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*}

¹Institut für Logik, Komplexität und Deduktionssysteme,
 ²Lehrstuhl 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

$$f(x_0, \dots, x_n, x'_0, \dots, x'_n) = 1 \Leftrightarrow (x_0, \dots, x_n, x'_0, \dots, x'_n) \in R$$

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:

$$f(V,V') = \bigvee_{(x_0,\dots,x_n,x'_0,\dots,x'_n)\in R} (\bigwedge_{i=0}^n (1 \Leftrightarrow x_i) \neg v_i + x_i v_i) \land (\bigwedge_{i=0}^n (1 \Leftrightarrow x'_i) \neg v'_i + x'_i v'_i)$$

Example 2.1

 $\begin{array}{l} V = \{ v_{0}, v_{1} \}, R = \{ (0, 0, 0, 1), (0, 1, 0, 1) \} \Rightarrow f(v_{0}, v_{1}, v_{0}^{'}, v_{1}^{'}) = \neg v_{0} \land \neg v_{1} \land \neg v_{0}^{'} \land v_{1}^{'} \lor \neg v_{0} \land v_{1} \land \neg v_{0}^{'} \land v_{1}^{'} \end{cases}$

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:

$$f(V, V') = \bigwedge f_i(V, V')$$
$$f_i(V, V') = (v'_i \Leftrightarrow g_i(V)) \lor (v'_i \Leftrightarrow v_i)$$

• interleaving:

$$f(V, V') = \bigvee f_i(V, V')$$
$$f_i(V, V') = (v'_i \Leftrightarrow g_i(V)) \land \bigwedge_{j \neq i} (v'_j \Leftrightarrow v_j)$$

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 $f: C_0 \times \ldots \times C_n \times C_0 \times \ldots \times C_n \mapsto$ $\{0, 1\}$ where

 $f(x_0, \dots, x_n, x'_0, \dots, x'_n) = 1 \Leftrightarrow (x_0, \dots, x_n, x'_0, \dots, x'_n) \in R$

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:

$$f(C, C') = \bigvee_{(x_0, \dots, x_n, x'_0, \dots, x'_n) \in R} (\bigwedge_{i=0}^n c_i = x_i) \land (\bigwedge_{i=0}^n c'_i = x'_i)$$

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_i} 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 \rightarrow . If a state has several ingoing arcs this state will appear on the right side of \rightarrow 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:

$$f(C, C') = \bigwedge f_i(C, C') \land K(C, C')$$

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

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

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:

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

$$f_{i}(C, C') = (\bigwedge_{j=0}^{m_{i}} (g_{i_{j}}(C) \to \bigvee_{k=0}^{n_{j}} c_{i}' = x_{k})) \land \bigwedge_{j \neq i} (c_{j}' = c_{j})$$

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 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.

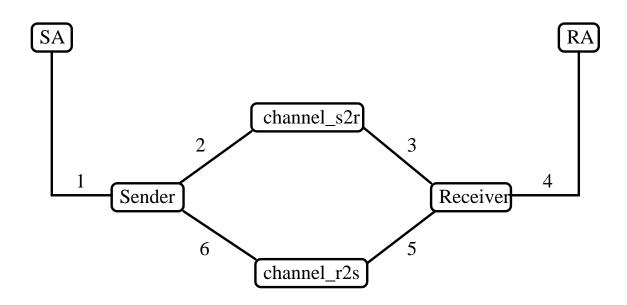


Figure 1: The configuration

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, state of the receiver in lower level, state of receiver in upper level.

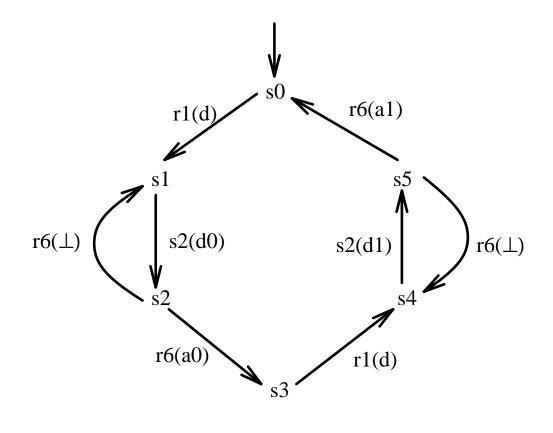
4.2 Different ways of description of the ABP in SMV

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.

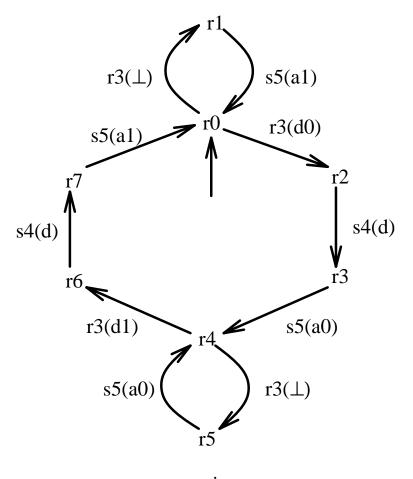
4.2.1 Standard way of description in SMV

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



sender

Figure 2: The sender



receiver

Figure 3: The receiver

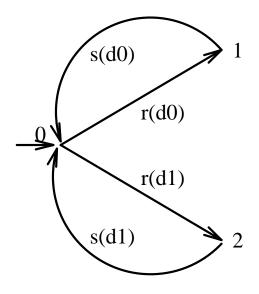


Figure 4: The channel for messages from sender to receiver

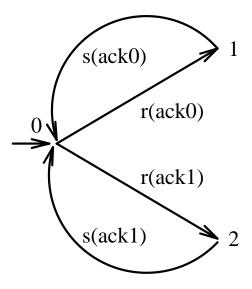


Figure 5: The channel for acknowledgements from receiver to sender

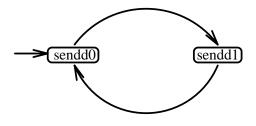


Figure 6: The sender of the upper level

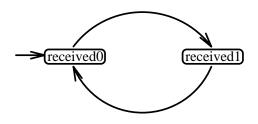


Figure 7: The receiver of the upper level

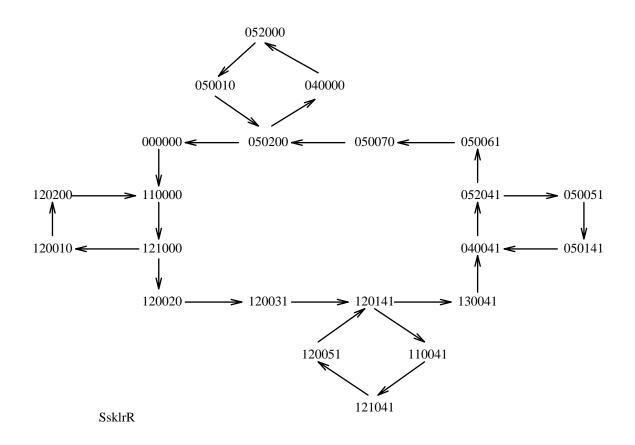


Figure 8: The product automaton of the alternating bit protocol in interleaving semantics

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..., 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 \land after sd0 nothing else can happen except the transition rd0 \land after rd0 nothing else can happen except the transition sd1 \land after sd1 nothing else can happen except the transition rd1 \land 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:

$$\begin{array}{lll} AG(sa=sendd0 \rightarrow A[sa=sendd0 & U & ra=received0]) \land \\ AG(ra=received0 \rightarrow A[ra=received0 & U & sa=sendd1]) \land \\ AG(sa=sendd1 \rightarrow A[sa=sendd1 & U & ra=received1]) \land \\ AG(ra=received1 \rightarrow A[ra=received1 & U & sa=sendd0]) \end{array}$$

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 $ra = ... \land sa = ...$ 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.

The CTL formula for C3 is:

$$AG((sa = sendd0 \rightarrow EFsa = sendd1) \land (sa = sendd1 \rightarrow EFsa = sendd0))$$

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... 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 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...). 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 expressive-ness.

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 processe 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_{req} . The sender (A) wants to set V_{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_{req} then we get the following cases:

A wants to write $V_{req} \iff V_{req} = 0$ B wants to write $V_{req} \iff V_{req} = 1$

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

A has the privilege of writing $\iff V_{req} = 0$ B has the privilege of writing $\iff V_{req} = 1$

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;
           boolean;
  set
  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

$\operatorname{semantic}$	upper	lower		signal	reachable	checking
model	layer	layer	data	model	states	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(\perp)).S0 + r6(0)
    S1 = r1(d).s2(d1).T1
    T0 = (r6(0) + r6(\perp)).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: 10²⁰ 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 specifica-

