them to be of different length, which we then have to rule out explicitly.
– Second, one could imagine a sequence of transition triples:

  (action,state)execution = ((state ∗ action ∗ state)lift)sequence

  The advantage is that (state ∗ action ∗ state) triples are already part of the automaton definition. But an important drawback is the redundancy of the representation. It has to be guaranteed that the transitions coincide on the intermediate states: a sequence ...#Some(s1,a1,s2)#Some(s3,a2,s4)#... is an execution only if s2 = s3.
– Finally, a pair of a start state and a sequence of action/state pairs turned out to be most appropriate:

  (action,state)execution = state ∗ ((action ∗ state)lift)sequence

  In the sequel exec stands for variables of type execution, whereas s denotes the start state and ex the sequence of action/state pairs. The additional start state is necessary because otherwise the first transition starts from an unknown state. However, this additional start state would have been necessary for a sequence of transition triples as well, in order to associate a state with the empty execution. This is necessary for simulation steps, where the empty execution is used to simulate a step of the implementation. Here it would be very complicated with an empty execution without state (nil) to keep track of the connection to the state of the preceding simulation step.

### 5.3 HOLCF Formalization of Executions and Traces

The predicate is_execution_fragment is realized by an operation is_ex_fr that "runs down" a sequence checking if all of its transitions are transition of the automaton A. The predicate is true if the operation terminates and returns TT (for finite executions) or if the search does not terminate (⊥ — for infinite executions).

```
is_execution_fragment A (s,ex) ≡ is_ex_fr A‘ex s ≠ FF
```

The operation is_ex_fr is defined as a fixpoint. The following rewriting rules can be deduced immediately from the definition.

```
is_ex_fr A‘⊥ s = ⊥
is_ex_fr A‘nil s = TT
is_ex_fr A‘(Def(a,t)#ex) s =
          bool_lift ((s,a,t)∈trans_of(A))
          andalso is_ex_fr A‘ex t
```

Executions are execution fragments that begin in a start state:

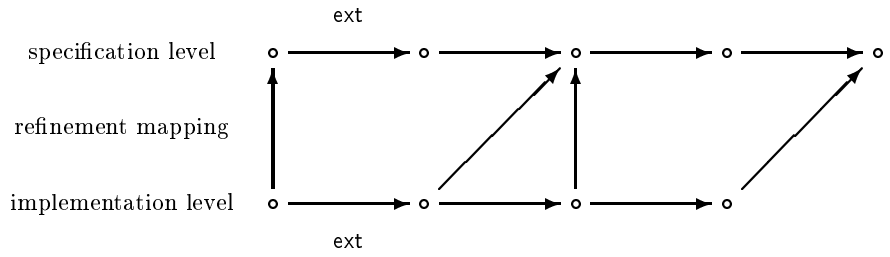


**Fig. 3.** Simulation by a refinement mapping: ext external action, int's are omitted

executions(A) ≡ {(s,ex) . s∈starts_of(A) ∧ is_execution_fragment A (s,ex)}

To obtain the traces of A, a mapping operation filter_act is defined that projects every pair in the execution sequence onto the action component:

filter_act‘ex ≡ map‘(fun_lift_2 fst)‘ex

Afterwards every non-external action of A is filtered out:

mk_trace A‘ex ≡ filter‘(pred_lift(λa.a∈externals(asig_of A)))‘(filter_act ex)

The traces of A are the results of applying mk_trace to the executions of A:

traces(A) ≡ {mk_trace A‘ex. ∃s . (s,ex)∈executions(A)}

As the definitions show, the formalization makes heavy use of the lifting functionals fun_lift_i, pred_lift and bool_lift.

## 6 Refinement Mappings in HOLCF

In order to demonstrate the advantages of our formalization, this section shows the proof of a more general refinement notion than weak refinement mappings.

### 6.1 Refinement Mappings

The notion of a *refinement mapping* is illustrated in Fig. 3. A refinement mapping f allows to simulate a step (s,a,t) of an concrete automaton C not only by another step of the abstract automaton A, but by a complete *move* of A.

```
is_refmap f C A ≡
        (∀s∈starts_of(C). f(s)∈starts_of(A)) ∧
        (∀s t a. reachable C s ∧ (s,a,t)∈trans_of(C)
          ⟶ ∃ex. move A ex (f s) a (f t) )
```

Moves are finite execution-fragments that begin in state f(s), end in state f(t), and perform only internal actions, except the action a, if that is external. This implies in particular that a single internal actions can be simulated by a finite number of internal actions.

```
move A ex s a t ≡
    is_execution_fragment A (s,ex) ∧ finite(ex) ∧
    laststate(s,ex)=t ∧
    mk_trace A‘ex = (if a∈externals(asig_of(A)) then Def(a)#nil else nil)
```

The predicate finite characterizes only the finite sequences that explicitly terminate with nil and excludes partial sequences. The precise definition will be given later on in the context of induction principles. The function laststate extracts the last state of an execution:

```
laststate (s,⊥) = s
laststate (s,nil) = s
laststate (s,Def(a,t)#ex) = laststate (t,ex)
```

