Verifying Part of the ACCESS.bus Protocol using PVS

Jozef Hooman

Dept. of Mathematics and Computing Science
Eindhoven University of Technology
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
e-mail: wsinjh@win.tue.nl

Abstract. Based on a compositional framework for the formal specification of distributed real-time systems, we present a method for protocol verification. To be able to deal with realistic examples, the method is supported by the interactive proof checker PVS. In this paper we illustrate our approach by a protocol of the ACCESS.bus which is used for the communication between a computer host and its peripheral devices (e.g., keyboards, mice, joysticks, etc.). The bus supports dynamic reconfiguration while the system is operating. We specify and verify a safety property and a real-time progress property of this industrial example.

1 Introduction

In previous work we have addressed the formal specification and verification of distributed real-time systems (see, e.g., [Hoo91]). A framework based on Hoare triples has been applied to several examples such as a water level monitoring system [Hoo93] and a chemical batch processing system [Hoo94c]. In the current paper we only consider the basic ideas behind this formalism and concentrate on the application of such a method to protocol verification.

Essentially, we use an assertional method in which a system is characterized by expressing its properties in a particular logic. Such a method basically deals with formulas of the form P **sat** spec, denoting that process P satisfies specification spec. For a formal derivation of a program satisfying a certain specification, a proof system is given, i.e., a set of axioms and rules axiomatizing the programming constructs. To be able to reason with the specifications of components, without knowing their implementation, we use a compositional verification method. This means that in a rule for a compound programming construct only the specifications of its components are used. Compositionality makes it possible to verify design steps during the process of top-down system development. A large number of compositional proof systems have been developed, for instance, for sequential systems [Hoa69], untimed systems [Zwi89, PJ91], and real-time systems [Hoo91].

The only programming construct used in this paper is parallel composition. Under certain conditions we have the following compositional rule.

Rule 1 (Parallel Composition). $$\frac{P_1 \ \mathbf{sat}\ \mathsf{spec}_1, \quad P_2 \ \mathbf{sat}\ \mathsf{spec}_2}{P_1 \| P_2 \ \mathbf{sat}\ \mathsf{spec}_1 \wedge \mathsf{spec}_2}$$

The main restriction on the application of this rule is that the specification spec_i of process P_i should only refer to the observable actions of process P_i itself, for $i = 1, 2$ (see, e.g., [Hoo91, Hoo94c] for more details).
We also need a *consequence rule* to weaken a specification.

Rule 2 (Consequence). $\dfrac{P \ \mathbf{sat}\ \mathsf{spec}_0, \mathsf{spec}_0 \rightarrow \mathsf{spec}_1}{P \ \mathbf{sat}\ \mathsf{spec}_1}$

To verify a protocol for a network of processes $P_1 \| \cdots \| P_n$, we first model the application domain and then proceed as follows:

1. Formulate a top-level specification spec_{tl} for the network $P_1 \| \cdots \| P_n$.
2. Axiomatize the communication mechanism between the processes $P_1, \ldots, P_n$ by a set of axioms $COMMAX$.
3. Find a suitable specification for each process P_i of the form P_i **sat** spec_i, where spec_i refers to the external communication interface of P_i only, for $i = 1, \ldots, n$.
4. Prove $\mathsf{spec}_1 \wedge \cdots \wedge \mathsf{spec}_n \rightarrow \mathsf{spec}_{tl}$, using axioms of $COMMAX$ and perhaps other assumptions needed for correctness.
5. Derive a correct implementation of process P_i according to its specification P_i **sat** spec_i, for $i = 1, \ldots, n$.

By the parallel composition rule and the consequence rule, this leads to a proof of $P_1 \| \cdots \| P_n$ **sat** spec_{tl}. Hence we have derived a distributed program satisfying the top-level specification. Observe that in the fourth step the protocol is verified on an abstract level, independent of the implementation of the processes. The correctness of this step is based on compositionality of the parallel composition rule.