tions. 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

```
1
     -- interleaving
\mathbf{2}
     ___
\overline{3}
     -- sender and receiver of upper level: as variables
4
     -- sender and receiver of lower level: as modules
5
     -- channels: as variables
\mathbf{6}
     ---
7
     -- channel corruption modelled by non-determinism in sender and receiver of
8
     -- lower level
9
10
     -- fairness for all non-deterministic choices
11
12
13
     MODULE sender(ch_s2r,ch_r2s,sa)
14
15
     VAR.
16
       state : {s0,s1,s2,s3,s4,s5};
17
18 ASSIGN
19
       init(state) := s0;
\frac{20}{21}
      next(state) :=
          case
\overline{2}\overline{2}
           state = s0 \& sa = sendd0 : s1;
\overline{23}
            state = s1 & (ch_s2r = empty) : s2;
24
            -- corrupted -> s1, right ack -> s3; nondeterminism represents channel
-- corruption
            state = s2 & (ch_r2s = ack0) : {s1,s3};
            -- wrong ack -> s1
\overline{28}
            state = s2 \& (ch_r 2s = ack1) : {s1};
\overline{29}
           state = s3 \& sa = sendd1 : s4;
30
          state = s4 & (ch_s2r = empty) : s5;
\frac{31}{32}\\ 33
           state = s5 & (ch_r2s = ack1): {s4,s0};
            state = s5 \& (ch_r 2s = ack0): \{s4\};
           1 : state;

    \begin{array}{r}
      34 \\
      35 \\
      36 \\
      37 \\
      37 \\
      \end{array}

          esac:
       next(ch_s2r) :=
          case
            ch_s2r = empty & state = s1 : data0;
38
            ch_s2r = empty & state = s4 : data1;
           1 : ch_s2r;
39
40
          esac;
41
       next(ch_r2s) :=
4\overline{2}
          case
43
            (ch_r2s in {ack0, ack1}) & (state = s2 | state = s5): empty;
44
            1 : ch_r2s;
45
          esac;
46
       next(sa) :=
47
          case
48
           state = s0 & sa = sendd0 : sendd1;
49
           state = s3 & sa = sendd1 : sendd0;
50
           1 : sa:
\frac{51}{52}
          esac;
53 FAIRNESS state = s3
54
     FAIRNESS state = s0
55
```

```
56
    FAIRNESS running
57
\begin{array}{c} 58 \\ 59 \end{array}
60
61
     MODULE receiver(ch_s2r,ch_r2s,ra)
62
63
     VAR
64
       state : {r0,r1,r2,r3,r4,r5,r6,r7};
65
66
    ASSIGN
67
       init(state) := r0;
68
       next(state) :=
69
         case
70
            -- non-determinism represents channel corruption
\dot{7}1
            state = r0 & (ch_s2r = data0) : {r1,r2};
\overline{72}
           state = r0 & (ch_s2r = data1) : {r1};
\begin{array}{c} 73 \\ 74 \\ 75 \\ 76 \\ 77 \\ 78 \\ 79 \end{array}
           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;
80
            state = r6 & ra = received1 : r7;
81
           state = r7 \& (ch_r2s = empty) : r0;
\tilde{82}
           1 : state;
\bar{8}3
         esac;
84
       next(ch_r2s) :=
\tilde{85}
         case
86
           ch_r2s = empty & (state in {r1,r7}): ack1;
87
           ch_r2s = empty & (state in {r3,r5}): ack0;
88
           1 : ch_r2s;
89
         esac;
<u>90</u>
       next(ch_s2r) :=
91
         case
\tilde{92}
            (ch_s2r in {data0,data1}) & (state = r0 | state = r4): empty;
93
            1 : ch_s2r;
94
         esac;
\tilde{95}
       next(ra) :=
96
         case
\tilde{97}
           state = r2 & ra = received0 : received1;
98
           state = r6 & ra = received1 : received0;
99
           1 : ra;
100
         esac;
101
102
103 FAIRNESS state = r2
104 FAIRNESS state = r6
105
106 FAIRNESS running
107
108
109 MODULE main
110
111 VAR
112
       ch_s2r : {empty,data0,data1};
113
       ch_r2s : {empty,ack0,ack1};
114
       sen : process sender(ch_s2r,ch_r2s,sa);
115
       rec : process receiver(ch_s2r,ch_r2s,ra);
116
       sa : {sendd0, sendd1};
117
       ra : {received0, received1};
118
1\overline{19} assign
120 init(ch_s2r) := empty;
```

```
121
      init(ch_r2s) := empty;
122
      init(sa) := sendd0;
123
      init(ra) := received0;
\frac{124}{125}
126\, -- no deadlock, there are paths fulfilling the fairness conditions
\overline{127} spec
128
      EG 1
129
130 -- sender can always send if it wants to
131 spec
132
      AG ((sa = sendd0 -> EF sa = sendd1) & (sa = sendd1 -> EF sa = sendd0))
133
134 -- data is transmitted in right order
135 spec
136
      AG (sa = sendd1 -> A [sa = sendd1 U ra = received1]) &
137
      AG (ra = received1 -> A [ra = received1 U sa = sendd0]) &
138
      AG (sa = sendd0 -> A [sa = sendd0 U ra = received0]) &
139
      AG (ra = received0 \rightarrow 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: 0.4666667 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

```
1
     -- interleaving
\tilde{2}
     ___
\frac{1}{3}
     -- direct representation of global transition relation as formula
     ----
-- sender and receiver of upper level: none
     -- sender and receiver of lower level
     -- channels
     -----
<u>9</u>
     -- channel corruption modelled by non-determinism in sender and receiver of
10
    -- lower level
11
     ---
12
     -- fairness only for state s0
13
14
15
    MODULE main
16
17
    VAR
18
      s2r : {empty,data0,data1};
19
      r2s : {empty,ack0,ack1};
20
      s : \{s0, s1, s2, s3, s4, s5\};
21
     r : {r0,r1,r2,r3,r4,r5,r6,r7};
```

```
22
23
     TNTT
24
       s = s0 \& r = r0 \& s2r = empty \& r2s = empty
25
\overline{26}
     TRANS
27
       s = s0 &
28
              next(s) = s1 & next(s2r) = s2r & next(r2s) = r2s & next(r) = r |
\overline{29}
       s = s1 \& (s2r = empty) \&
30
              (next(s) = s2) \& (next(s2r) = data0) \& next(r2s) = r2s \& next(r) = r |
31
       s = s2 \& r2s = ack0 \&
32
              (next(s) = s1 | next(s) = s3) \&
33
              next(s2r) = s2r \& next(r2s) = empty \& next(r) = r |
\frac{34}{35}
        s = s2 \& (r2s = ack1) \&
              next(s) = s1 & next(s2r) = s2r & next(r2s) = empty & next(r) = r |
\frac{36}{37}
       s = s3 &
              next(s) = s4 \& next(s2r) = s2r \& next(r2s) = r2s \& next(r) = r
\frac{38}{39}
       s = s4 & (s2r = empty) &
              next(s) = s5 & next(s2r) = data1 & next(r2s) = r2s & next(r) = r |
40
       s = s5 \& (r2s = ack1) \&
41
              (next(s) = s4 | next(s) = s0) \&
42
              next(s2r) = s2r \& next(r2s) = empty \& next(r) = r |
43
        s = s5 \& (r_{2s} = a_{ck}0) \&
44
              next(s) = s4 & next(s2r) = s2r & next(r2s) = empty & next(r) = r |
45
46
        (r = r0 & (s2r = data0) &
47
              (next(r) in {r1,r2}) & next(s2r) = empty & next(r2s) = r2s |
48
       r = r0 & (s2r = data1) &
49
              next(r) = r1 & next(s2r) = empty & next(r2s) = r2s |
\begin{array}{c} 50 \\ 51 \end{array}
       r = r1 & (r2s = empty) &
              next(r) = r0 \& next(s2r) = s2r \& next(r2s) = ack1 |
52
       r = r^{2} k
53
             next(r) = r3 & next(s2r) = s2r & next(r2s) = r2s |
54
       r = r3 & (r2s = empty) &
55
             next(r) = r4 \& next(s2r) = s2r \& next(r2s) = ack0
56
       r = r4 & (s2r = data1) &
57
          (next(r) in {r5,r6}) & next(s2r) = empty & next(r2s) = r2s |
58
       r = r4 & (s2r = data0) &
\overline{59}
              next(r) = r5 \& next(s2r) = empty \& next(r2s) = r2s |
60
       r = r5 \& (r2s = empty) \&
61
              next(r) = r4 \& next(s2r) = s2r \& next(r2s) = ack0
62
       r = r6 &
63
             next(r) = r7 & next(s2r) = s2r & next(r2s) = r2s |
64
       r = r7 \& (r2s = empty) \&
65
              next(r) = r0 \& next(s2r) = s2r \& next(r2s) = ack1) \& next(s) = s
66
67
     FAIRNESS
68
       s = s0
-- no deadlock, there are paths fulfilling the fairness conditions
     SPEC
72
       EG 1
7\overline{3}
74
75
     -- data is transmitted in right order
     SPEC
76
       AG (r in \{r0, r1\} \rightarrow A [r in \{r0, r1\} \cup s = s2]) &
77
        AG (s in \{s1,s2\} \rightarrow A [s in \{s1,s2\} U r = r4]) &
\frac{78}{79}
        AG (r in \{r4, r5\} \rightarrow A [r in \{r4, r5\} U s = s5]) &
        AG (s in \{s4, s5\} \rightarrow 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<sup>4</sup>.45943) out of 432 (2<sup>8</sup>.75489)
```

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

A.3.1 The SMV program

```
1
     -- interleaving
\mathbf{2}
3
     -- sender and receiver of upper level: none
4
     -- sender and receiver f lower level: as modules
56789
     -- channels: as variables
    ---
     -- channel corruption modelled by non-determinism in sender and receiver of
     -- lower level
     ---
10
    -- fairness only for state s0 and running for sender of lower level
11
12
13 MODULE sender(ch_s2r,ch_r2s)
14
15 var
16
       state : {s0,s1,s2,s3,s4,s5};
17
18 ASSIGN
19
     init(state) := s0;
\frac{20}{21}
      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};
\frac{\overline{27}}{28}
           -- wrong ack -> s1
          state = s2 & (ch_r2s = ack1) : {s1};
\tilde{29}
           state = s3 : s4;
30
           state = s4 & (ch_s2r = empty) : s5;
31 \\ 32 \\ 33 \\ 34 \\ 35
           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;
40
         esac;
41
       next(ch_r2s) :=
42
         case
43
           (ch_r2s in {ack0, ack1}) & (state = s2 | state = s5): empty;
44
           1 : ch_r2s;
45
         esac;
46
47
     -- ensures that all nondeterministic choices in sender and receiver are fair
48 FAIRNESS state = s0
```

