# Combining Model Checking and Deduction for I/O-Automata

Olaf Müller and Tobias Nipkow*
TU München†

**Abstract**

We propose a combination of model checking and interactive theorem proving where the theorem prover is used to represent finite and infinite state systems, reason about them compositionally and reduce them to small finite systems by verified abstractions. As an example we verify a version of the Alternating Bit Protocol with unbounded lossy and duplicating channels: the channels are abstracted by interactive proof and the resulting finite state system is model checked.

## 1 Introduction

The purpose of this paper is to combine the two major paradigms for the verification of distributed systems: *model checking* and *theorem proving*. The advantages of each approach are well known: model checking is automatic but limited to finite state processes, theorem proving requires user interaction but can deal with arbitrary processes. Recently attempts have been made to combine the strength of both methods by using the deductive machinery of theorem provers to reduce "large" correctness problems to ones which are small enough for model checking. The key idea is *abstraction* whereby the state space is partitioned to obtain a smaller automaton which is amenable to model checking. Of course the abstraction has to be sound w.r.t. the property we want to check: if the abstracted automaton satisfies the property so should the original automaton.

In our approach the theorem prover provides a common representation language and tools for

- both finite and infinite state systems,
- checking the soundness of abstractions,
- reasoning about systems in a compositional manner.

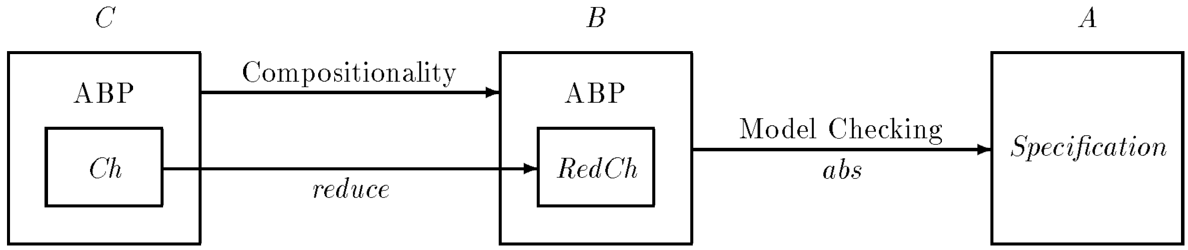
Our work is based on Lynch and Tuttle's *Input/Output-Automata* (*IOA*) [14] as model of distributed processes which have been embedded in the theorem prover Isabelle/HOL [15]. We are interested in verifying safety properties of IOA. These safety properties are not expressed by temporal logic formulae but again by IOA. Hence we need to check that the traces of one IOA $C$ (the implementation) are included in the traces of another IOA $A$ (the specification). Assuming that $C$ is infinite or at least too large to check $traces(C) \subseteq traces(A)$ automatically, we define an intermediate automaton $B$ which is an abstraction of $C$ and should satisfy $traces(C) \subseteq traces(B) \subseteq traces(A)$. Thus we have achieved the following division of labor: $traces(C) \subseteq traces(B)$, i.e. the soundness of the abstraction, is proved interactively in Isabelle; $traces(B) \subseteq$

*Research supported by ESPRIT BRA 6453, *Types*.
†Address: Institut für Informatik, Technische Universität München, 80290 München, Germany. Email: {Mueller,Nipkow}@Informatik.TU-Muenchen.De

*traces*$(A)$ is verified automatically by a model checker; transitivity of $\subseteq$ yields the desired *traces*$(C) \subseteq$ *traces*$(A)$.

The main distinguishing feature of our approach is the ability to reason about the soundness of arbitrary abstractions because we have the meta-theory of IOA at our disposal. Assuming that the theorem prover and the formalization of IOA in it are correct, the only remaining source of errors is the model checker which is treated like an oracle by the theorem prover. Note that this includes the interface between model checker and theorem prover, which is particularly critical because we need to ensure that the theorem prover formalizes exactly the logic the model checker is based on.

