I/O Automata in Isabelle/HOL

Tobias Nipkow* and Konrad Slind**

Technische Universität München***

Abstract. We have embedded the meta-theory of I/O automata, a model for describing and reasoning about distributed systems, in Isabelle's version of higher order logic. On top of that, we have specified and verified a recent network transmission protocol which achieves reliable communication using single-bit-header packets over a medium which may reorder packets arbitrarily.

1 Introduction

This paper describes a formalization of *Input/Output automata* (*IOA*), a particular model for concurrent and distributed discrete event systems due to Lynch and Tuttle [9], inside Isabelle/HOL, a theorem prover for higher-order logic [12].

The motivation for our work is twofold:

- The verification of distributed systems is a challenging application for formal methods because in that area informal arguments are notoriously unreliable.
- This area is doubly challenging for interactive general purpose theorem provers because model-checking [4] already provides a successful automatic approach to the verification of finite state systems.

IOA were chosen as the vehicle for our study because they have become popular for specifying and verifying distributed algorithms both on paper [8, 2] and with machine assistance [10, 6, 13, 7]. The unique aspect of our work is the fact that we have formalized and verified the meta-theory of IOA on top of which we carried out our case study. Thus IOA are objects in the logic, just like natural numbers or lists, which can be manipulated by operators such as parallel composition and hiding.

The first half of the paper describes how (parts of) the rigorous mathematical theory of IOA is completely formalized in higher-order logic. The second half describes a protocol proof built on top of this formalization.

* Research supported by ESPRIT BRA 6453, *TYPES*

** Research supported by DFG grant Br 887/4-2, *Deduktive Programmentwicklung*

*** Institut für Informatik, 80290 München, Germany.
E-mail: {nipkow,slind}@informatik.tu-muenchen.de

1.1 Related work

The same protocol had been verified by the first author 4 years ago, but without the meta-theory, i.e. Isabelle was then merely used to prove some "random" verification conditions. When comparing the two proofs, we noticed that at the time the first author had accidentally missed out one proof obligation to do with initial states. This is a typical example of the kind of mistake that formalized meta-theory helps to avoid.

Helmink et al. [6] follow the same approach: they verify a communication protocol using the Coq system [5] to discharge proof obligations set up by hand.

An interesting compromise is the approach by Garland et al. [13, 7] based on the Larch Prover (LP). They formalize some of the meta-theory (e.g. finite execution fragments and simulation maps) using the Larch Shared Language (LSL). They can then generate verification conditions automatically from LSL specifications and discharge them using LP. In contrast to our work, IOA are a meta-linguistic notion and are not available as objects in the logic. Also, they do not formalize infinite behaviours and do not have an independent notion of implementation based on inclusion of traces. Instead they formalize the notion of a simulation map, which is a sufficient condition for implementation. Part of our work has been to formalize both concepts, simulation maps and implementations, and prove they are related.

2 I/O automata

Lynch and Tuttle [9] give a set-theoretic definition of I/O automata. An IOA is a possibly infinite state automaton where each transition between states is labelled from a set of actions. The set of actions that an automaton A may use is structured into an *action signature* by partitioning the actions into three disjoint sets: *inputs*, *outputs*, and *internals*. The *external* actions are the union of *inputs* and *outputs*.

An IOA A can be modelled with a quintuple $<Asig, St, St_0, \rightarrow, \Xi>$ where

- $Asig$ is an action signature. $acts(Asig) \equiv inputs(Asig) \cup outputs(Asig) \cup internals(Asig)$.
- St is a set of states
- $St_0 \subseteq St$ is a non-empty set of start states
- $\rightarrow \subseteq St \times acts(Asig) \times St$ is a transition relation. An element $s \xrightarrow{a} t$ of this relation is also known as a *step*.
- Ξ is an equivalence relation used to define *fairness*. We do not consider fairness in this paper.

An important property of $\rightarrow$ is that it must be *input-enabled*. By this we mean that for every state s and every input action a, A must have a transition $s \xrightarrow{a} t$. This expresses that an automaton must be able to respond to any input offered by its environment.

A 'run' of an automaton consists of a sequence of linked steps; formally, such an *execution fragment* of A is a finite sequence $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots \xrightarrow{a_n} s_n$ or an infinite sequence $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots$ of alternating states and actions of A such that $s_i \xrightarrow{a_i} s_{i+1}$ is a step of A for every i. An *execution* of A is just an execution fragment of A that begins in a start state of A. A state is *reachable* if it occurs in some execution. A property P of A is *invariant* if it holds in every reachable state of A.

IOA provide a notion of parallel composition. The central idea in this is highlighted by considering two different automata A_i and A_j in a set of automata to be composed. If A_i performs an output action a that is equal to an input action of A_j, then in the composed automaton, A_i produces a (and moves to a new state) and A_j consumes it, in the same moment, by moving along some transition it has labelled a. The action a remains as an output action of the composed automaton: hiding is provided as a separate operation.

We must define a notion of composition for action signatures before we can formally define composition for automata. A countable set $\{S_i\}_{i \in I}$ of action signatures is *strongly compatible* if for all i, j such that $i \neq j$

- $\mathit{outputs}(S_i) \cap \mathit{outputs}(S_j) = \{\}$,
- $\mathit{internals}(S_i) \cap \mathit{acts}(S_j) = \{\}$, and
- each action is contained only in a finite number of signatures.