```
49
50
    FAIRNESS running
MODULE receiver(ch_s2r,ch_r2s)
     VAR
\frac{56}{57}
       state : {r0,r1,r2,r3,r4,r5,r6,r7};
58
     ASSIGN
59
       init(state) := r0;
60
       next(state) :=
61
         case
62
           -- non-determinism represents channel corruption
63
            state = r0 & (ch_s2r = data0) : {r1,r2};
64
            state = r0 & (ch_s2r = data1) : {r1};
65
           state = r1 & (ch_r2s = empty) : r0;
66
           state = r2 : r3;
67
           state = r3 & (ch_r2s = empty) : r4;
68
            -- non-determinism represents channel corruption
69
           state = r4 & (ch_s2r = data1): {r5,r6};
\begin{array}{c} 70\\71\\72\\73\\74\\75\\76\\77\\80\\81\\82\\83\\84\\85\end{array}
           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;
86
87
88
89
         esac;
90
     MODULE main
91
\tilde{92}
     VAR
93
       ch_s2r : {empty,data0,data1};
94
       ch_r2s : {empty,ack0,ack1};
95
       sen : process sender(ch_s2r,ch_r2s);
96
       rec : process receiver(ch_s2r,ch_r2s);
\tilde{97}
98 ASSIGN
99
       init(ch_s2r) := empty;
100
      init(ch_r2s) := empty;
101
102
103
104 -- no deadlock, there are paths fulfilling the fairness conditions
105 \,\,\mathrm{spec}
106
       EG 1
107
108\, -- data is transmitted in right order
109 spec
110
      AG (rec.state in {r0,r1} -> A [rec.state in {r0,r1} U sen.state = s2]) &
111
       AG (sen.state in \{s1,s2\} \rightarrow A [sen.state in \{s1,s2\} U rec.state = r4]) &
112
       AG (rec.state in {r4,r5} -> A [rec.state in {r4,r5} U sen.state = s5]) &
113 AG (sen.state in \{s4,s5\} \rightarrow 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

```
1
     -- interleaving
\mathbf{2}
     ---
\overline{3}
     -- sender and receiver of upper level: none
4
     -- sender and receiver of lower level: as modules
5
    -- channels: as modules
6
    --
7
     -- channel corruption modelled by non-determinism in the channel modules
8
9
     -- fairness only for state s0 and running for sender of lower level
10
    -- additional fairness in order to forbid an infinite sequence of channel
11
    -- corruptions
12
13
14
    MODULE sender(cs2r,cr2s,s2rnew)
15
16
    VAR
17
      state : {s0,s1,s2,s3,s4,s5};
18
19 ASSIGN
\frac{20}{21}
      init(state) := s0;
      next(state) :=
22
       case
23
           state = s0 : s1;
24
           state = s1 & (cs2r = empty) : s2;
```

```
25
             state = s2 \& cr2s = ack0 : s3;
26
             state = s2 & (cr2s in {ack1,cor_ack}) : s1;
state = s3 : s4;
             state = s4 & (cs2r = empty) : s5;
             state = s5 & (cr2s = ack1): s0;
30
             state = s5 & (cr2s in {ack0,cor_ack}): s4;

    \begin{array}{r}
      31 \\
      32 \\
      33 \\
      34 \\
      35
    \end{array}

            1 : state;
          esac;
        next(cs2r) :=
          case
             cs2r = empty & state = s1 : data0;
36
            cs2r = empty & state = s4 : data1;
\frac{37}{38}
            1 : cs2r;
          esac;
39
        next(s2rnew) :=
40
          case
41
            cs2r = empty \& state = s1 : 1;
42
            cs2r = empty \& state = s4 : 1;
43
            1 : s2rnew;
44
          esac;
45
        next(cr2s) :=
46
          case
47
            state = s2 & cr2s = ack0 : empty;
48
            state = s2 & (cr2s in {ack1,cor_ack}) : empty;
49
             state = s5 & (cr2s = ack1): empty;
50
            state = s5 & (cr2s in {ack0,cor_ack}): empty;
1 : cr2s;
          esac;
     FAIRNESS state = s0
\frac{56}{57}
     FAIRNESS running
\frac{58}{59}
     MODULE receiver(cs2r,cr2s,r2snew)
60
61
      VAR
62
       state : {r0,r1,r2,r3,r4,r5,r6,r7};
63
64
     ASSIGN
65
       init(state) := r0;
66
        next(state) :=
67
          case
68
            state = r0 \& (cs2r = data0) : r2;
69
             state = r0 & (cs2r in {data1,cor_data}) : r1;
\overline{70}
            state = r1 & (cr2s = empty) : r0;
\begin{array}{c} 71 \\ 72 \\ 73 \\ 74 \\ 75 \\ 76 \\ 77 \\ 78 \\ 79 \\ 80 \end{array}
            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) :=
\begin{array}{c} 81\\ 82\\ 83\end{array}
          case
            state = r1 & (cr2s = empty) : ack1;
            state = r3 & (cr2s = empty) : ack0;
\frac{84}{85}
            state = r5 \& (cr2s = empty) : ack0;
            state = r7 & (cr2s = empty) : ack1;
86
            1 : cr2s;
87
          esac;
88
        next(r2snew) :=
89
          case
```

```
90
           state = r1 & (cr2s = empty) : 1;
91
           state = r3 & (cr2s = empty) : 1;
92
           state = r5 \& (cr2s = empty) : 1;
<u>9</u>3
           state = r7 \& (cr2s = empty) : 1;
94
           1 : r2snew;
95
         esac;
96
       next(cs2r) :=
97
         case
98
           state = r0 & (cs2r = data0) : empty;
99
           state = r0 & (cs2r in {data1,cor_data}) : empty;
100
           state = r4 & (cs2r = data1): empty;
101
           state = r4 & (cs2r in {data0,cor_data}): empty;
102
           1 : cs2r;
103
         esac;
104
105
106
107 MODULE ch_s2r
108
109~{\rm var}
110 \quad \texttt{s} \; : \; \{\texttt{empty},\texttt{cor_data},\texttt{data0},\texttt{data1}\} \; ;
      -- corr and new are necessary to make the use of the language
111
112
      -- construct FAIRNESS possible
113
       -- if corr could be changed more often than data is received then corr
114
      -- could always be 0 when s is changed (corrupted)
115
      corr : boolean;
116
      new : boolean;
117
118~{\rm assign}
119
       init(s) := empty;
120
       next(s) :=
121
         case
\frac{122}{123}
           (s = data0 | s = data1) & new & corr: cor_data;
           s = data0 & new & !corr : data0;
124
           s = data1 & new & !corr : data1;
125
           1 : s;
126
         esac;
\frac{127}{128}
       next(new) :=
         case
129
           new : 0;
130
           new = 0 : 0;
131
         esac:
132
       -- random decision always for each next datum (Zufallsentscheidung immer
133
       -- f"ur das jeweils n"achste Datum)
134
       next(corr) :=
135
         case
136
           new : {0,1};
137
           1 : corr;
138
         esac;
139
140 FAIRNESS corr = 0
141 -- we do not need a FAIRNESS running here since if this module is not
142 -- executed at all means that no corruption has occurred;
143 -- corr and new can be reset after the data has already been read
144 -- this is no problem with respect to the fairness of corr
145
146 MODULE ch_r2s
147
148 var
149 \quad \texttt{s} \ : \ \{\texttt{empty},\texttt{cor\_ack},\texttt{ack0},\texttt{ack1}\}; \\
150 corr : boolean;
151
      new : boolean;
152
153 assign
154 init(s) := empty;
```

```
155
      next(s) :=
156
        case
157
          (s = ack0 | s = ack1) & new & corr: cor_ack;
158
           s = ack0 & new & !corr : ack0;
159
          s = ack1 & new & !corr : ack1;
160
          1 : s;
161
        esac;
162
      next(new) :=
163
        case
164
          new : 0;
165
          new = 0 : 0;
166
        esac;
167
      -- random decision always for each next datum
168
      next(corr) :=
169
        case
170
          new : {0,1};
171
          1 : corr;
172
        esac;
173
174 FAIRNESS corr = 0
175
176
177\, MODULE main
178
179 var
180 sen : process sender(s2r.s,r2s.s,s2r.new);
181 rec : process receiver(s2r.s,r2s.s,r2s.new);
182
      s2r : process ch_s2r;
183
      r2s : process ch_r2s;
184
185
186
187 -- no deadlock, there are paths fulfilling the fairness conditions
188 spec
189
      EG 1
190
191 -- data is transmitted in right order
192 {\rm \ spec}
193
      AG (rec.state in \{r0, r1\} \rightarrow A [rec.state in \{r0, r1\} U sen.state = s2]) &
194
      AG (sen.state in {s1,s2} -> A [sen.state in {s1,s2} U rec.state = r4]) &
195
      AG (rec.state in {r4,r5} -> A [rec.state in {r4,r5} U sen.state = s5]) &
196
      AG (sen.state in \{s4, s5\} \rightarrow 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

1

```
\frac{\overline{2}}{3}
     ___
                NAME: ABP_NM_NU.ni.gv.smv
     ---
              AUTHOR: Armin Biere (armin@ira.uka.de)
4
5 \\ 6 \\ 7
     -- ABP Alternating Bit Protocol
     -- NM No Media modelled
     -- NU No users modelled
8
     -- ni non interleaving
ğ
     -- gv synchronize via global variables
10
11
     -- This is the alternating Bit Protokoll as described in:
12
     -- Automatic Verification of Finite-State Concurrent Systems Using
13
     -- Temporal Logic Specifications, by E.M. Clarke, E.A. Emerson
14
     -- and A.P. Sistla, in ACM Transactions on Programming Languages
15
     -- and Systems. Volume 8. No.2. April 1986. Pages 244--263.
16
     -- No lower or higher media is simulated.
17
18
     -- But we do include the transmission of the data.
19
20
     -- The main difference between this description of the Alternating
\overline{2}1
     -- Bit Protocol and that mentioned above is that no interleaving
\overline{22}
     -- semantic is used. Because the smv system restricts multiple
23
     -- assignement of a variable in different modules we can't use
24
     -- global variables to exchanges signals between modules.
\frac{2}{25}
\overline{26}
     27
     -- This time we don't use modules at all. So we can use the global
28
     -- variable approach to synchronize sender and receiver. This is
29
     -- possible since the sender of a signal only wants to write a signal
\overline{30}
31
     -- if it is zero and the receiver vice versa.
     32
33
     MODULE main
\frac{34}{35}\frac{36}{36}
     VAR.
      snd : boolean;
                          -- signal from sender to receiver:
\frac{37}{38}\\ 39\\ 39
                          -- set by the sender and reset by the receiver
                          -- signal from receiver to sender:
      rcv : boolean;
                          -- set by the receiver and reset by the sender
40
41
      SNDstate : {
42
                          -- there must be an extra state to generate the
         prepareSend,
43
                          -- the data we want to transmit
44
                          -- send data and control bit ( see data )
          send.
45
          receive,
                          -- receive acknowledgement of the receiver
46
          transmitted
                          -- we got the right acknowledgement
47
      };
48
       Smsg : boolean;
                          -- what will be transmitted
49
       SNDdata : { dm00, dm01, dm10, dm11, err };
50
       SNDcontrol : boolean;
51
\frac{52}{53}
      RCVstate : {
54 \\ 55
        receive.
                          -- wait for data to receive
                          -- generate an acknowledgement according
         prepareAck,
5\overline{6}
                          -- to the control bit and the received data.
57
                          -- Also it is possible to generate an error.
58
         send,
                          -- send the acknowledgement to the sender.
59
                          -- got data with the right control bit.
        received
60
      };
```

