A case study on different modeling approaches based on model checking
- verifying numerous versions of the alternating bit protocol with SMV

January 23, 1995

Armin Biere[1*] Alexander Kick[2*]

[1]Institut für Logik, Komplexität und Deduktionssysteme,
[2]Lehrstuhl Informatik für Ingenieure und Naturwissenschaftler,
Universität Karlsruhe, Am Fasanengarten 5,D-76128 Karlsruhe, Germany
Email: {`armin,kick`}`@ira.uka.de`

Abstract

Recently, outstanding results have been achieved in the formal verification of concurrent systems by model checking techniques. In this paper we report our experience with SMV, a symbolic model verifier, applied to a communication protocol, the alternating bit protocol. We investigated different approaches of modeling the alternating bit protocol in SMV. We describe the problems encountered because of the restrictions of SMV. As a consequence, we call for a more general language for model checking, which both overcomes these disadvantages of SMV and enhances the possibility of optimizations, and more specific input languages on top of it, easing the application of model checking for the end user.

---

[*] Supported by DFG Vo 287/5-2

1 Introduction

Model checking [Clarke *et al.*, 1993] has been successfully applied to the verification of large and complex systems. This has been made possible mainly by the introduction of OBDD based techniques [Bryant, 1986]. How model checking functions in general is well explained in [Clarke *et al.*, 1993] and [Zucker, 1993]. SMV is a tool for model checking.

SMV [McMillan, 1993] has been developed for sequential circuit verification. In order to evaluate the usefulness of SMV for protocol verification (for other case studies in symbolic model checking see [Gopalakrishnan *et al.*, 1994]), to discover other deficiencies of SMV and to learn about the appropriateness of model checking [Clarke *et al.*, 1993] in general we tried to verify numerous versions of the alternating bit protocol.

The major outcome of our investigation was the demand for a more general language in which to describe Kripke models (implementations) and specifications. We claim that the μ-calculus [Burch *et al.*, 1990] is this appropriate language and that more specific interfaces (languages such as SMV, state charts, process algebras) should be built on top of the μ-calculus model checker.

The rest of the paper is structured as follows. In Section 2 we show how the boolean function representation of a reactive system can be composed from the boolean function representation of its components. In Section 3 we describe some peculiarities of the SMV system. In Section 4 we investigate different possibilities in verifying the alternating bit protocol: we verify it at different levels of abstractions and different ways of description for both the interleaving and synchronous execution model. In Section 5 we draw some general conclusions about protocols. Due to the deficiencies of SMV in modeling protocols on a high level we not only call for specific input languages, one of which is presented in Section 6, but go even further and call for a more general description language, the μ-calculus, into which the specific input languages can be translated in Section 7.

2 Representing Kripke models as functions with boolean range and component states as domain

As already mentioned, in SMV Kripke models are represented as boolean functions, which have an efficient data structure, the OBDDs. Below, we repeat how this is done for boolean variables [Burch *et al.*, 1994] and show how we can obtain a similar representation for the case of components that have a finite number of states.

2.1 With boolean variables only

Components of sequential circuits have only two possible states: 0 or 1. The state of the whole system can therefore be represented as boolean vectors $\{0,1\}^n$, the transition relation R of the whole system as a boolean function f operating on vectors of length 2n with the property

$$f(x_0,\ldots,x_n,x_0^{'},\ldots,x_n^{'}) = 1 \Leftrightarrow (x_0,\ldots,x_n,x_0^{'},\ldots,x_n^{'}) \in R$$

2.1.1 Representing R as a disjunction of conjuncts where each conjunct represents an element of R

Thus, f can be constructed in a very simple way:

$$f(V,V^{'}) = \bigvee_{(x_0,\ldots,x_n,x_0^{'},\ldots,x_n^{'})\in R} (\bigwedge_{i=0}^{n}(1 \Leftrightarrow x_i)\neg v_i + x_i v_i) \wedge (\bigwedge_{i=0}^{n}(1 \Leftrightarrow x_i^{'})\neg v_i^{'} + x_i^{'} v_i^{'})$$

Example 2.1
$V = \{v_0, v_1\}, R = \{(0,0,0,1),(0,1,0,1)\} \Rightarrow f(v_0,v_1,v_0^{'},v_1^{'}) = \neg v_0 \wedge \neg v_1 \wedge \neg v_0^{'} \wedge v_1^{'} \vee \neg v_0 \wedge v_1 \wedge \neg v_0^{'} \wedge v_1^{'}$

2.1.2 Representing R as what changes and what does not change

Above we have constructed the boolean function from the transition relation of the whole system. Below, we construct the boolean transition function for the whole system from the boolean functions representing the transition relations of the components (f_i). The g_i are boolean functions operating on V, the boolean variables representing the components, which determine the output of component i.

- synchronous circuits:
$$f(V,V^{'}) = \bigwedge f_i(V,V^{'})$$
$$f_i(V,V^{'}) = v_i^{'} \Leftrightarrow g_i(V)$$
- asynchronous circuits:
$$f(V,V^{'}) = \bigwedge f_i(V,V^{'})$$
$$f_i(V,V^{'}) = (v_i^{'} \Leftrightarrow g_i(V)) \vee (v_i^{'} \Leftrightarrow v_i)$$
- interleaving:
$$f(V,V^{'}) = \bigvee f_i(V,V^{'})$$
$$f_i(V,V^{'}) = (v_i^{'} \Leftrightarrow g_i(V)) \wedge \bigwedge_{j\neq i}(v_j^{'} \Leftrightarrow v_j)$$

  Note that the last conjunct is very sensitive for BDDs without a special variable ordering.

2.2 With variables with finite domains (which can be represented as boolean vectors)

If we have a more abstract description of components of a system these components can be automata with a small number of states. Variables ($c_i \in C_i$) represent independent components. The states of the whole system can be described as tuples of the component states $(x_0, \ldots, x_n) \in C_0 \times \ldots \times C_n$, the transition relation R ($R \subseteq C_0 \times \ldots \times C_n \times C_0 \times \ldots \times C_n$) as a function $f\colon C_0 \times \ldots \times C_n \times C_0 \times \ldots \times C_n \mapsto \{0, 1\}$ where

$$f(x_0, \ldots, x_n, x_0^{'}, \ldots, x_n^{'}) = 1 \Leftrightarrow (x_0, \ldots, x_n, x_0^{'}, \ldots, x_n^{'}) \in R$$

2.2.1 Representing R as a disjunction of conjuncts where each conjunct represents an element of R

Let C be the set of variables $\{c_0, \ldots, c_n\}$.

f can be constructed similarly to the boolean case:

$$f(C, C^{'}) = \bigvee_{(x_0,\ldots,x_n,x_0^{'},\ldots,x_n^{'}) \in R} (\bigwedge_{i=0}^{n} c_i = x_i) \wedge (\bigwedge_{i=0}^{n} c_i^{'} = x_i^{'})$$

In the interleaving case this formula can be modified as follows.

Non communicating components In this case the states of all other components do not matter for the transition of one component. The above formula simplifies to

$$f(C, C^{'}) = \bigvee_{i=0}^{n} \bigvee_{k=0}^{m_i} c_i = x_{k_1} \wedge c_i^{'} = x_{k_2} \wedge \bigwedge_{j \neq i} c_j^{'} = c_j$$

where the second or ranges over the number of states (m_i) in which a component can be.

Communicating components The future state of a component is determined by the components with which it communicates. The communicating components can change at the same time.

$$f(C, C^{'}) = \bigvee_{i=0}^{n} \bigvee_{k=0}^{m_i} (c_i = x_{k_0} \wedge c_{i_0} = x_{k_1} \wedge \ldots \wedge c_{i_m} = x_{k_{m+1}}$$
$$\wedge c_i^{'} = x_{k_{m+2}} \wedge c_{i_0}^{'} = x_{k_{m+3}} \wedge \ldots \wedge c_{i_m}^{'} = x_{k_{2m+2}} \wedge \bigwedge_{j \notin \{i, i_0, \ldots i_m\}} c_j^{'} = c_j)$$

where c_{i_j} are exactly the components with which c_i communicates.

In such a description we do not have any problems in describing non-determinism and communication actions (for each non-deterministic action just one more disjunct above) in contrast to SMV as we will see below.

2.2.2 Representing R as what changes and what does not change

In all three cases below $\bigwedge_{j=0}^{m_i} g_{i_j}(C)$ is a tautology and for fixed i all $g_{i_j}(C)$ are mutually exclusive. The $g_{i_j}(C)$s denote the preconditions for a change of a certain component c_i. To ensure that there is always a true precondition we can take $\neg \bigvee_{j=0}^{m_i-1} g_{i_j}(C)$ as the last precondition for $c_i^{'} = c_i$.

For a given state and given component i $(x_0, \ldots, x_n)$ *exactly one* g_{i_j} delivers 1. Always one, because the next state of a component is always determined by the previous state of the system and $c_i^{'}$ would be left unspecified in the next state, i.e., it would be a random state, otherwise (As in [Clarke *et al.*, 1993] we also consider only Kripke models with total transition relation. Especially all runs (paths) are considered to be infinite.); not more than one because otherwise the transition relation of the component would be false (empty). Nondeterminism can be represented by a disjunct on the right side of →. If a state has several ingoing arcs this state will appear on the right side of → in several conjuncts.

- synchronous:

  In a synchronous circuit all components proceed at the same time.

$$f(C, C^{'}) = \bigwedge f_i(C, C^{'})$$

$$f_i(C, C^{'}) = \bigwedge_{j=0}^{m_i} (g_{i_j}(C) \to \bigvee_{k=0}^{n_j} c_i^{'} = x_k)$$

- asynchronous:

$$f(C, C^{'}) = \bigwedge f_i(C, C^{'}) \wedge K(C, C^{'})$$

$$f_i(C, C^{'}) = (\bigwedge_{j=0}^{m_i} (g_{i_j}(C) \to \bigvee_{k=0}^{n_j} c_i^{'} = x_k)) \vee (c_i^{'} = c_i)$$

$$K(C, C^{'}) = \bigwedge_{\substack{\text{possible} \\ \text{communications}}} (c_a = x_{a_1} \wedge c_a^{'} = x_{a_2}) \leftrightarrow \ldots \leftrightarrow (c_e = x_{e_1} \wedge c_e^{'} = x_{e_2})$$

When components communicate the participating components have to transition at the same time and *not* transition individually. K ensures that communication transitions occur at the same time. E.g., $(c_a = x_{a_1} \wedge c_a^{'} = x_{a_2})$

in the definition of K is one transition of a component. The $\leftrightarrow$ ensures that the transitions participating in a communication occur only simultaneously. This formula allows the description of non-deterministic choice for one c_i whose non-deterministic transition has to occur at the same time as the transition of another component.

A direct description with a formula as in 2.2.1 is prohibitive: if there are two components with n and m states we could possibly have $n \cdot m$ disjuncts. The formula just presented is therefore more convenient.

- interleaving:

$$f(C, C') = \bigvee f_i(C, C')$$

$$f_i(C, C') = (\bigwedge_{j=0}^{m_i} (g_{i_j}(C) \rightarrow \bigvee_{k=0}^{n_j} c_i' = x_k)) \wedge \bigwedge_{j \neq i} (c_j' = c_j)$$

This formula is similar to the boolean case. Note, however, that this formula above can only be used if there are no communication transitions. Generally, in the interleaving case the direct description in 2.2.1 is therefore the easiest.

3 The SMV tool

The SMV tool is well described in [McMillan, 1993] and [McMillan, 1992]. Here, we just give a short overview over the internal functioning of SMV and consider two points concerning the input language which attracted our attention.

3.1 Internal functioning of SMV

The next and init statements describing the transition relation of the various modules are first translated into a tree like data structure. Vectors are translated into boolean variables. The tree representations of each module are then translated into an OBDD representation for the transition relation of the product automaton. Thus, SMV implements global model checking. The CTL specification, finally, is checked by fixpoint iterations on the OBDDs.

3.2 Non-determinism in SMV

Non-determinism can be represented in SMV by assigning a set of possible states to the next state of a variable. To ensure fairness of such a nondeterministic transition we have to put a fairness constraint into the SMV program. We could put fairness on all the states between which there is a non-deterministic choice. However, in general, it suffices to put fairness on those states leaving a loop.

Even more: One fairness constraint on a state outside internal loops suffices (e.g. the start state of the sender is reached infinitely often) since this fairness constraint can only be fulfilled if all other nondeterministic transitions are fair (`FAIRNESS running` is needed in addition, see below). Such a fairness constraint ensures liveness at the same time. To ensure that the model we deal with is not empty we always have to check that there exist infinite paths ($EG \quad true$ in the specification).

However, we run into problems when we want to change the next states of two variables in different modules at the same time (This is only possible in the strict synchronous mode.). The complicated SMV program where channels are represented as modules is an example for this problem (see appendix A.4).

Note that putting fairness on all states allows to find non-reachable states in the protocol (and thus allows the minimization of protocols).

3.3 Total case statement

In the case statement the cases always have to be complete. As a consequence we almost always need the case `1 : state;` as the last in a case statement. Otherwise, SMV will not translate the program.

4 The alternating bit protocol as an example

We tested the advantages and disadvantages of SMV by performing a case study on the alternating bit protocol. In this paper we refer to the description of the alternating bit protocol as it is presented in [Baeten and Weijland, 1990] or [Clarke *et al.*, 1986].

We first describe shortly the alternating bit protocol (ABP), then we investigate different ways of description of the ABP in SMV and model it at different levels of abstractions in various models of execution and types of communication.

4.1 Description of the alternating bit protocol

The alternating bit protocol shall ensure that incoming data is delivered, but also in the right order.

4.1.1 The configuration

The configuration of the six automata is as in Figure 1. SA and RA are sender and receiver of the upper level, respectively. The lower level has to ensure via the alternating bit protocol that the exact sequence of data sent by SA is correctly delivered to RA, i.e. they have to manage the disturbances of the two channels.