If $\{S_i\}_{i \in I}$ is a countable collection of strongly compatible action signatures, its *composition* $\Delta_{i \in I} S_i$ is the action signature S with

$$\begin{aligned} \mathit{inputs}(S) &= (\bigcup_{i \in I} \mathit{inputs}(S_i)) - (\bigcup_{i \in I} \mathit{outputs}(S_i)) \\ \mathit{outputs}(S) &= \bigcup_{i \in I} \mathit{outputs}(S_i) \\ \mathit{internals}(S) &= \bigcup_{i \in I} \mathit{internals}(S_i) \end{aligned}$$

A set of automata with strongly compatible action signatures is also said to be *strongly compatible.* If $\{A_i\}_{i \in I}$ is a strongly compatible set of automata, then the composition $||_{i \in I} A_i$ is the following automaton:

$$\begin{aligned} \mathit{Asig} &= \Delta_{i \in I} \mathit{Asig}(A_i) \\ \mathit{St} &= \prod_{i \in I} \mathit{St}(A_i) \\ \mathit{St}_0 &= \prod_{i \in I} \mathit{St}_0(A_i) \\ \rightarrow &= \{<s, a, t> \mid \forall i \in I.\ \text{if } a \in \mathit{acts}(\mathit{Asig}(A_i)) \text{ then } <s_i, a, t_i> \text{ is a step of } A_i \\ &\qquad\qquad \text{else } s_i = t_i\} \end{aligned}$$

In the above definition, $\prod$ refers to the set-theoretic product of a family of sets, s and t refer to elements of the state product, and s_i and t_i refer to the ith projections of these elements. Typically, a finite composition $||_{i \in 1 \ldots n} A_i$ is written as $A_1 \mid\mid \ldots \mid\mid A_n$.

Hiding some actions W of an automaton is accomplished by an operation on its action signature S:

$$hide(S, W) \equiv <inputs(S) - W, outputs(S) - W, internals(S) \cup W>$$

Similarly, restricting the actions of an automaton is performed by the following operation on its action signature:

$$\begin{aligned} restrict(S, W) \equiv <&inputs(S) \cap W, outputs(S) \cap W, \\ &internals(S) \cup (externals(S) - W)> \end{aligned}$$

Note that *hide* and *restrict* may lead to complications if input actions are allowed to be hidden. However, since we do not use them in this fashion we do not encounter any problems.

Now we turn to notions of behaviour. If we strip out the subsequence of actions $a_i, a_{i+1}, \ldots$ of an execution fragment of A, we obtain a *schedule* of A. If we filter the schedule of an execution of A so that it has only external actions, we obtain a *behaviour* of A. The set of behaviours of A represents the externally observable activities that A can participate in and is defined

$$behaviours(A) \equiv \{beh.\ \exists ex \in executions(A).\ beh = behaviour(A, ex)\}.$$

For various cases (finite, infinite, fair) Lynch and Tuttle present definitions of the notions of *implementation* and *specification* that, in each case, amount to requiring that the behaviours of an implementation automaton be a subset of the behaviours of the specification. In the case important to our verification, the definition is as follows:

$$\begin{aligned} &implements(A, B) \equiv \\ &\quad inputs(Asig(A)) = inputs(Asig(B)) \wedge \\ &\quad outputs(Asig(A)) = outputs(Asig(B)) \wedge \\ &\quad behaviours(A) \subseteq behaviours(B) \end{aligned}$$

The formal presentation of IOA gives rise to a very large and complex theory: there are approximately 85 definitions in the base theory, before considering any bifurcations resulting from the consideration of fairness or from supporting the distinction between finite and infinite sequences.

2.1 I/O automata in Isabelle

Our formalization is in Isabelle's version of higher-order logic, a variant of Church's Simple Type theory supporting ML-style polymorphism and Haskell-style type classes[12]. The formalization was largely straightforward, since Isabelle's higher-order logic already supports a well-developed theory of sets. We will touch on only a few of the most important definitions.

One important aspect in this formalization is that our logic requires definitions to be total, i.e. without conditions on them, unlike many given by Lynch and Tuttle. This has the drawback that some definitions may not be exactly as in the presentation of Lynch and Tuttle, but has the advantage that the definitions are smaller and more general; conditions only appear in the statements of theorems that require them. For example, in the definition of composition for IOA, we do not require that the set of automata be countable. This condition is necessary for some meta-theoretic results, but not all: e.g. it is not necessary for the meta-theory we base our verification on.

Isabelle notation. Set comprehension has the shape $\{e.\ P\}$, where e is an expression and P a predicate. Tuples are written between angle brackets, e.g. $<s, a, t>$, and are nested pairs with projection functions *fst* and *snd*. If f is a function of type $\tau_1 \rightarrow \tau_2 \rightarrow \tau_3$, application is written $f(x, y)$ rather than $f\ x\ y$. Conditional expressions are written $if(A, B, C)$. The empty list is written $[]$, and "cons" is written infix: $h :: tl$. Function composition is another infix, e.g. $f \circ g$.