```
61
      Rmsg : boolean;
62
      RCVcontrol : boolean;
63
      RCVdata : { am0, am1, err };
64
65
66
    ASSIGN
67
      68
      -- the manipulation of rcv and snd are the only global operations
69
      70
      init(snd) := 0;
71 \\ 72
      next(snd) :=
        case
73 \\ 74 \\ 75 \\ 76 \\ 77 \\ 77
         SNDstate = send & ! snd : 1;
        RCVstate = receive & snd : 0;
         1
                                   : snd;
        esac;
      init(rcv) := 0;
\begin{array}{c} 78 \\ 79 \end{array}
      next(rcv) :=
        case

        SNDstate = receive & rcv : 0;

        RCVstate = send & ! rcv : 1;

        1
        : rcv;

80
81
82
83
84
         1
        esac;
\tilde{85}
      -- *************
\breve{86}
      -- this is the sender
87
      -- **************
88
      next(SNDdata) :=
89
        case
90
          SNDstate = prepareSend & Smsg & ! SNDcontrol : { err, dm10 };
          SNDstate = prepareSend & ! Smsg & ! SNDcontrol : { err, dm00 };
91
92
          SNDstate = prepareSend & Smsg & SNDcontrol : { err, dm11 };
93
          SNDstate = prepareSend & ! Smsg & SNDcontrol : { err, dmO1 };
94
          1
                                                         : SNDdata;
95
        esac;
96
      next(Smsg) :=
97
        case
98
          SNDstate = transmitted : { 0, 1}; -- generate new data to send
99
                                            -- Keep it the same so that
          1
                : Smsg;
100
                                            -- the receiver gets the
101
                                             -- right one. We don't have
102
                                             -- buffer for the data!
103
        esac:
104
      init(SNDcontrol) := 0;
105
      next(SNDcontrol) :=
106
        case
107
         SNDstate = transmitted : ! SNDcontrol;
108
                     : SNDcontrol;
          1
109
        esac;
      init(SNDstate) := prepareSend;
110
111
      next(SNDstate) :=
112
        case
113
                                          : send;
          SNDstate = prepareSend
114
          SNDstate = send & ! snd
                                           : receive;
          SNDstate = receive & rcv :
115
116
           case
117
             SNDcontrol :
118
               case
119
                 RCVdata = am1
                                          : transmitted;
120
                 RCVdata = am0
                                                           -- we got a wrong ack:
121
                RCVdata = err
                                           : send;
                                                         -- send again
122
               esac;
123
            ! SNDcontrol :
124
              case
1\bar{2}5
                 RCVdata = amO
                                           : transmitted:
```

```
126
                RCVdata = am1
                                                       -- we got a wrong ack:
127
                RCVdata = err
                                    : send;
                                                      -- send again
128
             esac:
129
          esac;
         SNDstate = transmitted
130
                                       : prepareSend;
131
         1
                                         : SNDstate;
132
        esac;
133
134
      135
      -- the description of the receiver follows
136
      137
      init(RCVcontrol) := 0;
138
      next(RCVcontrol) :=
139
       case
140
         RCVstate = received : ! RCVcontrol;
141
         1
                             : RCVcontrol;
142
       esac:
143
      next(RCVdata) :=
144
       case
145
         RCVstate = prepareAck :
146
           case
147
            RCVcontrol :
148
              case
149
                SNDdata in { dm11, dm01 } : { am1, err };
150
                SNDdata = err : err;
151
                                       : { am0, err };
                1
152
              esac;
153
            ! RCVcontrol :
154
              case
155
                SNDdata in { dm10, dm00 } : { am0, err };
156
                SNDdata = err
                                : err;
157
               1
                                       : { am1, err };
158
              esac;
159
           esac:
160
         1
                                       : RCVdata;
161
        esac;
162
      next(Rmsg) :=
163
       case
164
         RCVstate = receive & snd :
165
           case
166
            SNDdata in { dm10, dm11 }
                                     : 1;
167
             SNDdata in { dmOO, dmO1 } : 0;
168
           esac;
169
         1
                                      : Rmsg;
170
        esac;
171
      init(RCVstate) := receive;
172
      next(RCVstate) :=
\frac{173}{174}
        case
         RCVstate = receive & snd
                                      : prepareAck;
175
         RCVstate = prepareAck
                                       : send;
176
         RCVstate = send & ! rcv
                                      :
177
           case
\frac{178}{179}
            RCVcontrol :
              case
                          -- the receiver has choosen nondeterministically
180
                          -- to generate an error or not. This means we have
181
                          -- to check our own data that we have prepared for
182
                          -- acknowledging.
183
                RCVdata = am1 : received;
184
                1
                              : receive; -- receive again
185
               esac:
186
             ! RCVcontrol :
187
              case
188
               RCVdata = amO : received;
189
                1
                              : receive; -- receive again
190
              esac:
```

```
191
             esac;
192
           RCVstate = received
                                   : receive;
193
          1
                                   : RCVstate;
194
         esac;
195
196 fairness
197 ! SNDdata = err
198 FAIRNESS
199 ! RCVdata = err
200
201 -- first of all liveness specifications
202 SPEC -- ensure that the transition relation is not empty
203 EF SNDstate = transmitted
204 \ {\rm spec}
205
      AG AF SNDstate = transmitted
206 spec
207
      AG AF SNDstate = send
208
209
           -- correct transmission of a one bit
210
           -- this means that when the sender sends a one bit the
211
           -- receiver does not enter his received state without
212
           -- having received one bit:
213~{\rm spec} --
214
      AG ( (SNDstate = send & Smsg) ->
215
            A [ (! RCVstate = received ) U (RCVstate = received & Rmsg) ] )
\overline{216}
217\, SPEC \, -- correct transmission of a zero bit \,
218
      AG ( (SNDstate = send & ! Smsg) ->
219
            A [ (! RCVstate = received ) U (RCVstate = received & ! Rmsg) ] )
```

B.2 Global variables, no users

```
1
               NAME: ABP_M_NU.ni.gv.smv
     ---
\frac{\overline{2}}{3}
    ___
              AUTHOR: Armin Biere (armin@ira.uka.de)
\frac{4}{5}
    -- ABP Alternating Bit Protocol
    -- M the medium is supported
\frac{6}{7}
    -- NU No users modelled
    -- ni non interleaving
8
    -- gv synchronize via global variables
9
10
    -- This is an *extended* version of the alternating Bit Protokoll
11
    -- as described in
12
    -- Automatic Verification of Finite-State Concurrent Systems Using
13
    -- Temporal Logic Specifications, by E.M. Clarke, E.A. Emerson
14
    -- and A.P. Sistla, in ACM Transactions on Programming Languages
15
    -- and Systems. Volume 8. No.2. April 1986. Pages 244--263.
16
17
    -- Only a lower Media is modeled but no users of the service.
18
    -- In this version the data is realy transported between the
19
    -- sender and the receiver. The lower medium can loose messages
20
    -- and error detaction is performed. So this describes a transport
\overline{21}
    -- protocol over a loosy channel.
\overline{22}
23
    -- The main difference between this description of the Alternating
24
    -- Bit Protocol and that mentioned above is that no interleaving
25
    -- semantic is used. Because the smv system restricts multiple
\overline{26}
    -- assignement of a variable in different modules we can't use
27
    -- global variables to exchanges signals between modules.
28
29
    30
    -- This time we don't use modules at all. So we can use the global
31
    -- variable approach to synchronize sender and receiver. This is
```

```
32
    -- possible since the sender of a signal only wants to write a signal
33
    -- if it is zero and the receiver vice versa.
34
    \overline{35}
36
    -- On the other hand we would like to ensure syntactically that
37
     -- the sender and the receiver only comunicate via the medium and
38
    -- dont't inspect the data or states of the partner.
39
    -- With the SMV system this is only possible if we use a complicated
40
    -- signaling approach which does't seem realy appropriate (compare
41
     -- with the specific version using this mechanism)
42
43
    MODULE main
44
45
    VAR
46
      sndReq : boolean;
                             -- Request from the sender to the media
47
                            -- set by the sender and reset by the medium
48
                            -- this is the acknowledge provided by the
      sndAck : boolean;
49
                            -- lower medium when the receiver sends
50
                             -- a higher level acknowledge to the sender
\frac{51}{52}
                             -- it is set by the medium and reset by the
                            -- sender.
5\overline{3}
5\overline{4}
                            -- the receiver tells medium via this signal
      rcvRes : boolean:
                             -- that he wants to send an ackowledgement
-- the medium reports a data to the receiver
      rcvInd : boolean;
      SNDstate : {
58
         prepareSend,
                          -- there must be an extra state to generate the
\overline{59}
                          -- the data we want to transmit
60
                          -- send data and control bit ( see data )
         send
61
         receive,
                         -- receive acknowledgement of the receiver
62
                         -- we got the right acknowledgement
          transmitted
63
      };
64
                          -- what will be transmitted
       Smsg : boolean;
       SNDdata : { dm00, dm01, dm10, dm11 };
65
66
       SNDcontrol : boolean;
67
68
\overline{69}
      RCVstate : {
-- wait for data to receive
        receive,
         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.
        send.
        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
      }:
\frac{88}{89}
       RCV2SNDdata : { am0, am1, err };
90
      RCV2SNDstate : {
91
         receive.
92
         error,
93
         noerror,
94
          send
\tilde{95}
      };
```