The method described above has been used to specify and verify a distributed real-time arbitration protocol [Hoo94a] in our Hoare-style framework. This protocol is inspired by an algorithm of the IEEE 896 Futurebus specification. In [ZH95] the first four steps of the method have been applied to an atomic broadcast protocol which involves both real-time and fault-tolerance. Whereas these examples have been verified manually, it is clear that the application of our formal method to large realistic systems requires some form of mechanical support. One should have a tool to construct proofs, to discharge simple verification conditions automatically, and to check proofs mechanically (e.g., after small changes in the specifications).

Therefore we investigate the use of the verification system PVS[1] (Prototype Verification System) [ORS92] for our method. The PVS specification language is a strongly-typed higher-order logic. Specifications can be structured into a hierarchy of theories. There are a number of built-in theories (e.g., reals, lists, sets, ordering relations, etc.) and a mechanism for automatically generating theories for abstract datatypes. The PVS system contains an interactive proof checker

[1] PVS is free available, see WWW page `http://www.csl.sri.com/sri-csl-pvs.html`

with, for instance, induction rules, automatic rewriting, and powerful decision procedures.

The first four steps of the method described above, supported by PVS, are applied to a part of the ACCESS.bus [ACC94]. ACCESS.bus is used for the communication between a computer host and its peripheral devices, such as keyboards, mice, joysticks, modems, monitors and printers. The ACCESS.bus protocol includes a physical layer and several software layers, including a base protocol, device driver interface, and several specific device protocols. The base protocol defines standard messages for device initialization, device identifications, address assignment, and a message envelope for device reports and control information. Usually the host computer is simply called the *computer*, all other partners on the bus are called *devices*, and together they are called *components*. A bus transaction is called a *message*.

The bus supports dynamic reconfiguration while the system is operating. The base protocol contains a configuration process which is used to assign a unique address to each device. Configuration shall occur at system start-up, reinitialization, or at any time when the computer detects the addition or removal of a device. At system start-up or reinitialization the computer shall send a Reset message to all addresses. When a device is reset or powered-up (henceforth we often refer to this as the *start* of the device), it shall always revert to the power-up default address. Then the device shall send an Attention message to the computer which shall reply with an Identification Request. This shall lead to an Identification Reply message of the device containing its identification string. Finally, the computer shall send an Assign Address message to the device containing its identification string and a new address.

Since there is a possibility that two devices are assigned to the same address, a device first transmits a Reset message to its assigned address before it may send data. This self-addressed Reset message forces other devices at the same address back to the power-up default address. We say that a process is in *data mode* if it may send data (i.e. it has obtained a new address, transmitted a self-addressed Reset message, and has not received any Reset to its new address during some period of time).

We investigate two properties of this protocol:

- A safety property which expresses that if two devices are in data mode then their addresses will be different.
- A progress property, expressing that a powered-up device will be in data mode within a bounded amount of time, provided the host does not send any Reset message.

Since program design is not relevant in the example of the ACCESS.bus protocol, we do not consider the fifth step of the method described above, but concentrate on the specifications. The main point here is step four of this method and we focus on the proof of the required implication in PVS. Therefore all primitives and specifications are defined in the PVS specification language. Although PVS has a facility to generate LaTeX output of theories and proofs, we present our material in the original PVS format to give an impression of the user interface of this tool. First we formulate a few simple real-time primitives in Sect. 2. The configuration of the ACCESS.bus is defined in Sect. 3. Next the safety property and the progress property are specified and verified in, respectively, Sect. 4 and Sect. 5. Concluding remarks can be found in Sect. 6.

2 Real-Time Primitives

To reason about timing properties, we use a time domain which equals the real numbers. In the PVS theory `rt` below this is specified by defining the type `Time` to be equal to the built-in type `real`. Also nonnegative time points are needed, represented by the type `NonNegTime`. Further we define time intervals, using `co` to represent left-closed right-open intervals, etc. (Comments appear after '%'.) The types `setof[Time]` and `pred[Time]` are equivalent to the type `[Time -> bool]` denoting functions from `Time` to the built-in type `bool`.

