

Proof-Checking Protocols using Bisimulations

Christine Röckl and Javier Esparza

Technische Universität München,
Fakultät für Informatik, D-80290 München,
{roeckl,esparza}@in.tum.de

Abstract. We report on our experience in using the Isabelle/HOL theorem prover to mechanize proofs of observation equivalence for systems with infinitely many states, and for parameterized systems. We follow the direct approach: An infinite relation containing the pair of systems to be shown equivalent is defined, and then proved to be a weak bisimulation. The weak bisimilarity proof is split into many cases, corresponding to the derivatives of the pairs in the relation. Isabelle/HOL automatically proves simple cases, and guarantees that no case is forgotten. The strengths and weaknesses of the approach are discussed.

1 Introduction

Observation equivalence (or weak bisimilarity) is a natural notion of behavioural equivalence; it has been extensively studied and applied in the literature (see, for instance, [Mil89,MS92,San95]). There exist two general ways of showing that a system is observationally equivalent to its specification. One, semantically oriented, way is to follow the definition: exhibit a relation containing as a pair the system and its specification, and prove that it is a bisimulation. Conceptually, the difficult part is to exhibit the relation, while proving that the relation is indeed a bisimulation reduces to a (usually large) number of simple checks of the form “for each derivative there exists a matching derivative”. In order to apply this method the only requirement is to have a good intuition about the system. The second, syntactically oriented, way is to use algebraic (equational) reasoning: the system is proved to be observationally equivalent to the specification by exhibiting a (usually long) chain of equalities starting with the system and ending with the specification. In this case no bisimulation relation has to be guessed but deep insight into the proof system is required.

Both ways can be applied to systems with a finite or infinite state space. In the second case, however, they cannot be completely automatized due to well known undecidability results. Still, algebraic techniques have been used to verify various infinite systems, including a variation of the ABP (see, for instance, [GS95,BG93]), and tool support has been developed. In contrast, and to the best of our knowledge, the semantic way has not yet been mechanized for infinite systems, even though it could be very useful: In order to establish that a relation is a bisimulation, a large number of cases may have to be considered, even if the relation can be partitioned into uniformly representable infinite subsets.

For instance, the infinite-state version of the alternating bit protocol (ABP) considered in [Mil89], yields a relation consisting of the union of 12 groups of pairs, leading to 94 proof obligations of the form ‘the strong transition $\xrightarrow{\mu}$ on one side can be matched by a weak transition $\xRightarrow{\hat{\mu}}$ on the other side’. Proofs by hand consider only a few cases (Milner considers 6), leaving the others as obvious or similar. This procedure is of course prone to errors; in fact, Milner remarks that his proof of the ABP is on the verge of what is tractable by hand, and he proposes to mechanize the procedure with the help of theorem provers.

In this paper we follow Milner’s suggestion. We report on our experience with the mechanization in Isabelle/HOL [Pau94,Pau93] of three examples concerning communication protocols, including Milner’s proof of the ABP. We examine to what extent the theorem prover is able to verify bisimulation relations automatically, and at what point the user has to provide additional information. We shall see that Isabelle automatically finds derivatives that are reachable by transition sequences of length 0 or 1, while longer transition sequences and their derivatives have to be specified by the user.

We have chosen examples from the area of communication protocols for several reasons. First, they are often used as ‘real-life’ examples to demonstrate the expressiveness and merits, or weaknesses, of concurrent frameworks and their behavioural or algebraic equivalences [BK85,Mil89,BW90,NC95,Sne95,Nam97]. Second, they often serve as benchmarks for proof environments, especially for algebraic proof systems in theorem provers. Examples of mechanizations are presented in [PS88,NS94,GS95,BG93,Gim96,HM98]. Finally, the bisimulation approach seems to be particularly suitable for communication protocols (see the discussion at the end of the paper).

We model both communication protocols and their specifications in terms of *labelled transition systems* [Plo81], using a concurrent normal form that usually suffices for reactive systems (see also [GS95]): a finite number of sequential, yet non-deterministic, value-passing processes (i.e., processes with infinite summation) is connected by parallel composition, with an embracing restriction hiding the internal actions. Note that other models like communicating automata, for instance, would have done as well. Note further that the systems and their specifications need not even be described within the same model. Also, our experiments do not rely especially on Isabelle/HOL; we could as well have applied any other generic prover offering higher order logic, like PVS or Coq.

The paper is organized as follows: In Section 2 we give a short overview of the features of Isabelle/HOL we are going to exploit in our case study. The process algebraic framework is introduced in Section 3. The main part of this work is Section 4 where we establish bisimulation relations for three examples, and discuss their proofs in Isabelle: we show that a channel that loses or duplicates messages is observationally equivalent to a channel that further — detectably — garbles messages but applies a filter to discard the garbled ones before delivery. The channels are assumed to be of arbitrary length, thus both systems are infinite-state (Section 4.1). The second example is a mechanization of Milner’s correctness proof for the Alternating Bit Protocol (ABP) [Mil89] in

Isabelle/HOL. Also this example is infinite-state as, again, the channels are assumed to be of arbitrary length (Section 4.2). The last example deals with a specification of a Sliding Window Protocol (SWP) in terms of a parallel composition of several channels applying the ABP [PS88]. It is a parameterized system (the parameter being essentially the window size) containing infinite-state components. Due to the compositionality of observation equivalence we can replace the ABP components with their specifications (i.e., with one-place buffers), and thus obtain a parameterized system of finite-state components (Section 4.3).