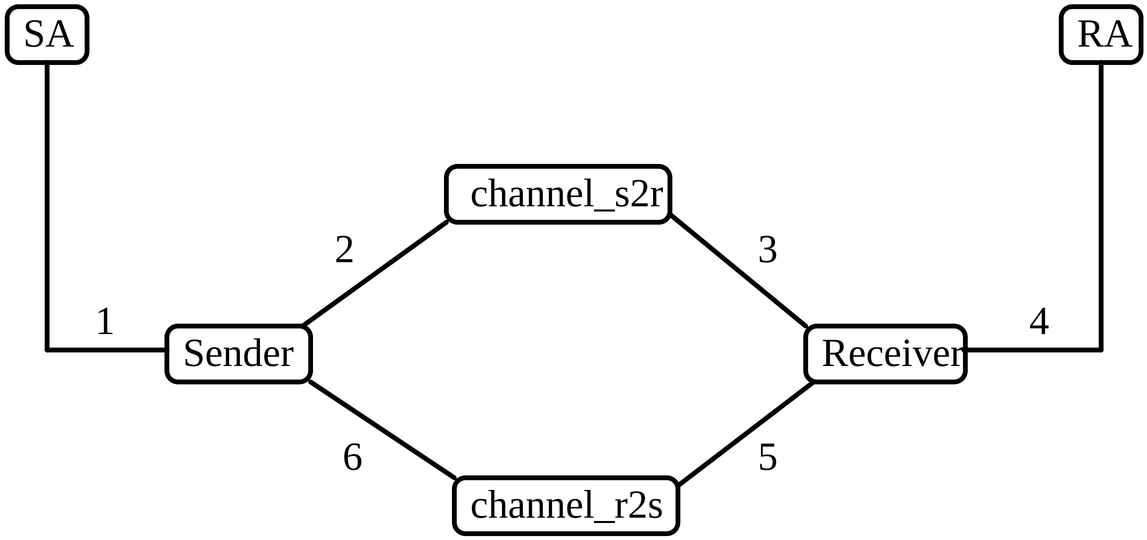

Figure 1: The configuration

4.1.2 The automata, processes

There are six automata: sender, receiver of the two levels, and 2 one-way channels, as can be seen in Figures 2, 3, 4,5,6 and 7. In these figures the transitions are labeled with different types of action. E.g., r1(d) stands for r(ead) data d at port 1. Synchronization is achieved by the fact that read and s(end) with the same port number (e.g., r1(d) and s1(d)) have to occur simultaneously. The product automaton for the whole alternating bit protocol has been constructed for the interleaving semantics in Figure 8. In this figure the states are described in the form (state of sender in upper level, state of sender in lower level, channel from sender in lower level to receiver in lower level, channel from receiver in lower level to sender in lower level, state of the receiver in lower level, state of receiver in upper level).

4.2 Different ways of description of the ABP in SMV

There are two main ways of describing a model in SMV: with next and init statements and with the TRANS statement. We consider both in this subsection.

This and the following subsection are explained in terms of the interleaving model of the ABP.

4.2.1 Standard way of description in SMV

In the standard way of description recommended by the author of SMV the next and init statements and modules are used.

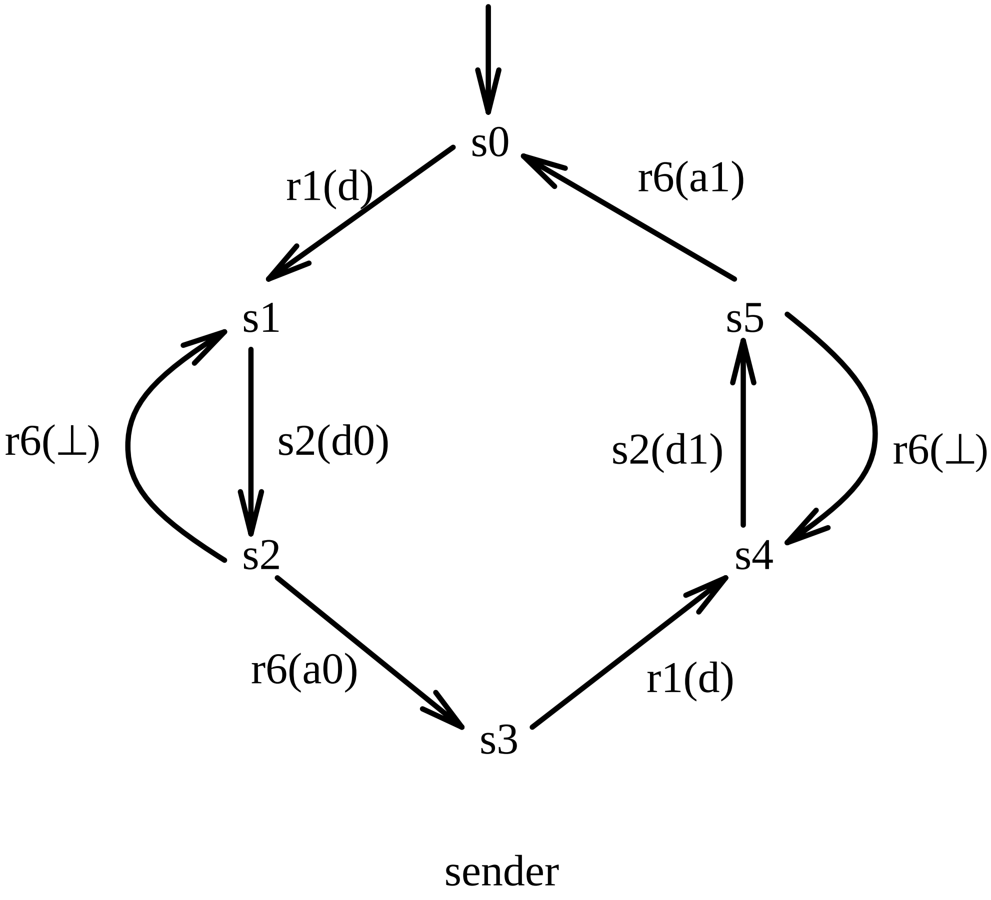

Figure 2: The sender

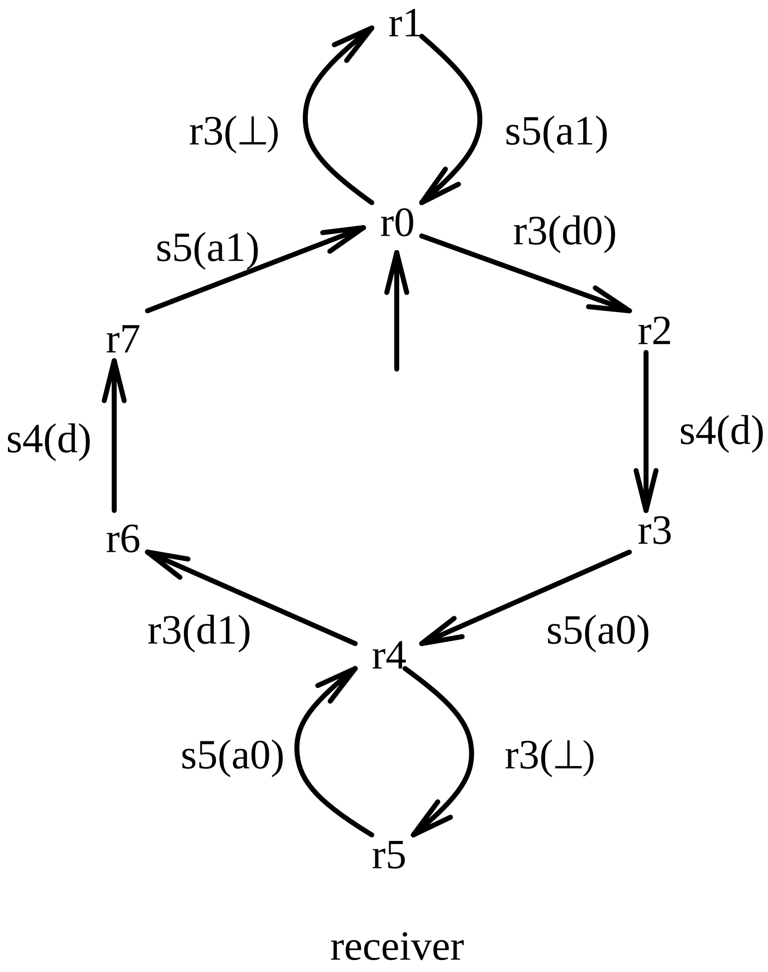

Figure 3: The receiver

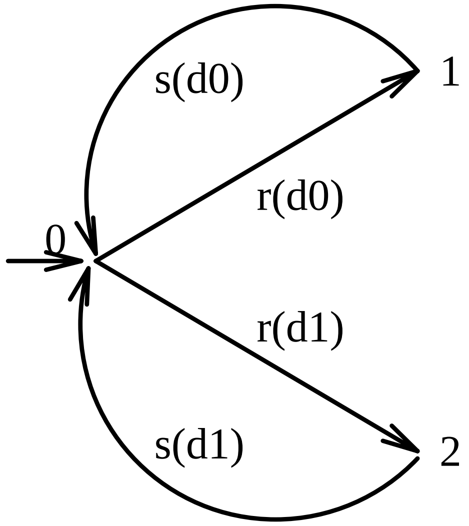

Figure 4: The channel for messages from sender to receiver

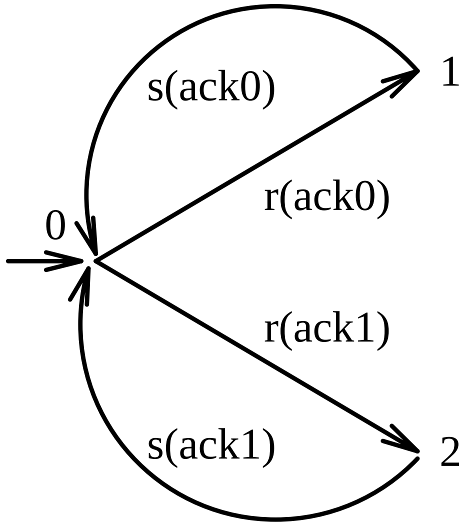

Figure 5: The channel for acknowledgements from receiver to sender

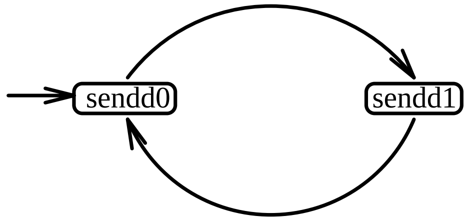

Figure 6: The sender of the upper level

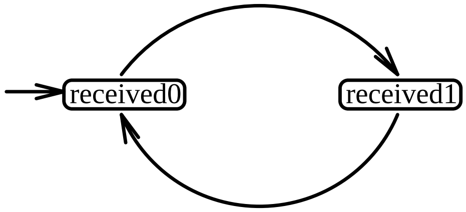

Figure 7: The receiver of the upper level

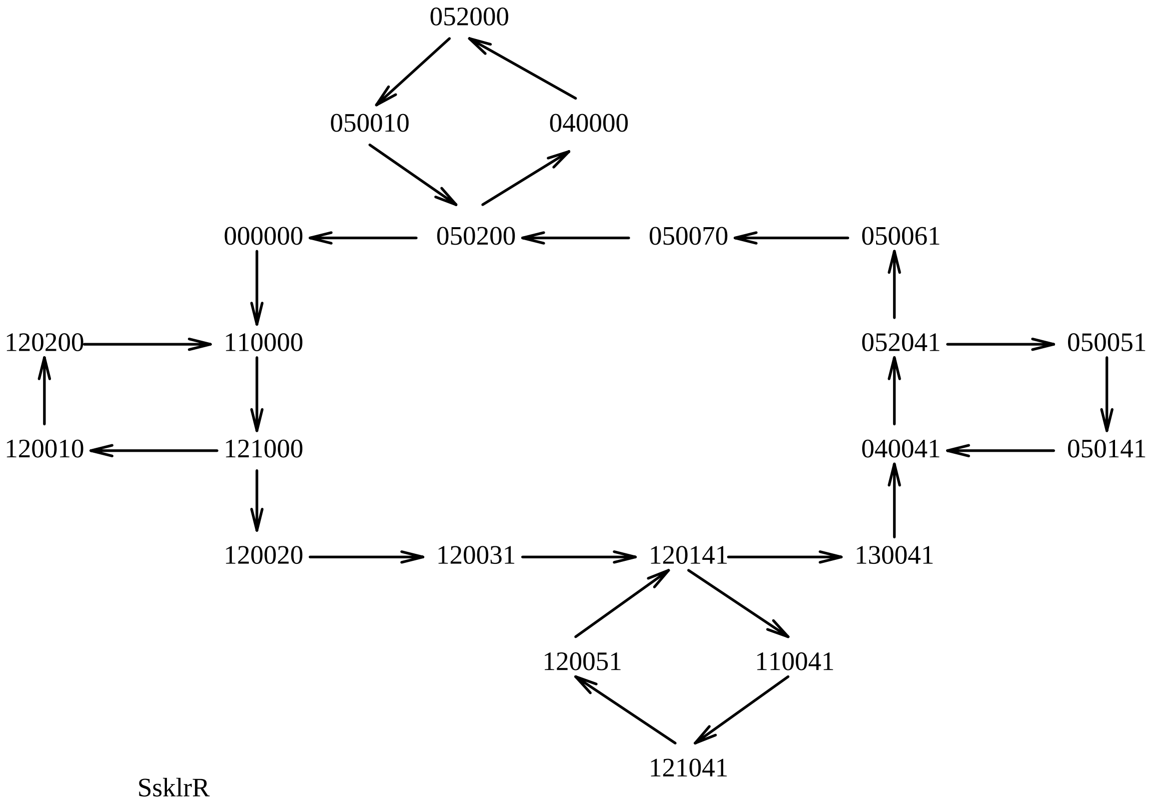

Figure 8: The product automaton of the alternating bit protocol in interleaving semantics

There are two modules: one for the sender and one for the receiver. The channels are modeled as global variables, also sender and receiver of the upper level. As data is sent to the channel, the content of the channel changes at the same time. This is modeled in our SMV program by the simultaneous change of the state of the sender and the corresponding channel. The modeling of the channels and the sender and receiver of the upper level as SMV variables (and not as modules) is only possible because these processes change their state exactly when sender or receiver of the lower level do.

Modeling the possible corruption of data and acknowledgements within the receiver and sender modules is much easier and clearer than having a module for each channel (compare appendix A.1 with A.4).