Action signatures. An action signature is described by the type

$$(\alpha)signature \equiv (\alpha)set \times (\alpha)set \times (\alpha)set.$$

The first, second and third components of an action signature S may be extracted with *inputs*, *outputs*, and *internals*. Furthermore, $actions(S)$ is defined to be $inputs(S) \cup outputs(S) \cup internals(S)$. Action signatures are defined as follows:

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

I/O automata. An IOA is a triple with type defined by

$$(\alpha, \sigma)ioa \equiv (\alpha)signature \times (\sigma)set \times (\sigma \times \alpha \times \sigma)set$$

and it is further required that the first member of the triple be an action signature, the second be a non-empty set of start states and the third be an input-enabled state transition relation:

$$\begin{aligned} & IOA(<asig, starts, trans>) \equiv \\ & \quad is\_asig(asig) \wedge starts \neq \{\} \wedge is\_state\_trans(asig, trans). \end{aligned}$$

Notice that we have employed the polymorphic type system of higher order logic in the representation of the actions and states of an IOA. Among other benefits, this enables the automatic type inference mechanism of Isabelle to detect some inconsistencies. The property of being an input-enabled state transition relation is defined as follows:

$$\begin{aligned} & is\_state\_trans(asig, R) \equiv \\ & \quad (\forall <s, a, t> \in R.\ a \in actions(asig)) \wedge \\ & \quad (\forall a \in inputs(asig). \forall s. \exists t.\ <s, a, t> \in R) \end{aligned}$$

The projections from an IOA are $asig\_of$, $starts\_of$, and $trans\_of$. The actions of an IOA are defined $acts \equiv actions \circ asig\_of$. We will use $\longrightarrow$ interchangeably with $trans\_of$.

Sequences. Lynch and Tuttle define the executions and behaviours of IOA in terms of underlying theories of finite and infinite sequences. We felt that developing two separate types of sequences and hence two parallel theories of IOA would result in a clumsy framework for verification; accordingly, we searched for a single representation that would accommodate both kinds of sequence, while still being able to make the necessary distinctions. Besides the finite/infinite distinction, sequences must support filtering by a predicate, indexing, and concatenation operations. Under our criteria, lists are not usable to model sequences since the theories of finite and infinite lists are different; the theory of *lazy lists* accomodates both finite and infinite lists, but does not support filtering.

We chose to model sequences with functions: an infinite sequence of elements, each of type α, is represented by a function of type $nat \longrightarrow \alpha$. However, the consideration of finite sequences forces us to address the issue of partiality: our chosen logic (HOL) provides only total functions, but a finite sequence, modelled with the built-in notion of function in the logic, must be a partial function.

There are three basic approaches that we know of for handling partial functions in a logic of total functions:

- Regard the function as a relation; members of the relation are points where the function is defined.
- Expand the range. Extending the range of the function by adding a special value that stands for "undefined" allows a functional style, at the cost of having to support the "undefined" value in some manner.
- Decline to say how the function behaves over part of its domain. This allows, for example, describing the predecessor function by: $\forall n.\, pred(n+1) = n$.

We chose the second approach. Sequences have type $nat \longrightarrow (\alpha)option$. The *option* datatype can be defined in Isabelle as $(\alpha)option = None \mid Some(\alpha)$ using an ML-like notation. A finite sequence in this representation ends with an infinite number of consecutive $None$s.

Execution fragments. Consider modelling a *step* of an execution as a triple $<s, a, s'>$. Then an execution fragment could be modelled as a sequence of such triples subject to the condition that for every i, $s'_i = s_{i+1}$. This condition makes for an awkward representation. More seriously, such a representation is not able to model an empty execution, which consists of only a single state.

We eventually decided to represent an execution fragment with a pair of sequences: an *action sequence* with type $nat \longrightarrow (action)option$ and an infinite *state sequence* having type $nat \longrightarrow state$. Using this representation, a step of fragment $ex = <A, S>$ is $<S_i, act, S_{i+1}>$ when $A_i = Some(act)$. This representation makes it easy to distinguish between infinite and finite executions (ex is finite iff the sequence A is); filtering ex reduces to filtering A; and the empty execution consisting only of state s is $<\lambda i.None, \lambda i.s>$. The Isabelle definition is

$$
\begin{aligned}
&is\_execution\_fragment(A, <act, state>) \equiv \\
&\ \forall n\ a.\ (act(n) = None \supset state(Suc(n)) = state(n)) \wedge \\
&\qquad (act(n) = Some(a) \supset\ <state(n), a, state(Suc(n))> \in \longrightarrow_A)
\end{aligned}
$$

Note that there is no requirement that $None$ be followed only by $None$: $None$s 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].

Reachability. Using an inductive definition, reachability could be defined by the following two rules:

$$
\begin{aligned}
s \in starts\_of(A) &\supset reachable(A, s) \\
reachable(A, s) \wedge <s, a, t> \in \longrightarrow_A &\supset reachable(A, t)
\end{aligned}
$$

However, we derive the two rules from a more direct definition:

$$reachable(A, s) \equiv \exists ex \in executions(A).\ \exists n.\ snd(ex)(n) = s.$$

Invariants in IOA are defined to be properties that hold in all reachable states:

$$invariant(A, P) \equiv \forall s.\ reachable(A, s) \supset P(s)$$

The following important theorem allows the proof of invariant properties by induction:

$$
\begin{aligned}
&(\forall s. s \in starts\_of(A) \supset P(s)) \wedge \\
&(\forall s\ t\ a. reachable(A, s) \wedge P(s) \wedge <s, a, t> \in \longrightarrow_A \supset P(t)) \\
&\supset\ invariant(A, P)
\end{aligned}
$$

Parallel composition. Many IOA definitions use indexed families. Higher order logic gives a neat approach to this, since an α-indexed family can be represented by the type $\alpha \rightarrow (\beta)set$ (or indeed, $\alpha \rightarrow \beta \rightarrow bool$, but we want to retain some concrete indication that we are using sets). In Isabelle, *general union* is defined $UNION(A, B) \equiv \{y.\ \exists x \in A.\ y \in B(x)\}$. A trivial specialization of A to the universal set of a type gives $UNION1$, a 'functional' version of general union.