The rest of the paper illustrates this approach using a particular example, namely an implementation of the Alternating Bit Protocol using unbounded channels. This is in contrast to pure model checking approaches where the channels are always of a fixed capacity (usually 1). The key to the success of our approach is the fact that channels may lose and duplicate, but not reorder messages. Thus is is possible to compactify channels without altering their behaviour by collapsing all adjacent identical messages. This is what our abstraction from $C$ to $B$ does. The full picture looks like this:




The implementation $C$ contains unbounded channels *Ch* which are abstracted/compactified by a function *reduce*. It is shown interactively that *reduce* is indeed an abstraction function, i.e. *traces*$(Ch) \subseteq$ *traces*$(RedCh)$. $B$ is the same as $C$ except that collapsing channels are used. Compositionality proves that $C$ must be an implementation of $B$, i.e. *traces*$(C) \subseteq$ *traces*$(B)$. Although *RedCh* is not a finite state system, it behaves like one if used in the context of the ABP because at any one time there are at most two different messages on each channel. Thus $B$ is a finite state system. Note however, that we never need to prove this explicitly. This is merely an intuition which is later confirmed by the model checker which is given a description of $B$ and $A$ together with an abstraction function *abs* between them. The model checker explores the full state space of $B$ verifying transition by transition that *abs* is indeed an abstraction. It is only the successful termination of the model checker which tells us that $B$ must be finite.

## 1.1 Related work

Our paper is closely related to the work by Hungar [11] who embeds a subset of OCCAM in the theorem prover LAMBDA and combines it with an external model checker. The key difference is that Hungar relies much more on unformalized meta-theory than we do: he axiomatizes OCCAM's proof rules instead of deriving them from a semantics, and does not verify the soundness of his data abstractions.

The literature on abstraction for model checking is already quite extensive (see, for example, [4, 8, 5]). The general idea is to compute an abstract program given a concrete one together with an abstraction function/relation. The approach of Clarke et al. is in principle also applicable to infinite concrete systems. However, since they compute an approximation to the real abstract program, the result is not necessarily finite state. Nevertheless it would be interesting to rephrase their ideas in terms of IOA and apply them to our example. In this case we would not give $B$ explicitly but would compute (via the rewriting machinery of the theorem prover) a (hopefully finite state) approximation of it.

Our work differs from most approaches to model checking because we do not check if an automaton satisfies a temporal logic formula but if its traces are included in those of another automaton. Although theoretically equivalent, automata can be compared by providing an explicit abstraction function (or simulation relation), *abs* above. The same approach is followed in [12] where abstraction functions are also used for reduction, and in [9] where liveness is taken into account. If the documentation aspect of an explicit abstraction function is not considered important, one could also use a model checker which searches for an abstraction function using, for example, the techniques of [6], although this is bound to be less efficient.

Finally there is the result by Abdulla and Jonsson [1] that certain properties of finite state systems communicating via unbounded lossy channels are decidable, which they apply to the Alternating Bit Protocol. However, in our work the channels can both lose and duplicate messages. Hence their result does not apply directly.

# 2 I/O-Automata in Isabelle/HOL

**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 $\tau_1 \to \tau_2 \to \tau_3$, application is written $f(x, y)$ rather than $f\ x\ y$. Conditional expressions are written $\mathit{if}(A, B, C)$. The empty list is written $[\,]$, and "cons" is written infix: $h :: \mathit{tl}$. Function composition is another infix, e.g. $f \circ g$.

## 2.1 I/O Automata

An IOA is a finite or infinite state automaton with labelled transitions. I/O automata, initially introduced by Lynch and Tuttle [14], are still under development, and the formalization we used represents only a fragment of the theory one can find in recent papers [7]. For example, we do not take care of fairness or time constraints. The details of the formalization can be found in a previous paper [15], so that we give only a brief sketch of the essential definitions inside Isabelle/HOL.

An action signature is described by the type

$$(\alpha)\mathit{signature} \equiv (\alpha)\mathit{set} \times (\alpha)\mathit{set} \times (\alpha)\mathit{set}.$$

The first, second and third components of an action signature $S$ may be extracted with *inputs*, *outputs*, and *internals*. Furthermore, $\mathit{actions}(S) = \mathit{inputs}(S) \cup \mathit{outputs}(S) \cup \mathit{internals}(S)$, and $\mathit{externals}(S) = \mathit{inputs}(S) \cup \mathit{outputs}(S)$. Action signatures have to satisfy the following condition:

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

An IOA is a triple with type defined by

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

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:

$$\begin{aligned}
&\mathit{IOA}(<\mathit{asig}, \mathit{starts}, \mathit{trans}>) \equiv \\
&\quad \mathit{is_asig}(\mathit{asig}) \wedge \mathit{starts} \neq \{\} \wedge \mathit{is_state_trans}(\mathit{asig}, \mathit{trans}).
\end{aligned}$$

The property of being an input-enabled state transition relation is defined as follows:

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

The projections from an IOA are *asig_of*, *starts_of*, and *trans_of*. The actions of an IOA are defined $acts \equiv actions \circ asig\_of$.