In the introducing SMV program (cf. appendix A.1) we are careful: There are fairness conditions for every non-deterministic transition and also fairness running.

The specification The protocol has to fulfill the following specification:

- C1: All sent data arrives at the receiver.
- C2: Data is received in the order it was sent. All sequences are of the form $sd0.rd0.sd1.rd1.sd0\ldots$, i.e. data with alternating bit 0 is sent by the sender of the upper level, then data with alternating bit 0 is received by the receiver of the upper layer, ...
- C3: The sender can always send when it wants to.

The informal specification for C2 can subsequently be transformed into a formal CTL specification:

C2
$\Leftrightarrow$
in the beginning nothing else can happen except the transition sd0 $\wedge$
after sd0 nothing else can happen except the transition rd0 $\wedge$
after rd0 nothing else can happen except the transition sd1 $\wedge$
after sd1 nothing else can happen except the transition rd1 $\wedge$
after rd1 nothing else can happen except the transition sd0

Note that the state of SA and RA after the observation path sd0.rd0.sd1.rd1 is the same as in the beginning. The first and last conjunct therefore collapse into one when we translate the conjunction into CTL:

$$\begin{gathered}AG(sa = sendd0 \rightarrow A[sa = sendd0 \quad U \quad ra = received0]) \wedge \\ AG(ra = received0 \rightarrow A[ra = received0 \quad U \quad sa = sendd1]) \wedge \\ AG(sa = sendd1 \rightarrow A[sa = sendd1 \quad U \quad ra = received1]) \wedge \\ AG(ra = received1 \rightarrow A[ra = received1 \quad U \quad sa = sendd0])\end{gathered}$$

This CTL formula is stronger than C2 because of the boundedness of the Until operator in CTL semantics. Because of this property C1 is also captured.

Note that in the synchronous and asynchronous operational semantics we need a different specification for C2. We need to replace the propositions after the Until operator by conjuncts of the form $ra = \ldots \wedge sa = \ldots$. This is necessary because in contrast to interleaving semantics the states of sa and ra could change at the same time in synchronous and asynchronous execution model. This would violate the partial order of events.

$$\begin{gathered}AG(sa = sendd0 \rightarrow A[sa = sendd0 \quad U \quad ra = received0 \wedge sa = sendd0]) \wedge \\ AG(ra = received0 \rightarrow A[ra = received0 \quad U \quad sa = sendd1 \wedge ra = received0]) \wedge \\ AG(sa = sendd1 \rightarrow A[sa = sendd1 \quad U \quad ra = received1 \wedge sa = sendd1]) \wedge \\ AG(ra = received1 \rightarrow A[ra = received1 \quad U \quad sa = sendd0 \wedge ra = received1])\end{gathered}$$

The CTL formula for C3 is:

$$AG((sa = sendd0 \rightarrow EF sa = sendd1) \wedge (sa = sendd1 \rightarrow EF sa = sendd0))$$

The specification strongly depends on the description of the implementation. The same is true for the formulation of fairness. So, great care has to be taken in formulating these. For example, the above specification can be trivially fulfilled if the protocol does not contain any paths fulfilling the fairness conditions. $AG\ldots$ is also true if there are no paths at all. This situation easily occurs if the fairness constraints are not fulfilled. Whether this is the case can be detected by checking for the specification $EG \quad true$.

4.2.2 Representing each transition as conjunct - direct representation with TRANS

Instead of using init and next statements we can encode the transition relation directly as a boolean function as described in section 2.2.1 by using the `TRANS` statement. The appropriate example program appears in appendix A.2.

4.3 Different levels of abstraction

The alternating bit protocol can be modeled at different abstraction levels:

- Sender and receiver of the upper level, sender, receiver of the lower level and the 2 channels are modeled. As an example see the program in appendix A.1.
- Receiver, sender of the lower level are modeled as modules, the 2 channels as variables – sender and receiver of the upper level are not represented. The program in appendix A.3, e.g., has this abstraction level.

  The transitions of the sender and receiver in the upper level always occur at the same time with the appropriate transitions of sender and receiver in the lower level, respectively. So we do not need to represent sender and receiver of the upper level.

  In all interleaving programs except A.1 we reduced the number of fairness conditions and specification formulas to the ones really needed.

  Executing a module which does not change any state when executed also produces a path in SMV (e.g. $r0.r0.r0\dots$). The fairness on s0 thus would not be enough. In this case, the receiver could execute for ever without changing any state, thus making the specification false. This is why we need `FAIRNESS running` for the sender. The fairness constraint on one state of the sender (s0) suffices to make *all* non-deterministic choices fair, also that the receiver is executed infinitely often.

  These two fairness conditions also ensure that the sender continuously sends new messages, thus making formula C2 true.
- Receiver, sender of the lower level and the 2 channels are modeled as modules (see appendix A.4).

4.4 Interleaving, asynchronous or synchronous description

4.4.1 Interleaving, synchronous and asynchronous models cause different verification results!

The difference between synchronous and asynchronous and between synchronous and interleaving execution model should be clear. We will therefore only look into the difference between asynchronous and interleaving execution model.

If there is no interaction of the processes the reachable states of the asynchronous and interleaving models are the same. Otherwise, this is not the case.

To see this, consider two processes P and Q, both having 2 states (p1,p2,q1,q2) and both infinitely alternating between their 2 states. If we allow P to go into p2 only if it is in p1 and Q is in q1, and similar for Q, the states reachable in the interleaving execution model are a strict subset of the reachable states in the asynchronous execution model ((p2,q2) is never reached in the interleaving model) – if checking for the state of the other process and its own action needs an atomic time unit.

By refinement there is no more such difference. Therefore, one has always to bear in mind the interrelation between execution model and how fine the actions of the components are.

For process algebra interleaving is enough since there is no such dependence between components.

4.4.2 Interleaving descriptions: conclusions, comparisons

Our interleaving descriptions in SMV have already been presented in previous sections. Here we draw some conclusions with respect to the different implementations.

The direct representation with variables might be error-prone and cumbersome. The specification where all processes are modules (channels too) has a clear not justifiable overhead, is cumbersome and error-prone. The SMV program where the channels are represented as variables and automata are described with case and next statements is tedious as well because of the simultaneity of transitions with the channels. The latter two representations also fair badly with nondeterminism (because of simultaneity of communication actions, see above). Of these three, the direct representation seems best.

The SMV programs also differ in the time and BDD nodes needed to compute the truth of the specification (see different outputs of SMV in appendix A). Although programs A.2 and A.3 have the same number of variables program A.2 has half of the BDD nodes of program A.3. That the latter program needs two additional variables running for each process in the internal representation and that in program A.2 the transition relation contains only the transitions between the *reachable* states are probably the reason. The case where channels are also modules is even worse.

4.4.3 Synchronous description of the ABP

The above are all descriptions in the interleaving execution model. A synchronous description of a communication protocol does not make sense since different computer stations do not need to have the same tact cycle and same global time. Nevertheless, we tried to describe it in the SMV language to test its expressiveness.

Especially here we had to struggle with the restriction of SMV that no two modules can write on a common variable in conjunction with synchronous processes. Sender and receiver, however, never change the content of the channel at the same time (because of exclusive preconditions (guards)). This, however, should in fact be allowed, since in implementations mutual exclusion has to be ensured only among different writers!

The description is not difficult if we have just one main module. Otherwise, we need a module for each signal. The latter is a tedious description, making things more complicated than the abstract functioning. I.e., we have to misuse the SMV language to get it done.

We now describe in detail our synchronous formulation of the ABP. This means that all proccesse are acting simultaneously – in contrast to interleaving semantics. We already mentioned a problem that arises with this approach. When two processes want to communicate, they have to do this via some global instance which in its simplest form could be modeled by a global variable. But that results in the common problem of shared resources. So there should be some mechanism to ensure mutual exclusion.

The SMV-Language imposes a strong restriction to overcome this problem. It does not allow that synchronous processes (modules) have a common writable variable. But in the case of protocols one often needs the concept of a *signal* that could be sent by one process to another. For example if we have a binary signal `request` which can be communicated from process A to process B then the simplest representation would be a global boolean variable $V_{\texttt{req}}$. The sender (A) wants to set $V_{\texttt{req}}$ and the receiver (B) wants to reset it. So it seems that we run into the same problem that a global variable should be writable by two different processes. But there is a fundamental difference between this case and other mutual exclusion problems. If we distinguish the occurrences of the writing efforts of the sender resp. the receiver by the value of $V_{\texttt{req}}$ then we get the following cases:

$$\begin{aligned} A \text{ wants to write } V_{\texttt{req}} &\iff V_{\texttt{req}} = 0 \\ B \text{ wants to write } V_{\texttt{req}} &\iff V_{\texttt{req}} = 1 \end{aligned}$$

So $V_{\texttt{req}}$ serves itself as a semaphore for enabling writing access to $V_{\texttt{req}}$.

$$\begin{aligned} A \text{ has the privilege of writing} &\iff V_{\texttt{req}} = 0 \\ B \text{ has the privilege of writing} &\iff V_{\texttt{req}} = 1 \end{aligned}$$

The conclusion of this discussion is that the SMV language is not very well suited for describing signalling. It should however be mentioned that the concept of describing a signal in this way can be translated into the SMV language – but only with the drawback of loosing the module concept. In this case the protocol has to be described in one module. So it can not syntactically be checked that the transition relation is implementable by different processes.[†]

[†] However this should be no problem if the SMV language is used as an intermediate language into which descriptions of real implementations are translated and not the other way around. So our synchronous descriptions of the ABP are hiding the danger that they do not represent any implementation at all.

```
MODULE signal
VAR
  sig  :  boolean;
  set  :  boolean;
  reset:  boolean;
ASSIGN
  init(sig) := 0;
  next(sig) :=
    case
      ! sig &   set :  1;
        sig & reset :  0;
      1             :  sig;
    esac;
```

Figure 9: signal module

If we want to be sure that a synchronous description can lead to an implementation we can use the module concept in combination with an additional semaphore for each signal. This method has not only the disadvantage of increasing the number of states but it also considerably complicates the description of the modules. If we want to use such a mechanism, then first of all we should describe a class of signal modules as it is shown in Figure 9.

If the sender wants to set the `set` bit then he must ensure that a previously sent message is not lost. The best way to achieve this is that the sender waits until the `signal` bit is released before he sets the `set` bit. Before he can carry on he has to wait until the signal object has set the `signal` bit. On the other hand if the receiver wants to reset the signal and has set its `reset` bit then he must wait until this happens. So this scheme works as a 1 bit queue.

Such an implementation would be overloaded by instructions to handle correct signalling. For an example of such an awkward description of the ABP see the program in appendix B.4. It has not only an awkward description but it also needs more states and thus results in a longer checking time (compare with table 1 for more details). After all, this approach does not seem appropriate.

One could think of a third method to communicate signals. In this case the directly communicating processes investigate each others state to decide when a signal has been sent (see appendix B.5). This results in some sort of a rendezvous principle because both processes have to wait until the corresponding partner is willing to send resp. to receive. One major drawback of this method is that it has no implementation at all. The only advantage is that it gives the least number of states.

If we do without modules, the signalling can be achieved by global boolean variables. We distinguish the descriptions by the number of involved processes. In the simple case there are only two processes: one for the sender and one for

semantic model	upper layer	lower layer	data	signal model	reachable states	checking time in s
sync.	no	no	yes	investigation	136	< 1
sync.	no	no	yes	global vars	184	< 1
sync.	no	yes	yes	global vars	1220	3
sync.	yes	yes	yes	global vars	10246	??(70)
sync.	no	no	yes	signal module	472	3
sync.	yes	yes	yes	signal module	??	??
interl.	no	yes	no	global vars	22	< 1
interl.	yes	yes	no	global vars	22	< 1
interl.	no	yes	no	channel module	320	< 2

Table 1: Comparison of the different descriptions of the ABP. All tests were run on a Sparc 10 (50Mhz) with 64 MB main memory. SMV was always used with options -r -f, i.e., the reachable states of our programs were always calculated before model checking.

the receiver (appendix B.1). The complexity rises by inventing a lower layer, consisting of two error producing channels (appendix B.2). The third version additionally describes the higher layer that consists of two abstract users of the offered protocol (appendix B.3). In all cases, the transported data consist of one bit.

Because of the different complexity it is not possible to give one specification that all versions have to fulfill. On the contrary, the specifications had to be reworked heavily in order to be correct. The question marks in Tabel 1 indicate that there might be an error in the specification formula or the model. We were tired looking for the error. We include the two models with the question marks in Table 1 nevertheless so that the reader can compare the number of reachable states and include the corresponding global variables program in the appendix so that the reader can get a feeling for how it is written. In the beginning of this research we tried to implement the ABP with signal objects as described above (last line of the synchronous models in Tabel 1). With this version we were not able to generate any results (we could not check it nor generate any counterexamples) when we used a Sparc 10 with 64MB main memory (The program is not included in the appendix.).

From Table 1 one can see that the number of states from the most simple to the most complex description increases roughly by a factor of 10 at each level.

4.4.4 Asynchronous description

Circumventing the restriction of SMV that no two modules can write to a common variable by signal modules does not help in making an asynchronous execution model possible (The representation would be false.). The only way to do the job is one big complicated module where it is difficult to see that it actually represents the implementation.

4.5 Synchronous or asynchronous send between sender and receiver

With buffers sending is asynchronous.

But we can also have a synchronous send and receive between sender and receiver, i.e., the receiver has to receive the message at the same time as the sender sends the message. This is simply obtained by leaving out the channels. This is modeled by Clarke in their CSP like description language for model checking [Clarke *et al.*, 1986]. This can also be described in SMV.

5 General conclusions about protocols

5.1 Similar structure

Since protocols have a similar structure they have a similar representation in SMV. The translation of the communication structure into SMV is the same (e.g. channels as variables), but also most of the specification (E.g., that the sequence of incoming messages is the same as the sequence of outgoing messages.). As a consequence, a special input language for protocol verification would be advantageous.

5.2 The size of a channel

In some protocols it suffices that the size of a channel is just 1, e.g., in the alternating bit protocol. However, this is not a correct model for many other protocols, e.g., sliding window protocols. In this case we may need induction over the size of the channel or size of the sliding window.