$$
\begin{aligned}
&asig\_composition(asigs) \equiv \\
&\ <UNION1(inputs \circ asigs) - UNION1(outputs \circ asigs), \\
&\quad UNION1(outputs \circ asigs), \\
&\quad UNION1(internals \circ asigs)>
\end{aligned}
$$

The direct product of a family of sets is also quite simple to define: $DP(F) \equiv \{tuple\ .\ \forall i.\ tuple(i) \in F(i)\}$. We therefore have the following definition of the parallel composition of an indexed family of automata:

$$
\begin{array}{l}
ioa\_composition(ioas) \equiv \\
\quad < asig\_composition(asig\_of \circ ioas), \\
\quad\quad DP(starts\_of \circ ioas), \\
\quad\quad \{<s, act, t>\ .\ \forall i.\ if(act \in acts(ioas(i)), \\
\quad\quad\quad\quad\quad\quad <s(i), act, t(i)> \in trans\_of(ioas(i)), \\
\quad\quad\quad\quad\quad\quad s(i) = t(i))\} >
\end{array}
$$

This definition of composition is appealingly compact, but hard to use in verification. The following definition is tailored for the binary case and was used in the example verification.

$$
\begin{array}{l}
A \parallel B \equiv \\
\quad < asig\_comp(asig\_of(A), asig\_of(B)), \\
\quad\quad \{<u, v>\ .\ u \in starts\_of(A) \wedge v \in starts\_of(B)\}, \\
\quad\quad \{<s, act, t>\ .\ (act \in acts(A) \vee act \in acts(B)) \wedge \\
\quad\quad\quad if(act \in acts(A), <fst(s), act, fst(t)> \in \longrightarrow_A, fst(s) = fst(t)) \wedge \\
\quad\quad\quad if(act \in acts(B), <snd(s), act, snd(t)> \in \longrightarrow_B, snd(s) = snd(t))\}>
\end{array}
$$

where a specialization of action signature composition is needed:

$$
\begin{array}{l}
asig\_comp(A_1, A_2) \equiv \\
\quad <(inputs(A_1) \cup inputs(A_2)) - (outputs(A_1) \cup outputs(A_2)), \\
\quad\quad outputs(A_1) \cup outputs(A_2), internals(A_1) \cup internals(A_2)>
\end{array}
$$

Notice that this binary composition operator gives more flexibility in the states than the infinite case. Letting α stand for the type of actions, and σ and τ for types of states, binary composition has type

$$(\alpha, \sigma)ioa \rightarrow (\alpha, \tau)ioa \rightarrow (\alpha, \sigma \times \tau)ioa$$

whereas the infinite version has

$$(\beta \rightarrow (\alpha, \sigma)ioa) \rightarrow (\alpha, \beta \rightarrow \sigma)ioa$$

and thus requires that every member of the indexed set have the same type, i.e. all automata are defined over the same state space and action set. This is fine for replicating the same IOA many times but awkward when composing a system from many different components.

Note that in our formalization both of binary and arbitrary parallel composition, each process to be composed must already be defined over all actions in the composition, even those not in its own action signature. This is feasible for a system built from a fixed number of processes but precludes the reuse of generic components. The latter can be achieved with an operation for renaming actions:

$$rename : (\alpha, \sigma)ioa \rightarrow (\beta \rightarrow (\alpha)option) \rightarrow (\beta, \sigma)ioa$$

The details are contained in a forthcoming report; *rename* is not used in the sequel.

2.2 Meta-theory

Possibility maps [9] give a means of doing refinement proofs of IOA systems. The set of possibility maps we make use of is described by the following predicate, which takes a function f (from *concrete* states to *abstract* states), a concrete automaton C, and an abstract automaton A.

$$\begin{array}{l} is\_weak\_pmap(f, C, A) \equiv \\ \quad (\forall s_0 \in starts\_of(C).\ f(s_0) \in starts\_of(A)) \wedge \\ \quad (\forall s\ t\ a.\ reachable(C, s) \wedge <s, a, t> \in trans\_of(C) \\ \quad \supset if(a \in externals(asig\_of(C)), <f(s), a, f(t)> \in trans\_of(A), f(s) = f(t))) \end{array}$$

The following theorem states that the existence of a possibility map between C and A implies that the behaviours of C do not exceed those prescribed by A.

$$\begin{array}{l} IOA(C) \wedge IOA(A) \wedge \\ externals(asig\_of(C)) = externals(asig\_of(A)) \wedge \\ is\_weak\_pmap(f, C, A) \\ \supset behaviours(C) \subseteq behaviours(A) \end{array}$$

The proof proceeds by using f to give an execution of A in terms of an execution of C. After that, the behaviours are shown to coincide, and the proof finishes by demonstrating that the abstract execution is in fact an execution.

Note that the above theorem merely requires the external actions of both automata to be the same but does not preclude that some actions are inputs of one automaton and outputs of the other. In such unusual cases, however, it will not be possible to prove that C implements A because the definition of *implements* requires that the inputs and outputs are identical, not just the externals.

3 An example verification

In the sequel we specify and verify a communication protocol originally due to Attiya et al. [3]. This protocol achieves FIFO-communication over channels which may lose and reorder messages, but may not duplicate them. Let us call such channels *non-duplicating* (*ND*). The remarkable fact is that the protocol requires only a single header-bit. Starting from [3], Nipkow verified a slightly optimized version of this protocol some 4 years ago, using Isabelle merely as a theorem prover for the hand-generated verification conditions. To simplify matters he also assumed that the underlying channels do not lose messages. In the following we recast Nipkow's specification and verification in the IOA formalization presented above.