### 6.2 Proof Sketch of Correctness

In Isabelle we proved the following correctness theorem:

```
IOA(C) ∧ IOA(A) ∧
externals(asig_of(C)) = externals(asig_of(A)) ∧
is_refmap f C A
⟶ traces(C) ⊆ traces(A)
```

By the way, this theorem shows how to use HOLCF only for metatheory: Whereas the conclusion traces(C) ⊆ traces(A) is formalized using HOLCF, the premises, which have to be fulfilled for refinement proofs, can in most cases be proved in HOL only. Let us now analyze the proof in a backwards direction. By elementary set equalities the claim reduces to

```
IOA(C) ∧ IOA(A) ∧
externals(asig_of(C)) = externals(asig_of(A)) ∧
is_refmap f C A  ∧ exec1∈executions(C)
⟶ ∃exec2∈executions(A) . mk_trace C‘(snd exec1)=mk_trace A‘(snd exec2)
```

That is, for every execution exec1 of C we have to show the existence of a state/sequence pair exec2 that has

- **Subgoal 1**: the same trace as exec1 and
- **Subgoal 2**: is an execution of A.

This "corresponding" execution exec2 can be constructed (in the spirit of the Execution Correspondence Theorem of [6]) by concatenating all the finite moves of A that simulate the single steps of C. The function corresp_ex simply takes care of the start state, whereas corresp_ex2 does all the work by running down the concrete execution:

corresp_ex A f (s,ex) ≡ (f(s),corresp_ex2 A f‘ex (f(s)))

corresp_ex2 A f‘⊥ s = ⊥
corresp_ex2 A f‘nil s = nil
corresp_ex2 A f‘(Def(a,t)#ex) s =
snd(εexec. move A exec s a t) @ corresp_ex2 A f‘ex t

Here ε again denotes Hilbert's description operator. Note that εexec always exists because the definition of is_refmap exactly states the existence of a simulation move for every reachable state of C.

Note that corrsp_ex2 constructs an *infinite* concatenation, which would have been more complicated to define in pure HOL.

**Subgoal 1.** To prove trace equality we mainly need distributivity of trace generation over concatenation:

**Lemma1**
mk_trace A‘(ex1@ex2) = (mk_trace A‘ex1) @ (mk_trace A‘ex2)

Whereas the move property guarantees trace equality already for every move of A and its simulated step of C, lemma 1 extends these stepwise equalities to the global equality of the whole traces of ex1 and ex2.

**Subgoal 2.** Just as before, the move property yields already the property of being an execution-fragment for every simulation move. To prove the property for the whole corresponding execution, we need a lemma that propagates it from single executions ex1 and ex12 to their concatenation ex1@ex2. Of course, ex1 and ex2 have to be related in such a way that the last state of ex1 is at the same time first state of ex2.

**Lemma2**
finite(ex1)
⟶ is_execution_fragment A‘(s,ex1) ∧ is_execution_fragment A‘(t,ex2)
∧t=laststate(s,ex1)
⟶ is_execution_fragment A (s,ex1@ex2)

Notice that the assumption finite(ex1) is not necessary, as the proof goal of Lemma 2 is_execution_fragment A (s,ex1@ex2) reduces to is_execution_fragment A‘(s,ex1) if ex1 is partial finite or infinite. But in our context we need the lemma only under this assumption, as we argue about moves, and the move property includes the finiteness requirement. We use the finiteness assumption because it facilitates the proof, as we will see in the next section.

### 6.3 Structural Induction Principles

This section shows two different induction principles that were used in the proof. For Lemma 1 and most of the other lemmas not mentioned here a structural induction rule can be used that is automatically generated by the datatype package of HOLCF:

$$\mathsf{adm(P)} \wedge \mathsf{P(nil)} \wedge \mathsf{P}(\bot) \wedge (\forall \mathsf{x\ xs.\ x} \neq \bot \wedge \mathsf{P(xs)} \longrightarrow \mathsf{P(x\#xs)}) \longrightarrow \forall \mathsf{y.P(y)}$$

Here adm(P) denotes the admissibility of the predicate P, that is P has to hold for the least upper bound of every chain satisfying P. Often the proof of adm(P) can be reduced to the continuity of all functions occuring in P.

Exactly this continuity condition cannot be fulfilled for Lemma 2, as the function laststate is not continous in ex1. Nevertheless Lemma 2 is admissible, so we could prove it using the admissibility definition directly. But an easier and smarter way is to generate a weaker induction principle that takes advantage of the fact that we need Lemma 2 only for finite ex1.