An *execution-fragment* of an IOA $A$ is a finite or infinite sequence that consists of alternating states and actions. In Isabelle it is represented as a pair of sequences: an infinite *state sequence* of type $nat \to state$ and an *action sequence* of type $nat \to (action)option$. Here the *option* datatype is defined 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. Using this representation, a step of an execution-fragment ${<}as, ss{>}$ is ${<}ss(i), a, ss(i+1){>}$ if $as(i) = Some(a)$. Formally:

$$\begin{array}{l} is\_execution\_fragment(A, {<}as, ss{>}) \equiv \\ \quad \forall n\ a.\ \ (as(n) = None \supset ss(Suc(n)) = ss(n)) \wedge \\ \quad \phantom{\forall n\ a.\ \ } (as(n) = Some(a) \supset {<}ss(n), a, ss(Suc(n)){>} \in trans\_of(A)) \end{array}$$

An *execution* of $A$ is an execution-fragment of $A$ that begins in a start state of $A$. 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* replaces $Some(a)$ by $None$ if $a$ is not an external action.

## 2.2 Composition and Refinement

I/O automata provide a notion of parallel composition. In Isabelle this mechanism is realized by a binary operator $\|$. The definition simply reflects the fact that each component performs its locally defined transitions if the relevant action is part of its actions signature, otherwise it remains unchanged.

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

where an action signature composition is needed:

$$\begin{array}{l} asig\_comp(S_1, S_2) \equiv \\ \quad {<}(inputs(S_1) \cup inputs(S_2)) - (outputs(S_1) \cup outputs(S_2)), \\ \quad\ outputs(S_1) \cup outputs(S_2), internals(S_1) \cup internals(S_2){>} \end{array}$$

Action signature composition presumes compatibility of actions, which is defined by

$$\begin{array}{l} compatible(S_1, S_2) \equiv \\ \quad (outputs(S_1) \cap outputs(S_2) = \{\}) \wedge \\ \quad (outputs(S_1) \cap internals(S_2) = \{\}) \wedge \\ \quad (outputs(S_2) \cap internals(S_1) = \{\}) \end{array}$$

and is trivially extended to compatibility of automata.

For the aim of refinement, we make use of abstraction functions which Lynch and Tuttle call "weak possibility mappings". The set of these maps 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 $A$.

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

The following theorem proved in Isabelle states that the existence of an abstraction function from $C$ to $A$ implies that the traces of $C$ are contained in those of $A$.

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

### 2.3 Renaming

As in [13] we define an operation for renaming actions. The motivation for this is modularity: name clashes can be avoided and generic components can be plugged into different environments.

$$\mathit{rename} : (\alpha, \sigma)\mathit{ioa} \to (\beta \to (\alpha)\mathit{option}) \to (\beta, \sigma)\mathit{ioa}$$

In contrast to [13] we define the action renaming function with type $\beta \to (\alpha)\mathit{option}$ instead of $\alpha \to \beta$. Therefore it does not have to be injective, which facilitates reasoning about such functions.

$$\begin{array}{l} \mathit{rename}(A, f) \equiv \\ \quad \lt\lt \{\mathit{act}\ .\ \exists \mathit{act}'.\ f(\mathit{act}) = \mathit{Some}(\mathit{act}') \wedge \mathit{act}' \in \mathit{inputs}(\mathit{asig\_of}(A))\}, \\ \quad\quad \{\mathit{act}\ .\ \exists \mathit{act}'.\ f(\mathit{act}) = \mathit{Some}(\mathit{act}') \wedge \mathit{act}' \in \mathit{outputs}(\mathit{asig\_of}(A))\}, \\ \quad\quad \{\mathit{act}\ .\ \exists \mathit{act}'.\ f(\mathit{act}) = \mathit{Some}(\mathit{act}') \wedge \mathit{act}' \in \mathit{internals}(\mathit{asig\_of}(A))\}\gt, \\ \quad \mathit{starts\_of}(A), \\ \quad \{\lt s, \mathit{act}, t \gt\ .\ \exists \mathit{act}'.\ f(\mathit{act}) = \mathit{Some}(\mathit{act}') \wedge \lt s, \mathit{act}', t \gt \in \mathit{trans\_of}(A)\}\gt \end{array}$$

## 3 Specification

The Alternating Bit Protocol [3] is designed to ensure that messages are delivered in order, from a sender to a receiver, in the presence of channels that can lose and duplicate messages. This FIFO-communication can be specified by a simple queue and therefore a single automaton *Spec*. As we are aiming for a finite state system, we have to consider an additional point: The sender buffer of the implementation will not be able to store an unbounded number of incoming messages. Restricting the number of input actions to yield a finite sender buffer is not allowed because of the input-enabledness required of IOA.