In the mean time the above protocol has been simplified considerably [2] using the following divide-and-conquer approach:

1. FIFO-communication can be implemented on top of *order-preserving* (*OP* = duplication and loss, but no reordering) channels using standard protocols like the Alternating Bit.

2. OP-communication can be implemented on top of ND-channels using a cut down version of the protocol in [3].

Putting the two protocols on top of each other, we obtain FIFO-communication using ND-channels. In retrospect, the original protocol can be seen as an explicit merge of the two layers.

Despite this recent simplification, we have stuck to the original protocol for two reasons: we had an Isabelle reference version already, which would allow interesting comparisons, and because the original protocol, being more complex, would be more of a challenge for Isabelle's theorem proving technology.

In the protocol, the Sender and the Receiver progress through a series of *rounds*. The key idea is that processes count and compare the number of packets sent and received. When to switch to a new round, i.e. when a message has been sent, is determined by this relationship. The precise logic is too intricate to summarize concisely.

3.1 Concise descriptions of transition relations

We need some notation to describe the transition relations of the processes. In our implementation, a transition relation is given by a set comprehension having the following shape:

$$\begin{array}{ll} \{<s, act, s'> \ . & \texttt{case}\ act\ \texttt{of} \\ & a_1 \Rightarrow R_1(s, s') \\ & \quad \vdots \\ & a_n \Rightarrow R_n(s, s')\} \end{array}$$

where each R_i gives the relationship between s and s' when a_i occurs. This style of representation is clear and easy to work with. Unfortunately, it entails much redundancy: if part or all of the state does not change, that must explicitly be expressed. Therefore, in the interests of readability, we will use the precondition-effect format of Lynch and Tuttle to describe a transition relation:

action	(input | output | internal)
Precondition:	C
Effect:	$p_1 := v_1, \ldots, p_n := v_n$.

Predicate C is the constraint on the state s that must hold for the transition to apply. The p_i are the state components and the v_i are the new values taken on in the transition. We omit mention of all parts of the state that are not changed in the transition, all empty transitions, and all vacuous preconditions. A formal translation from this style to the set comprehension representation is straightforward, but unnecessary for the purposes of reading this paper. Although it would be desirable to have a tool to perform such translations automatically, we consider this a user-interface question which should be separated from the proof tool.

3.2 The Specification

The intended FIFO-behaviour of the system is given by defining an IOA named *Spec*. The state of Spec is a message queue q, initially empty, modelled with the type $(\mu)list$, where the parameter μ represents the message type. The only actions performed in the abstract system are: $S\_msg(m)$, putting message m at the end of q, and $R\_msg(m)$, taking message m from the head of q. Formally:

$S\_msg(m)$	input	$R\_msg(m)$	output
Effect:	$q := q@[m]$	Precondition:	$q = m :: rst$
		Effect:	$q := rst$

3.3 The Implementation

The system being proved correct is a parallel composition of 4 processes:

$$Impl \equiv Sender \parallel Schannel \parallel Receiver \parallel Rchannel$$

a sender, a receiver, and proprietary channels for both. The "dataflow" in the system is depicted in Fig. 1

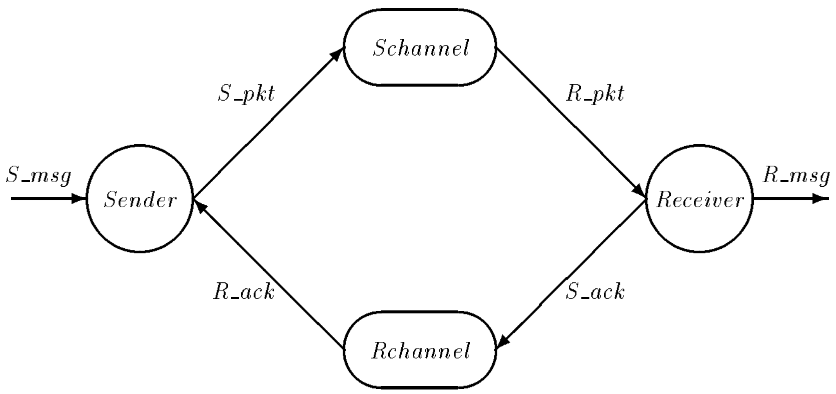

Fig. 1. The Implementation

The objects going over the medium from Sender to Receiver are modelled with the type $(\mu)packet \equiv bool \times \mu$. This expresses that messages (represented by μ) are sent with a single header bit. The header bit of a packet is extracted with hdr and the message field is extracted with msg.

The type of system actions, $(\mu)action$, is described in Isabelle by the following ML-style datatype:

```
'm action =
   S_msg ('m)        (* Sender inputs msg               *)
 | R_msg ('m)        (* Receiver outputs msg            *)
 | S_pkt ('m packet) (* Sender sends packet             *)
 | R_pkt ('m packet) (* Receiver receives packet        *)
 | S_ack (bool)      (* Receiver sends acknowledgment   *)
 | R_ack (bool)      (* Sender receives acknowledgment  *)
 | C_m_s             (* Change mode in Sender           *)
 | C_m_r             (* Change mode in Receiver         *)
 | C_r_s             (* Change round in Sender          *)
 | C_r_r ('m)        (* Change round in Receiver        *)
```

The current datatype definition facility in Isabelle/HOL directly asserts the necessary freeness and induction axioms. It is planned to port Paulson's datatype package [11], which *proves* these axioms from a construction of the datatype, from ZF to HOL. This would not change the interface but would guarantee the consistency of the system.