2 Isabelle/HOL

Using the generic theorem prover Isabelle [Pau94] we conduct all proofs in its instantiation HOL for higher-order logic [Pau93]. Proofs in Isabelle are based on unification, and are usually conducted in a backward resolution style: the user formulates the goal he/she intends to prove, and then — in interaction with Isabelle — continuously reduces it to simpler subgoals until all of the subgoals have been accepted by the tool. Upon this the goal can be stored in Isabelle’s database as a theorem. Isabelle offers various tactics, most of them applying to single subgoals. The basic tactics allow the user to instantiate a theorem from Isabelle’s database so that its conclusion can be applied to transform a current subgoal into instantiations of its premises. Further there exist automatic tactics using the basic tactics to prove given subgoals according to different heuristics. These heuristics have in common that a provable goal is always transformed into a set of provable subgoals; ‘unsafe’ rules (rules that might yield unprovable subgoals) are only applied if none of the resulting subgoals has to be reported to the user as currently unproved. Besides these *classical tactics* Isabelle offers *simplification tactics* based on algebraic transformations. The most general and powerful tactic is `Auto_tac` which interweaves classical and simplification tactics, and reasons about all subgoals of the current proof state simultaneously.

Isabelle’s instantiation for higher-order logic offers a number of modules, called *theories*, including among others frameworks for arithmetics, sets, and lists. These modules include databases with basic theorems that have already been proved, and that can be referenced by the user when working with the modules. The arithmetic module of Isabelle/HOL is of particular interest, as it allows the user to give inductive definitions. Note that transition systems are defined inductively, i.e., as the *least set* satisfying certain axioms and rules. Isabelle then automatically generates various forms of the transition rules including case exhaustions stating all reasons that may have led to a given transition; e.g., “ $P \parallel Q \xrightarrow{\tau} R$ because $P \xrightarrow{\tau} P'$ and $R = P' \parallel Q$, or ...”. Further it generates tactics for structural induction. Although Isabelle/HOL offers a framework for coinduction, we do not make use of it, sticking to the original definition of observation equivalence given in terms of a predicate over binary relations.

Isabelle is a *generic* theorem prover, i.e., the user can define theories of his/her own. Such theory modules consist of two parts: in a definition part new types, constants, and rules for the constants are introduced; in a second part theorems

are proved and added to the theorem database. A preamble to the definition part lists the theories upon which the new module is based; these can include built-in as well as user-defined theories.

3 Transition Systems and Observation Equivalence

Reactive systems generally consist of a finite set of *process components* which, according to their *states*, send and receive data along *connections*¹ between them (see also [GS95]).

Let A be a countably infinite set of *signals* (visible actions) ranged over by a, b, \dots . Further, in order to transmit messages of some (unspecified) type α we introduce a countably infinite set of *connections* $(\alpha)\mathcal{C}on$ ranged over by $c1, c2, \dots$. The type of the connections is parameterized over the type variable α , thus we use $(\alpha)\mathcal{C}on$ instead of simply writing $\mathcal{C}on$. The set of *visible labels*, $(\alpha)\mathcal{L}$ is given by all inputs a and $c(v)$, and outputs \bar{a} and $\bar{c}\langle v \rangle$ of signals and messages along the connections. We use $\mathcal{A}ct \stackrel{\text{def}}{=} A \cup \{\bar{a} \mid a \in A\}$ to denote the set of inputs and outputs on signals.

(States of) *systems* are defined inductively in terms of (states of) their *components* P_i — given in terms of constant identifiers, and representing the basic units — and parallel compositions between them. We use the term *processes* to refer to components and systems equally. Formally, process components and states have the following syntax:

$$\begin{aligned} (\alpha)\mathcal{P}\mathcal{C} &::= P_1(\tilde{x}_1) \mid \dots \mid P_n(\tilde{x}_n), \\ (\alpha)\mathcal{S} &::= (\alpha)\mathcal{P}\mathcal{C} \mid (\alpha)\mathcal{S} \parallel (\alpha)\mathcal{S}, \end{aligned}$$

where the \tilde{x}_i are variables of type α . In the Isabelle formalization of the protocols, components are sequential yet possibly nondeterministic processes, where each state is denoted by a constant of its own.

We use a Plotkin-style transition semantics [Plo81] given in terms of a *strong transition relation* $\xrightarrow{\mu} \subseteq (\alpha)\mathcal{S} \times ((\alpha)\mathcal{L} \cup \{\tau\}) \times (\alpha)\mathcal{S}$. The transition rules are defined inductively via axioms for the components in $(\alpha)\mathcal{P}\mathcal{C}$ (describing the input and output as well as the silent behaviour of the components), and rules for parallel composition including communication, where $\mu \in (\alpha)\mathcal{L} \cup \{\tau\}$, $a \in A$, $c \in (\alpha)\mathcal{C}on$, and $v \in \alpha$:

$$\frac{P \xrightarrow{\mu} P'}{P \parallel Q \xrightarrow{\mu} P' \parallel Q} \text{ P1} \quad \frac{P \xrightarrow{a} P' \quad Q \xrightarrow{\bar{a}} Q'}{P \parallel Q \xrightarrow{\tau} P' \parallel Q'} \text{ C1} \quad \frac{P \xrightarrow{c(v)} P' \quad Q \xrightarrow{\bar{c}\langle v \rangle} Q'}{P \parallel Q \xrightarrow{\tau} P' \parallel Q'} \text{ C3}$$

The rules P2, C2, and C4 are symmetric versions of P1, C1, and C3.

Note that the transition rules are defined inductively, i.e., the transition relation is defined as the least set satisfying the given axioms and rules (cf. Section 2). This implies that besides the constructive rules such as P1, P2, or

¹ We do not use the term *channel* here, in order to avoid ambiguities with the channels used by the protocols.

C1, one can give analysis rules telling for a given transition how it can be derived. Let, e.g., a be a signal, and let P and Q be processes. Then,

$$\frac{P \parallel Q \xrightarrow{a} R}{(\exists P' . R = P' \parallel Q \wedge P \xrightarrow{a} P') \vee (\exists Q' . R = P \parallel Q' \wedge Q \xrightarrow{a} Q')}$$

is such a case analysis. The general analysis rule comprising silent steps is more complicated, as it further considers possible communications between the components.