When the sender always waits for an acknowledgement until the next data is sent (i.e., sending and receiving alternate: s.r.s.r) then channel size 1 is enough.

5.3 Asynchrous models

For protocols, the asynchronous execution model is most appropriate. Unfortunately, this model is also computationally most expensive and most difficult to represent in SMV.

6 Automatic translation of PA terms into a μ-calculus (or SMV) program

6.1 A special input language for PA specifications for model checking

The translation of PA descriptions into SMV is cumbersome, as we have seen above. Furthermore, to prove properties in PA cannot be recommended and is difficult for large descriptions. Therefore, we demand a special translation for PA terms into CTL. The kind of input language we have in mind is of the following form:

```
MODULE S
    S = S0.S1.S
    S0 = r1(d).s2(d0).T0
    T0 = (r6(1) + r6(⊥)).S0 + r6(0)
    S1 = r1(d).s2(d1).T1
    T0 = (r6(0) + r6(⊥)).S1 + r6(1)

MODULE R
    ...

...

COMMUNICATION
    (s2(d),r2(d))
    (s3(d), r3(d))
    ...

SPECIFICATION
    ...

MODEL
    asynchronous
```

The `MODULEs` are the process descriptions. The pairs below `COMMUNICATION` represent the transitions which have to occur simultaneously. Such an input language allows much simpler descriptions than SMV. In particular, this avoids the hazzle we had with SMV to specify simultaneous transitions of e.g. receiver and channel automaton.

In order to enable a specification in CTL for a PA description one could enhance the PA description with state points - or just using variables (describing the state of a PA process) for specification.

6.2 How to enable a translation of the above language into the μ-calculus or SMV and how to draw advantages from such a translation

An easy way of producing an SMV program from a PA term is to first produce the product automaton from the parallel components and the description of their synchronous interaction (r(d), s(d)) and then to translate the product automaton into a boolean function representation. Such a description would be simpler than with init and next statements in SMV. Note that this is not recommendable since the product automaton can be huge. We should therefore use a direct translation of components with subsequent combination of the translated components by OBDD operations.

The use of more general trees representing the transition relation where the nodes are the components and the arcs to the successor nodes are labeled with the possible values a state component can have should be investigated.

7 Summary

The SMV language was developed mainly for the purpose of verification of sequential circuits. As a consequence, this poses problems for the application to other verification problems.

The main problems with the SMV language are:

- Difficulty in combining non-determinism with simultaneous transitions (communication) of modules.
  Example: In the Amoeba protocol [Mulder, 1990] there is a state of the server interface (TLSN) where there is a non-deterministic choice between 2 receipts and a timeout. Note that the appropriate channel has to be emptied when a datum is received via a certain port. There is no direct way to express (with `ASSIGN`), the non-deterministic choice between the 3 possibilities and at the same time the simultaneity of the emptying of the channels. (It can be done by having an explicit choice variable for *every* such kind of non-deterministic choice. This, however, would be extremely cumbersome!)

- Several processes cannot write to shared variables. This is appropriate for sequential circuits but not for protocols. This has also been considered as disadvantageous by [Gopalakrishnan *et al.*, 1994] and [Campos, 1993]. In [Campos, 1993] it is stated that "support from the definition language in defining and using shared variables would be very useful. The language could generate the control modules for each variable declared shared, and simplify the exchange of information."

- Difficulty in representing an asynchronous execution model. Well, this could be achieved by making each transition non-deterministic, allowing a component to stay in the same state. Another possibility is stuttering [Campos, 1993] by which asynchronous behaviour can be introduced and finer granularity of time can be achieved.

All these disadvantages of SMV make the modeling of many protocols not only notationally extremely tedious and complicated but can also increase the size of the model considerably (see our synchronous examples).

[Gopalakrishnan *et al.*, 1994] state that SMV must be interfaced to design systems (They have developed a Petri-nets interface to SMV.). In [Campos, 1993] it is believed that a "language with a syntax closer to that of a general programming language could increase the efficiency of the verification of programs." We go further and call for a more general description language, the μ-calculus, and more specific languages on top of it. The input language of SMV is one of these: useful for the application to senquential circuit verification. Other such specific languages can be process algebra, state charts or other specification languages for the verification of communication protocols.

This has several advantages:

- The underlying system is much more general and many more things can be investigated. It can thus serve as a tool for experimentation.

- The system can be easily extended to other interfaces. Note that 'misusing' a language for purposes for which it was not defined can result in many specification errors since specifications become less understandable. This can be easily seen in our synchronous specifications, but also in [Campos, 1993]. This is why it is important that specific interfaces for special purpose types of verification can be *easily* added. This will make verification much more convenient and - what is more - less error-prone.

- When translating into the μ-calculus we can ensure the most concise representation of the states and transition relation. Automatic abstraction and many reductions can be performed on the μ-calculus level such as automatic reduction of the number of variables before the translation into the BDD representation. E.g., the program in appendix A.1 could be transformed into the program in appendix A.3, i.e., variables sa and ra could be eliminated.

- Using the μ-calculus as an intermediate language allows the following optimizations. When a CTL formula has been translated into the μ-calculus it is possible to simplify it according to the semantics of the μ-calculus. With the SMV system such an optimization is difficult because model checking is done along the structure of CTL terms (the evaluation is syntax driven!). On the other hand it is possible to enrich the μ-calculus with operators that preserve most of the information that allows SMV to apply special purpose OBDD operations. One example are *modal operators* as in the modal μ-calculus in [Cleaveland, 1990]. These can be evaluated by special purpose OBDD operations and correspond to a next state calculation in a state space search (compare with the '`collapse_bdd`'-function in the SMV system). If the μ-calculus is seen as functional program and not as a logical term such a modal operator corresponds to a functional. So we are looking for a μ-calculus with higher types. These higher types can express information about a transition relation that can not be used by the SMV system. We think that for some examples this approach will result in an even faster model checking algorithm (we do not stress the possibly greater expressivenes of such an enriched μ-calculus as it is the case in [Hungar, 1994]).

References

[Baeten and Weijland, 1990] J. C. M. Baeten and W. P. Weijland. *Process Algebra*, volume 18 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, Cambridge, England, 1990.

[Bryant, 1986] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8), 1986.

[Burch *et al.*, 1990] 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 *5th Annual IEEE Symposium on Logic in Computer Science, Philadelphia*, pages 428 – 439, 1990.

[Burch *et al.*, 1994] J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan, and D. L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(4):401 – 424, April 1994.

[Campos, 1993] S. V. Campos. The priority inversion problem and real-time symbolic model checking. Technical Report CMU-CS-93-125, Carnegie Mellon University, April 1993.

[Clarke *et al.*, 1986] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244 – 263, April 1986.

[Clarke *et al.*, 1993] E. Clarke, O. Grumberg, and D. Long. Verification tools for finite-state concurrent systems. In de Bakker, editor, *A Decade of Concurrency, REX School/Symposium*, volume 803 of *LNCS*, pages 124 – 175. Springer, 1993.

[Cleaveland, 1990] Rance Cleaveland. Tableau-based model checking in the propositional mu-calculus. *Acta Inf.*, 27:725–747, 1990.

[Gopalakrishnan *et al.*, 1994] G. Gopalakrishnan, D. Khandekar, R. Kuramkote, and R. Nalumasu. Case studies in symbolic model checking. Technical Report UUCS-94-009, Department of Computer Science, University of Utah, 1994.

[Hungar, 1994] H. Hungar. Model checking of macro processes. In D. L. Dill, editor, *Comuter Aided Verification, CAV'94*, pages 169–181. Springer-Verlag, 1994.

[McMillan, 1992] K. L. McMillan. The SMV system. Technical report, Carnegie Mellon University, 1992.

[McMillan, 1993] K. L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.

[Mulder, 1990] J. C. Mulder. On the Amoeba protocol. In J. C. M. Baeten, editor, *Applications of Process Algebra*, volume 17 of *Cambridge Tracts in Theoretical Computer Science*, pages 147 – 172. Cambridge University Press, 1990.

[Zucker, 1993] J. Zucker. Propositional temporal logics and their use in model checking. In P. E. Lauer, editor, *Functional Programming, Concurrency, Simulation and Automated Reasoning*, pages 108–116. Springer-Verlag, Berlin, DE, 1993. Proceedings of International Lecture Series 1991-92, McMaster UniversityLecture Notes in Computer Science 693.

A Interleaving model of ABP in SMV

A.1 Sender and receiver of the upper level, sender, receiver and the 2 channels

A.1.1 The SMV program

```
-- interleaving
--
-- sender and receiver of upper level: as variables
-- sender and receiver of lower level: as modules
-- channels: as variables
--
-- channel corruption modelled by non-determinism in sender and receiver of
-- lower level
--
-- fairness for all non-deterministic choices


MODULE sender(ch_s2r,ch_r2s,sa)

VAR
  state : {s0,s1,s2,s3,s4,s5};

ASSIGN
  init(state) := s0;
  next(state) :=
    case
      state = s0 & sa = sendd0 : s1;
      state = s1 & (ch_s2r = empty) : s2;
      -- corrupted -> s1, right ack -> s3; nondeterminism represents channel
      -- corruption
      state = s2 & (ch_r2s = ack0) : {s1,s3};
      -- wrong ack -> s1
      state = s2 & (ch_r2s = ack1) : {s1};
      state = s3 & sa = sendd1 : s4;
      state = s4 & (ch_s2r = empty) : s5;
      state = s5 & (ch_r2s = ack1): {s4,s0};
      state = s5 & (ch_r2s = ack0): {s4};
      1 : state;
    esac;
  next(ch_s2r) :=
    case
      ch_s2r = empty & state = s1 : data0;
      ch_s2r = empty & state = s4 : data1;
      1 : ch_s2r;
    esac;
  next(ch_r2s) :=
    case
      (ch_r2s in {ack0, ack1}) & (state = s2 | state = s5): empty;
      1 : ch_r2s;
    esac;
  next(sa) :=
    case
      state = s0 & sa = sendd0 : sendd1;
      state = s3 & sa = sendd1 : sendd0;
      1 : sa;
    esac;

FAIRNESS state = s3
FAIRNESS state = s0

```

```
FAIRNESS running




MODULE receiver(ch_s2r,ch_r2s,ra)

VAR
  state : {r0,r1,r2,r3,r4,r5,r6,r7};

ASSIGN
  init(state) := r0;
  next(state) :=
    case
      -- non-determinism represents channel corruption
      state = r0 & (ch_s2r = data0) : {r1,r2};
      state = r0 & (ch_s2r = data1) : {r1};
      state = r1 & (ch_r2s = empty) :  r0;
      state = r2 & ra = received0 : r3;
      state = r3 & (ch_r2s = empty) : r4;
      -- non-determinism represents channel corruption
      state = r4 & (ch_s2r = data1): {r5,r6};
      state = r4 & (ch_s2r = data0): {r5};
      state = r5 & (ch_r2s = empty) : r4;
      state = r6 & ra = received1 : r7;
      state = r7 & (ch_r2s = empty) : r0;
      1 : state;
    esac;
  next(ch_r2s) :=
    case
      ch_r2s = empty & (state in {r1,r7}): ack1;
      ch_r2s = empty & (state in {r3,r5}): ack0;
      1 : ch_r2s;
    esac;
  next(ch_s2r) :=
    case
      (ch_s2r in {data0,data1}) & (state = r0 | state = r4): empty;
      1 : ch_s2r;
    esac;
  next(ra) :=
    case
      state = r2 & ra = received0 : received1;
      state = r6 & ra = received1 : received0;
      1 : ra;
    esac;


FAIRNESS state = r2
FAIRNESS state = r6

FAIRNESS running


MODULE main

VAR
  ch_s2r : {empty,data0,data1};
  ch_r2s : {empty,ack0,ack1};
  sen : process sender(ch_s2r,ch_r2s,sa);
  rec : process receiver(ch_s2r,ch_r2s,ra);
  sa : {sendd0, sendd1};
  ra : {received0, received1};

ASSIGN
  init(ch_s2r) := empty;
```

```
  init(ch_r2s) := empty;
  init(sa) := sendd0;
  init(ra) := received0;


-- no deadlock, there are paths fulfilling the fairness conditions
SPEC
  EG 1

-- sender can always send if it wants to
SPEC
  AG ((sa = sendd0 -> EF sa = sendd1) & (sa = sendd1 -> EF sa = sendd0))

-- data is transmitted in right order
SPEC
  AG (sa = sendd1 -> A [sa = sendd1 U ra = received1]) &
  AG (ra = received1 -> A [ra = received1 U sa = sendd0]) &
  AG (sa = sendd0 -> A [sa = sendd0 U ra = received0]) &
  AG (ra = received0 -> A [ra = received0 U sa = sendd1])
```

A.1.2 The performance

```
i90s11:~/public/bin>smvo -f -r smvd/examples/own/abp/correct/ulcaf.smv
-- specification EG 1 is true
-- specification AG ((sa = sendd0 -> EF sa = sendd1) & (s... is true
-- specification AG (sa = sendd1 -> A(sa = sendd1 U ra = ... is true

resources used:
user time: 1.21667 s, system time: 0.466667 s
BDD nodes allocated: 4209
Bytes allocated: 917504
BDD nodes representing transition relation: 330 + 1
reachable states: 22 (2^4.45943) out of 1728 (2^10.7549)
```

A.2 Direct representation of the transition relation, no sender and receiver in the upper level