It may come as a surprise that we introduce all the actions in one go and not on a 'per automaton' basis. This is a consequence of the type system which enforces that each component automaton A of a composition is defined over the same *type* of actions. Of course the transition of each A contains only triples $<s, a, t>$ where $a \in acts(A)$. At the end of Section 2.1 we indicated how *rename* could be used to achieve a modular style of composition.

There is a related problem to do with the implementation relationship which requires that both automata (the specification and the implementation) are defined over the same type of actions; otherwise their traces are incomparable. Of course the specification's action signature (which is a *set*) is independent of the implementation, but the *type* of actions is not. Depending on the exact syntax used in the definition of the transition relation, this may even require additional 'no-op'-clauses when new internal actions are added to the implementation. To get around this problem requires a notion of "implementation modulo a renaming" where the renaming relates the action types of both automata.

If all this explicit renaming appears very cumbersome, the alternative should be considered: a single global type of actions. Not only would this approach rob us of the benefits of type checking, but it would also pose the problem of how to associate individual application-specific actions, e.g. `S_msg`, with elements of the global type of actions. It is even less clear how to model parameterized actions in such an approach.

3.4 Multisets

The verification uses a theory of finite multisets. The axiomatization provides operations to add and delete elements from the multiset, as well as to count the number of members of the multiset having a particular property. The empty multiset is written $\emptyset$. The number of members of multiset M with property P is written $|M;P|$.

$$\begin{aligned} delm(\emptyset, x) &\equiv \emptyset \\ delm(addm(M, x), y) &\equiv if(x = y, M, addm(delm(M, y), x)) \\ |\emptyset; P| &\equiv 0 \\ |addm(M, x); P| &\equiv |M;P| \ + \ if(P(x), 1, 0) \end{aligned}$$

By abuse of notation, we will sometimes write $|M;x|$ to stand for $|M;\lambda y.y = x|$, the number of occurrences of items equal to x in M. There is also an induction theorem for the finite multisets:

$$P(\emptyset) \supset (\forall M\ x.\ P(M) \supset P(addm(M, x))) \supset P(N)$$

We derived a collection of theorems about multisets; among the more useful in our verification is the following, which enables the reduction of cardinality relationships to logical relationships:

$$(\forall x.\ P(x) \supset Q(x)) \supset |M;P| \leq |M;Q|$$

3.5 The Sender

The state of the process *Sender* is a 5-tuple:

Field	Type	Initial Value
messages:	$(\mu)list$	[]
sent:	$(bool)multiset$	$\emptyset$
received:	$(bool)multiset$	$\emptyset$
header:	*bool*	*false*
sending:	*bool*	*true*

The Sender makes the following transitions:

S_msg(m)	input
Effect:	$messages := messages@[m]$
S_pkt(pkt)	output
Precondition:	$hdr(pkt) = header \wedge msg(pkt) = hd(messages)$ $\wedge\ sending$
Effect:	$sent := addm(sent, hdr(pkt))$
R_ack(b)	input
Effect:	$received := addm(received, b)$
C_m_s	internal
Precondition:	$\vert sent; \neg header\vert < \vert received; \neg header\vert \ \wedge\ sending$
Effect:	$sending := False$
C_r_s	internal
Precondition:	$\vert sent; header\vert \leq \vert received; \neg header\vert \ \wedge\ \neg sending$
Effect:	$header := \neg header,\ sending := True,$ $messages := tl(messages)$

3.6 The Receiver

The state of the process *Receiver* is also a 5-tuple:

Field	Type	Initial Value
$messages$:	$(\mu)list$	$[\,]$
$replies$:	$(bool)multiset$	$\emptyset$
$received$:	$((\mu)packet)multiset$	$\emptyset$
$header$:	$bool$	$false$
$replying$:	$bool$	$false$

The Receiver makes the following transitions:

$R\_msg(m)$	output
Precondition:	$messages = m :: rst$
Effect:	$messages := rst$
$R\_pkt(pkt)$	input
Effect:	$received := addm(received, pkt)$
$S\_ack(b)$	output
Precondition:	$b = header \ \wedge \ replying$
Effect:	$replies := addm(replies, b)$
C_m_r	internal
Precondition:	$\lvert replies; \neg header\rvert < \lvert received; \lambda y.hdr(y) = header\rvert$ $\wedge \ replying$
Effect:	$replying := False$
$C\_r\_r(m)$	internal
Precondition:	$\lvert replies; header\rvert \leq \lvert received; \lambda y.hdr(y) = header\rvert \ \wedge$ $\lvert replies; \neg header\rvert < \lvert received; <header, m>\rvert \ \wedge \ \neg replying$
Effect:	$header := \neg header, \ replying := True,$ $messages := messages@[m]$

3.7 The Channels

The Sender and Receiver each have a proprietary channel, named *Schannel* and *Rchannel* respectively. The messages sent by the Sender and Receiver are never lost, but the channels may mix them up. Accordingly, multisets are used in modelling the channel states: the state of Schannel is of type $((\mu)packet)multiset$, the state of Rchannel is of type $(bool)multiset$, expressing that replies from the Receiver are just one bit. Schannel makes the following transitions:

$S\_pkt(pkt)$	input	$R\_pkt(pkt)$	output
Effect:	$M := addm(M, pkt)$	Precondition:	$\lvert M; pkt\rvert \neq 0$
		Effect:	$M := delm(M, pkt)$

Similarly, Rchannel makes the following transitions:

$S\_ack(b)$	input	$R\_ack(b)$	output
Effect:	$M := addm(M, b)$	Precondition:	$\lvert M; b\rvert \neq 0$
		Effect:	$M := delm(M, b)$

The initial state of both channels is an empty multiset.