We model inputs in an early style, i.e., input rules are of the form $\forall w. P \xrightarrow{c(w)} P'(w)$. When formalizing the transition systems in Isabelle the user need not state the \forall -quantification explicitly but applies a formal parameter for which the quantification is then automatically provided by the prover. Note that Isabelle is able to distinguish between constants and formal parameters.

In order to abstract from internal activities of the systems (like, e.g., communications, idle loops, or the processing of data), we use a *weak transition relation* $\xRightarrow{\mu} \subseteq (\alpha)\mathcal{S} \times ((\alpha)\mathcal{L} \cup \{\tau, \epsilon\}) \times (\alpha)\mathcal{S}$ which allows for arbitrarily many τ -steps before and after each transition. As usual $\xRightarrow{\epsilon} \stackrel{\text{def}}{=} (\xrightarrow{\tau})^*$ denotes the reflexive transitive closure of (strong) internal steps. Below we give the rules for introducing ϵ , lifting strong transitions to weak ones, and for the expansion of weak transitions by silent steps, where $\nu \in (\alpha)\mathcal{L} \cup \{\tau\}$, and $\mu \in (\alpha)\mathcal{L} \cup \{\tau, \epsilon\}$,

$$\frac{}{P \xRightarrow{\epsilon} P} \text{E} \quad \frac{P \xrightarrow{\tau} P'}{P \xRightarrow{\epsilon} P'} \text{TE} \quad \frac{P \xrightarrow{\nu} P'}{P \xRightarrow{\nu} P'} \text{SW} \quad \frac{P \xRightarrow{\epsilon} P', P' \xRightarrow{\mu} P'', P'' \xRightarrow{\epsilon} P'''}{P \xRightarrow{\mu} P'''} \text{EX}$$

Further there exist weak versions of the rules P1, P2, C1, C2, C3, and C4.

We use the common abbreviation $\hat{\mu}$ to denote ϵ if $\mu = \tau$, and μ if μ is a visible label.

Isabelle allows a mixfix representation of constants (like actions, processes, or transitions). Exploiting this we can write $P \text{ -}[c\langle v \rangle]\text{ -> } P'$ for an output of value v along channel c , or $P \text{ -}[c\#\{v\}]\text{ -> } P'$ (v) for an input of v , or $P \text{ -}[tau]\text{ -> } P'$ for a silent transition. We write $P \text{ =[u]=> } P'$ for a weak transition with label u .

So far we have not distinguished between *internal* and *external* signals or connections. In order to avoid an additional restriction operator, for which we would have to formalize additional rules in Isabelle, we defer the matter of interface to the definition of bisimilarity. Let $A \subseteq \Lambda$ be a set of signals, and $C \subseteq (\alpha)\mathcal{Con}$ a set of connections of type α . Then we define $(\alpha)\mathcal{L}|_{(A,C)}$ to contain all possible inputs and outputs on signals in A , as well as on connections in C .

Definition 1 (Observation Equivalence). A relation $\mathcal{R} \subseteq (\alpha)\mathcal{S} \times (\alpha)\mathcal{S}$ is a (weak) bisimulation wrt. $(A, C) \in \Lambda \times (\alpha)\mathcal{Con}$, if for all $(P, Q) \in \mathcal{R}$ and $\mu \in (\alpha)\mathcal{L}|_{(A,C)} \cup \{\tau\}$, the following holds:

- If $P \xrightarrow{\mu} P'$, for some P' , there exists a Q' s.t. $Q \xRightarrow{\hat{\mu}} Q'$ and $(P', Q') \in \mathcal{R}$.
- If $Q \xrightarrow{\mu} Q'$, for some Q' , there exists a P' s.t. $P \xRightarrow{\hat{\mu}} P'$ and $(P', Q') \in \mathcal{R}$.

Two processes P and Q are *observation equivalent wrt. (A, C)* , written $P \approx_{(A, C)} Q$, if there exists a weak bisimulation containing (P, Q) .

Let \backslash be the restriction operator from CCS, extended to be applicable both to signals and connections. Though we find it convenient not to have a restriction operator on $(\alpha)\mathcal{S}$, we are still close to the original notion of observation equivalence. Let \mathcal{R} be a bisimulation wrt. (A, C) . Then

$$\mathcal{R}' \stackrel{\text{def}}{=} \{(P \backslash (A - A, (\alpha)\mathcal{C}on - C), Q \backslash (A - A, (\alpha)\mathcal{C}on - C) \mid (P, Q) \in \mathcal{R}\}$$

is a weak bisimulation in the sense of [Mil89]. This is due to observation equivalence being a congruence wrt. restriction. Further every bisimulation in the sense of [Mil89] is a bisimulation wrt. $(A, (\alpha)\mathcal{C}on)$. This implies that two systems are observationally equivalent in the usual sense iff they are so in our sense, allowing us to adapt congruence properties and proof techniques without having to prove them explicitly in our framework.

4 A Case Study

In all of the following examples the sets of labels that are visible to the observer either consist of signals, or of messages sent along channels. We can thus project observation equivalence wrt. (A, C) either to observation equivalence wrt. A , written \approx_A , or to observation equivalence wrt. C , written \approx_C .

All proofs in this section follow a uniform pattern: The user sets up the systems and their specifications by giving their states and transition rules in one module. From this definition Isabelle computes sets of rules that can be used to reason about the transitions in a constructive as well as in an analysing style; further Isabelle generates schemes for structural induction (cf. Sections 2, 3). Another module contains as a predicate the criterion for a relation to be a bisimulation. The bisimulation relation itself is defined and proved in a third module which relies on the two previous modules. Often a further module is necessary to provide additional theorems about the data types used to model the transition systems (e.g., about insertion into or deletion from the finite lists representing communication channels).