```
<u>96</u>
```

```
97
98 ASSIGN
99
     -- *****
100
    -- here we have to manage the seting and reseting of all signals
101
     102
     init(sndReq) := 0; -- Request from sender
103
     next(sndReq) :=
104
      case
105
        SNDstate = send & ! sndReq
                                : 1;
106
        SND2RCVstate = receive & sndReq : 0;
107
        1
                                 : sndReq;
108
      esac:
109
     init(sndAck) := 0; -- Ackowledgement reached sender
110
     next(sndAck) :=
111
      case
112
        SNDstate = receive & sndAck : 0;
113
        RCV2SNDstate = send & ! sndAck : 1;
114
        1
                                 : sndAck;
115
      esac;
116
     init(rcvInd) := 0; -- Indication of request from sender
117
     next(rcvInd) :=
118
      case
11\bar{9}
        RCVstate = receive & rcvInd
                               : 0;
120
        SND2RCVstate = send & ! rcvInd : 1;
121
        1
                                 : rcvInd;
122
      esac:
123
     init(rcvRes) := 0; -- response to sender from receiver
124
     next(rcvRes) :=
125
      case
\overline{126}
        RCVstate = send & ! rcvRes : 1;
127
        RCV2SNDstate = receive & rcvRes : 0;
128
       1
                                : rcvRes;
129
      esac;
130
131
     132
     -- this is the channel from the sender to the receiver
133
     134
     init(SND2RCVstate) := receive;
135
     next(SND2RCVstate) :=
136
      case
137
        SND2RCVstate = receive & sndReq
                                            : { error, noerror };
138
        SND2RCVstate = error
                                              : send;
139
        SND2RCVstate = noerror
                                              : send;
140
        SND2RCVstate = send & ! rcvInd
                                              : receive;
141
                                              : SND2RCVstate;
        1
142
      esac:
143
     next(SND2RCVdata) :=
144
      case
145
        SND2RCVstate = receive & sndReq :
146
         case
147
          SNDdata = dmOO : dmOO;
148
          SNDdata = dm01 : dm01;
149
          SNDdata = dm10 : dm10;
150
          SNDdata = dm11 : dm11;
151
         esac:
152
        SND2RCVstate = error
                                               : err;
153
        1
                                               : SND2RCVdata;
154
       esac:
155
156
     157
     -- here comes the channel from the receiver to the sender
158
     159
     init(RCV2SNDstate) := receive;
160
     next(RCV2SNDstate) :=
161
      case
```

```
162
           RCV2SNDstate = receive & rcvRes
                                                               : { error, noerror };
163
           RCV2SNDstate = error
                                                               : send;
164
           RCV2SNDstate = noerror
                                                               : send;
165
           RCV2SNDstate = send
                                                               : receive;
166
                                                               : RCV2SNDstate;
           1
167
         esac;
168
       next(RCV2SNDdata) :=
169
         case
170
          RCV2SNDstate = receive & rcvRes :
171
           case
172
              RCVdata = amO : amO;
173
              RCVdata = am1 : am1;
174
             esac;
175
         RCV2SNDstate = error
                                                               : err;
\begin{array}{c} 176 \\ 177 \end{array}
                                                                : RCV2SNDdata;
           1
         esac;
178
179
       -- **************
180
       -- this is the sender
181
       -- ***********
182
       next(SNDdata) :=
                                   -- here we don't have to generate errors
183
                                   -- because the medium does it
184
         case
185
           SNDstate = prepareSend & Smsg & ! SNDcontrol : dm10;
186
           SNDstate = prepareSend & ! Smsg & ! SNDcontrol : dm00;
SNDstate = prepareSend & Smsg & SNDcontrol : dm11;
187
188
           SNDstate = prepareSend & ! Smsg & SNDcontrol : dm01;
189
           1
                                                               : SNDdata;
190
         esac:
191
       next(Smsg) :=
192
         case
193
          SNDstate = transmitted : { 0, 1}; -- generate new data to send
194
                                                 -- keep it the same so that
           1
                                   : Smsg;
195
                                                 -- the receiver gets the
196
                                                 -- right one. We don't have
197
                                                 -- buffer for the data!
198
         esac;
199
       init(SNDcontrol) := 0;
200
       next(SNDcontrol) :=
201
         case
202
         SNDstate = transmitted : ! SNDcontrol;
203
          1
                                     : SNDcontrol;
204
         esac:
\overline{2}05
       init(SNDstate) := prepareSend;
206
       next(SNDstate) :=
207
         case
208
                                         : send;
: receive;
           SNDstate = prepareSend
209
           SNDstate = send & ! sndReq
\overline{2}10
           SNDstate = receive & sndAck :
211
             case
212
              SNDcontrol :

    \begin{array}{c}
      \overline{213} \\
      214
    \end{array}

                 case
                   RCV2SNDdata = am1
                                               : transmitted;
\bar{2}\bar{1}\bar{5}
                    RCV2SNDdata = am0
                                                                  -- we got a wrong ack:
\bar{2}\bar{1}6
                  RCV2SNDdata = err
                                                                 -- send again
                                                : send;
217
                 esac;
! SNDcontrol :
                case
\overline{2}20
                   RCV2SNDdata = amO
                                                : transmitted;
221
                   RCV2SNDdata = am1
                                                                  -- we got a wrong ack:
222
                  RCV2SNDdata = err
                                                : send;
                                                               -- send again
223
                 esac;
\overline{2}24
             esac:
225
           SNDstate = transmitted
                                                : prepareSend;
\bar{2}\bar{2}\bar{6}
           1
                                                 : SNDstate:
```

```
227
         esac;
228
\bar{2}\bar{2}\bar{9}
       -- ************
230
       -- the description of the receiver follows
\overline{231}
       232
       init(RCVcontrol) := 0;
233
      next(RCVcontrol) :=
\bar{2}\bar{3}\bar{4}
        case
235
          RCVstate = received : ! RCVcontrol;
\tilde{2}\tilde{3}6
                  : RCVcontrol;
          1
\bar{2}37
         esac;
238
      next(RCVdata) :=
\overline{239}
        case
240
         RCVstate = prepareAck :
\bar{2}41
           case
242
              RCVcontrol :
243
               case
244
                  SND2RCVdata in { dm11, dm01 } : am1;
245
                  1
                                                : am0;
esac;
              ! RCVcontrol :
248
                case
\bar{2}49
                 SND2RCVdata in { dm10, dm00 } : am0;
\overline{250}
                  1
                                                  : am1;
\bar{2}51
                esac;
252
             esac;
253
           1
                                                  : RCVdata;
254
         esac;
255
       next(Rmsg) :=
\overline{256}
        case
257
          RCVstate = receive & rcvInd :
258
            case
259
              SND2RCVdata in { dm10, dm11 }
                                                : 1;
\overline{2}60
               SND2RCVdata in { dm00, dm01 }
                                                  : 0;
\bar{2}61
               1
                                                  : Rmsg;
262
             esac;
263
           1
                                                  : Rmsg;
264
         esac;
\overline{2}65
       init(RCVstate) := receive;
\overline{2}66
      next(RCVstate) :=
267
        case
                                              : prepareAck;
268
          RCVstate = receive & rcvInd
269
           RCVstate = prepareAck
                                                 : send;
570
           RCVstate = send & ! rcvRes
                                                  :
271
            case
272
              RCVcontrol :
273
                case
                              -- the receiver has choosen nondeterministically
\frac{\overline{274}}{275}
                               -- to generate an error or not. This means we have
                              -- to check our own data that we have prepared for
                              -- acknowledging.
276
\bar{2}\bar{7}\bar{7}
                 RCVdata = am1 : received;
\overline{278}
                 1
                                  : receive; -- receive again
279
                esac;
\overline{280}
               ! RCVcontrol :
\bar{2}81
                 case
282
                 RCVdata = am0 : received;
283
                  1
                                  : receive; -- receive again
\bar{284}
                esac:
\overline{2}85
             esac;
286
           RCVstate = received
                                  : receive;
287
          1
                                   : RCVstate;
288
         esac;
289
290 \ {\rm fairness}
291 SND2RCVstate = noerror
```

```
292 fairness
293
      RCV2SNDstate = noerror
294
295 -- first of all liveness specifications
296 SPEC -- ensure that the transition relation is not empty
297
      EF SNDstate = transmitted
298 SPEC
299
      AG AF SNDstate = transmitted
300 \text{ spec}
301
      AG AF SNDstate = send
302
303 spec --
304
      AG ( (SNDstate = prepareSend & Smsg) ->
305
           A [ (! RCVstate = received ) U (RCVstate = received & Rmsg) ] )
306
307 SPEC -- correct transmission of a zero bit
308
      AG ( (SNDstate = prepareSend & ! Smsg) ->
309
           A [ (! RCVstate = received ) U (RCVstate = received & ! Rmsg) ] )
```

B.3 Global variables

```
1
                NAME: ABP_M_U.ni.gv.smv
     ___
\frac{1}{2}
     ---
              AUTHOR: Armin Biere (armin@ira.uka.de)

  \frac{4}{5} \frac{5}{6} \frac{7}{7}

    -- ABP Alternating Bit Protocol
    -- M the medium is supported
    -- II
           users modelled
     -- ni non interleaving
8
    -- gv synchronize via global variables
9
10
    -- This is an *extended* version of the alternating Bit Protokoll
11
    -- as described in
19
     -- Automatic Verification of Finite-State Concurrent Systems Using
13
    -- Temporal Logic Specifications, by E.M. Clarke, E.A. Emerson
14
    -- and A.P. Sistla, in ACM Transactions on Programming Languages
15
    -- and Systems. Volume 8. No.2. April 1986. Pages 244--263.
16
17
    -- Only a lower Media is modeled but no users of the service.
18
    -- In this version the data is realy transported between the
19
    -- sender and the receiver. The lower medium can loose messages
20
    -- and error detaction is performed. So this describes a transport
21
     -- protocol over a loosy channel.
22
23
    -- The main difference between this description of the Alternating
\frac{\overline{24}}{25}
    -- Bit Protocol and that mentioned above is that no interleaving
     -- semantic is used. Because the smv system restricts multiple
26
     -- assignement of a variable in different modules we can't use
27
     -- global variables to exchanges signals between modules.
28
29
    -- ***********
                     *******
30
    -- This time we don't use modules at all. So we can use the global
31
     -- variable approach to synchronize sender and receiver. This is
32
     -- possible since the sender of a signal only wants to write a signal
33
     -- if it is zero and the receiver vice versa.
34
     \frac{35}{36}
     -- On the other hand we would like to ensure syntactically that
\overline{37}
     -- the sender and the receiver only comunicate via the medium and
38
    -- dont't inspect the data or states of the partner.
39
    -- With the SMV system this is only possible if we use a complicated
40
    -- signaling approach which does't seem realy appropriate (compare
41
     -- with the specific version using this mechanism)
42
```