What we really need is an assumption about the behaviour of the environment, namely that it will only send the next message if requested to do so by an explicit action *Next* issued by the system. In the IOA-model this can be expressed by including an environment IOA which embodies this assumption. Therefore the specification is a parallel composition of two processes:

$$\mathit{Specification} \equiv \mathit{Env} \parallel \mathit{Spec}$$

and the interaction between them is shown in Fig. 1. The two components *Env* and *Spec* are described in the following subsections.

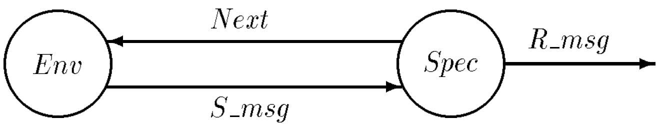



Figure 1: The Specification

### 3.1 The Environment

*Env* models the assumption that the environment only outputs *S_msg* when allowed to do so by *Next*. The state of *Env* is a single boolean variable *send_next*, initially true, which is set to true by every incoming *Next*. *S_msg* is enabled only if *send_next* is true and sets *send_next* to false as a result:

| | | | |
|---|---|---|---|
| *Next* | input | $S\_msg(m)$ | output |
| post: | $send\_next'$ | pre: | $send\_next$ |
| | | post: | $\neg send\_next'$ |

where we use the following format to describe transition relations:

| | |
|---|---|
| *action* | (input \| output \| internal) |
| pre: | $P$ |
| post: | $Q$ |

Predicate $P$ is the constraint on the state $s$ that must hold for the transition to apply. If it is true, it is omitted. Predicate $Q$ relates the state components before and after the transition; we refer to the state components after the transition by decorating their names with a $'$. If no state component changes, post is omitted.

### 3.2 The Specification

The state of the IOA *Spec* is a message queue $q$, initially empty, modelled with the type $(\mu)list$, where the parameter $\mu$ represents the message type. The only actions performed in the abstract system are: $S\_msg(m)$, putting message $m$ at the end of $q$, $R\_msg(m)$, taking message $m$ from the head of $q$, and *Next*, signaling the world outside to send the next message. Formally:

| | | | | | |
|---|---|---|---|---|---|
| *Next* | output | $S\_msg(m)$ | input | $R\_msg(m)$ | output |
| pre: | *true* | post: | $q' = q@[m]$ | pre: | $q = m :: rst$ |
| | | | | post: | $q' = rst$ |

## 4 Implementation

The system being proved correct also contains the component *Env* described in the previous section.

$$Implementation \equiv Env \parallel Impl$$

*Impl* represents the Alternating Bit Protocol and is itself a parallel composition of 4 processes:

$$Impl \equiv Sender \parallel S\_Ch \parallel Receiver \parallel R\_Ch$$

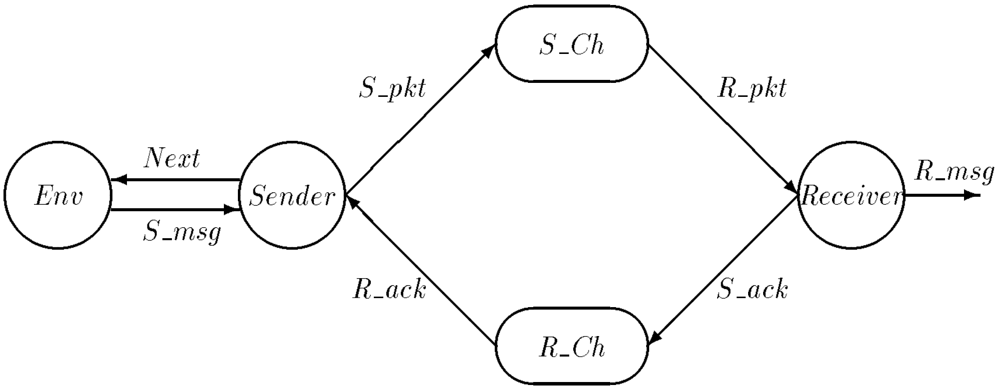



Figure 2: The Implementation

a sender, a receiver, and proprietary channels for both. The "dataflow" in the system is depicted in Fig. 2