The proof that a relation is a bisimulation usually falls into the following parts: in a separate theorem for each label μ , we prove symbolically for every $(P, Q) \in \mathcal{R}$ that if $P \xrightarrow{\mu} P'$, for some P' , there exists a Q' s.t. $Q \xrightarrow{\hat{\mu}} Q'$ and $(P', Q') \in \mathcal{R}$; and similarly for Q . Then the main theorem of each theory stating that \mathcal{R} is a bisimulation is instantiated according to Definition 1 and reduced to the above proof obligations. This, including the necessary swapping of quantifiers, is done automatically by Isabelle.

In the following we present three examples demonstrating the generality of the approach: in the first example we compare two infinite-state systems that are both non-deterministic; the second example deals with quite a large composed system containing non-deterministic components and a small deterministic

specification; in the third example we consider a composed system which is parameterized wrt. the number of its components. The three examples are closely related, e.g., the system studied in our third example contains as its components the second one which can be replaced by its specification due to the compositionality of observation equivalence. It should be noted that, unlike in many proof mechanizations in theorem provers, our Isabelle proofs are not much different from proofs as one would perform them by hand. Yet, what is different is the emphasis put on different parts of the proofs. Whereas in proofs by hand one has to be careful not to forget about any strong transition, Isabelle automatically takes care of this. On the other hand, an Isabelle user has to spend a lot of time interacting with the tool in order to prove very simple theorems about the data structures manipulated during a transition. In both cases the weak transitions have to be found by the person conducting the proof. However, presenting them to Isabelle is not much more time-consuming than writing them down on a piece of paper, and often it even suffices to provide the prover with a scheme so to enable it to generate the transitions automatically.

4.1 Faulty channels of unbounded size

Our first example is taken from [Sne95]. It is of interest to us as it compares two indeterminate infinite-state systems operating on similar data structures. Most of the resulting proof obligations refer to strong transitions in weak disguise.

Consider two channels of unbounded capacity, say K and L . We model their contents by finite lists of arbitrary length. Both may lose or duplicate messages, but K is further able to garble data. This is reflected by an additional bit attached to each message in K .

```

"L(s)           -[ci#{x}]->    L(x # s)"           (* accept *)
"L(s @ a # t)   -[tau]->      L(s @ t)"           (* lose *)
"L(s @ a # t)   -[tau]->      L(s @ a # a # t)"       (* dupl *)
"L(s @ [x])     -[co<x>]->    L(s)"                 (* deliver *)

"K(s)           -[ci#{x}]->    K((x, True) # s)"      (* accept *)
"K(s @ a # t)   -[tau]->      K(s @ t)"             (* lose *)
"K(s @ a # t)   -[tau]->      K(s @ a # a # t)"       (* dupl *)
"K(s @ (x, b) # t) -[tau]->    K(s @ (x, False) # t)" (* garble *)
"K(s @ [(x, b)]) -[cf<(x, b)>]-> K(s)"                 (* deliver *)

```

A filter attached to K delivers correctly transmitted messages and discards garbled ones. We consider a version of a filter that discards garbled messages immediately when it receives them, but may arbitrarily lose or duplicate a message once it has accepted it. A filter that delivers all messages it has accepted would not lead to a system observationally equivalent to L : a state in which a message has just been transferred to the filter could not match a losing action by L .

```

"Filter         -[cf#{(x, True)}]-> FF{x, 0}"        (* accept *)

```

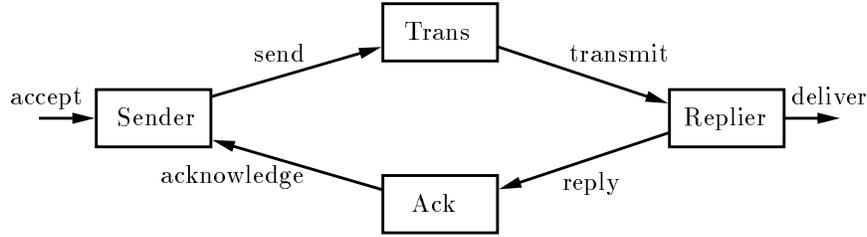


Fig. 1. Components of the ABP

```

"Filter      -[cf#{(x, False)}]-> Filter"          (* discard *)
"FF{x, 0}    -[co<x>]->      Filter"              (* deliver *)
"FF{x, Suc n} -[co<x>]->    FF{x, n}"            (* deliver *)
"FF{x, 0}    -[tau]->      Filter"                (* lose *)
"FF{x, Suc n} -[tau]->    FF{x, n}"              (* lose *)
"FF{x, n}    -[tau]->    FF{x, Suc n}"          (* dupl *)

```

By demonstrating that the relation `BS_Filter`, given below, is a bisimulation relation wrt. `{ci, co}`, we can conclude that $L \approx (K \parallel \text{Filter}) \setminus \{cf\}$.

```

"BStr == {(P, Q) . (EX s      . P = K(s) || Filter
                    & Q = L(map fst (filter snd s)))}"
"BSdl == {(P, Q) . (EX s x n . P = K(s) || FF{x,n}
                    & Q = L((map fst (filter snd s))
                              @ (replicate (Suc n) x)))}"
"BS_Filter == BStr Un BSdl"

```

In the relation `BS_Filter`, `L` contains lists which are obtained from those stored in `K` by first eliminating all garbled messages (`filter snd s`; garbled messages are tagged with a `False` bit), and then projecting all elements of the resulting list to their first components (`map fst`). The list x^{n+1} , denoted by `replicate (Suc n) x`, models the $n + 1$ copies of message `x` stored in the filter.