```
43
     MODULE main
44
45
     VAR.
46
       sndReq : boolean;
                                -- Request from the sender to the media
47
                                 -- set by the sender and reset by the medium
48
        sndAck : boolean;
                                 -- this is the acknowledge provided by the
49
                                 -- lower medium when the receiver sends
50 \\ 51 \\ 51
                                 -- a higher level acknowledge to the sender
                                 -- it is set by the medium and reset by the
52 \\ 53 \\ 54
                                 -- sender.
        rcvRes : boolean;
                                -- the receiver tells medium via this signal
                                 -- that he wants to send an ackowledgement
55
        rcvInd : boolean;
                                 -- the medium reports a data to the receiver
56
                                 -- The user {\tt A} sends a request and the user
       usrAReq : boolean;
57
                                 -- B gets an indication
\overline{58}
        usrBInd : boolean;
59
60
       SNDstate : {
61
           waitForReq,
                             -- Wait for user request of a transmission
62
                             -- there must be an extra state to generate the
           prepareSend,
                             -- the data we want to transmit
63
64
                             -- send data and control bit ( see data )
           send
65
           receive,
                             -- receive acknowledgement of the receiver
66
           transmitted
                             -- we got the right acknowledgement
67
       };
68
                            -- what will be transmitted
       Smsg : boolean;
        SNDdata : { dm00, dm01, dm10, dm11 };
69

    \begin{array}{c}
      70 \\
      71 \\
      72 \\
      73 \\
      74
    \end{array}

        SNDcontrol : boolean;
       RCVstate : {
74 \\ 75 \\ 76 \\ 77 \\ 78 \\ 79
         receive,
          prepareAck,
                             -- generate an acknowledgement according
                             -- to the control bit and the received data.
                             -- send the acknowledgement to the sender.
          send,
                             -- got data with the right control bit.
         received,
                             -- send indication to the user
          sendInd
};
        Rmsg : boolean;
        RCVcontrol : boolean;
        RCVdata : { am0, am1 };
        \texttt{SND2RCVdata} \ : \ \{ \ \texttt{dm00}, \ \texttt{dm01}, \ \texttt{dm10}, \ \texttt{dm11}, \ \texttt{err} \ \} \ ;
86
        SND2RCVstate : {
\bar{87}
          receive,
88
           error.
89
           noerror,
90
           send
\tilde{9}\tilde{1}
        };
92
93
        RCV2SNDdata : { am0, am1, err };
94
        RCV2SNDstate : {
95
          receive,
\tilde{96}
           error,
97
           noerror,
98
           send
99
       };
100
101
        USRAdata : boolean;
102
        USRAstate : {
          prepareData,
103
104
          send
105
       };
106
107
        USRBdata : boolean:
```

```
108
    USRBstate : {
109
     receive,
110
      processData
111
     };
112
113 assign
114
    -- ******************
115
     -- communication between the lower medium and the service
116
     117
     init(sndReq) := 0; -- Request from sender
118
     next(sndReq) :=
119
      case
120
       SNDstate = send & ! sndReq
                                : 1;
121
        SND2RCVstate = receive & sndReq : 0;
122
       1
                                 : sndReg;
1\bar{2}\bar{3}
      esac;
124
     init(sndAck) := 0; -- Ackowledgement reached sender
125
     next(sndAck) :=
126
      case
1\overline{2}7
        SNDstate = receive & sndAck : 0;
RCV2SNDstate = send & ! sndAck : 1;
128
129
       1
                                 : sndAck:
130
      esac;
131
     init(rcvInd) := 0; -- Indication of request from sender
132
     next(rcvInd) :=
133
      case
134
       RCVstate = receive & rcvInd : 0;
135
        SND2RCVstate = send & ! rcvInd : 1;
136
       1
                                 : rcvInd:
137
      esac;
138
     init(rcvRes) := 0; -- response to sender from receiver
139
     next(rcvRes) :=
140
      case
141
       RCVstate = send & ! rcvRes
                                : 1;
142
       RCV2SNDstate = receive & rcvRes : 0;
143
       1
                                 : rcvRes;
144
      esac;
145
146
     147
     -- communication between the medium and the user instances
148
     149
     init(usrAReq) := 0;
150
     next(usrAReq) :=
151
      case
152
       USRAstate = send & ! usrAReq
                                        : 1;
                                       : 0;
153
       SNDstate = waitForReq & usrAReq
154
       1
                                         : usrAReq;
155
      esac;
156
     init(usrBInd) := 0;
157
     next(usrBInd) :=
158
      case
159
       RCVstate = sendInd & ! usrBInd : 1;
160
       USRBstate = receive & usrBInd
                                         : 0:
161
        1
                                         : usrBInd;
162
      esac;
163
164
     165
     -- this is the channel from the sender to the receiver
166
     167
     init(SND2RCVstate) := receive;
168
     next(SND2RCVstate) :=
169
      case
                                            : { error, noerror };
      SND2RCVstate = receive & sndReq
170
171
        SND2RCVstate = error
                                               : send;
172
        SND2RCVstate = noerror
                                               : send:
```

```
173
          SND2RCVstate = send & ! rcvInd
                                          : receive;
174
                                                      : SND2RCVstate;
         1
175
       esac;
176
      next(SND2RCVdata) :=
177
      case
178
         SND2RCVstate = receive & sndReq :
        case
179
         SNDdata = dm00 : dm00;
SNDdata = dm01 : dm01;
SNDdata = dm10 : dm10;
SNDdata = dm11 : dm11;
esac;
180
181
182
183
184
185
       SND2RCVstate = error
                                                     : err:
186
                                                      : SND2RCVdata;
        1
187
       esac;
188
189
      -- *********************
190
      -- here comes the channel from the receiver to the sender
191
      192
      init(RCV2SNDstate) := receive;
193
      next(RCV2SNDstate) :=
194
      case
195
       RCV2SNDstate = receive & rcvRes : { error, noerror };
196
        RCV2SNDstate = error
                                                      : send;
197
         RCV2SNDstate = noerror
                                                      : send;
198
        RCV2SNDstate = send
                                                      : receive;
199
        1
                                                      : RCV2SNDstate;
200
       esac;
201
     next(RCV2SNDdata) :=
202
       case
       RCV2SNDstate = receive & rcvRes :
203
204
         case
205
        RCVdata = am0 : am0;
RCVdata = am1 : am1;
esac;
\overline{206}
\overline{2}07
       RCV2SNDstate = error
1
208
                                                      : err:
209
                                                      : RCV2SNDdata;
210
      esac;
\bar{2}\bar{1}\bar{1}
212
      213
      -- This is the user A who wants to send data to the user B
214
      215
      init(USRAstate) := prepareData;
\overline{2}\overline{1}\overline{6}
      next(USRAstate) :=
217
       case
       218
219
220
                                                : USRAstate;
        1
5\bar{2}\bar{1}
       esac;
222
      next(USRAdata) :=
\bar{2}\bar{2}\bar{3}
      case
\bar{2}\bar{2}\bar{4}
       USRAstate = prepareData : { 1, 0 };
225
                                                : USRAdata;
        1
\overline{226}
       esac;
\overline{2}\overline{2}\overline{7}
228
      -- *************
229
      -- here comes user B
230
      -- ************
231
      init(USRBstate) := receive;
232
     next(USRBstate) :=
233
      case
                                          : processData;
: receive;
       USRBstate = receive & usrBInd
USRBstate = processData
1
234
\overline{235}
\overline{2}36
                                                : USRBstate;
237
      esac:
```

```
238
      next(USRBdata) :=
239
       case
240
       USRBstate = receive & usrBInd : Rmsg;
241
                                                : USRBdata;
         1
\bar{2}4\bar{2}
       esac;
243
244
      -- ************
245
      -- this is the sender
246
      -- *************
\bar{2}47
      next(SNDdata) :=
                               -- here we don't have to generate errors
248
                               -- because the medium does it
249
        case
\overline{250}
         SNDstate = prepareSend & Smsg & ! SNDcontrol : dm10;
          SNDstate = prepareSend & ! Smsg & ! SNDcontrol : dm00;
251
\overline{2}\overline{5}\overline{2}
         SNDstate = prepareSend & Smsg & SNDcontrol : dm11;
SNDstate = prepareSend & ! Smsg & SNDcontrol : dm01;
\overline{2}53
254
                                                        : SNDdata;
         1
255
        esac;
256
      next(Smsg) :=
\overline{257}
        case
\overline{2}58
         SNDstate = waitForReq & usrAReq : USRAdata;
259
                                                  : Smsg;
         1
260
        esac;
261
      init(SNDcontrol) := 0;
262
      next(SNDcontrol) :=
263
        case
264
        SNDstate = transmitted : ! SNDcontrol;
265
         1
                    : SNDcontrol;
266
        esac:
267
      init(SNDstate) := waitForReq;
268
      next(SNDstate) :=
269
        case
SNDstate = waitForReq & usrAReq : prepareSend;
          SNDstate = prepareSend : send;
272
          SNDstate = send & ! sndReq
                                          : receive;
          SNDstate = receive & sndAck :
273
274
          case
275
            SNDcontrol :
\bar{2}\bar{7}\bar{6}
            case
\overline{2}77
              RCV2SNDdata = am1 : transmitted;
RCV2SNDdata = am0 |
278
                                                           -- we got a wrong ack:
                                        : send; -- send again
279
            RCV2SNDdata = err
280^{-1}
         esac;
! SNDcontrol :
               esac;
581
282
             case
                                       : transmitted;
283
                RCV2SNDdata = amO
\bar{2}84
                RCV2SNDdata = am1
                                                          -- we got a wrong ack:
285
                RCV2SNDdata = err
                                                         -- send again
                                          : send;
\overline{286}
               esac;
287
           esac:
                                      : waitForReq;
288
        SNDstate = transmitted
\bar{289}
         1
                                          : SNDstate;
290
        esac:
291
292
      -- *****
293
      -- the description of the receiver follows
\bar{2}94
      295
      init(RCVcontrol) := 0;
296
      next(RCVcontrol) :=
297
       case
298
       RCVstate = received : ! RCVcontrol;
299
                             : RCVcontrol;
         1
300
       esac;
301
      next(RCVdata) :=
302
       case
```

303 RCVstate = prepareAck : 304 case 305RCVcontrol : 306case SND2RCVdata in { dm11, dm01 } : am1; 3073081 : am0; 309esac: 310! RCVcontrol : $311 \\ 312$ case SND2RCVdata in { dm10, dm00 } : am0; 313 1 : am1; 314 esac; $\frac{315}{316}$ esac; 1 : RCVdata; $\begin{array}{c} 317\\ 318 \end{array}$ esac; next(Rmsg) := 319case 320RCVstate = receive & rcvInd : 321case 322SND2RCVdata in { dm10, dm11 } : 1; 323 SND2RCVdata in { dm00, dm01 } : 0; 3241 : Rmsg; $\overline{3}25$ esac; 3261 : Rmsg; $\tilde{3}\bar{2}\tilde{7}$ esac; 328init(RCVstate) := receive; $\bar{3}2\bar{9}$ next(RCVstate) := 330case 331RCVstate = receive & rcvInd : prepareAck; RCVstate = prepareAck 332: send; : 333RCVstate = send & ! rcvRes 334 case $\bar{3}\bar{3}5$ RCVcontrol : 336case RCVdata = am1 : received; 3373381 : receive; -- receive again 339esac; 340! RCVcontrol : 341case 342 RCVdata = am0 : received; 3431 : receive; -- receive again 344esac; 345esac; 346 RCVstate = received : sendInd; 347 RCVstate = sendInd & 348! usrBInd : receive; 3491 : RCVstate; 350esac; 351352 fairness 353 SND2RCVstate = noerror 354 fairness $\frac{355}{356}$ RCV2SNDstate = noerror 357 SPEC -- transition relation is not empty under the fairness constraints 358 EF USRBstate = processData $359 \,\, {\rm spec}$ 360 -AG AF USRBstate = processData 361362 spec 363 AG ((USRAstate = send & USRAdata) -> 364A [!USRBstate = processData U (USRBstate = processData & USRBdata)]) 365 spec 366 AG ((USRAstate = send & ! USRAdata) -> 367A [!USRBstate = processData U (USRBstate = processData & ! USRBdata)])

B.4 With signal modules