A.2.1 The SMV program

```
-- interleaving
--
-- direct representation of global transition relation as formula
--
-- sender and receiver of upper level: none
-- sender and receiver of lower level
-- channels
--
-- channel corruption modelled by non-determinism in sender and receiver of
-- lower level
--
-- fairness only for state s0


MODULE main

VAR
  s2r : {empty,data0,data1};
  r2s : {empty,ack0,ack1};
  s : {s0,s1,s2,s3,s4,s5};
  r : {r0,r1,r2,r3,r4,r5,r6,r7};
```

```

INIT
  s = s0 & r = r0 & s2r = empty & r2s = empty

TRANS
  s = s0 &
        next(s) = s1 & next(s2r) = s2r & next(r2s) = r2s & next(r) = r |
  s = s1 & (s2r = empty) &
        (next(s) = s2) & (next(s2r) = data0) & next(r2s) = r2s & next(r) = r |
  s = s2 & r2s = ack0 &
        (next(s) = s1 | next(s) = s3) &
        next(s2r) = s2r & next(r2s) = empty & next(r) = r |
  s = s2 & (r2s = ack1) &
        next(s) = s1 & next(s2r) = s2r & next(r2s) = empty & next(r) = r |
  s = s3 &
        next(s) = s4 & next(s2r) = s2r & next(r2s) = r2s & next(r) = r|
  s = s4 & (s2r = empty) &
        next(s) = s5 & next(s2r) = data1 & next(r2s) = r2s & next(r) = r |
  s = s5 & (r2s = ack1) &
        (next(s) = s4 | next(s) = s0) &
        next(s2r) = s2r & next(r2s) = empty & next(r) = r |
  s = s5 & (r2s = ack0) &
        next(s) = s4 & next(s2r) = s2r & next(r2s) = empty & next(r) = r |

  (r = r0 & (s2r = data0) &
        (next(r) in {r1,r2}) & next(s2r) = empty & next(r2s) = r2s |
  r = r0 & (s2r = data1) &
        next(r) = r1 & next(s2r) = empty & next(r2s) = r2s |
  r = r1 & (r2s = empty) &
        next(r) = r0 & next(s2r) = s2r & next(r2s) = ack1 |
  r = r2 &
        next(r) = r3 & next(s2r) = s2r & next(r2s) = r2s |
  r = r3 & (r2s = empty) &
        next(r) = r4 & next(s2r) = s2r & next(r2s) = ack0 |
  r = r4 & (s2r = data1) &
    (next(r) in  {r5,r6}) & next(s2r) = empty & next(r2s) = r2s |
  r = r4 & (s2r = data0) &
        next(r) = r5 & next(s2r) = empty & next(r2s) = r2s |
  r = r5 & (r2s = empty) &
        next(r) = r4 & next(s2r) = s2r & next(r2s) = ack0 |
  r = r6 &
        next(r) = r7 & next(s2r) = s2r & next(r2s) = r2s |
  r = r7 & (r2s = empty) &
        next(r) = r0 & next(s2r) = s2r & next(r2s) = ack1) & next(s) = s

FAIRNESS
  s = s0

-- no deadlock, there are paths fulfilling the fairness conditions
SPEC
  EG 1

-- data is transmitted in right order
SPEC
  AG (r in {r0,r1} -> A [r in {r0,r1} U s = s2]) &
  AG (s in {s1,s2} -> A [s in {s1,s2} U r = r4]) &
  AG (r in {r4,r5} -> A [r in {r4,r5} U s = s5]) &
  AG (s in {s4,s5} -> A [s in {s4,s5} U r = r0])
```

A.2.2 The performance

```
i90s11:~/public/bin>smvo -f -r smvd/examples/own/abp/correct/ulctsf.smv
-- specification EG 1 is true
-- specification AG (r in (r0 union r1) -> A(r in (r0 uni... is true
```

```
resources used:
user time: 0.583333 s, system time: 0.283333 s
BDD nodes allocated: 1756
Bytes allocated: 917504
BDD nodes representing transition relation: 152 + 1
reachable states: 22 (2^4.45943) out of 432 (2^8.75489)
```

A.3 Receiver, sender as modules, the 2 channels as variables

A.3.1 The SMV program

```
-- interleaving
--
-- sender and receiver of upper level: none
-- sender and receiver f lower level: as modules
-- channels: as variables
--
-- channel corruption modelled by non-determinism in sender and receiver of
-- lower level
--
-- fairness only for state s0 and running for sender of lower level


MODULE sender(ch_s2r,ch_r2s)

VAR
  state : {s0,s1,s2,s3,s4,s5};

ASSIGN
  init(state) := s0;
  next(state) :=
    case
      state = s0 : s1;
      state = s1 & (ch_s2r = empty) : s2;
      -- corrupted -> s1, right ack -> s3; nondeterminism represents channel
      -- corruption
      state = s2 & (ch_r2s = ack0) : {s1,s3};
      -- wrong ack -> s1
      state = s2 & (ch_r2s = ack1) : {s1};
      state = s3 : s4;
      state = s4 & (ch_s2r = empty) : s5;
      state = s5 & (ch_r2s = ack1): {s4,s0};
      state = s5 & (ch_r2s = ack0): {s4};
      1 : state;
    esac;
  next(ch_s2r) :=
    case
      ch_s2r = empty & state = s1 : data0;
      ch_s2r = empty & state = s4 : data1;
      1 : ch_s2r;
    esac;
  next(ch_r2s) :=
    case
      (ch_r2s in {ack0, ack1}) & (state = s2 | state = s5): empty;
      1 : ch_r2s;
    esac;

-- ensures that all nondeterministic choices in sender and receiver are fair
FAIRNESS state = s0
```

```

FAIRNESS running


MODULE receiver(ch_s2r,ch_r2s)

VAR
  state : {r0,r1,r2,r3,r4,r5,r6,r7};

ASSIGN
  init(state) := r0;
  next(state) :=
    case
      -- non-determinism represents channel corruption
      state = r0 & (ch_s2r = data0) : {r1,r2};
      state = r0 & (ch_s2r = data1) : {r1};
      state = r1 & (ch_r2s = empty) :  r0;
      state = r2 : r3;
      state = r3 & (ch_r2s = empty) : r4;
      -- non-determinism represents channel corruption
      state = r4 & (ch_s2r = data1): {r5,r6};
      state = r4 & (ch_s2r = data0): {r5};
      state = r5 & (ch_r2s = empty) : r4;
      state = r6 : r7;
      state = r7 & (ch_r2s = empty) : r0;
      1 : state;
    esac;
  next(ch_r2s) :=
    case
      ch_r2s = empty & (state in {r1,r7}): ack1;
      ch_r2s = empty & (state in {r3,r5}): ack0;
      1 : ch_r2s;
    esac;
  next(ch_s2r) :=
    case
      (ch_s2r in {data0,data1}) & (state = r0 | state = r4): empty;
      1 : ch_s2r;
    esac;



MODULE main

VAR
  ch_s2r : {empty,data0,data1};
  ch_r2s : {empty,ack0,ack1};
  sen : process sender(ch_s2r,ch_r2s);
  rec : process receiver(ch_s2r,ch_r2s);

ASSIGN
  init(ch_s2r) := empty;
  init(ch_r2s) := empty;



-- no deadlock, there are paths fulfilling the fairness conditions
SPEC
  EG 1

-- data is transmitted in right order
SPEC
  AG (rec.state in {r0,r1} -> A [rec.state in {r0,r1} U sen.state = s2]) &
  AG (sen.state in {s1,s2} -> A [sen.state in {s1,s2} U rec.state = r4]) &
  AG (rec.state in {r4,r5} -> A [rec.state in {r4,r5} U sen.state = s5]) &
  AG (sen.state in {s4,s5} -> A [sen.state in {s4,s5} U rec.state = r0])
```

A.3.2 The performance

- **With just two fairness conditions as described above**

```
i90s11:~/public/bin>smvo -f -r smvd/examples/own/abp/correct/lcsf.smv
-- specification EG 1 is true
-- specification AG (rec.state in (r0 union r1) -> A(rec.... is true

resources used:
user time: 0.916667 s, system time: 0.216667 s
BDD nodes allocated: 2739
Bytes allocated: 917504
BDD nodes representing transition relation: 243 + 1
reachable states: 22 (2^4.45943) out of 432 (2^8.75489)
```

- **With additional superfluous fairness conditions on s0, s3, r0, r4, running on both receiver and sender**

```
i90s11:~/public/bin>smvo -f -r smvd/examples/own/abp/correct/lcaf.smv
-- specification EG 1 is true
-- specification AG (rec.state in (r0 union r1) -> A(rec.... is true

resources used:
user time: 1.03333 s, system time: 0.383333 s
BDD nodes allocated: 3455
Bytes allocated: 917504
BDD nodes representing transition relation: 243 + 1
reachable states: 22 (2^4.45943) out of 432 (2^8.75489)
```

A.4 Receiver, sender, and the 2 channels as modules

A.4.1 The SMV program

```
-- interleaving
--
-- sender and receiver of upper level: none
-- sender and receiver of lower level: as modules
-- channels: as modules
--
-- channel corruption modelled by non-determinism in the channel modules
--
-- fairness only for state s0 and running for sender of lower level
-- additional fairness in order to forbid an infinite sequence of channel
-- corruptions


MODULE sender(cs2r,cr2s,s2rnew)

VAR
  state : {s0,s1,s2,s3,s4,s5};

ASSIGN
  init(state) := s0;
  next(state) :=
    case
      state = s0 : s1;
      state = s1 & (cs2r = empty) : s2;
```

```
      state = s2 & cr2s = ack0 : s3;
      state = s2 & (cr2s in {ack1,cor_ack}) : s1;
      state = s3 : s4;
      state = s4 & (cs2r = empty) : s5;
      state = s5 & (cr2s = ack1): s0;
      state = s5 & (cr2s in {ack0,cor_ack}): s4;
      1 : state;
    esac;
  next(cs2r) :=
    case
      cs2r = empty & state = s1 : data0;
      cs2r = empty & state = s4 : data1;
      1 : cs2r;
    esac;
  next(s2rnew) :=
    case
      cs2r = empty & state = s1 : 1;
      cs2r = empty & state = s4 : 1;
      1 : s2rnew;
    esac;
  next(cr2s) :=
    case
      state = s2 & cr2s = ack0 : empty;
      state = s2 & (cr2s in {ack1,cor_ack}) : empty;
      state = s5 & (cr2s = ack1): empty;
      state = s5 & (cr2s in {ack0,cor_ack}): empty;
      1 : cr2s;
    esac;

FAIRNESS state = s0

FAIRNESS running


MODULE receiver(cs2r,cr2s,r2snew)

VAR
  state : {r0,r1,r2,r3,r4,r5,r6,r7};

ASSIGN
  init(state) := r0;
  next(state) :=
    case
      state = r0 & (cs2r = data0) : r2;
      state = r0 & (cs2r in {data1,cor_data}) : r1;
      state = r1 & (cr2s = empty) :  r0;
      state = r2 : r3;
      state = r3 & (cr2s = empty) : r4;
      state = r4 & (cs2r = data1): r6;
      state = r4 & (cs2r in {data0,cor_data}): r5;
      state = r5 & (cr2s = empty) : r4;
      state = r6 : r7;
      state = r7 & (cr2s = empty) : r0;
      1 : state;
    esac;
  next(cr2s) :=
    case
      state = r1 & (cr2s = empty) : ack1;
      state = r3 & (cr2s = empty) : ack0;
      state = r5 & (cr2s = empty) : ack0;
      state = r7 & (cr2s = empty) : ack1;
      1 : cr2s;
    esac;
  next(r2snew) :=
    case
```

```
      state = r1 & (cr2s = empty) : 1;
      state = r3 & (cr2s = empty) : 1;
      state = r5 & (cr2s = empty) : 1;
      state = r7 & (cr2s = empty) : 1;
      1 : r2snew;
    esac;
  next(cs2r) :=
    case
      state = r0 & (cs2r = data0) : empty;
      state = r0 & (cs2r in {data1,cor_data}) : empty;
      state = r4 & (cs2r = data1): empty;
      state = r4 & (cs2r in {data0,cor_data}): empty;
      1 : cs2r;
    esac;



MODULE ch_s2r

VAR
  s : {empty,cor_data,data0,data1};
  -- corr and new are necessary to make the use of the language
  -- construct FAIRNESS possible
  -- if corr could be changed more often than data is received then corr
  -- could always be 0 when s is changed (corrupted)
  corr : boolean;
  new : boolean;

ASSIGN
  init(s) := empty;
  next(s) :=
    case
      (s = data0 | s = data1) & new & corr: cor_data;
      s = data0 & new & !corr : data0;
      s = data1 & new & !corr : data1;
      1 : s;
    esac;
  next(new) :=
    case
      new : 0;
      new = 0 : 0;
    esac;
  -- random decision always for each next datum (Zufallsentscheidung immer
  -- f"ur das jeweils n"achste Datum)
  next(corr) :=
    case
      new : {0,1};
      1 : corr;
    esac;

FAIRNESS corr = 0
-- we do not need a FAIRNESS running here since if this module is not
-- executed at all means that no corruption has occurred;
-- corr and new can be reset after the data has already been read
-- this is no problem with respect to the fairness of corr

MODULE ch_r2s

VAR
  s : {empty,cor_ack,ack0,ack1};
  corr : boolean;
  new : boolean;

ASSIGN
  init(s) := empty;
```

```
  next(s) :=
    case
      (s = ack0 | s = ack1) & new & corr: cor_ack;
      s = ack0 & new & !corr : ack0;
      s = ack1 & new & !corr : ack1;
      1 : s;
    esac;
  next(new) :=
    case
      new : 0;
      new = 0 : 0;
    esac;
  -- random decision always for each next datum
  next(corr) :=
    case
      new : {0,1};
      1 : corr;
    esac;

FAIRNESS corr = 0


MODULE main

VAR
  sen : process sender(s2r.s,r2s.s,s2r.new);
  rec : process receiver(s2r.s,r2s.s,r2s.new);
  s2r : process ch_s2r;
  r2s : process ch_r2s;



-- no deadlock, there are paths fulfilling the fairness conditions
SPEC
  EG 1

-- data is transmitted in right order
SPEC
  AG (rec.state in {r0,r1} -> A [rec.state in {r0,r1} U sen.state = s2]) &
  AG (sen.state in {s1,s2} -> A [sen.state in {s1,s2} U rec.state = r4]) &
  AG (rec.state in {r4,r5} -> A [rec.state in {r4,r5} U sen.state = s5]) &
  AG (sen.state in {s4,s5} -> A [sen.state in {s4,s5} U rec.state = r0])
```

A.4.2 The performance

```
i90s11:~/public/bin>smvo -f -r smvd/examples/own/abp/correct/lcmaf.smv
-- specification EG 1 is true
-- specification AG (rec.state in (r0 union r1) -> A(rec.... is true

resources used:
user time: 6.13333 s, system time: 0.383333 s
BDD nodes allocated: 10115
Bytes allocated: 983040
BDD nodes representing transition relation: 407 + 1
reachable states: 320 (2^8.32193) out of 12288 (2^13.585)
```