4 The Verification

An assortment of theories are used to structure the verification. The theory structure can be seen in Fig. 2. Theories with a '*' are already installed in the Isabelle system.

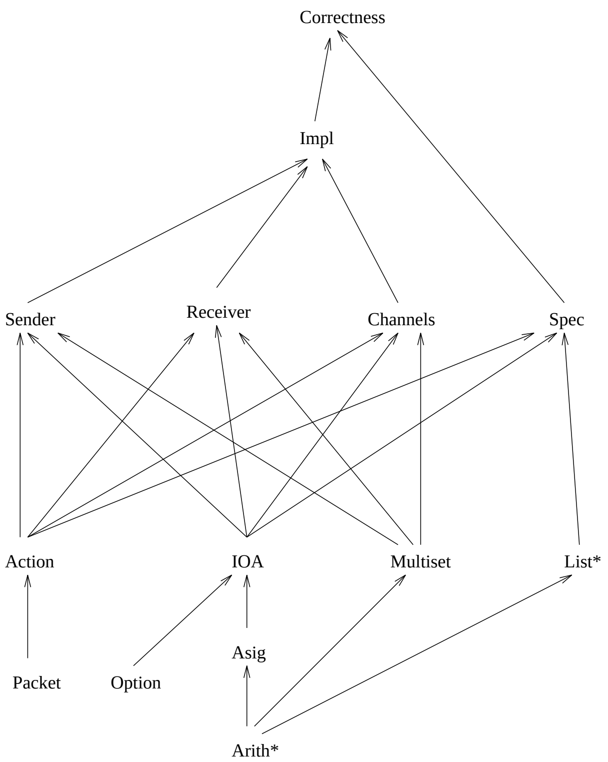

Fig. 2. Theory Structure

In the theory *Correctness* we define a mapping *hom* from the implementation state to the specification state:

$$hom(s) \equiv R.messages@if(R.header = S.header, S.messages, ttl(S.messages))$$

In this, *ttl* is a function defined by the equations $ttl([\,]) = [\,]$ and $ttl(h :: rst) = rst$. To distinguish between components of the Receiver state and the Sender state that have the same field names, e.g. *header*, we use a 'dotted identifier' notation, e.g. *S.header* and *R.header*. The statement of correctness says that *hom* is a possibility map from the implementation IOA to the specification IOA (under the restriction of the action signature of the implementation to the external action signature of the specification).

$$is\_weak\_pmap(hom, restrict(Impl, externals(Spec\_sig)), Spec)$$

The verification depends on several system invariants (lemmas 5.1 to 5.4 in [3]) that relate the states of the 4 processes in all reachable states of the system. Although we attempt to give loose English paraphrases of these invariants, they can be difficult to make sense of.

1. No packets from the Receiver to the Sender are dropped by Rchannel. The analogous statement for Schannel is also true. We employ the abbreviatory function $hdr\_sum(M, b) \equiv |M; \lambda pkt.\, hdr(pkt) = b|$ to improve readability.

$$\begin{array}{l}(\forall b.\, |R.replies; b| = |S.received; b| + |Rchannel; b|) \wedge \\ (\forall b.\, |S.sent; b| = hdr\_sum(R.received, b) + hdr\_sum(Schannel, b))\end{array}$$

2. This invariant expresses a complicated relationship about how many messages are sent and the relative status of header bits in the Sender and Receiver.

$$\begin{array}{l}R.header = S.header \wedge S.sending \wedge \\ |R.replies; \neg S.header| \leq |S.sent; \neg S.header| \leq |R.replies; S.header| \\ \vee \\ R.header = \neg S.header \wedge R.replying \wedge \\ |S.sent; \neg S.header| \leq |R.replies; S.header| \leq |S.sent; S.header|\end{array}$$

3. The number of received messages in the Receiver plus the number of those messages in transit (in Schannel) is not greater than the number of replies, provided the message isn't current and the header bits agree.

$$\begin{array}{l}R.header = S.header \supset \\ \forall m.\, (S.messages = [\,] \vee m \neq hd(S.messages)) \supset \\ \quad |R.received; < S.header, m > | + |Schannel; < S.header, m > | \\ \quad \leq |R.replies; \neg S.header|\end{array}$$

4. If the headers are opposite, then the Sender queue has a message in it.

$$R.header = \neg S.header \supset S.messages \neq [\,]$$

4.1 Proofs

The proofs of the invariants and the final theorem followed a common pattern. Each proof proceeded by induction on the structure of the reachability relation, followed by case analysis on the type of actions. This gave ten cases. Most of the vacuous cases (those arising from transitions not changing any states mentioned in the invariant) were cleared off automatically by the conditional and contextual rewriting of Isabelle. In the remaining cases, one or two precise applications of simple arithmetic inequalities in the hypotheses were necessary before another application of rewriting or Isabelle's automatic reasoning tactics would finish off the proof. Usually, the reasoning steps in Isabelle were larger than in [3], but once in a while, usually when doing intricate reasoning among the hypotheses, quite low level steps had to be made, sometimes requiring explicit instantiations of hypotheses to be given.

As a measure of complexity, we counted the number of tactic applications. In the development of the meta-theory, there are about 80 proof steps. In the example, there are a little over 400 proof steps. Building the meta-theory and verifying the system takes approximately twenty minutes on a Sparc2 with 64 Megabytes of memory.

A problem we encountered, echoed in accounts of other large verifications, was identifying what to do next when confronted by a large goal with many hypotheses. Of course, one answer to this is 'keep your goals small', but then one has the problem of expending too much effort on proof management. Furthermore, we were often in a situation where there were lots of hypotheses that *needed* to be used, and isolating (first visually, then computationally) the small number that were immediately relevant to the goal at hand took a lot of effort.