The standard PVS operator `NOT` on `bool` is overloaded in `rt` such that it is defined on predicates over `Time`, too. Finally it is defined when a time predicate holds `inside` or `during` an interval.

```
rt  : THEORY
BEGIN
Time            : TYPE = real
NonNegTime      : TYPE = {t : Time | 0 <= t}
Interval        : TYPE = setof[Time]

t, t0, t1    : VAR Time
cc( t0, t1 ) : Interval = {t | t0 <= t AND t <= t1} %closed-closed
co( t0, t1 ) : Interval = {t | t0 <= t AND t < t1}  %closed-open
oc( t0, t1 ) : Interval = {t | t0 < t AND t <= t1}  %open-closed
oo( t0, t1 ) : Interval = {t | t0 < t AND t < t1}   %open-open

P          : VAR pred[Time]
NOT ( P ) : pred[Time] = (LAMBDA t : NOT P(t) ) ;

I               : VAR Interval
inside( P, I ) : bool = (EXISTS t : I(t) AND P(t))
dur( P, I )     : bool = (FORALL t : I(t) IMPLIES P(t))
END rt
```

3 Configuration of the ACCESS.bus

To apply the method of Sect. 1 to the ACCESS.bus protocol, first the application domain is modelled. The configuration of the ACCESS.bus is defined in theory `config` which imports the theory `rt` (i.e., the definitions given in `rt` are also available in `config`). Two uninterpreted types `Components` and `Addresses` are defined (for our purpose it is irrelevant how addresses are represented in the ACCESS.bus). The + behind TYPE indicates that the types are assumed to be nonempty. The host computer hc is defined as a constant of type Components and all other components are represented by the type Devices. Further, assume given two functions addr and datamode to denote, respectively, the address of a component at a certain point in time and whether a device is in data mode.

```
config : THEORY
BEGIN
IMPORTING rt
Components, Addresses : TYPE+

hc      : Components           % host computer
Devices : TYPE = {c : Components | c /= hc}

addr     : [ Components -> [Time -> Addresses] ]
datamode : [ Devices -> pred[Time] ]
```

In the ACCESS.bus, components communicate by means of messages which consist in our model of five fields: message identification, source address, destination address, identification string, and new address. Identification strings of devices are defined as an uninterpreted nonempty type. To define message identifications we use an enumeration type with five elements. Such an enumeration type is in fact an abstract datatype in PVS and by typechecking it the PVS system internally stores the information that all elements are different (so, e.g., message identification Reset is different from Attention). This information is used by the decision procedures of PVS. Messages are represented by a record with five fields. Primitives send and rec express that a component sends, respectively receives, a message at a particular point in time.

```
IdStrings  : TYPE+
MessageIds : TYPE = {Reset, Attention, IdReq, IdReply, AssignAddr}
Messages   : TYPE = [# mid     : MessageIds,
                       source  : Addresses,
                       dest    : Addresses,
                       idstr   : IdStrings,
                       newaddr : Addresses
                     #]

send, rec : [ Components, Messages -> pred[Time] ]
```

Henceforth, in this and subsequent theories, the following variables are used:

```
t, t0, t01, t02, t1, t2, t3    : VAR Time
c, c0                          : VAR Components
d, d0, d1, d2                  : VAR Devices
a, a1, a2, newa, newa1, newa2  : VAR Addresses
```

```
mi                              : VAR MessageIds
istr, istr1, istr2              : VAR IdStrings
m                               : VAR Messages
```

By overloading the equality symbol, we can write `(addr(d) = a)(t)` instead of `addr(d)(t) = a`.

```
A           : VAR [Time -> Addresses]
= ( A, a ) : pred[Time] = (LAMBDA t : A(t) = a);
```

Since usually not all five fields of a message are relevant, we introduce a number of abbreviations. Similar abbreviations are defined (but not shown here) for `rec`.

```
send(c,mi)(t) : bool = (EXISTS m: send(c, m WITH [mid := mi])(t) )

send(c,mi,a)(t) : bool = (EXISTS m:
     send(c, m WITH [mid := mi, dest := a])(t) )

send(c,mi,a1,a2)(t) : bool = (EXISTS m:
     send(c, m WITH [mid := mi, source := a1, dest := a2])(t) )

send(c,mi,a1,a2,istr)(t) : bool = (EXISTS m:
     send(c, m WITH [mid := mi, source := a1,
                     dest := a2, idstr := istr])(t) )

send(c,mi,a,istr,newa)(t) : bool = (EXISTS m:
     send(c, m WITH [mid := mi, dest := a, idstr := istr,
                     newaddr := newa])(t) )

send(c,mi,a,istr)(t) : bool =(EXISTS m:
      send(c, m WITH [mid := mi, dest := a, idstr := istr] )(t) )
```