To get such a principle we define the predicate finite inductively as the least set satisfying the rules finite(nil) and finite(xs) $\wedge$x$\neq\bot \longrightarrow$ finite(x#xs). In this case the inductive datatype package of HOL generates an induction rule of the following shape (which has been used for Lemma 2):

$$\mathsf{P(nil)} \wedge (\forall \mathsf{x\ xs.\ x} \neq \bot \wedge \mathsf{P(xs)} \wedge \mathsf{finite(xs)} \longrightarrow \mathsf{P(x\#xs)})$$
$$\longrightarrow (\forall \mathsf{y}\ .\ \mathsf{finite(y)} \longrightarrow \mathsf{P(y)})$$

### 6.4 Proof Statistics

The formalization of I/O automata in HOLCF turns out to be rather compact: There are about 40 definitions on 8 pages including sequences, automata, traces and refinement. The correctness proof of the refinment mapping includes 180 proof commands on 7 pages and therefore seems to be very concise compared to the handwritten formal proof of [6] of about 5 pages (only counting the relevant parts, as a more general refinement notion is proved there). We argue that this is an advantage of our formalization of sequences as lazy lists. For example, an infinite concatenation in our context is easily defined as done for corresp_ex, whereas in [6] a limit construction of intervals given by indexes is needed.

## 7 Conclusion

We formalized the metatheory of I/O automata in Isabelle/HOLCF and proved the correctness of refinement mappings within this model. The proof appears to be rather concise compared to handwritten proofs which is due to our formalization of potentially infinite sequences in domain theory. This sequence formalization applies to every trace based model of distributed systems that distinguishes between internal and external actions. We argue that an alternative modelling in a setting of total function would be more complicated and less natural.

Furthermore, we provide a methodology to move between HOL, a logic of total functions, and HOLCF, a logic of partial functions. In our context this permits to use the more adequate logic for metatheory and for refinement proofs, respectively. Besides, this allows for the automation of continuity proofs in such a combination of HOL and HOLCF, which compensates the drawback of continuity and admissibility proofs in domain theory.

## References


1. M. Abadi and L. Lamport. The existence of refinement mappings. In *Proc. 3rd IEEE Symp. LICS*, pages 165–177. IEEE Computer Society Press, 1988.
2. S. Brooks. Full abstraction for a shared variable parallel language. In *Proc. 8th IEEE Symp. Logic in Computer Science*, pages 98–109, 1993.
3. C.-T. Chou and D. Peled. Formal verification of a partial-order reduction technique for model checking. In T. Margaria and B. Steffen, editors, *Proc. 2nd TACAS*, volume 1055 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.
4. I. P. D.J.B. Bosscher 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 FTRTFT'94*, volume 863 of *Lecture Notes in Computer Science*, pages 170–192. Springer, 1994.
5. S. Feferman. Computation on abstract data types. the extensional approach, with an application to streams. *Annals of Pure and Applied Logic*, 81:75–113, 1996.
6. 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., 1993. Extended abstract in Proceedings ICALP'94.
7. M. Gordon and T. Melham. *Introduction to HOL: a theorem-proving environment for higher-order logic*. Cambridge University Press, 1993.
8. P. Loewenstein. A formal theory of simulations between infinite automata. *Formal Methods in System Design*, 3(1):117–149, 1993.
9. V. Luchangco, E. Söylemez, S. Garland, and N. Lynch. Verifying timing properties of concurrent algorithms. In *Proc. 7th Int. Conf. Formal Description Techniques*, pages 259–273. IFIP WG6.1, Chapman and Hall, 1994.
10. N. Lynch and M. Tuttle. An introduction to Input/Output automata. *CWI Quarterly*, 2(3):219–246, 1989.
11. N. Lynch and F. Vaandrager. Forward and backward simulations – part I: Untimed systems. *Information and Computation*, 121(2):214–233, 1995.
12. O. Müller and T. Nipkow. Combining model checking and deduction for I/O-automata. In *Proc. 1st Workshop Tools and Algorithms for the Construction and Analysis of Systems*, volume 1019 of *Lecture Notes in Computer Science*, pages 1–16. Springer-Verlag, 1995.
13. T. Nipkow and K. Slind. I/O automata in Isabelle/HOL. In P. Dybjer, B. Nordström, and J. Smith, editors, *Types for Proofs and Programs*, volume 996 of *Lecture Notes in Computer Science*, pages 101–119. Springer-Verlag, 1995.
14. D. v. Oheimb. Datentypspezifikationen in Higher-Order LCF. Master's thesis, Computer Science Department, Technical University Munich, 1995.
15. L. Paulson. Co-induction and co-recursion in higher-order logic. Technical Report TR-334, Univ. of Cambridge, Computer Lab., 1994.
16. L. C. Paulson. *Logic and Computation*. Cambridge University Press, 1987.
17. L. C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.
18. F. Regensburger. HOLCF: Higher Order Logic of Computable Functions. In E. Schubert, P. Windley, and J. Alves-Foss, editors, *Higher Order Logic Theorem Proving and its Applications*, volume 971 of *Lecture Notes in Computer Science*, pages 293–307. Springer-Verlag, 1995.


This article was processed using the LaTeX macro package with LLNCS style