Proving that `BS_Filter` is a bisimulation is not difficult, yet one has to take care of the lists of messages in the channels. As mentioned above most of the involvement by the user goes into theorems telling, for instance, how `map fst (filter snd s)` looks like if an element has been lost from `s`. Provided with these theorems, however, Isabelle proves by one single application of `Auto_tac` that `BS_filter` is a bisimulation. In particular, the user does not have to find the weak transitions.

The proof script contains less than 300 lines, and has been set up within a few hours only.

4.2 The Alternating Bit Protocol

The Alternating Bit Protocol (ABP), introduced in [BSW69], is a well-established benchmark for proof methodologies implemented in theorem provers (see, for

instance, [PS88,BG93,NS94,Gim96]). It turns unreliable channels into reliable communication lines. We consider an infinite-state variant in which the channels can hold arbitrarily many messages. The model as well as the outline of the proof follow [Mil89].

The behaviour of the ABP can be specified in terms of a one-place buffer. This allows to abstract from the data to be transmitted. Note however that including them would not further complicate the proof, neither by hand nor in a theorem prover.

```
"accBuff -[accept]-> delBuff"          (* accept a message *)
"delBuff -[deliver]-> accBuff"         (* deliver a message *)
```

The ABP is designed around two faulty channels — one *transmitting* the messages and the other returning *acknowledgements* — a *sender*, and a *replier* module. A schematic view is given in Figure 1.

The channels, **Trans** and **Ack**, may both lose or duplicate but never swap messages. They behave exactly like channel **L** in the previous Section. We refer to the input and output connections of **Trans** as **cs** and **ct**, and to those of **Ack** as **cr** and **ca**, respectively.

The *sender* module continuously accepts messages from the environment, transmits them over the channel **Trans**, and waits for an acknowledgement along **Ack**, before accepting a new message. If an acknowledgement does not arrive within a certain time the sender assumes that the message has been lost and resends it. Yet, as the message may only have been delayed, the sender tags all messages with a bit so that new messages can be distinguished from old ones. An alternating bit suffices for this purpose, as the messages may not be swapped.

```
"Accept(b) -[accept]-> Send(b)"        (* accept a message *)
"Send(b) -[cs<b>]-> Sending(b)"       (* send message *)
"Sending(b) -[tau]-> Send(b)"         (* timeout *)
"Sending(b) -[ca#{b}]-> Accept(~b)"   (* correct acknowledge *)
"Sending(b) -[ca#{~b}]-> Sending(b)"  (* old acknowledge *)
```

After having delivered a message to the environment, the *replier* module repeatedly transmits tagged acknowledgements to the sender until a new message arrives.

```
"Deliver(b) -[deliver]-> Reply(b)"     (* deliver a message *)
"Reply(b) -[cr<b>]-> Replying(b)"     (* acknowledge *)
"Replying(b) -[tau]-> Reply(b)"       (* timeout *)
"Replying(b) -[ct#{~b}]-> Deliver(~b)" (* receive new message *)
"Replying(b) -[ct#{b}]-> Replying(b)" (* receive old message *)
```

The bisimulation relation, **BS_ABP**, is the union of the relations **BSaccept**, in which the processes are potentially able to accept a new message, and **BSdeliver**, in which the processes may deliver the current message. In every channel there are at most two types of messages or acknowledgements: those that are currently being delivered, and possibly copies of the previous ones. The finite lists are

thus either of the form x^n , or $x^n y^m$; in Isabelle this is expressed by using the `replicate` operator defined in the built-in module for finite lists.

```
"BSaccept == {(P, Q) . Q = accBuff &
  ((EX b n p . P = Accept(~b) || Trans(replicate n b) ||
    Ack(replicate p b) || Reply(b))
 | (EX b n p . P = Accept(~b) || Trans(replicate n b) ||
    Ack(replicate p b) || Replaying(b))
 | (EX b m p q . P = Send(~b) || Trans(replicate m (~b)) ||
    Ack(replicate p b @ replicate q (~b)) ||
    Reply(~b))
 | (EX b m p q . P = Send(~b) || Trans(replicate m (~b)) ||
    Ack(replicate p b @ replicate q (~b)) ||
    Replaying(~b))
 | (EX b m p q . P = Sending(~b) || Trans(replicate m (~b)) ||
    Ack(replicate p b @ replicate q (~b)) ||
    Reply(~b))
 | (EX b m p q . P = Sending(~b) || Trans(replicate m (~b)) ||
    Ack(replicate p b @ replicate q (~b)) ||
    Replaying(~b))}"

"BSdeliver == {(P, Q) . Q = delBuff &
  ((EX b m p . P = Send(~b) || Trans(replicate m (~b)) ||
    Ack(replicate p b) || Deliver(~b))
 | (EX b m p . P = Sending(~b) || Trans(replicate m (~b)) ||
    Ack(replicate p b) || Deliver(~b))
 | (EX b m n p . P = Send(~b) ||
    Trans(replicate m (~b) @ replicate n b) ||
    Ack(replicate p b) || Reply(b))
 | (EX b m n p . P = Send(~b) ||
    Trans(replicate m (~b) @ replicate n b) ||
    Ack(replicate p b) || Replaying(b))
 | (EX b m n p . P = Sending(~b) ||
    Trans(replicate m (~b) @ replicate n b) ||
    Ack(replicate p b) || Reply(b))
 | (EX b m n p . P = Sending(~b) ||
    Trans(replicate m (~b) @ replicate n b) ||
    Ack(replicate p b) || Replaying(b))}"

"BS_ABP == BSaccept Un BSdeliver"
```