```
\frac{1}{2}
                    NAME: ABP_NM_NU.ni.sm.smv
     ___
     ---
                  AUTHOR: Armin Biere (armin@ira.uka.de)
\overline{3}
4
    -- ABP Alternating Bit Protocol
5
    -- NM No Media modelled
6
    -- NU No users modelled
7
     -- ni non interleaving
8
    -- sm synchronize via signal modules
ğ
10
    -- This is the alternating Bit Protokoll as described in:
11
     -- Automatic Verification of Finite-State Concurrent Systems Using
12
     -- Temporal Logic Specifications, by E.M. Clarke, E.A. Emerson
13
     -- and A.P. Sistla, in ACM Transactions on Programming Languages
14
    -- and Systems. Volume 8. No.2. April 1986. Pages 244--263.
15
16
    -- No lower or higher media is simulated.
17
     -- But we do include the transmission of the data.
18
19
    -- The main difference between this description of the Alternating
20
    -- Bit Protocol and that mentioned above is that no interleaving
21
    -- semantic is used. Because the smv system restricts multiple
22
     -- assignement of a variable in different modules we can't use
\overline{2}\overline{3}
    -- global variables to exchanges signals between modules.
24
\overline{25}
     26
    -- Here we use a seperate signal module which is responsible
27
     -- for a signal. The sender (respectively the receiver)
\overline{28}
    -- can request to set (reset) the signal. Than the signal
29
    -- module will do this for him.
30
    -- The reason for this complicated scheme is that the smv
31
     -- language does not support multiple assignement of variable
32
     -- in different modules (though both modules who want to
33
    -- write to the variable don't do it simoustanly).
\frac{34}{35}
     36
    MODULE main
37
38
    VAR.
39
      snd : signal;
40
      rcv : signal;
41
      SND : sender(RCV.data, rcv, snd);
                                           -- No media supported. So we have to
42
                                           -- access the data in common store.
43
       RCV : receiver(SND.data, snd, rcv);
44
45
     -- first of all four liveness specifications
46
     SPEC -- ensure that the transition relation is not empty
47
      EF SND.state = transmitted
48
     SPEC
49
      AG AF SND.state = transmitted
50
    SPEC
51
      AG AF SND.state = sendSet
52
53
           -- correct transmission of a one bit
54
           -- this means that when the sender sends a one bit the
55
           -- receiver does not enter his received state without
56
           -- having received one bit:
57
     SPEC --
58
      AG ( (SND.state = sendSet & SND.Smsg) ->
59
           A [ (! RCV.state = received ) U (RCV.state = received & RCV.Rmsg) ] )
60
61
    SPEC -- correct transmission of a zero bit
62
      AG ( (SND.state = sendSet & ! SND.Smsg) ->
63
           A [ (! RCV.state = received ) U (RCV.state = received & ! RCV.Rmsg) ] )
```

```
64
65
66
     MODULE signal
67
     VAR
68
       sig : boolean;
69
       set : boolean;
                           -- the sender of this signal *owns* this bit.
\begin{array}{c} 70 \\ 71 \\ 72 \\ 73 \\ 74 \\ 75 \\ 76 \\ 77 \\ 78 \\ 79 \end{array}
                           -- 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;
: sig;
           1
          esac;
\tilde{83}
84
85
     MODULE sender(rdata, IN, OUT) -- rdata is the data of the receiver
                                      -- IN = rcv. OUT = snd
\breve{8}\breve{6}
     VAR
87
       Smsg : boolean;
88
       control : boolean;
89
       state : {
90
          prepareSend, -- there must be an extra state to generate the
91
                           -- the data we want to transmit
92
\tilde{9}\bar{3}
                           -- the next to sates send the data ( OUT = snd )
94
                           -- Wait for ! snd.sig
           sendWait.
95
                           -- Wait for snd.sig
           sendSet,
96
97
                            -- the next two states receive the acknowledgement
98
                           -- ( IN = rcv )
99
           receiveWait, -- Wait for rcv.sig
100
           receiveReset, -- Wait for ! rcv.sig
101
102
          transmitted -- we got the right acknowledgement
103
       } :
104
       data : { dm00, dm01, dm10, dm11, err };
105
106~{\rm assign}
107
       next(data) :=
108
         case
109
           state = prepareSend & Smsg & ! control : { err, dm10 };
110
            state = prepareSend & ! Smsg & ! control
                                                          : { err, dmOO };
           state = prepareSend & Smsg & control : { err, dm11 };
state = prepareSend & ! Smsg & control : { err, dm01 };
111
112
113
           1
                                                            : data:
114
          esac;
115
       next(Smsg) :=
116
          case
117
           state = transmitted : { 0, 1}; -- generate new data to send
1 : Smsg; -- keep it the same so that
118
119
                                                 -- the receiver gets the
120
                                                 -- right one. We don't have
121
                                                 -- buffer for the data!
\overline{1}\overline{2}\overline{2}
          esac;
123
       init(control) := 0;
124
       next(control) :=
125
         case
126
          state = transmitted : ! control;
127
           1
                                     : control;
1\bar{2}8
          esac:
```

```
129
      init(state) := prepareSend;
130
      init(IN.reset) := 0;
131
      next(IN.reset) :=
132
        case
133
         state = receiveReset & IN.sig
                                          : 1;
134
          1
                                            : 0;
135
        esac:
136
      init(OUT.set) := 0;
137
      next(OUT.set) :=
138
        case
139
          state = sendSet & ! OUT.sig : 1;
140
         1
                                           : 0:
141
        esac:
142
      next(state) :=
143
        case
144
         state = prepareSend
                                           : sendWait;
145
         state = sendWait & ! OUT.sig
                                           : sendSet;
146
         state = sendSet & OUT.sig
                                          : receiveWait;
147
         state = receiveWait & IN.sig :
148
           case
149
             control :
150
               case
                 rdata = am1
151
                                          : receiveReset;
152
                 rdata = amO
                                                           -- we got a wrong ack:
153
                 rdata = err
                                           : sendWait; -- send again
154
               esac:
155
            ! control :
156
               case
157
                 rdata = amO
                                           : receiveReset;
158
                 rdata = am1
                                                           -- we got a wrong ack:
159
                 rdata = err
                                                          -- send again
                                           : sendWait:
160
               esac;
161
           esac;
162
          state = receiveReset & ! IN.sig : transmitted;
163
          state = transmitted
                                           : prepareSend;
164
          1
                                           : state;
165
        esac;
166
             -- the sender is responsible on his own that an error
167
             -- in the underlying media is generated.
168
            -- This is accomplished by allowing the data to be
169
             -- nondeterministically choosen between the real data (data bit
170
             -- and control bit) and an error. To ensure a correct transmission
171
             -- of data by the ABP we have to impose a restriction. Namely that
172
             -- the media does not always generate an error:
173 fairness --
174 ! data = err
175
176 MODULE receiver(sdata, IN, OUT) -- sort of pram model for data transfer
177
                                  -- IN = snd, OUT = rcv
178 var
179 Rmsg : boolean;
180
      control : boolean;
181
      state : {
182
        receiveWait,
                        -- wait for data to come in ( snd.sig = 1 )
183
                      -- reset snd.sig
        receiveReset,
184
185
        prepareAck,
                        -- generate an acknowledgement according
186
                        -- to the control bit and the received data.
187
                        -- Also it is possible to generate an error.
188
189
                        -- send the acknowledgement to the sender.
190
        sendWait,
                        -- wait for signal to be read by sender
191
        sendSet,
                        -- now set it ( rcv.sig = 1 )
192
193
        received
                        -- got data with the right control bit.
```