Messages are transmitted from the sender to the receiver with a single header bit as packets of type $bool \times \mu$. The type of system actions, $(\mu)action$, is described in Isabelle by the following ML-style datatype:

$$(\mu)action \equiv \; Next \mid S\_msg(\mu) \mid R\_msg(\mu) \mid S\_pkt(bool, \mu) \mid R\_pkt(bool, \mu) \mid S\_ack(bool) \mid R\_ack(bool)$$

### 4.1 The Sender

The state of the process *Sender* is a pair:

| Field | Type | Initial Value |
|---|---|---|
| *message*: | $(\mu)option$ | *None* |
| *header*: | *bool* | *true* |

The Sender makes the following transitions:

| | |
|---|---|
| *Next* | output |
| pre: | $message = None$ |
| $S\_msg(m)$ | input |
| post: | $message' = Some(m) \;\wedge\; header' = header$ |
| $S\_pkt(b, m)$ | output |
| pre: | $message = Some(m) \;\wedge\; b = header$ |
| $R\_ack(b)$ | input |
| post: | **if** $b = header$ |
| | **then** $message' = None \;\wedge\; header' = \neg header$ |
| | **else** $message' = message \;\wedge\; header' = header$ |

Note that the presence of *Env*, i.e. the fact that the sender can control the flow of incoming messages via *Next*, enables us to get by with a buffer of length 1 (modelled by $(\mu)option$) in the sender; *Next* is only sent if the buffer is empty, i.e. $message = None$.

### 4.2 The Receiver

The state of the process *Receiver* is also a pair differing from the Sender simply in the initial value of the header variable:

| Field | Type | Initial Value |
|---|---|---|
| $message$: | $(\mu)option$ | $None$ |
| $header$: | $bool$ | $false$ |

The Receiver makes the following transitions:

| | |
|---|---|
| $R\_msg(m)$ | output |
| pre: | $message = Some(m)$ |
| post: | $message' = None \ \wedge \ header' = header$ |
| $R\_pkt(b, m)$ | input |
| post: | **if** $b \neq header \wedge message = None$ |
| | **then** $message' = Some(m) \ \wedge \ header' = \neg header$ |
| | **else** $message' = message \ \wedge \ header' = header$ |
| $S\_ack(b)$ | output |
| pre: | $b = header$ |

Note that *R_pkt* does not change the state unless $message = None$. This ensures that the receiver has passed the last message on via *R_msg* before accepting a new one. Alternatively, one could add the precondition $message = None$ to *S_ack* which would preclude the sender getting an acknowledgment and sending a new message before the receiver has actually passed the old one on.

### 4.3 The Channels

The channels, *R_Ch* and *S_Ch*, have very similar functionality. Roughly speaking, messages are added to a queue by an input action and removed from it by the corresponding output action. In addition, there can be no change at all in order to model the possibility to lose messages in case of the adding action and to duplicate messages in case of the removing action. The only differences between the channels are the type of the messages delivered, packets for *S_Ch* and booleans for *R_Ch*, and the specific names for input and output actions, *S_pkt* and *R_pkt* or *S_ack* and *R_ack*, respectively. Therefore both channels can be designed as instances of a generic channel using the renaming function described in section 2.

This is done by introducing a new datatype $(\alpha)act \equiv S(\alpha) \mid R(\alpha)$ of abstract actions and defining an IOA *Ch* with a single state component $q : (\alpha)list$ by the following transition relation:

| | | | |
|---|---|---|---|
| $S(a)$ | input | $R(a)$ | output |
| post: | $q' = q \vee q' = q@[a]$ | pre: | $q \neq [\,] \wedge a = hd(q)$ |
| | | post: | $q' = q \vee q' = tl(q)$ |

In Isabelle we use a set comprehension format to describe transition relations. In the case of *Ch* it looks like this:

$$
\begin{array}{lll}
Ch\_trans \ \equiv \ \{<s, act, s'> \, . & \texttt{case}\ act\ \texttt{of} & \\
& S(a) \ \Rightarrow & s' = s \vee s' = s@[a] \\
& R(a) \ \Rightarrow & s \neq [\,] \wedge a = hd(s) \wedge \\
& & (s' = s \vee s' = tl(s))\,\}
\end{array}
$$

An automatic translation of the pre/post style into the set comprehension format is possible and desirable but not the focus of our research.