To show that `BS_ABP` is indeed a bisimulation wrt. `{accept, deliver}` we follow our usual scheme. As a typical example consider the case where the ABP performs a strong `accept` transition. We have to prove the obligation, if $(P, Q) \in \text{BS_ABP}$ and $P \xrightarrow{\text{accept}} P'$, then there exists a Q' s.t. $Q \xrightarrow{\text{accept}} Q'$, and $(P', Q') \in \text{BS_ABP}$. Out of the six subrelations in `BSaccept`, differing in the shape of P , Isabelle automatically extracts the first two as those in which P can do an `accept`. It remains to show that in both cases the resulting process P' fits the shape of P in the third and fourth subrelations of `BSdeliver`, the

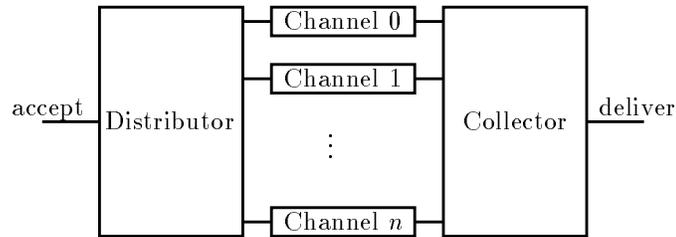


Fig. 2. A Specification of the SWP

difficulty being that the lists of messages in the channels look differently from those in $\text{BS}_{\text{accept}}$. However, once provided with the necessary theorems about finite lists, Isabelle manages to complete the proof fully automatically.

Another interesting example is the reverse case where $Q \text{ -}[\text{accept}]\text{ -} > Q'$, and $P \text{ =}[\text{accept}]\text{ => } P'$. For the third through sixth case of $\text{BS}_{\text{accept}}$ the user has to provide suitable sequences of weak transitions leading to the acceptance of a new message. In all of the cases we can apply the following scheme: remove all messages from **Trans** and **Ack** (that this is possible can be shown by an induction on the length of the lists stored in the channels), then have the replier transmit an acknowledgement to the sender, and finally execute the **accept** transition.

For the invisible transitions of the ABP we essentially have to show that they yield derivatives that still lie within $\text{BS}_{\text{accept}}$ or $\text{BS}_{\text{deliver}}$, respectively. As for each of the processes there are several possibilities, we examine each of the twelve cases separately. Note, moreover, that a simultaneous treatment of all the cases may exceed the capacity of the prover, as also the hypothetical cases like “component **Ack** communicates with component **Trans**” have to be considered, resulting in an exponential blow-up of cases. Again, provided with the necessary theorems about lists, Isabelle proves the cases fully automatically.

The proof script contains about 800 lines. As a large part of it consists of theorems about the finite lists used in the channels, some experience with theorem provers is necessary to set up the proofs. The bisimulation part itself contains a bit more than 400 lines, and can be set up within a few days by a user experienced both in the bisimulation proof method and theorem proving. Notice, however, that this is only possible if the concept of the proof has already been clear before Isabelle is brought into play.

4.3 A Specification of the Sliding Window Protocol

In [PS88] a specification of the Sliding Window Protocol (SWP) is presented, given by the parallel composition of n communication lines which in turn use the ABP on faulty channels. Figure 2 gives a schematical view of the system. Incoming messages are cyclically distributed to the communication lines by a *distributor* module, and are recollected and delivered by a *collector* module. The system specifies the behaviour of an SWP with input and output windows of

equal size. A far simpler specification consists of an $(n + 2)$ -place buffer, if n is the number of parallel channels in our implementation (the distributor and the collector contribute with one place each).

We now reason that an implementation using n copies of the ABP and an $(n + 2)$ -place buffer are observation equivalent. Applying the compositionality of observation equivalence, we can use one-place buffers instead of the n copies of the ABP. We split the proof into three parts, again exploiting the compositionality of observation equivalence.

This time we cannot abstract from data, as the system need not deliver a message before accepting a new one. We have to guarantee that messages are not swapped. The one-place buffers EB^2 are thus of the following form, where ci_i and co_i are denoted by $ci\{i\}$ and $co\{i\}$, respectively:

```
"EB{i, None} - [ci{i}#x]-> EB{i, Some x}"          (* accept *)
"EB{i, Some x} - [co{i}<x>-> EB{i, None}"          (* deliver *)
```

Distributor and collector possess parameters h and l telling to which of the buffers a message is to be sent, or from which one it is to be taken. Each such transition increments the parameters by 1 modulo the number of buffers, yielding the cyclic behaviour of distributor and collector.

```
"D{n, h, None} - [ca#x]-> D{n, h, Some x}"        (* accept *)
"D{n, h, Some x} - [ci{h}<x>-> D{n, h {+n} 1, None}" (* distr *)

"C{n, l, None} - [co{l}#x]-> C{n, l {+n} 1, Some x}" (* collect *)
"C{n, l, Some x} - [cd<x>-> C{n, l, None}"         (* deliver *)
```

In the sequel we discuss the three parts of the proof:

(1) We need a finite representation of the n one-place buffers put in parallel. They can be described by a single component AB containing an array with n elements, one for the place of each buffer.

```
"[ i < length xs ; xs!i = None ] ==>
  AB{xs} - [ci{i}#x]-> AB{xs [i := Some x]}"      (* accept *)
"[ i < length xs ; xs!i = Some x ] ==>
  AB{xs} - [co{i}<x>-> AB{xs [i := None]}"        (* deliver *)
```

The first rule reads as follows: for all positions numbered $0 \leq i < n$ (with $n = \text{length } xs$), if the position is empty ($xs!i = \text{None}$), then AB can read a value on connection $ci\{i\}$ and store it in place i . The second rule is the corresponding rule for destructive output.

In order to show that a parallel composition of n one-place buffers and an array buffer of size n are observation equivalent wrt. $\{ci\{i\} \mid 0 \leq i < n\} \cup \{co\{i\} \mid 0 \leq i < n\}$, we show that for every array of length n , there is a bisimulation containing $(EB\{(n), \text{None}\} \parallel AB\{\text{None}^n\}, AB\{\text{None}^{n+1}\})$:

```
"BS_BuffInduct == {(P, Q) .
  (EX xs x . P = (EB{(length xs), x}) || AB{xs} &
    Q = AB{xs @ [x]})}"
```

² EB stands for 'element buffer', as opposed to the 'array buffer' AB introduced later.