B Synchronous model of ABP in SMV

B.1 Global variables, no medium, no users

```

--         NAME: ABP_NM_NU.ni.gv.smv
--       AUTHOR: Armin Biere (armin@ira.uka.de)

-- ABP Alternating Bit Protocol
-- NM  No Media modelled
-- NU  No users modelled
-- ni  non interleaving
-- gv  synchronize via global variables

-- This is the alternating Bit Protokoll as described in:
-- Automatic Verification of Finite-State Concurrent Systems Using
-- Temporal Logic Specifications, by E.M. Clarke, E.A. Emerson
-- and A.P. Sistla, in ACM Transactions  on Programming Languages
-- and Systems. Volume 8. No.2. April 1986. Pages 244--263.

-- No lower or higher media is simulated.
-- But we do include the transmission of the data.

-- The main difference between this description of the Alternating
-- Bit Protocol and that mentioned above is that no interleaving
-- semantic is used. Because the smv system restricts multiple
-- assignement of a variable in different modules we can't use
-- global variables to exchanges signals between modules.

-- *******************************************************************
-- This time we don't use modules at all. So we can use the global
-- variable approach to synchronize sender and receiver. This is
-- possible since the sender of a signal only wants to write a signal
-- if it is zero and the receiver vice versa.
-- *******************************************************************

MODULE main

VAR
  snd : boolean;     -- signal from sender to receiver:
                     -- set by the sender and reset by the receiver
  rcv : boolean;     -- signal from receiver to sender:
                     -- set by the receiver and reset by the sender

  SNDstate : {
     prepareSend,    -- there must be an extra state to generate the
                     -- the data we want to transmit
     send,           -- send data and control bit ( see data )
     receive,        -- receive acknowledgement of the receiver
     transmitted     -- we got the right acknowledgement
  };
  Smsg : boolean;   -- what will be transmitted
  SNDdata : { dm00, dm01, dm10, dm11, err };
  SNDcontrol : boolean;


  RCVstate : {
    receive,         -- wait for data to receive
    prepareAck,      -- generate an acknowledgement according
                     -- to the control bit and the received data.
                     -- Also it is possible to generate an error.
    send,            -- send the acknowledgement to the sender.
    received         -- got data with the right control bit.
  };
```

```
  Rmsg : boolean;
  RCVcontrol : boolean;
  RCVdata : { am0, am1, err };


ASSIGN
  -- **************************************************************
  -- the manipulation of rcv and snd are the only global operations
  -- **************************************************************
  init(snd) := 0;
  next(snd) :=
    case
      SNDstate = send  & ! snd   : 1;
      RCVstate = receive & snd   : 0;
      1                          : snd;
    esac;
  init(rcv) := 0;
  next(rcv) :=
    case
      SNDstate = receive & rcv   : 0;
      RCVstate = send & ! rcv    : 1;
      1                          : rcv;
    esac;

  -- ******************
  -- this is the sender
  -- ******************
  next(SNDdata) :=
    case
      SNDstate = prepareSend &   Smsg & ! SNDcontrol   : { err, dm10 };
      SNDstate = prepareSend & ! Smsg & ! SNDcontrol   : { err, dm00 };
      SNDstate = prepareSend &   Smsg &   SNDcontrol   : { err, dm11 };
      SNDstate = prepareSend & ! Smsg &   SNDcontrol   : { err, dm01 };
      1                                                : SNDdata;
    esac;
  next(Smsg) :=
    case
      SNDstate = transmitted :  { 0, 1};  -- generate new data to send
      1                      :  Smsg;     -- keep it the same so that
                                          -- the receiver gets the
                                          -- right one. We don't have
                                          -- buffer for the data!
    esac;
  init(SNDcontrol) := 0;
  next(SNDcontrol) :=
    case
      SNDstate = transmitted   : ! SNDcontrol;
      1                        :   SNDcontrol;
    esac;
  init(SNDstate) := prepareSend;
  next(SNDstate) :=
    case
      SNDstate = prepareSend             : send;
      SNDstate = send & ! snd            : receive;
      SNDstate = receive & rcv :
        case
          SNDcontrol :
            case
              RCVdata = am1              : transmitted;
              RCVdata = am0 |                            -- we got a wrong ack:
              RCVdata = err              : send;         -- send again
            esac;
        ! SNDcontrol :
            case
              RCVdata = am0              : transmitted;
```

```
            RCVdata = am1 |                        -- we got a wrong ack:
            RCVdata = err             : send;         -- send again
          esac;
      esac;
    SNDstate = transmitted            : prepareSend;
    1                                 : SNDstate;
  esac;

-- **************************************
-- the description of the receiver follows
-- **************************************
init(RCVcontrol) := 0;
next(RCVcontrol) :=
  case
    RCVstate = received   : ! RCVcontrol;
    1                     :   RCVcontrol;
  esac;
next(RCVdata) :=
  case
    RCVstate = prepareAck :
      case
        RCVcontrol :
          case
            SNDdata in { dm11, dm01 } : { am1, err };
            SNDdata = err             : err;
            1                         : { am0, err };
          esac;
        ! RCVcontrol :
          case
            SNDdata in { dm10, dm00 } : { am0, err };
            SNDdata = err             : err;
            1                         : { am1, err };
          esac;
      esac;
    1                                 : RCVdata;
  esac;
next(Rmsg) :=
  case
    RCVstate = receive & snd :
      case
        SNDdata in { dm10, dm11 }     : 1;
        SNDdata in { dm00, dm01 }     : 0;
      esac;
    1                               : Rmsg;
  esac;
init(RCVstate) := receive;
next(RCVstate) :=
  case
    RCVstate = receive & snd          : prepareAck;
    RCVstate = prepareAck             : send;
    RCVstate = send & ! rcv           :
      case
        RCVcontrol :
          case         -- the receiver has choosen nondeterministically
                       -- to generate an error or not. This means we have
                       -- to check our own data that we have prepared for
                       -- acknowledging.
            RCVdata = am1   : received;
            1               : receive;  -- receive again
          esac;
        ! RCVcontrol :
          case
            RCVdata = am0   : received;
            1               : receive;  -- receive again
          esac;
```

```
        esac;
      RCVstate = received     : receive;
      1                       : RCVstate;
    esac;

FAIRNESS
 ! SNDdata = err
FAIRNESS
 ! RCVdata = err

-- first of all liveness specifications
SPEC -- ensure that the transition relation is not empty
  EF SNDstate = transmitted
SPEC
  AG AF SNDstate = transmitted
SPEC
  AG AF SNDstate = send

      -- correct transmission of a one bit
      -- this means that when the sender sends a one bit the
      -- receiver does not enter his received state without
      -- having received one bit:
SPEC  --
  AG ( (SNDstate = send & Smsg) ->
       A [ (! RCVstate = received ) U (RCVstate = received & Rmsg) ] )

SPEC  -- correct transmission of a zero bit
  AG ( (SNDstate = send & ! Smsg) ->
       A [ (! RCVstate = received ) U (RCVstate = received & ! Rmsg) ] )
```

B.2 Global variables, no users

```
--         NAME: ABP_M_NU.ni.gv.smv
--       AUTHOR: Armin Biere (armin@ira.uka.de)

-- ABP Alternating Bit Protocol
-- M   the medium is supported
-- NU  No users modelled
-- ni  non interleaving
-- gv  synchronize via global variables

-- This is an *extended* version of the alternating Bit Protokoll
-- as described in
-- Automatic Verification of Finite-State Concurrent Systems Using
-- Temporal Logic Specifications, by E.M. Clarke, E.A. Emerson
-- and A.P. Sistla, in ACM Transactions  on Programming Languages
-- and Systems. Volume 8. No.2. April 1986. Pages 244--263.

-- Only a lower Media is modeled but no users of the service.
-- In this version the data is realy transported between the
-- sender and the receiver. The lower medium can loose messages
-- and error detaction is performed. So this describes a transport
-- protocol over a loosy channel.

-- The main difference between this description of the Alternating
-- Bit Protocol and that mentioned above is that no interleaving
-- semantic is used. Because the smv system restricts multiple
-- assignement of a variable in different modules we can't use
-- global variables to exchanges signals between modules.

-- *******************************************************************
-- This time we don't use modules at all. So we can use the global
-- variable approach to synchronize sender and receiver. This is
```

```
-- possible since the sender of a signal only wants to write a signal
-- if it is zero and the receiver vice versa.
-- *******************************************************************

-- On the other hand we would like to ensure syntactically that
-- the sender and the receiver only comunicate via the medium and
-- dont't inspect the data or states of the partner.
-- With the SMV system this is only possible if we use a complicated
-- signaling approach which does't seem realy appropriate (compare
-- with the specific version using this mechanism)

MODULE main

VAR
  sndReq : boolean;     -- Request from the sender to the media
                        -- set by the sender and reset by the medium
  sndAck : boolean;     -- this is the acknowledge provided by the
                        -- lower medium when the receiver sends
                        -- a higher level acknowledge to the sender
                        -- it is set by the medium and reset by the
                        -- sender.
  rcvRes : boolean;     -- the receiver tells medium via this signal
                        -- that he wants to send an ackowledgement
  rcvInd : boolean;     -- the medium reports a data to the receiver

  SNDstate : {
     prepareSend,    -- there must be an extra state to generate the
                     -- the data we want to transmit
     send,           -- send data and control bit ( see data )
     receive,        -- receive acknowledgement of the receiver
     transmitted     -- we got the right acknowledgement
  };
  Smsg : boolean;    -- what will be transmitted
  SNDdata : { dm00, dm01, dm10, dm11 };
  SNDcontrol : boolean;


  RCVstate : {
    receive,         -- wait for data to receive
    prepareAck,      -- generate an acknowledgement according
                     -- to the control bit and the received data.
                     -- Also it is possible to generate an error.
    send,            -- send the acknowledgement to the sender.
    received         -- got data with the right control bit.
  };
  Rmsg : boolean;
  RCVcontrol : boolean;
  RCVdata : { am0, am1 };

  SND2RCVdata : { dm00, dm01, dm10, dm11, err };
  SND2RCVstate : {
     receive,
     error,
     noerror,
     send
  };

  RCV2SNDdata : { am0, am1, err };
  RCV2SNDstate : {
     receive,
     error,
     noerror,
     send
  };

```

```

ASSIGN
  -- **************************************************************
  -- here we have to manage the seting and reseting of all signals
  -- **************************************************************
  init(sndReq) := 0; -- Request from sender
  next(sndReq) :=
    case
      SNDstate = send & ! sndReq      : 1;
      SND2RCVstate = receive & sndReq : 0;
      1                               : sndReq;
    esac;
  init(sndAck) := 0; -- Ackowledgement reached sender
  next(sndAck) :=
    case
      SNDstate = receive & sndAck     : 0;
      RCV2SNDstate = send & ! sndAck  : 1;
      1                               : sndAck;
    esac;
  init(rcvInd) := 0; -- Indication of request from sender
  next(rcvInd) :=
    case
      RCVstate = receive & rcvInd     : 0;
      SND2RCVstate = send & ! rcvInd  : 1;
      1                               : rcvInd;
    esac;
  init(rcvRes) := 0; -- response to sender from receiver
  next(rcvRes) :=
    case
      RCVstate = send & ! rcvRes      : 1;
      RCV2SNDstate = receive & rcvRes : 0;
      1                               : rcvRes;
    esac;

  -- **************************************************************
  -- this is the channel from the sender to the receiver
  -- **************************************************************
  init(SND2RCVstate) := receive;
  next(SND2RCVstate) :=
    case
      SND2RCVstate = receive &   sndReq                : { error, noerror };
      SND2RCVstate = error                             : send;
      SND2RCVstate = noerror                           : send;
      SND2RCVstate = send    & ! rcvInd                : receive;
      1                                                : SND2RCVstate;
    esac;
  next(SND2RCVdata) :=
    case
      SND2RCVstate = receive & sndReq :
        case
          SNDdata = dm00 : dm00;
          SNDdata = dm01 : dm01;
          SNDdata = dm10 : dm10;
          SNDdata = dm11 : dm11;
        esac;
      SND2RCVstate = error                             : err;
      1                                                : SND2RCVdata;
    esac;

  -- ***************************************************************
  -- here comes the channel from the receiver to the sender
  -- ***************************************************************
  init(RCV2SNDstate) := receive;
  next(RCV2SNDstate) :=
    case
```

```
    RCV2SNDstate = receive &   rcvRes                : { error, noerror };
    RCV2SNDstate = error                             : send;
    RCV2SNDstate = noerror                           : send;
    RCV2SNDstate = send                              : receive;
    1                                                : RCV2SNDstate;
  esac;
next(RCV2SNDdata) :=
  case
    RCV2SNDstate = receive & rcvRes :
      case
        RCVdata = am0 : am0;
        RCVdata = am1 : am1;
      esac;
    RCV2SNDstate = error                             : err;
    1                                                : RCV2SNDdata;
  esac;

-- ******************
-- this is the sender
-- ******************
next(SNDdata) :=           -- here we don't have to generate errors
                           -- because the medium does it
  case
    SNDstate = prepareSend &   Smsg & ! SNDcontrol   : dm10;
    SNDstate = prepareSend & ! Smsg & ! SNDcontrol   : dm00;
    SNDstate = prepareSend &   Smsg &   SNDcontrol   : dm11;
    SNDstate = prepareSend & ! Smsg &   SNDcontrol   : dm01;
    1                                                : SNDdata;
  esac;
next(Smsg) :=
  case
    SNDstate = transmitted :  { 0, 1};  -- generate new data to send
    1                      :  Smsg;     -- keep it the same so that
                                        -- the receiver gets the
                                        -- right one. We don't have
                                        -- buffer for the data!
  esac;
init(SNDcontrol) := 0;
next(SNDcontrol) :=
  case
    SNDstate = transmitted   : ! SNDcontrol;
    1                        :   SNDcontrol;
  esac;
init(SNDstate) := prepareSend;
next(SNDstate) :=
  case
    SNDstate = prepareSend             : send;
    SNDstate = send & ! sndReq         : receive;
    SNDstate = receive & sndAck :
      case
        SNDcontrol :
          case
            RCV2SNDdata = am1          : transmitted;
            RCV2SNDdata = am0 |                        -- we got a wrong ack:
            RCV2SNDdata = err          : send;         -- send again
          esac;
      ! SNDcontrol :
          case
            RCV2SNDdata = am0          : transmitted;
            RCV2SNDdata = am1 |                        -- we got a wrong ack:
            RCV2SNDdata = err          : send;         -- send again
          esac;
      esac;
    SNDstate = transmitted             : prepareSend;
    1                                  : SNDstate;
```

```
    esac;

  -- ***************************************
  -- the description of the receiver follows
  -- ***************************************
  init(RCVcontrol) := 0;
  next(RCVcontrol) :=
    case
      RCVstate = received   : ! RCVcontrol;
      1                     :   RCVcontrol;
    esac;
  next(RCVdata) :=
    case
      RCVstate = prepareAck :
        case
          RCVcontrol :
            case
              SND2RCVdata in { dm11, dm01 } : am1;
              1                             : am0;
            esac;
          ! RCVcontrol :
            case
              SND2RCVdata in { dm10, dm00 } : am0;
              1                             : am1;
            esac;
        esac;
      1                                     : RCVdata;
    esac;
  next(Rmsg) :=
    case
      RCVstate = receive & rcvInd :
        case
          SND2RCVdata in { dm10, dm11 }     : 1;
          SND2RCVdata in { dm00, dm01 }     : 0;
          1                                 : Rmsg;
        esac;
      1                                     : Rmsg;
    esac;
  init(RCVstate) := receive;
  next(RCVstate) :=
    case
      RCVstate = receive & rcvInd           : prepareAck;
      RCVstate = prepareAck                 : send;
      RCVstate = send & ! rcvRes            :
        case
          RCVcontrol :
            case         -- the receiver has choosen nondeterministically
                         -- to generate an error or not. This means we have
                         -- to check our own data that we have prepared for
                         -- acknowledging.
              RCVdata = am1   : received;
              1               : receive;  -- receive again
            esac;
          ! RCVcontrol :
            case
              RCVdata = am0   : received;
              1               : receive;  -- receive again
            esac;
        esac;
      RCVstate = received     : receive;
      1                       : RCVstate;
    esac;

FAIRNESS
  SND2RCVstate = noerror
```

```
FAIRNESS
  RCV2SNDstate = noerror

-- first of all liveness specifications
SPEC -- ensure that the transition relation is not empty
  EF SNDstate = transmitted
SPEC
  AG AF SNDstate = transmitted
SPEC
  AG AF SNDstate = send

SPEC  --
  AG ( (SNDstate = prepareSend & Smsg) ->
       A [ (! RCVstate = received ) U (RCVstate = received & Rmsg) ] )

SPEC  -- correct transmission of a zero bit
  AG ( (SNDstate = prepareSend & ! Smsg) ->
       A [ (! RCVstate = received ) U (RCVstate = received & ! Rmsg) ] )
```