The concrete channels are obtained from the abstract channel by $rename(Ch, S\_acts)$ and $rename(Ch, R\_acts)$, where

$$
\begin{array}{lcl}
S\_acts & : & (\mu)action \rightarrow (bool \times \mu)\ act\ option \\
R\_acts & : & (bool)action \rightarrow (bool)\ act\ option
\end{array}
$$

map the concrete actions to the corresponding abstract actions. For example $S\_acts$ is defined by $S\_acts(S\_pkt(b,m)) = Some(S(<b,m>))$, $S\_acts(R\_pkt(b,m)) = Some(R(<b,m>))$ and $S\_acts(act) = None$ for all other actions $act$.

# 5 Abstraction

What we are aiming for is a finite-state description of the Alternating Bit Protocol that is refined by the given implementation described in the previous section. On the way there we have to remove two obstacles:

1. The channel queues have to be finite.

2. The message alphabet has to be finite.

## 5.1 Finite Channels

Our attention is focused on this requirement. We defined an abstract version $RedCh$ of $Ch$ and an abstraction function $reduce$ from $Ch$ to $RedCh$ and prove $is\_weak\_pmap(reduce, Ch, RedCh)$. The idea is based on the observation that at most two different messages are held in each channel. This is easily explained: each message is repeatedly sent to $S\_Ch$, until the corresponding acknowledgment arrives. Once we switch to the next message, $S\_Ch$ can only contain copies of the previous message. Hence, $S\_Ch$'s queue is always of the form $old^*new^*$. The same is true for $R\_Ch$. Thus, if all adjacent identical messages are merged, the channels have size at most 2. Fortunately, this reasoning never needs to be formalized but is implicitly performed by the model checker.

### 5.1.1 Refinement of Channels

A compacting channel $RedCh$ is obtained from $Ch$ if new messages are only added provided they differ from the last one added. Thus $RedCh$ is identical to $Ch$ except for action $S$:

$$\begin{array}{lll} S(\alpha) & \text{input} & \\ \text{post:} & q' = q \vee & \textbf{if } a \neq hd(reverse(q)) \vee q = [\,] \\ & & \textbf{then } q' = q@[a] \\ & & \textbf{else } q' = q \end{array}$$

By renaming $RedCh$ we obtain the collapsed versions of $R\_Ch$ and $S\_Ch$, called $R\_RedCh$ and $S\_RedCh$. Notice that the description is a priori not finite, as $q$ is an unbounded list. Finiteness is only implied by the context, i.e. the behaviour of the protocol.

With the definition of an abstraction function $reduce$

$$\begin{array}{lcl} reduce([\,]) & \equiv & [\,] \\ reduce([x :: xs]) & \equiv & \texttt{case } xs \texttt{ of} \\ & & \quad [\,] \;\Rightarrow\; [x] \\ & & \quad y :: ys \;\Rightarrow\; if(x = y, reduce(xs), x :: reduce(xs)) \end{array}$$

we get the following refinement goal:

$$is\_weak\_pmap(reduce, Ch, RedCh)$$

The proof of this obligation is rather straightforward. It proceeds by case analysis on the type of actions. Using some lemmata on how $reduce$ behaves when combined with operators like @ or

$tl$, most cases are automatically solved by the conditional and contextual rewriting of Isabelle. Finally, using the meta-theorem

$$\begin{array}{l} is\_weak\_pmap(abs, C, A) \\ \supset is\_weak\_pmap(abs, rename(C, f), rename(A, f)) \end{array}$$

we get the appropriate refinement results for the concrete channels $S\_Ch$, $R\_Ch$ and their collapsed versions $S\_RedCh$ and $R\_RedCh$.