Exploiting the compositionality of observation equivalence, we can conclude by induction on the number of parallel components that

$$\text{EB}\{0, \text{None}\} \parallel \dots \parallel \text{EB}\{n - 1, \text{None}\} \approx_{\{ci\{i\}, co\{i\}\}} \text{AB}\{\text{None}^n\}.$$

(2) As a consequence of (1) our implementation reduces to a system consisting of a distributor, a collector, and an array buffer **AB** of size n . We proceed by comparing this system to another system given by a variation of an n -place buffer **BP**, and two barrier one-place buffers, one attached to its ‘front’ (**FB**), and another attached to its ‘back’ (**BB**). Internally the n -place buffer is organized like the array buffer, yet it possesses only one input and one output connection, and stores and retrieves messages in a cyclic order; the parameters h and l indicate where to store messages and from where to retrieve them.

```
"cs!h = None ==> BP{h, l, cs} -[ci#{x}]->
    BP{h {+length(cs)} 1, l, cs[h := Some(x)]}"      (* accept *)
"cs!l = Some(x) ==> BP{h, l, cs} -[co<x>]->
    BP{h, l {+length(cs)} 1, cs[l := None]}"        (* deliver *)

"FB{None} -[ca#{x}]-> FB{Some x}"                    (* accept *)
"FB{Some x} -[ci<x>]-> FB{None}"                    (* deliver *)

"BB{None} -[co#{x}]-> BB{Some x}"                    (* accept *)
"BB{Some x} -[cd<x>]-> BB{None}"                    (* deliver *)
```

We can show that $D\{n, h, \text{None}\} \parallel \text{AB}\{\text{None}^n\} \parallel C\{n, l, \text{None}\}$ and $\text{FB}\{\text{None}\} \parallel \text{BP}\{h, l, \text{None}^n\} \parallel \text{BB}\{\text{None}\}$ are observation equivalent wrt. $\{ca, cd\}$ by exhibiting the following bisimulation relation:

```
"BS_SWP == {(P, Q) . (EX xs x y h l .
    h < length xs & l < length xs &
    P = (FB{x}) || (BP{h, l, xs}) || BB{y} &
    Q = (D{length xs, h, x}) || (AB{xs}) || C{length xs, l, y})}"
```

(3) To complete the proof we have to show that **BP** behaves like an n -place buffer, **nB**, modelled as follows:

```
"length(s) < n ==> nB{n, s} -[ci#{x}]-> nB{n, s @ [x]}"
    "nB{n, x # s} -[co<x>]-> nB{n, s}"
```

A list $\text{None}^k \circ s \circ \text{None}^l$ stored in **BP** is reflected in **nB** by a list \hat{s} (see **BS1_1** below); \hat{s} is obtained from s by mapping all elements **Some x** to x (s does not contain **None** anyway); a list $s_1 \circ \text{None}^k \circ s_2$ in **BP** is reflected in **nB** by $\hat{s}_2 \circ \hat{s}_1$ (see **BS1_2** below). The bisimulation relation looks as follows:

```
"BS1_1 == {(P, Q) . (EX n cs h l .
    P = nB{n, map the cs}
    & Q = BP{h, l, replicate l None @ cs @ replicate (n - h) None}
    & list_all (% x . x ~= None) cs
    & h < n
    & l + length cs = h)}"
```

```

"BS1_2 == {(P, Q) . (EX n cs1 cs2 h l .
    P = nB{n, map the cs2 @ map the cs1}
    & Q = BP{h, l, cs1 @ replicate (l - h) None @ cs2}
    & list_all (% x . x ~= None) cs1
    & list_all (% x . x ~= None) cs2
    & length cs1 = h
    & h <= l & l < n
    & l + length cs2 = n)}}
"BS_nBuffer == BS1_1 Un BS1_2"

```

The proof script for the three bisimulations verifying the SWP contains about 600 lines, and has been set in less than two weeks. The proofs of (1) and (2) are rather straightforward, as P and Q behave similarly. Isabelle deduces the proofs automatically without the user having to split them into single theorems covering the obligations. Note that the weak transitions are directly derivable from strong ones applying the rules TE and SW (see Section 3), thus need not be given by the user. Also, almost no additional results about the data types have to be provided by the user. The most challenging part concerning the mechanization is (3), as here cyclic structures are mapped to linear lists. The corresponding theorems make up for nearly two thirds of the proof. For these proofs certain expertise in theorem proving is indispensable.

5 Discussion

In the previous section we have presented a mechanization of the verification of communication protocols in a process-algebraical framework based on exhibiting bisimulation relations. In this section we discuss several questions about our approach, going from the more general (is a bisimulation framework appropriate?) to the more concrete (how to further improve our techniques).

Are bisimulation techniques suitable? Bisimulations are often argued to be too discriminating for many practical applications. In the area of communication protocols this seems to be a lesser problem. Due to the rather deterministic behaviour of the specifications of communication protocols, observation and fair testing equivalence [NC95] — and sometimes even trace equivalence [Mil89] — coincide, no matter how the implementations of the protocols look like. Thus, in these cases one can profit from the bisimulation proof methodology to show the, usually less discriminating, notions of testing or trace equivalence.

Bisimulation equivalence does not preserve liveness properties since, e.g., one cannot infer from Q being divergence-free that necessarily P is so too, even if they are weakly bisimilar. However, in the area of communication protocols nondeterminism often models probabilistic choice (a message can be lost with a certain probability), and so it is often reasonable to assume that the system does not remain in any τ -loop indefinitely (as for instance in the ABP). Under this assumption bisimulation preserves liveness properties. It should also be mentioned that our approach can be extended to stronger bisimulation-like equivalences preserving liveness properties (see, for instance, [Wal90]).