B.3 Global variables

```
--         NAME: ABP_M_U.ni.gv.smv
--       AUTHOR: Armin Biere (armin@ira.uka.de)

-- ABP Alternating Bit Protocol
-- M   the medium is supported
-- U   users modelled
-- ni  non interleaving
-- gv  synchronize via global variables

-- This is an *extended* version of the alternating Bit Protokoll
-- as described in
-- Automatic Verification of Finite-State Concurrent Systems Using
-- Temporal Logic Specifications, by E.M. Clarke, E.A. Emerson
-- and A.P. Sistla, in ACM Transactions  on Programming Languages
-- and Systems. Volume 8. No.2. April 1986. Pages 244--263.

-- Only a lower Media is modeled but no users of the service.
-- In this version the data is realy transported between the
-- sender and the receiver. The lower medium can loose messages
-- and error detaction is performed. So this describes a transport
-- protocol over a loosy channel.

-- The main difference between this description of the Alternating
-- Bit Protocol and that mentioned above is that no interleaving
-- semantic is used. Because the smv system restricts multiple
-- assignement of a variable in different modules we can't use
-- global variables to exchanges signals between modules.

-- ******************************************************************
-- This time we don't use modules at all. So we can use the global
-- variable approach to synchronize sender and receiver. This is
-- possible since the sender of a signal only wants to write a signal
-- if it is zero and the receiver vice versa.
-- ******************************************************************

-- On the other hand we would like to ensure syntactically that
-- the sender and the receiver only comunicate via the medium and
-- dont't inspect the data or states of the partner.
-- With the SMV system this is only possible if we use a complicated
-- signaling approach which does't seem realy appropriate (compare
-- with the specific version using this mechanism)

```

```
MODULE main

VAR
  sndReq : boolean;     -- Request from the sender to the media
                        -- set by the sender and reset by the medium
  sndAck : boolean;     -- this is the acknowledge provided by the
                        -- lower medium when the receiver sends
                        -- a higher level acknowledge to the sender
                        -- it is set by the medium and reset by the
                        -- sender.
  rcvRes : boolean;     -- the receiver tells medium via this signal
                        -- that he wants to send an ackowledgement
  rcvInd : boolean;     -- the medium reports a data to the receiver
  usrAReq : boolean;    -- The user A sends a request and the user
                        -- B gets an indication
  usrBInd : boolean;

  SNDstate : {
     waitForReq,     -- Wait for user request of a transmission
     prepareSend,    -- there must be an extra state to generate the
                     -- the data we want to transmit
     send,           -- send data and control bit ( see data )
     receive,        -- receive acknowledgement of the receiver
     transmitted     -- we got the right acknowledgement
  };
  Smsg : boolean;    -- what will be transmitted
  SNDdata : { dm00, dm01, dm10, dm11 };
  SNDcontrol : boolean;


  RCVstate : {
    receive,
    prepareAck,      -- generate an acknowledgement according
                     -- to the control bit and the received data.
    send,            -- send the acknowledgement to the sender.
    received,        -- got data with the right control bit.
    sendInd          -- send indication to the user
  };
  Rmsg : boolean;
  RCVcontrol : boolean;
  RCVdata : { am0, am1 };

  SND2RCVdata : { dm00, dm01, dm10, dm11, err };
  SND2RCVstate : {
     receive,
     error,
     noerror,
     send
  };

  RCV2SNDdata : { am0, am1, err };
  RCV2SNDstate : {
     receive,
     error,
     noerror,
     send
  };

  USRAdata : boolean;
  USRAstate : {
    prepareData,
    send
  };

  USRBdata : boolean;
```

```
  USRBstate : {
    receive,
    processData
  };

ASSIGN
  -- **************************************************************
  -- communication between the lower medium and the service
  -- **************************************************************
  init(sndReq) := 0; -- Request from sender
  next(sndReq) :=
    case
      SNDstate = send & ! sndReq      : 1;
      SND2RCVstate = receive & sndReq : 0;
      1                               : sndReq;
    esac;
  init(sndAck) := 0; -- Ackowledgement reached sender
  next(sndAck) :=
    case
      SNDstate = receive & sndAck     : 0;
      RCV2SNDstate = send & ! sndAck  : 1;
      1                               : sndAck;
    esac;
  init(rcvInd) := 0; -- Indication of request from sender
  next(rcvInd) :=
    case
      RCVstate = receive & rcvInd     : 0;
      SND2RCVstate = send & ! rcvInd  : 1;
      1                               : rcvInd;
    esac;
  init(rcvRes) := 0; -- response to sender from receiver
  next(rcvRes) :=
    case
      RCVstate = send & ! rcvRes      : 1;
      RCV2SNDstate = receive & rcvRes : 0;
      1                               : rcvRes;
    esac;

  -- **************************************************************
  -- communication between the medium and the user instances
  -- **************************************************************
  init(usrAReq) := 0;
  next(usrAReq) :=
    case
      USRAstate = send & ! usrAReq              : 1;
      SNDstate = waitForReq & usrAReq           : 0;
      1                                         : usrAReq;
    esac;
  init(usrBInd) := 0;
  next(usrBInd) :=
    case
      RCVstate = sendInd & ! usrBInd            : 1;
      USRBstate = receive & usrBInd             : 0;
      1                                         : usrBInd;
    esac;

  -- **************************************************************
  -- this is the channel from the sender to the receiver
  -- **************************************************************
  init(SND2RCVstate) := receive;
  next(SND2RCVstate) :=
    case
      SND2RCVstate = receive &   sndReq                : { error, noerror };
      SND2RCVstate = error                             : send;
      SND2RCVstate = noerror                           : send;
```

```
    SND2RCVstate = send    & ! rcvInd                : receive;
    1                                                : SND2RCVstate;
  esac;
next(SND2RCVdata) :=
  case
    SND2RCVstate = receive & sndReq :
      case
        SNDdata = dm00 : dm00;
        SNDdata = dm01 : dm01;
        SNDdata = dm10 : dm10;
        SNDdata = dm11 : dm11;
      esac;
    SND2RCVstate = error                             : err;
    1                                                : SND2RCVdata;
  esac;

-- ***************************************************************
-- here comes the channel from the receiver to the sender
-- ***************************************************************
init(RCV2SNDstate) := receive;
next(RCV2SNDstate) :=
  case
    RCV2SNDstate = receive &   rcvRes                : { error, noerror };
    RCV2SNDstate = error                             : send;
    RCV2SNDstate = noerror                           : send;
    RCV2SNDstate = send                              : receive;
    1                                                : RCV2SNDstate;
  esac;
next(RCV2SNDdata) :=
  case
    RCV2SNDstate = receive & rcvRes :
      case
        RCVdata = am0 : am0;
        RCVdata = am1 : am1;
      esac;
    RCV2SNDstate = error                             : err;
    1                                                : RCV2SNDdata;
  esac;

-- *******************************************************
-- This is the user A who wants to send data to the user B
-- *******************************************************
init(USRAstate) := prepareData;
next(USRAstate) :=
  case
    USRAstate = prepareData                   : send;
    USRAstate = send & ! usrAReq              : prepareData;
    1                                         : USRAstate;
  esac;
next(USRAdata) :=
  case
    USRAstate = prepareData                   : { 1, 0 };
    1                                         : USRAdata;
  esac;

-- *****************
-- here comes user B
-- *****************
init(USRBstate) := receive;
next(USRBstate) :=
  case
    USRBstate = receive & usrBInd             : processData;
    USRBstate = processData                   : receive;
    1                                         : USRBstate;
  esac;
```

```
next(USRBdata) :=
  case
    USRBstate = receive & usrBInd             : Rmsg;
    1                                         : USRBdata;
  esac;

-- ******************
-- this is the sender
-- ******************
next(SNDdata) :=           -- here we don't have to generate errors
                           -- because the medium does it
  case
    SNDstate = prepareSend &   Smsg & ! SNDcontrol   : dm10;
    SNDstate = prepareSend & ! Smsg & ! SNDcontrol   : dm00;
    SNDstate = prepareSend &   Smsg &   SNDcontrol   : dm11;
    SNDstate = prepareSend & ! Smsg &   SNDcontrol   : dm01;
    1                                                : SNDdata;
  esac;
next(Smsg) :=
  case
    SNDstate = waitForReq & usrAReq           : USRAdata;
    1                                         : Smsg;
  esac;
init(SNDcontrol) := 0;
next(SNDcontrol) :=
  case
    SNDstate = transmitted   : ! SNDcontrol;
    1                        :   SNDcontrol;
  esac;
init(SNDstate) := waitForReq;
next(SNDstate) :=
  case
    SNDstate = waitForReq & usrAReq    : prepareSend;
    SNDstate = prepareSend             : send;
    SNDstate = send & ! sndReq         : receive;
    SNDstate = receive & sndAck :
      case
        SNDcontrol :
          case
            RCV2SNDdata = am1          : transmitted;
            RCV2SNDdata = am0 |                        -- we got a wrong ack:
            RCV2SNDdata = err          : send;         -- send again
          esac;
      ! SNDcontrol :
          case
            RCV2SNDdata = am0          : transmitted;
            RCV2SNDdata = am1 |                        -- we got a wrong ack:
            RCV2SNDdata = err          : send;         -- send again
          esac;
      esac;
    SNDstate = transmitted             : waitForReq;
    1                                  : SNDstate;
  esac;

-- ****************************************
-- the description of the receiver follows
-- ****************************************
init(RCVcontrol) := 0;
next(RCVcontrol) :=
  case
    RCVstate = received   : ! RCVcontrol;
    1                     :   RCVcontrol;
  esac;
next(RCVdata) :=
  case
```

```
      RCVstate = prepareAck :
        case
          RCVcontrol :
            case
              SND2RCVdata in { dm11, dm01 } : am1;
              1                             : am0;
            esac;
          ! RCVcontrol :
            case
              SND2RCVdata in { dm10, dm00 } : am0;
              1                             : am1;
            esac;
        esac;
      1                                     : RCVdata;
    esac;
  next(Rmsg) :=
    case
      RCVstate = receive & rcvInd :
        case
          SND2RCVdata in { dm10, dm11 }     : 1;
          SND2RCVdata in { dm00, dm01 }     : 0;
          1                                 : Rmsg;
        esac;
      1                                     : Rmsg;
    esac;
  init(RCVstate) := receive;
  next(RCVstate) :=
    case
      RCVstate = receive & rcvInd           : prepareAck;
      RCVstate = prepareAck                 : send;
      RCVstate = send & ! rcvRes            :
        case
          RCVcontrol :
            case
              RCVdata = am1   : received;
              1               : receive;  -- receive again
            esac;
          ! RCVcontrol :
            case
              RCVdata = am0   : received;
              1               : receive;  -- receive again
            esac;
        esac;
      RCVstate = received     : sendInd;
      RCVstate = sendInd &
        ! usrBInd             : receive;
      1                       : RCVstate;
    esac;

FAIRNESS
  SND2RCVstate = noerror
FAIRNESS
  RCV2SNDstate = noerror

SPEC -- transition relation is not empty under the fairness constraints
  EF USRBstate = processData
SPEC
  AG AF USRBstate = processData

SPEC
  AG ((USRAstate = send & USRAdata) ->
    A [ !USRBstate = processData U (USRBstate = processData & USRBdata) ])
SPEC
  AG ((USRAstate = send & ! USRAdata) ->
    A [ !USRBstate = processData U (USRBstate = processData & ! USRBdata) ])
```