### 5.1.2 Compositionality

In order to extend this refinement result from the channels to the whole system, we have to prove some compositionality results for refinements. Lynch and Tuttle [13] established the required lemma on the level of trace inclusions. We decided, however, to prove it on the level of abstraction functions for reasons of simplicity.

$$\begin{array}{l} IOA(C_1) \wedge IOA(C_2) \wedge IOA(A_1) \wedge IOA(A_2) \wedge \\ externals(asig\_of(C_1)) = externals(asig\_of(A_1)) \wedge \\ externals(asig\_of(C_2)) = externals(asig\_of(A_2)) \wedge \\ compatible(C_1, C_2) \wedge compatible(A_1, A_2) \wedge \\ is\_weak\_pmap(f, C_1, A_1) \wedge is\_weak\_pmap(g, C_2, A_2) \\ \supset is\_weak\_pmap(\lambda{<}c_1, c_2{>}.{<}f(c_1), g(c_2){>}, C_1 \parallel C_2, A_1 \parallel A_2) \end{array}$$

Unfortunately, trace inclusion does not imply the existence of an abstraction function. Hence the above theorem is not as general as the corresponding one about traces, in particular since $is\_weak\_pmap(id, A, A)$ only holds if $A$ has no internal actions. We intend to formalize and prove compositionality on the trace level in the near future.

Performing the proofs of abstraction and compositionality with Isabelle we encountered a mismatch between the time required for the refinement proof and that required for the compatibility checks. Nearly half the time (1.5 min on a SPARC station 10) was needed to establish that no component causes a name clash of input/output actions. These checks, although automated, are expensive if performed by a theorem prover. Partly this is caused by our decision to have *rename* translate action names in the opposite direction one would expect (see section 2.3), something we may need to rethink.

## 5.2 Finite Number of Messages

The second requirement, the problem of abstracting out data from a data-independent program has already been addressed by Wolper [17]. In his paper he shows how to reduce an infinite data domain to a small finite one if data independence is guaranteed and the properties to be checked are expressible in propositional temporal logic. In [2] and [16] this method is applied to the Alternating Bit Protocol. There, only three different message values are needed to verify the protocol's functional correctness.

Basically, a program is data-independent if its behaviour does not depend on the specific data it operates upon. A sufficient condition for a program described by an IOA to be data independent is that everywhere in the automaton the transitions are independent of the value of messages being transmitted. An inspection of our description of the protocol shows that it satisfies the condition.

In contrast to [2] our specification is not given as a collection of temporal formulae, but in terms of I/O automata. Thus, the methods above are not directly applicable to our formalization and until now, we did not investigate how to transfer them formally into our setting. However, it is intuitively plausible that Wolper's theory of data-independence holds generally, independently of the respective formalization. That is why we analogously restricted our model checking algorithm to deal with only three different message values.

A formal treatment of data-abstraction in Isabelle/HOL needs a modification of the way we model data. Currently the diversity of data is modelled by polymorphic types[1]. But since types are a meta-level notion and cannot be talked about (e.g. quantified) in HOL, even formalizing data independence seems to be impossible. Using object-level sets instead of polymorphism would cure this problem but is likely to complicate the theory.

# 6 Model Checking

The task of the model checker is to verify that $B$, the implementation with collapsing channels refines $A$, the specification. It is done by a generic ML-function *check*

$$check(actions, internal, startsB, nextsB, startA, transA, abs)$$

where $actions : (\alpha)list$ is the list of all actions, $internal : \alpha \to bool$ recognizes internal actions of $B$, $startsB : (\sigma)list$ is the list of start states of $B$, $nextsB : \sigma \to \alpha \to (\sigma)list$ produces the list of successor states in $B$, $startsA : \tau \to bool$ recognizes start states of $A$, $transA : \tau \to \alpha \to \tau \to bool$ recognizes transitions of $A$, and $abs : \sigma \to \tau$ is the abstraction function.

It is easy to translate Isabelle's predicative description of $A$'s transitions automatically into an ML-function *transA*. For *nextsB* this is only possible if the predicates have a certain recognizable form, for example disjunctions of assignments of values to the state components. Otherwise how are we to compute the set of next states satisfying an arbitrary predicate? If $\sigma$, the state space of $B$ (as opposed to the set of reachable states!) is infinite, this is impossible. That is the main reason why we need to specify $B$, i.e. *RedCh* explicitly; otherwise we could have described *RedCh* implicitly in terms of *Ch* and *reduce*.

The abstraction function *abs* is given by

$$abs(s) \equiv l(R.message)@if(R.header = S.header, l(S.message), tl(l(S.message)))$$

where $l : (\alpha)option \to (\alpha)list$ is defined by the equations $l(Some(x)) = [x]$ and $l(None) = [\,]$. To distinguish between components of the receiver state and the sender state that have the same field names, we use a 'dotted identifier' notation, e.g. $S.header$ and $R.header$.

It is also possible to generate *abs* automatically as a set of corresponding state pairs as done in [10]. This would not allow to document *abs* explicitly, but it would mean a step forward towards fully automatic support — the major advantage of model checking.