```
194 };
195
       data : { am0, am1, err };
196
197~{\rm assign}
198
       init(control) := 0;
199
       next(control) :=
200
         case
201
          state = received : ! control;
202
           1
                                : control;
\overline{2}03
         esac;
\overline{2}04
       next(data) :=
205
         case
206
          state = prepareAck :
207
            case
\overline{2}08
              control :
209
                  case
210
                   sdata in { dm11, dm01 } : { am1, err };
211
                     sdata = err : err;
212
                                               : { am0, err };
                    1
\frac{\overline{2}\overline{1}\overline{3}}{214}
                  esac;
               ! control :
215
                  case
\overline{2}16
                   sdata in { dm10, dm00 } : { am0, err };
\bar{2}\bar{1}\bar{7}
                   sdata = err : err;

    \begin{array}{c}
      218 \\
      219
    \end{array}

                    1
                                                : { am1, err };
                  esac;
220
              esac:
221
            1
                                               : data;
\overline{2}\overline{2}\overline{2}
         esac:
\bar{2}\bar{2}\bar{3}
       next(Rmsg) :=
224
         case
225
          state = receiveWait & IN.sig :
226
            case
\frac{1}{227}
                                              : 1;
              sdata in { dm10, dm11 }
228
                sdata in { dm00, dm01 }
                                                : 0;
229
              esac;
\overline{230}
            1
                                                 : Rmsg;
\frac{\bar{2}31}{232}
          esac;
        init(IN.reset) := 0;
233
       next(IN.reset) :=
234
         case
235
           state = receiveWait & IN.sig
                                               : 1;
236
           1
                                                 : 0;
\overline{2}\overline{3}7
         esac;
\overline{2}38
       init(OUT.set) := 0;
239
       next(OUT.set) :=
240
         case
241
          state = sendWait & ! OUT.sig : 1;
\overline{2}42
           1
                                                 : 0;
243
         esac:
244
       init(state) := receiveWait;
245
       next(state) :=
246
         case
247
           state = receiveWait & IN.sig : receiveReset;
\bar{2}48
            state = receiveReset & ! IN.sig : prepareAck;
249
           state = prepareAck
                                                : sendWait;
250
            state = sendWait & ! OUT.sig :
\bar{2}51
              case
\overline{2}52
                 control :
253
                                  -- the receiver has choosen nondeterministically
                  case
254
                                  -- to generate an error or not. This means we have
255
                                  -- to check our own data that we have prepared for
\overline{256}
                                 -- acknowledging.
                     data = am1 : sendSet;
1 : receiveWait; -- receive again
\overline{2}57
258
```

```
259
                esac;
260
               ! control :
261
                case
262
                  data = amO
                              : sendSet;
263
                               : receiveWait; -- receive again
                  1
\frac{1}{264}
                esac;
265
            esac;
266
           state = sendSet & OUT.sig : received;
267
           state = received
                                   : receiveWait;
268
          1
                                    : state;
269
         esac:
270 fairness
271
      ! data = err
```

B.5 Synchronization via state investigation

```
1
     ___
                NAME: ABP_NM_NU.ni.si.smv
{}^2_{3}_{4}
     ---
              AUTHOR: Armin Biere (armin@ira.uka.de)
     -- ABP Alternating Bit Protocol
5678
     -- NM No Media modelled
     -- NU No users modelled
     -- ni non interleaving
     -- si synchronize via state investigation
9
10\, -- This is the alternating Bit Protokoll as described in:
11
     -- Automatic Verification of Finite-State Concurrent Systems Using
12
     -- Temporal Logic Specifications, by E.M. Clarke, E.A. Emerson
13
     -- and A.P. Sistla, in ACM Transactions on Programming Languages
14
     -- and Systems. Volume 8. No.2. April 1986. Pages 244--263.
15
16
     -- This example is a small one because we use synchronous send
17
     -- and receive. In addition no lower or higher media is simulated.
18
     -- But we do include the transmission of the data.
19
20
    -- The main difference between this description of the Alternating
21
     -- Bit Protocol and that mentioned above is that no interleaving
22
     -- semantic is used. Because the smv system restricts multiple
\overline{23}
     -- assignement of a variable in different modules we can't use
24
     -- global variables to exchanges signals between modules.
25
     -- This version ensures the correct transmission of lower media messages
\frac{1}{26} 27
     -- by simultaneous investigation of the states of the sender and
     -- receiver. So this is no real implementation.
\overline{28}
\overline{29}
     MODULE main
VAR.
      SND : sender(RCV.state, RCV.data);
       RCV : receiver(SND.state, SND.data);
34
35
     -- first of all liveness specifications
36
     SPEC -- ensure that the transition relation is not empty
\frac{37}{38}
      EF SND.state = transmitted
     SPEC
39
      AG AF SND.state = transmitted
40
    SPEC
41
      AG AF SND.state = send
42
43
           -- correct transmission of a one bit
44
           -- this means that when the sender sends a one bit the
45
           -- receiver does not enter his received state without
46
           -- having received one bit:
47
     SPEC --
```

```
48
       AG ( (SND.state = send & SND.Smsg) ->
49
             A [ (! RCV.state = received ) U (RCV.state = received & RCV.Rmsg) ] )
50
51
     SPEC -- correct transmission of a zero bit
52
       AG ( (SND.state = send & ! SND.Smsg) ->
53
             A [ (! RCV.state = received ) U (RCV.state = received & ! RCV.Rmsg) ] )
54
55
     MODULE sender(rstate, rdata) -- rstate is the state of receiver
\overline{56}
                                        -- rdata is the data of the receiver
57 \\ 58
     V A R.
59
       Smsg : boolean;
60
       control : boolean;
61
       state : {
        prepareSend, -- there must be an extra state to generate the
62
63
                           -- the data we want to transmit
64
                          -- send data and control bit ( see data )
        send,
65
        receive,
                          -- receive acknowledgement of the receiver
66
          transmitted -- we got the right acknowledgement
67
       };
6\dot{8}
       data : { dm00, dm01, dm10, dm11, err };
\underline{69}
\begin{array}{ccc} 75 \\ 70 \\ 71 \\ 71 \\ 72 \\ 73 \\ 73 \\ 74 \end{array}
       next(data) :=
         case
           state = prepareSend & Smsg & ! control : { err, dm10 };
74 \\ 75 \\ 76 \\ 77 \\ 78
          state = prepareSend & ! Smsg & ! control : { err, dm00 };
          state = prepareSend & Smsg & control : { err, dm11 };
state = prepareSend & ! Smsg & control : { err, dm01 };
           1
                                                           : data;
         esac:
79
       next(Smsg) :=
80
         case
           state = transmitted : { 0, 1}; -- generate new data to send
1 : Smsg; -- keep it the same so that
81
\overline{82}
-- the receiver gets the
                                               -- right one. We don't have
                                               -- buffer for the data!
86
         esac;
\overline{87}
       init(control) := 0;
88
       next(control) :=
89
         case
90
         state = transmitted : ! control;
91
           1
                                   : control;
\overline{92}
         esac;
93
       init(state) := prepareSend;
94
       next(state) :=
95
       case
<u>96</u>
           state = prepareSend
                                                : send;
           state = send &
97
98
           rstate = receive
                                                : receive;
99
           state = receive &
100
           rstate = send :
101
             case
102
               control :
103
                 case
104
                   rdata = am1
                                                : transmitted;
105
                    rdata = amO |
                                                                   -- we got a wrong ack:
106
                    rdata = err
                                                  : send;
                                                                   -- send again
107
                 esac:
108
              ! control :
109
                 case
110
                   rdata = amO
                                                 : transmitted;
                                                                   -- we got a wrong ack:
111
                    rdata = am1 |
112
                   rdata = err
                                                 : send:
                                                                  -- send again
```

```
113
               esac;
114
            esac;
115
          state = transmitted
                                          : prepareSend;
116
          1
                                           : state;
117
        esac:
118
             -- the sender is responsible on his own that an error
119
             -- in the underlying media is generated.
120
             -- This is accomplished by allowing the data to be
121
             -- nondeterministically choosen between the real data (data bit
122
             -- and control bit) and an error. To ensure a correct transmission
123
             -- of data by the ABP we have to impose a restriction. Namely that
124
             -- the media does not always generate an error:
125 FAIRNESS --
126 ! data = err
127
\overline{128} MODULE receiver(sstate, sdata) \, -- sstate is state of sender
129
130 var
131
     Rmsg : boolean;
132
      control : boolean;
133
      state : {
134
                        -- wait for data to receive
       receive
135
        prepareAck,
                        -- generate an acknowledgement according
136
                        -- to the control bit and the received data.
137
                        -- Also it is possible to generate an error.
138
                        -- send the acknowledgement to the sender.
        send.
139
       received
                        -- got data with the right control bit.
140
      };
141
      data : { am0, am1, err };
142
143 Assign
144
      init(control) := 0;
145
      next(control) :=
146
        case
147
          state = received : ! control;
148
          1
                            : control:
149
       esac;
150
      next(data) :=
151
        case
152
         state = prepareAck :
153
           case
154
             control :
155
              case
156
                 sdata in \{ dm11, dm01 \} : \{ am1, err \};
157
                 sdata = err : err;
158
                 1
                                        : { am0, err };
159
               esac;
160
              ! control :
161
               case
162
                sdata in { dm10, dm00 } : { am0, err };
163
                 sdata = err : err;
164
                                       : { am1, err };
                 1
165
               esac:
166
            esac;
167
          1
                                        : data;
168
        esac;
169
      next(Rmsg) :=
170
        case
171
          state = receive & sstate = send :
172
           case
173
             sdata in { dm10, dm11 }
                                       : 1:
174
              sdata in { dm00, dm01 }
                                     : 0;
175
            esac:
176
          1
                                         : Rmsg;
177
        esac:
```

```
178
       init(state) := receive;
179
       next(state) :=
180
         case
181
            state = receive & sstate = send
                                                 : prepareAck;
182
            state = prepareAck
                                                   : send;
183
            state = send & sstate = receive :
184
             case
185
                control :
186
                                 -- the receiver has choosen nondeterministically
                 case
                                 -- to generate an error or not. This means we have
-- to check our own data that we have prepared for
187
188
189
                                 -- acknowledging.
190
                    data = am1 : received;
191
                   1
                                  : receive; -- receive again
192
                 esac;
\overline{193}
                ! control :
194
                  case
195
                    data = am0 : received;
196
                    1
                           : receive; -- receive again
197
                  esac;
198
              esac;
199
            state = received : receive;
\frac{100}{200}
201
           1
                                  : state;
        esac;
\begin{array}{c} 201 \\ 202 \\ FAIRNESS \\ 203 \\ ! \\ data = err \end{array}
```