A final remark is that automated support for arithmetic reasoning would have helped.

4.2 The use of meta-theory

What are the advantages and disadvantages of embedding the theory of a formalism, IOA in this case, in a mechanized logic? The advantages are several:

- Such an embedding is formal and hence non-ambiguous. Thus it can serve as a *standard* presentation.
- Such an embedding is relatively consistent, so worries about consistency of an embedded calculus can be ameliorated.
- Meta-theorems, such as the possibility map theorem, can make verifications easier since correctness can be reduced to establishing some simpler fact, e.g. that *hom* is a possibility map. An alternative is to have an *external* agent operating outside the logic that would take the system correctness statement and compute the necessary 'verification conditions'. The verification could not be considered to be totally formal in that case, since the verification conditions, when proven, could not be joined back together *using inference* to produce the theorem stating system correctness. Hence the correctness statement could not be used in further verifications without making a break in the chain of formal proofs.
- A consequence of having meta-theorems is that correctness statements are fairly simple to set up. In contrast, a person attempting a verification in an environment without either meta-theorems or a verification condition generator would in general need to have a good background in mathematics and logic.

Some disadvantages are

- The representation of objects being verified can be large.

– Work needs to be done to logically derive the verification conditions. In effect, the meta-theory needs to be *hidden* so that it doesn't clutter up the verification. Sometimes this work is not trivial.

5 Conclusions and Future Work

We have formalized the theory of IOA in Isabelle's higher order logic. Lynch and Tuttle give a quite abstract set-theoretical presentation of IOA; the Isabelle implementation of higher order logic met this challenge easily. We found that Isabelle's well-developed theory of sets was nicely complemented by the equivalence of sets and types in higher-order logic. For example, we chose to represent states and actions with type variables in the development of the meta-theory. This use of the type system highlights essential dependencies: one must choose actions and states for automata before describing their transitions. Also, having functions and predicates as first-class objects allowed a smooth treatment of such notions as invariants, infinitary composition, traces, and multisets.

In the theory of IOA, we have specified and verified a recent protocol. In this effort, we found that hiding the meta-theory was necessary; otherwise the formulae confronting the user were simply too complex. We also found that describing the transition relations of individual IOA had much redundance; therefore, it would be useful to support 'shorthand' descriptions of automata in the style of this paper. Another issue is that of *action management* in Lynch and Tuttle's theory. Both composition and implementation currently require all automata involved to be defined over the same type of actions. This has grave consequences for modularity; therefore a facility for explicit renaming is required.

Acknowledgments. We have benefitted from Manfred Broy's criticism, from many conversations with Bernhard Schätz and from the critical and perceptive comments by Frits Vaandrager.

References

1. Martin Abadi and Leslie Lamport. The existence of refinement mappings. In *Proc. 3rd IEEE Symp. Logic in Computer Science*, pages 165–177. IEEE Computer Society Press, 1988.
2. Yehuda Afek, Hagit Attiya, Alan Fekete, Michael Fischer, Nancy Lynch, Yishay Mansour, Da-Wei Wang, and Lenore Zuck. Reliable communication over unreliable channels. *Journal of the ACM.* To appear.
3. Hagit Attiya, Alan Fekete, Michael Fischer, Nancy Lynch, Yishay Mansour, Da-Wei Wang, and Lenore Zuck. Reliable communication over unreliable channels. Draft version, 1990.
4. J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J.Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proc. 5th IEEE Symp. Logic in Computer Science*, pages 428–439, 1990.

5. G. Dowek, A. Felty, H. Herbelin, G. Huet, C. Murthy, C. Parent, C. Paulin-Mohring, and B. Werner. The Coq proof assistant user's guide version 5.8. Technical Report 154, INRIA, May 1993.
6. Leen Helmink, Alex Sellink, and Frits Vaandrager. Proof-checking a data link protocol. In Henk Barendregt and Tobias Nipkow, editors, *Types for Proofs and Programs*, volume 806 of *Lect. Notes in Comp. Sci.*, pages 127–165. Springer-Verlag, 1994.
7. Victor Luchangco, Ekrem Söylemez, Stephen Garland, and Nancy Lynch. Verifying timing properties of concurrent algorithms. In *FORTE'94: Seventh International Conference on Formal Description Techniques for Distributed Systems and Communciations Protocols*, 1994. Submitted for publication.
8. Nancy Lynch, Michael Merritt, William Weihl, and Alan Fekete. *Atomic Transactions*. Morgan Kaufmann Publishers, 1994.
9. Nancy Lynch and Mark Tuttle. An introduction to Input/Output automata. *CWI Quarterly*, 2(3):219–246, 1989.
10. Tobias Nipkow. Formal verification of data type refinement — theory and practice. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems*, volume 430 of *Lect. Notes in Comp. Sci.*, pages 561–591. Springer-Verlag, 1990.
11. Lawrence C. Paulson. A fixedpoint approach to implementing (co)inductive definitions. In Alan Bundy, editor, *Proc. 12th Int. Conf. Automated Deduction*, volume 814 of *Lect. Notes in Comp. Sci.*, pages 148–161. Springer-Verlag, 1994.
12. Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 1994.
13. Jørgen Søgaard-Andersen, Stephen Garland, John Guttag, Nancy Lynch, and Anya Pogosyants. Computer-assisted simulation proofs. In *Fourth Conference on Computer-Aided Verification*, volume 697 of *Lect. Notes in Comp. Sci.*, pages 305–319. Springer-Verlag, 1993.

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