*check* itself realizes the predicate $is\_weak\_pmap(abs, B, A)$ by simply performing full state space exploration. Beginning with *startsB* the algorithm examines all reachable states, checking for every transition $<s_1, a, s_2> \in trans\_of(B)$ that either $<abs(s_1), a, abs(s_2)> \in trans\_of(A)$ (if $a$ is external) or $abs(s_1) = abs(s_2)$ (if $a$ is internal).

At the moment the ML-code for the different arguments of *check* is still generated manually. However, we intend to automate this, subject to the restrictions on $B$ described above. It should also be noted that *check* is just a prototype which should be replaced by some optimized model checker, for example the one described in [9].

# References


[1] P. Abdulla and B. Jonsson. Verifying programs with unreliable channels. In *Proc. 8th IEEE Symp. Logic in Computer Science*, pages 160–170. IEEE Press, 1993.


[1]It is not true that a polymorphic IOA is automatically data independent: HOL-formulae may contain the polymorphic equality "=" which destroys data independence.

[2] S. Aggarwal, C. Courcoubetis, and P. Wolper. Adding liveness properties to coupled finite-state machines. *ACM Transactions on Programming Languages and Systems*, 12(2):303–339, 1990.

[3] K. Bartlett, R. Scantlebury, and P. Wilkinson. A note on reliable full-duplex transmission over half-duplex lines. *Communications of the ACM*, 12(5):260–261, 1969.

[4] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. In *Proc. 19th ACM Symp. Principles of Programming Languages*, pages 343–354. ACM Press, 1992.

[5] D. Dams, O. Grumberg, and R. Gerth. Abstract interpretation of reactive systems: Abstractions preserving $\forall$CTL$^*$, $\exists$CTL$^*$ and CTL$^*$. In E.-R. Olderog, editor, *Programming Concepts, Methods and Calculi (PROCOMET)*, pages 573–593. North-Holland, 1994.

[6] J.-C. Fernandez and L. Mounier. "On the Fly" verification of behavioural equivalences and preorders. In K. G. Larsen, editor, *Proc. 3rd Workshop Computer Aided Verification*, volume 575 of *Lect. Notes in Comp. Sci.*, pages 181–191. Springer-Verlag, 1992.

[7] 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. Extended abstract in Proceedings ICALP'94.

[8] S. Graf and C. Loiseaux. A tool for symbolic program verification and abstraction. In C. Courcoubetis, editor, *Computer Aided Verification*, volume 697 of *Lect. Notes in Comp. Sci.*, pages 71–84. Springer-Verlag, 1993.

[9] P. Herrmann, T. Kraatz, H. Krumm, and M. Stange. Automated verification of refinements of concurrent and distributed systems. Technical Report 541, Fachbereich Informatik, Universität Dortmund, 1994.

[10] P. Herrmann and H. Krumm. Report on analysis and verification techniques. Technical Report 485, Fachbereich Informatik, Universität Dortmund, 1993.

[11] H. Hungar. Combining model checking and theorem proving to verify parallel processes. In C. Courcoubetis, editor, *Computer Aided Verification*, volume 697 of *Lect. Notes in Comp. Sci.*, pages 154–165. Springer-Verlag, 1993.

[12] R. Kurshan. Reducibility in analysis of coordination. In K. Varaiya, editor, *Discrete Event Systems: Models and Applications*, volume 103 of *Lecture Notes in Control and Information Science*, pages 19–39. Springer-Verlag, 1987.

[13] N. Lynch and M. Tuttle. Hierarchical correctness proofs for distributed algorithms. Technical Report MIT/LCS/TR-387, Laboratory for Computer Science, MIT, Cambridge, MA., 1987.

[14] N. Lynch and M. Tuttle. An introduction to Input/Output automata. *CWI Quarterly*, 2(3):219–246, 1989.

[15] T. Nipkow and K. Slind. I/O automata in Isabelle/HOL. In *Proc. TYPES Workshop 1994*, Lect. Notes in Comp. Sci. Springer-Verlag. To appear.

[16] K. Sabnani. An algorithmic technique for protocol verification. *IEEE Transactions on Communications*, 36(8):924–930, 1988.

[17] P. Wolper. Expressing interesting properties of programs in propositional temporal logic. In *Proc. 13th ACM Symp. Principles of Programming Languages*, pages 184–193. ACM Press, 1986.