4 A Safety Property

In this section we verify a safety property, expressing that never two devices will be in data mode with the same address. This is done in theory `access1`, following the first four steps of the method presented in Sect. 1. Theory `access1` imports the theory `config`, which means that all the defintions of `config` and `rt` (which is imported by `config`) are available.

```
access1 : THEORY
BEGIN
IMPORTING config
```

4.1 Top-level Specification (spec_{tl})

The first step in our approach is the definition of the top-level specification `top_level1`, expressing that when two different devices are in data mode they have different addresses.

```
top_level1 : bool = (FORALL d1, d2, t:
                  datamode(d1)(t) AND datamode(d2)(t) AND d1 /= d2
                  IMPLIES
                  addr(d1)(t) /= addr(d2)(t))
```

4.2 Axiomatization of the Communication Mechanism (*COMMAX*)

The second step requires the axiomatization of the communication mechanism. The physical layer of the ACCESS.bus is based on the I^2C serial bus, for which the following holds:

> The same message sent at the same time by one or more entities is always correctly received once by every entity with a matching address that does not belong to the set of the senders (of the same message).

To be able to prove real-time properties, we use a nonnegative constant `TD`, representing an upper bound on the transmission time. It is not completely clear at which point in time a receiver should have the matching address; here we take a strong assumption (thus obtaining a weak axiom) by requiring that the receiver has this address during the whole receiving interval of `TD` time units.

```
TD : NonNegTime          % Transmission Delay

comm_ax1 : AXIOM (FORALL c, m, t:
          send(c,m)(t)
          IMPLIES
          (FORALL (c0 | NOT send(c0,m)(t) AND
                        dur( addr(c0) = dest(m) , co(t,t+TD) ) ):
           inside( rec(c0,m) , co(t,t+TD) ) ) )
```

4.3 Specification of the Processes (spec_i)

In step three we specify the components, as far as needed to prove `top_level1`. For this safety property we only have to specify devices. For a device `d`, assertion `DS1(d)` expresses that `d` is in data mode only if it has sent a self-addressed Reset at least `DD` time units earlier and since then no Reset has been received.

```
DD : NonNegTime           % Data mode Delay

DS1(d) : bool = (FORALL t : datamode(d)(t)
                      IMPLIES
                      (EXISTS (t0 | t0 < t - DD), a :
                          dur( addr(d) = a, cc(t0,t) ) AND
                          send(d,Reset,a)(t0) AND
                          dur( NOT rec(d,Reset,a), cc(t0,t) ) ) )
```

4.4 Correctness Proof

The goal of step four is to prove that the conjunction of the properties of all devices (i.e. (FORALL d : DS1(d))) implies top-level specification top_level1. First observe that DD >= TD is required to make sure that the self-addressed Reset of a starting device is received by all possible receivers before the device goes to data mode.

This, however, is not sufficient to prove top_level1. If two devices have the same identification string, they obtain the same address, and if they send their Reset messages simultaneously, then none of them will receive it and both can start sending data from the same address. The documentation [ACC94] does not give a solution for this case, but it seems reasonable to assume that the probability that this situation occurs is very small. To avoid reasoning with probabilities, we assume here that the situation can be neglected, i.e. that simultaneous resets do not occur, as expressed by not_sim_reset.