Comparison with algebraic techniques. Algebraic techniques are generally considered more elegant. Furthermore, it has been claimed that they succeed in cases where it is hard to find a bisimulation relation [GS96]. Their main drawback with respect to our approach is that they require deep insight into a proof system for bisimulation; on the contrary, exhibiting a bisimulation relation requires only good intuition about the system. A further point is that algebraic techniques usually require the transformation of a system into its normal form applying expansion, which in the presence of parallel compositions leads to an explosion of the size of the process. The degree of the explosion corresponds to the number of proof obligations the system produces in a bisimulation relation. A direct approach has the advantage that the explosion problem can be attacked by splitting the proof obligations, as we have done it in Section 4.2.

Keeping bisimulations manageable. Keeping the size of relations manageable is an important problem of our approach. Notice, for instance, that the description of `BS_ABP` already takes almost a page. A first solution is to use the compositionality of observation equivalence. In our third example we were able to replace the ABP channels by one-place buffers. Without this the bisimulation would have been unmanageable. Furthermore — though we have not used them — there exist various ‘up to’ techniques that can be exploited to reduce the size of the relations [MS92,San95]. ‘Up to’ techniques combine the direct approach with algebraic reasoning.

Dealing with data structures. Although our approach does not require to master a proof system for bisimulation, it still requires a lot of expertise in theorem proving, as usually the systems to be verified manipulate data. Proving simple facts about the data structures of a system (list, stacks, etc.) may amount to more than half of the interaction with the theorem prover. At this point the user has to decide whether to perform the full proof, or whether to provide certain necessary theorems as unproved axioms. Usually the properties of the data structures are rather straightforward, and so the proof does not lose much credibility.

Improving the approach. In our case study we have modelled all transition systems from scratch, slightly modifying the definition of observation equivalence in order to avoid a restriction (or hiding) operator, and projecting the labels either on the signals or on the use of connections. A formalization of a framework for further proofs would of course have to contain a restriction operator, and to implement a general definition of observation equivalence. Further there would have to be a transition rule lifting the user-defined axioms to the transition systems. We have implemented such a prototypical framework — yet still with restriction integrated in the definition of bisimilarity — and have transferred some simple proofs from Section 4 to it.

Acknowledgements: We would like to thank Daniel Hirschhoff and two anonymous referees for very helpful comments.

References

- [BG93] M. Bezem and J. F. Groote. A formal verification of the alternating bit protocol in the calculus of constructions. Logic Group Preprint Series 88, Dept. of Philosophy, Utrecht University, 1993.
- [BK85] J. A. Bergstra and J. W. Klop. Verification of an alternating bit protocol by means of process algebra. In *Mathematical Methods of Specification and Synthesis of Software Systems '85*, volume 215 of *LNCS*. Springer, 1985.
- [BSW69] K. A. Bartlett, R. A. Scantlebury, and P. T. Wilkinson. A note on reliable full-duplex transmission over half-duplex links. *Comm. of the ACM*, 12(5):260–261, May 1969.
- [BW90] J. Baeten and W. Weijland. *Process Algebra*. Cambridge University Press, 1990.
- [Gim96] E. Gimenez. An application of co-inductive types in Coq: Verification of the alternating bit protocol. In *Proc. TYPES'95*, volume 1158 of *LNCS*, pages 135–152. Springer, 1996.
- [GS95] J. F. Groote and J. G. Springintveld. Focus points and convergent process operators. Logic Group Preprint Series 142, Dept. of Philosophy, Utrecht University, 1995.
- [GS96] J. F. Groote and J. G. Springintveld. Algebraic verification of a distributed summation algorithm. Technical Report CS-R9640, CWI, Amsterdam, 1996.
- [HM98] T. Hardin and B. Mammass. Proving the bounded retransmission protocol in the pi-calculus. In *Proc. INFINITY'98*, 1998.
- [Mil89] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.
- [MS92] R. Milner and D. Sangiorgi. The problem of weak bisimulation up-to. In *Proc. CONCUR'92*, volume 630 of *LNCS*. Springer, 1992.
- [Nam97] K. Namjoshi. A simple characterization of stuttering bisimulation. In *Proc. FSTTCS'97*, volume 1346 of *LNCS*, pages 284–296. Springer, 1997.
- [NC95] V. Natarajan and R. Cleaveland. Divergence and fair testing. In *Proc. ICALP'95*, volume 944 of *LNCS*, pages 648–659. Springer, 1995.
- [NS94] T. Nipkow and K. Slind. I/O automata in Isabelle/HOL. In *Proc. TYPES'94*, volume 996 of *LNCS*, pages 101–119. Springer, 1994.
- [Pau93] L. C. Paulson. Isabelle's object-logics. Technical Report 286, University of Cambridge, Computer Laboratory, 1993.
- [Pau94] L. C. Paulson. *Isabelle: a generic theorem prover*, volume 828 of *LNCS*. Springer, 1994.
- [Plo81] G. Plotkin. Structural operational semantics. Technical report, DAIMI, Aarhus University, 1981.
- [PS88] K. Paliwoda and J. Sanders. The sliding-window protocol. Technical Report PRG-66, Programming Research Group, Oxford University, March 1988.
- [San95] D. Sangiorgi. On the proof method for bisimulation. In *Proc. MFCS'95*, volume 969 of *LNCS*, pages 479–488. Springer, 1995.
- [Sne95] J. L. A. Snepscheut. The sliding-window protocol revisited. *Formal Aspects of Computing*, 7:3–17, 1995.
- [Wal90] D. Walker. Bisimulation and divergence. *Information and Computation*, 85(2):202–241, 1990.