B.4 With signal modules

```
--             NAME: ABP_NM_NU.ni.sm.smv
--           AUTHOR: Armin Biere (armin@ira.uka.de)

-- ABP Alternating Bit Protocol
-- NM  No Media modelled
-- NU  No users modelled
-- ni  non interleaving
-- sm  synchronize via signal modules

-- This is the alternating Bit Protokoll as described in:
-- Automatic Verification of Finite-State Concurrent Systems Using
-- Temporal Logic Specifications, by E.M. Clarke, E.A. Emerson
-- and A.P. Sistla, in ACM Transactions  on Programming Languages
-- and Systems. Volume 8. No.2. April 1986. Pages 244--263.

-- No lower or higher media is simulated.
-- But we do include the transmission of the data.

-- The main difference between this description of the Alternating
-- Bit Protocol and that mentioned above is that no interleaving
-- semantic is used. Because the smv system restricts multiple
-- assignement of a variable in different modules we can't use
-- global variables to exchanges signals between modules.

-- ***************************************************
-- Here we use a seperate signal module which is responsible
-- for a signal.  The sender (respectively the receiver)
-- can request to set (reset) the signal. Than the signal
-- module will do this for him.
-- The reason for this complicated scheme is that the smv
-- language does not support multiple assignement of variable
-- in different modules (though both modules who want to
-- write to the variable don't do it simoustanly).
-- ***************************************************

MODULE main

VAR
  snd : signal;
  rcv : signal;
  SND : sender(RCV.data, rcv, snd);    -- No media supported. So we have to
                                       -- access the data in common store.
  RCV : receiver(SND.data, snd, rcv);

-- first of all  four liveness specifications
SPEC -- ensure that the transition relation is not empty
   EF SND.state = transmitted
SPEC
   AG AF SND.state = transmitted
SPEC
  AG AF SND.state = sendSet

      -- correct transmission of a one bit
      -- this means that when the sender sends a one bit the
      -- receiver does not enter his received state without
      -- having received one bit:
SPEC  --
   AG ( (SND.state = sendSet & SND.Smsg) ->
       A [ (! RCV.state = received ) U (RCV.state = received & RCV.Rmsg) ] )

SPEC  -- correct transmission of a zero bit
  AG ( (SND.state = sendSet & ! SND.Smsg) ->
       A [ (! RCV.state = received ) U (RCV.state = received & ! RCV.Rmsg) ] )
```

```
MODULE signal
VAR
  sig  : boolean;
  set  : boolean;   -- the sender of this signal *owns* this bit.
                    -- he sets it when wants the signal module to
                    -- set sig to one.
  reset: boolean;   -- the receiver of this signal *owns* this bit.
                    -- he sets reset when wants to set sig to zero.
ASSIGN
  init(sig) := 0;
  next(sig) :=
    case
      ! sig &   set : 1;
        sig & reset : 0;
      1             : sig;
    esac;


MODULE sender(rdata, IN, OUT) -- rdata is the data of the receiver
                              -- IN = rcv, OUT = snd
VAR
  Smsg : boolean;
  control : boolean;
  state : {
     prepareSend,   -- there must be an extra state to generate the
                    -- the data we want to transmit

                    -- the next to sates send the data ( OUT = snd )
     sendWait,      -- Wait for  ! snd.sig
     sendSet,       -- Wait for    snd.sig

                    -- the next two states receive the acknowledgement
                    -- ( IN = rcv )
     receiveWait,   -- Wait for    rcv.sig
     receiveReset,  -- Wait for  ! rcv.sig

     transmitted    -- we got the right acknowledgement
  };
  data : { dm00, dm01, dm10, dm11, err };

ASSIGN
  next(data) :=
    case
      state = prepareSend &   Smsg & ! control   : { err, dm10 };
      state = prepareSend & ! Smsg & ! control   : { err, dm00 };
      state = prepareSend &   Smsg &   control   : { err, dm11 };
      state = prepareSend & ! Smsg &   control   : { err, dm01 };
      1                                          : data;
    esac;
  next(Smsg) :=
    case
      state = transmitted :  { 0, 1};  -- generate new data to send
      1                   :  Smsg;     -- keep it the same so that
                                       -- the receiver gets the
                                       -- right one. We don't have
                                       -- buffer for the data!
    esac;
  init(control) := 0;
  next(control) :=
    case
      state = transmitted   : ! control;
      1                     :   control;
    esac;
```

```
  init(state) := prepareSend;
  init(IN.reset) := 0;
  next(IN.reset) :=
    case
      state = receiveReset & IN.sig      : 1;
      1                                  : 0;
    esac;
  init(OUT.set) := 0;
  next(OUT.set) :=
    case
      state = sendSet      & ! OUT.sig   : 1;
      1                                  : 0;
    esac;
  next(state) :=
    case
      state = prepareSend                : sendWait;
      state = sendWait & ! OUT.sig       : sendSet;
      state = sendSet  &   OUT.sig       : receiveWait;
      state = receiveWait & IN.sig :
        case
          control :
            case
              rdata = am1                : receiveReset;
              rdata = am0 |                              -- we got a wrong ack:
              rdata = err                : sendWait;     -- send again
            esac;
        ! control :
            case
              rdata = am0                : receiveReset;
              rdata = am1 |                              -- we got a wrong ack:
              rdata = err                : sendWait;     -- send again
            esac;
        esac;
      state = receiveReset & ! IN.sig    : transmitted;
      state = transmitted                : prepareSend;
      1                                  : state;
    esac;
         -- the sender is responsible on his own that an error
         -- in the underlying media is generated.
         -- This is accomplished by allowing the data to be
         -- nondeterministically choosen between the real data (data bit
         -- and control bit) and an error. To ensure a correct transmission
         -- of data by the ABP we have to impose a restriction. Namely that
         -- the media does not always generate an error:
FAIRNESS --
 ! data = err

MODULE receiver(sdata, IN, OUT) -- sort of pram model for data transfer
                                -- IN = snd, OUT = rcv
VAR
  Rmsg : boolean;
  control : boolean;
  state : {
    receiveWait,     -- wait for data to come in ( snd.sig = 1 )
    receiveReset,    -- reset snd.sig

    prepareAck,      -- generate an acknowledgement according
                     -- to the control bit and the received data.
                     -- Also it is possible to generate an error.

                     -- send the acknowledgement to the sender.
    sendWait,        -- wait for signal to be read by sender
    sendSet,         -- now set it ( rcv.sig = 1 )

    received         -- got data with the right control bit.
```

```
  };
  data : { am0, am1, err };

ASSIGN
  init(control) := 0;
  next(control) :=
    case
      state = received   : ! control;
      1                  :   control;
    esac;
  next(data) :=
    case
      state = prepareAck :
        case
          control :
            case
              sdata in { dm11, dm01 } : { am1, err };
              sdata = err             : err;
              1                       : { am0, err };
            esac;
          ! control :
            case
              sdata in { dm10, dm00 } : { am0, err };
              sdata = err             : err;
              1                       : { am1, err };
            esac;
        esac;
      1                               : data;
    esac;
  next(Rmsg) :=
    case
      state = receiveWait & IN.sig :
        case
          sdata in { dm10, dm11 }     : 1;
          sdata in { dm00, dm01 }     : 0;
        esac;
      1                               : Rmsg;
    esac;
  init(IN.reset) := 0;
  next(IN.reset) :=
    case
      state = receiveWait & IN.sig    : 1;
      1                               : 0;
    esac;
  init(OUT.set) := 0;
  next(OUT.set) :=
    case
      state = sendWait & ! OUT.sig    : 1;
      1                               : 0;
    esac;
  init(state) := receiveWait;
  next(state) :=
    case
      state = receiveWait & IN.sig    : receiveReset;
      state = receiveReset & ! IN.sig : prepareAck;
      state = prepareAck              : sendWait;
      state = sendWait & ! OUT.sig :
        case
          control :
            case        -- the receiver has choosen nondeterministically
                        -- to generate an error or not. This means we have
                        -- to check our own data that we have prepared for
                        -- acknowledging.
              data = am1   : sendSet;
              1            : receiveWait;  -- receive again
```

```
          esac;
        ! control :
          case
            data = am0   : sendSet;
            1            : receiveWait;  -- receive again
          esac;
      esac;
    state = sendSet & OUT.sig : received;
    state = received          : receiveWait;
    1                         : state;
  esac;
FAIRNESS
  ! data = err
```

B.5 Synchronization via state investigation

```
--         NAME: ABP_NM_NU.ni.si.smv
--       AUTHOR: Armin Biere (armin@ira.uka.de)

-- ABP Alternating Bit Protocol
-- NM  No Media modelled
-- NU  No users modelled
-- ni  non interleaving
-- si  synchronize via state investigation

-- This is the alternating Bit Protokoll as described in:
-- Automatic Verification of Finite-State Concurrent Systems Using
-- Temporal Logic Specifications, by E.M. Clarke, E.A. Emerson
-- and A.P. Sistla, in ACM Transactions  on Programming Languages
-- and Systems. Volume 8. No.2. April 1986. Pages 244--263.

-- This example is a small one because we use synchronous send
-- and receive. In addition no lower or higher media is simulated.
-- But we do include the transmission of the data.

-- The main difference between this description of the Alternating
-- Bit Protocol and that mentioned above is that no interleaving
-- semantic is used. Because the smv system restricts multiple
-- assignement of a variable in different modules we can't use
-- global variables to exchanges signals between modules.
-- This version ensures the correct transmission of lower media messages
-- by simultaneous investigation of the states of the sender and
-- receiver.  So this is no real implementation.

MODULE main

VAR
  SND : sender(RCV.state, RCV.data);
  RCV : receiver(SND.state, SND.data);

-- first of all liveness specifications
SPEC -- ensure that the transition relation is not empty
  EF SND.state = transmitted
SPEC
  AG AF SND.state = transmitted
SPEC
  AG AF SND.state = send

      -- correct transmission of a one bit
      -- this means that when the sender sends a one bit the
      -- receiver does not enter his received state without
      -- having received one bit:
SPEC  --
```

```
  AG ( (SND.state = send & SND.Smsg) ->
       A [ (! RCV.state = received ) U (RCV.state = received & RCV.Rmsg) ] )

SPEC  -- correct transmission of a zero bit
  AG ( (SND.state = send & ! SND.Smsg) ->
       A [ (! RCV.state = received ) U (RCV.state = received & ! RCV.Rmsg) ] )

MODULE sender(rstate, rdata)    -- rstate is the state of receiver
                                -- rdata is the data of the receiver

VAR
  Smsg : boolean;
  control : boolean;
  state : {
     prepareSend,   -- there must be an extra state to generate the
                    -- the data we want to transmit
     send,          -- send data and control bit ( see data )
     receive,       -- receive acknowledgement of the receiver
     transmitted    -- we got the right acknowledgement
  };
  data : { dm00, dm01, dm10, dm11, err };

ASSIGN
  next(data) :=
    case
      state = prepareSend &   Smsg & ! control   : { err, dm10 };
      state = prepareSend & ! Smsg & ! control   : { err, dm00 };
      state = prepareSend &   Smsg &   control   : { err, dm11 };
      state = prepareSend & ! Smsg &   control   : { err, dm01 };
      1                                          : data;
    esac;
  next(Smsg) :=
    case
      state = transmitted :  { 0, 1};  -- generate new data to send
      1                   :  Smsg;     -- keep it the same so that
                                       -- the receiver gets the
                                       -- right one. We don't have
                                       -- buffer for the data!
    esac;
  init(control) := 0;
  next(control) :=
    case
      state = transmitted   : ! control;
      1                     :   control;
    esac;
  init(state) := prepareSend;
  next(state) :=
    case
      state = prepareSend                : send;
      state = send &
      rstate = receive                   : receive;
      state = receive &
      rstate = send :
        case
          control :
            case
              rdata = am1                : transmitted;
              rdata = am0 |                             -- we got a wrong ack:
              rdata = err                : send;        -- send again
            esac;
        ! control :
            case
              rdata = am0                : transmitted;
              rdata = am1 |                             -- we got a wrong ack:
              rdata = err                : send;        -- send again
```

```
            esac;
        esac;
      state = transmitted               : prepareSend;
      1                                 : state;
    esac;
         -- the sender is responsible on his own that an error
         -- in the underlying media is generated.
         -- This is accomplished by allowing the data to be
         -- nondeterministically choosen between the real data (data bit
         -- and control bit) and an error. To ensure a correct transmission
         -- of data by the ABP we have to impose a restriction. Namely that
         -- the media does not always generate an error:
FAIRNESS --
 ! data = err

MODULE receiver(sstate, sdata)   -- sstate is state of sender

VAR
  Rmsg : boolean;
  control : boolean;
  state : {
    receive,         -- wait for data to receive
    prepareAck,      -- generate an acknowledgement according
                     -- to the control bit and the received data.
                     -- Also it is possible to generate an error.
    send,            -- send the acknowledgement to the sender.
    received         -- got data with the right control bit.
  };
  data : { am0, am1, err };

ASSIGN
  init(control) := 0;
  next(control) :=
    case
      state = received   : ! control;
      1                  :   control;
    esac;
  next(data) :=
    case
      state = prepareAck :
        case
          control :
            case
              sdata in { dm11, dm01 } : { am1, err };
              sdata = err             : err;
              1                       : { am0, err };
            esac;
          ! control :
            case
              sdata in { dm10, dm00 } : { am0, err };
              sdata = err             : err;
              1                       : { am1, err };
            esac;
        esac;
      1                               : data;
    esac;
  next(Rmsg) :=
    case
      state = receive & sstate = send :
        case
          sdata in { dm10, dm11 }     : 1;
          sdata in { dm00, dm01 }     : 0;
        esac;
      1                               : Rmsg;
    esac;
```

```
  init(state) := receive;
  next(state) :=
    case
      state = receive & sstate = send    : prepareAck;
      state = prepareAck                 : send;
      state = send & sstate = receive :
        case
          control :
            case         -- the receiver has choosen nondeterministically
                         -- to generate an error or not. This means we have
                         -- to check our own data that we have prepared for
                         -- acknowledging.
              data = am1   : received;
              1            : receive;  -- receive again
            esac;
          ! control :
            case
              data = am0   : received;
              1            : receive;  -- receive again
            esac;
        esac;
      state = received     : receive;
      1                    : state;
    esac;
FAIRNESS
  ! data = err
```