Then we can prove theorem cortl1, based on lemma cortl1lem which represents the situation that d1 has received an address before d2 sends a Reset to its own address.

```
not_sim_reset : bool = (FORALL t, d1, d2, a:
                        send(d1,Reset,a)(t) AND send(d2,Reset,a)(t)
                        IMPLIES
                        d1 = d2)

cortl1lem : LEMMA DD >= TD AND t01 <= t02 AND d1 /= d2 AND
                  t01 < t - DD AND
                  dur( NOT rec(d1,Reset,a1), cc(t01,t) ) AND
                  dur( addr(d1) = a1, cc(t01,t) ) AND
                  t02 < t - DD AND
                  send(d2,Reset,a2)(t02) AND addr(d2)(t) = a2 AND
                  not_sim_reset
                  IMPLIES
                  addr(d1)(t) /= addr(d2)(t)

cortl1 : THEOREM DD >= TD AND (FORALL d : DS1(d)) AND
                 not_sim_reset
                 IMPLIES
                 top_level1
END access1
```

The keywords LEMMA and THEOREM are equivalent in PVS. All lemmas and theorems presented here have been proved by means of the PVS interactive prover.

For instance, for cortl1lem the application of axiom comm_ax1 leads to three subgoals. One subgoal corresponds to the case that the Reset message of d2 is received by d1. Then we can derive a contradiction with the assumption that d1 does not receive Resets from t01 till t. The other two subgoals are related to the conditions of comm_ax1. Using not_sim_reset we can easily show that d1 does not send a Reset at t02. Finally we prove that d1 has address a2 during a certain interval by simply expanding a few definitions and providing the proper instantiations. In general, the user of the PVS prover gives the main outline of the proof, whereas details can be proved automatically (often by the powerful command "grind" which combines, e.g., automatic rewriting, skolemization, instantiation, and propositional simplification).

5 A Progress Property

In theory access2, which imports access1, we specify and verify a real-time progress property. To express when a device is powered-up we introduce a time dependent predicate up. Predicate start is used to express when a device starts (i.e. becomes powered-up or is reset).

```
access2 : THEORY
BEGIN
IMPORTING access1
up, start : [ Devices -> pred[Time] ]
```

5.1 Top-level Specification (spec_{tl})

Specification top_level2 expresses that a device which starts at t1 and remains up till t2 will be in data mode during [t1+JD,t2), unless it receives a host Reset between t1 and t2, that is, as long as the host does not send a Reset between t1-TD and t2 (note that a transmission might take less than TD).

```
JD : NonNegTime    % Join Delay

top_level2 : bool = (FORALL d, t1, t2:
                         start(d)(t1) AND
                         dur( up(d), co(t1,t2)) AND
                         dur( NOT send(hc,Reset), co(t1-TD,t2))
                         IMPLIES
                         dur( datamode(d), co(t1+JD,t2) ))
```

5.2 Axiomatization of the Communication Mechanism (*COMMAX*)

Since access2 imports access1, axiom comm_ax1 is still available. We add an axiom to express that any message received by a component has been sent before by another component.

```
comm_ax2 : AXIOM (FORALL c, m, t:
       rec(c,m)(t)
       IMPLIES
       (EXISTS (c0 | c0 /= c) : inside( send(c0,m), oc(t-TD,t) )))
```

5.3 Specification of the Processes (spec_i)

In the third step we specify the components of the system. First the behaviour of devices is specified by a number of properties. We define a few constant addresses and a function `devid` to denote the identification string of a device.

```
pda     : Addresses     % power-up default address
hca     : Addresses     % host computer address

devid   : [ Devices -> IdStrings ]
```

A starting device has the power-up default address and sends an Attention message to the host computer address within `AD`, for some timing constant `AD`, provided this device stays up during at least `AD` time units.

```
AD : NonNegTime          % Attention Delay

DS2(d) : bool = (FORALL (t | dur( up(d), co(t,t+AD) )):
               start(d)(t)
               IMPLIES
               inside( send(d,Attention,pda,hca), co(t,t+AD) ) )
```

If a device receives an Identification Request message at the power-up default address, it responds within `IRD` by sending its device identification string to the computer address.

```
IRD : NonNegTime         % Id Reply Delay

DS3(d) : bool = (FORALL (t | dur( up(d), co(t,t+IRD) )):
                       rec(d,IdReq,pda)(t)
                       IMPLIES
                       inside( send(d,IdReply,pda,hca,devid(d)),
                               co(t,t+IRD) ) )
```

If device `d` receives an Assign Address message containing its own device identification string then it will be in data mode after `DD` time units as long as it did not receive any Reset messages to the newly received address. Recall that `DD` has been defined in theory `access1`.

```
DS4(d) : bool = (FORALL t1, t2, newa:
                 dur( up(d), co(t1,t2) ) AND
                 rec(d,AssignAddr,pda,devid(d),newa)(t1) AND
                 dur( NOT rec(d,Reset,newa), co(t1,t2) )
                 IMPLIES
                 dur( datamode(d), co(t1+DD,t2) ) )
```

Further a device only sends a Reset message to an address if it has received this address in an Assign Address message recently.

```
DS5(d) : bool = (FORALL t, a:
                        send(d,Reset,a)(t)
                        IMPLIES
                        inside( rec(d,AssignAddr,pda,devid(d),a),
                                cc(t-DD,t) ) )
```

Devices never send Identification Request or Assign Address messages.

```
DS6(d) : bool = (FORALL t, m:
                       send(d,m)(t)
                       IMPLIES
                       mid(m) /= IdReq AND mid(m) /= AssignAddr)
```

When a device starts, it has the power-up default address pda as long as it did not receive an Assign Address message at least TD time units before.

```
DS7(d): bool = (FORALL t0, t1:
        start(d)(t0) AND dur(up(d), cc(t0,t1)) AND
        dur( NOT rec(d, AssignAddr, pda, devid(d)), cc(t0,t1-TD))
        IMPLIES
        addr(d)(t1) = pda)
```

The delay TD is related to comm_ax1 in access1, which requires that the receiver has the matching address during the whole receiving interval of TD time units. Properties DS6 and DS7 are often used in combination with comm_ax1 to show that a message sent to a device is also received (because the device cannot be a sender of this message and it will have the right address).

```
DS(d) : bool = DS2(d) AND DS3(d) AND DS4(d) AND
               DS5(d) AND DS6(d) AND DS7(d)
```

Next we specify the behaviour of the host computer. Property CS1 expresses that it responds to an Attention message by sending an Identification Request to the address of the sender within HCD1.

```
HCD1, HCD2 : NonNegTime    % Host Computer Delays

CS1 : bool = (FORALL t, a:
                      rec(hc,Attention,a,hca)(t)
                      IMPLIES
                      inside( send(hc,IdReq,a), co(t,t+HCD1) ) )
```

After receiving an Identification Reply, the computer returns an Assign Address message with a new address in less than HCD2 time units.

```
CS2 : bool = (FORALL t, a, istr:
    rec(hc,IdReply,a,hca,istr)(t)
    IMPLIES
    (EXISTS newa :
      inside( send(hc,AssignAddr,a,istr,newa), co(t,t+HCD2) ) ) )
```

The properties specified describe the exchange of messages between the computer and a starting device, resulting in the receipt of an Assign Address message by this device. As expressed by DS4, this only leads to data mode if the device does not receive a Reset to its new address. Therefore we assume that two Assign Address messages which contain different device identification strings, also contain different addresses. Then in the proof of top_level2 it is required that different devices have different identification strings.

```
CS3 : bool = (FORALL t1, t2, istr1, istr2, newa1, newa2:
                   send(hc,AssignAddr,pda,istr1,newa1)(t1) AND
                   send(hc,AssignAddr,pda,istr2,newa2)(t2) AND
                   istr1 /= istr2
                   IMPLIES
                   newa1 /= newa2 )
```

In property CS3 we made a simplification by not considering the re-use of addresses (in contrast with the real protocol which detects the absence of devices by Presence Check messages).

Similar to the devices, we have two properties about messages that are not sent by the computer and about its address. They are used in combination with comm_ax1 to derive that certain messages are received by the computer.

```
CS4: bool = (FORALL t, m:
                    send(hc,m)(t)
                    IMPLIES
                    mid(m) /= Attention AND mid(m) /= IdReply)

CS5: bool = (FORALL t: addr(hc)(t) = hca)

CS : bool = CS1 AND CS2 AND CS3 AND CS4 AND CS5
```

5.4 Correctness Proof

The correctness of the protocol described above with respect to specification top_level2 is proved, provided different devices have different identification strings (as expressed by id_diff). Further there is a constraint on the timing constants, using abbreviation RAB to represent the maximal delay between the start of a device and the moment it receives a new address.

```
id_diff : bool = (FORALL d1, d2: d1 /= d2 IMPLIES
                                 devid(d1) /= devid(d2))

RAB : Time = AD + 4*TD + HCD1 + IRD + HCD2 % Receive Address Bound

cortl2 : THEOREM id_diff AND JD >= RAB + DD AND CS AND
                 (FORALL d: DS(d))
                 IMPLIES
                 top_level2
```

In the proof of this theorem we use a number of lemmas (in the PVS file they occur before the theorem). It is rather straightforward to show that a starting device `d` receives an Assign Address message in less than `RAB` time units.

```
recAssignAddrlem : LEMMA
    DS2(d) AND DS3(d)AND DS6(d) AND DS7(d) AND
    CS1 AND CS2 AND CS4 AND CS5
    IMPLIES
    (FORALL (t | dur( up(d), co(t,t+RAB) )):
        start(d)(t)
        IMPLIES
        inside( rec(d,AssignAddr,pda,devid(d)), co(t,t+RAB) ) ) )
```

Next we show that this leads to datamode if `d` does not receive any Reset message to its new address and provided `JD >= RAB + DD`.

```
datalem : LEMMA DS4(d) AND dur( up(d), co(t1,t2) ) AND
                (EXISTS newa:
                    inside(rec(d,AssignAddr,pda,devid(d),newa),
                           co(t1,t1+RAB)) AND
                    dur(NOT rec(d, Reset, newa), co(t1,t2))) AND
                JD >= RAB + DD
                IMPLIES
                dur( datamode(d), co(t1+JD,t2) )
```

Hence it remains to prove that `d` does not receive any Reset message to its new address. This follows from the fact that the host computer does not send Reset messages and assuming no other device sends Reset messages to this new address.

```
notrecResetlem : LEMMA dur( NOT send(hc,Reset), co(t1-TD,t2)) AND
                   (FORALL (d0 | d0 /= d):
                     dur( NOT send(d0,Reset,newa), co(t1-TD,t2) ))
                   IMPLIES
                   dur( NOT rec(d,Reset,newa), co(t1,t2) )
```

Finally, we show that if the host computer sends a new address to device `d` then any other device will not send Reset messages to this new address. Note that `id_diff` is used here in combination with `CS3`.

```
notsendResetlem : LEMMA CS3 AND id_diff AND
                    (FORALL d: DS5(d)) AND
                    (FORALL d: DS6(d)) AND
                    (EXISTS t:
                        send(hc,AssignAddr,pda,devid(d),newa)(t))
                    IMPLIES
                    (FORALL (d0 | d0 /= d):
                      dur( NOT send(d0,Reset,newa), co(t1-TD,t2) ))
```

6 Concluding Remarks

Part of the ACCESS.bus protocol has been specified and verified using an assertional method supported by PVS. The protocol has real-time aspects and allows dynamic reconfiguration. Recent overviews on industrial applications of formal methods can be found in [Rus93] and [GCR94]. The main aim here was to investigate the support of the interactive theorem prover PVS for our protocol verification method.

An important part of the work described here concerns the tranformation from an informal description [ACC94] into a formal specification. In this paper we have extracted the essential points needed to prove a few properties, ignoring irrelevant implementation details, thus clarifying the protocol and making assumptions for its correctness explicit. For instance, to show the safety property (devices in data mode have different addresses) we had to introduce an assumption about the impossibility of simultanous Reset messages. Further we have slightly generalized the formulation in [ACC94] by introducing timing constants instead of concrete numbers and by deriving constraints on these constants that ensure correctness of the protocol.

Since the step from informal document to formal specification is crucial, we have tried to define our specifications close to the informal formulation. It turns out that the PVS specification language is very suitable for this purpose and by defining appropriate abbreviations the gap with the informal text is reasonably small. Clearly the transformation from informal to formal is an iterative process, since ambiguities have to be resolved, mistakes have to be corrected, etc. To our experience the use of a tool such as PVS increases the speed of this process to a large extent, since it is easy to incorporate changes, to rerun the proofs and to modify proofs interactively.

In the ACCESS.bus example we have not worked out step five of our approach in which a program for the components is derived. An example of program derivation by means of PVS can be found in [Hoo94b] where a mixed formalism, in which programs and assertional specifications are combined, has been defined in the PVS specification language.

Related to the work described here is the use of PVS [ORS92] and its predecessor EHDM for a number of applications, such as an interactive convergence clock synchronization algorithm [RvH93], a Byzantine fault-tolerant clock synchronization [Sha93], and an algorithm for interactive consistency [LR93]. We also mention the specification language TLA (Temporal Logic of Actions) which has been applied to a large number of examples. See, e.g., the specification and the hierarchically structured proof of a Byzantine generals algorithm [LM94]. Another nice example of protocol verification can be found in [BPV94] where an industrial protocol is specified and verified based on timed I/O automata.

Acknowledgements

The ACCESS.bus protocol has been proposed by Ron Koymans (Philips Research, Eindhoven) to a number of academic researchers as an example of an industrial protocol. He is also thanked for valuable comments on preliminary versions of this paper. We are grateful to the anonymous referees for useful suggestions.

References

[ACC94] *ACCESS.bus*$^{\mathrm{TM}}$*, Specifications – Version 2.2.* Sunnyvale, California, 1994.

[BPV94] D. Bosscher, I. Polak, and F. Vaandrager. Verification of an audio control protocol. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 170–192. LNCS 863, 1994.

[GCR94] S. Gerhart, D. Craigen, and T. Ralston. Experience with formal methods in critical systems. *IEEE Software*, 11(1):21–39, 1994.

[Hoa69] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580,583, 1969.

[Hoo91] J. Hooman. *Specification and Compositional Verification of Real-Time Systems.* LNCS 558, Springer-Verlag, 1991.

[Hoo93] J. Hooman. A compositional approach to the design of hybrid systems. In *Workshop on Theory of Hybrid Systems*, pages 121–148. LNCS 736, 1993.

[Hoo94a] J. Hooman. Compositional verification of a distributed real-time arbitration protocol. *Real-Time Systems*, 6(2):173–205, 1994.

[Hoo94b] J. Hooman. Correctness of real time systems by construction. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 19–40. LNCS 863, 1994.

[Hoo94c] J. Hooman. Extending Hoare logic to real-time. *Formal Aspects of Computing*, 6(6A):801–825, 1994.

[LM94] L. Lamport and S. Merz. Specifying and verifying fault-tolerant systems. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 41–76. LNCS 863, 1994.

[LR93] P. Lincoln and J. Rushby. The formal verification of an algorithm for interactive consistency under a hybrid fault model. In *Computer Aided Verification '93*, pages 292–304. LNCS 697, Springer-Verlag, 1993.

[ORS92] S. Owre, J. Rushby, and N. Shankar. PVS: A prototype verification system. In *11th Conference on Automated Deduction*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer-Verlag, 1992.

[PJ91] P. Pandya and M. Joseph. P-A logic – a compositional proof system for distributed programs. *Distributed Computing*, 4(4), 1991.

[Rus93] J. Rushby. Formal methods and the certification of critical systems. Technical Report CSL-93-7, SRI International, November 1993.

[RvH93] J. Rushby and F. von Henke. Formal verification of algorithms for critical systems. *IEEE Transactions on Software Engineering*, 19(1):13–23, 1993.

[Sha93] N. Shankar. Verification of real-time systems using PVS. In *Computer Aided Verification '93*, pages 280–291. LNCS 697, Springer-Verlag, 1993.

[ZH95] P. Zhou and J. Hooman. Formal specification and compositional verification of an atomic broadcast protocol. *Real-Time Systems*, 9(2):119–145, 1995.

[Zwi89] J. Zwiers. *Compositionality, Concurrency and Partial Correctness.* LNCS 321, Springer-Verlag, 1989.