Using the B Method to Model Protocols

Jean-Louis Lanet
Gemplus Research Group,
Av du Pic de Bertagne, 13881 Gémenos Cedex France
tel: +33 (0)4 42 36 64 22 - fax: +33 (0)4 42 36 55 55
jean-louis.lanet@gemplus.com

Abstract: In this paper we suggest to use the B formal method to model a protocol dedicated to smart cards. We use a pragmatic approach to prove the dynamic properties of the protocol by using historical variables to express the past. We check manually that those variables have been correctly updated in the different operations. With this approach we can avoid the use of a model checker to verify the dynamic properties. We show the advantages of this method to express general properties and service properties. We focused on the formalisation of the rules given in the standard and on the refinement process. We introduce new events and invariants at each level. Using this method enabled us to bring to the fore some ambiguities and errors of the protocol.

1. Introduction

The use of formal methods in software development for smart cards is a way to improve the quality of the software and thereby the system security. Moreover the ITSEC [1] state that for the assurance correctness level 4 a semi-formal notation like SDL shall be used. A formal notation like B must be used in order to obtain the certification for the highest level. The B method is suitable to model our critical applications due to the fact that most of these applications are sequential programs. We believe that it is possible to use this method for other applications like protocols as described in [2].

In a previous paper [3] we made a comparison between the C code generator of two formal methods (the B method and SDL) to specify a smart card protocol. The code generation is of a prime importance for a smart card because of the strong size constraints. Herein, we present our approach to verify dynamic properties on this protocol without using a model checker as proposed by [4]. We focus on the different refinement steps and the methodology used.

The application is a protocol used between a smart card and a reader. Two protocols are used within a smart card, and both of them are defined by the standard ISO 7816-3 [5]. The T=0 is character-oriented and is commonly used. But it does not allow data transfer on both directions in a same command. Moreover, it does not optimise exchanges especially in the case of a small I/O buffer in the card RAM memory. The T=1 (block-oriented) allows more control on data exchanged. The standard that defines the protocol uses some general procedures, specific chaining procedures, a set of rules and several informative scenarios to explain some cases. But in fact, certain behaviours are defined neither by the rules nor by the scenarios

(e.g. the order to handle the different errors). Sometimes, the scenarios are ambiguous and do not correspond with the rules.

In the rest of the paper we present the protocol in section 2 and then we describe the formal model of the B specification in section 3. Our conclusions are given in section 4.

2. Overview of the T=1 protocol

The T=1 is a half duplex protocol used to transfer messages sliced in several blocks. To start an exchange, the reader waits for the Answer To Reset (ATR) command including the Protocol Type Selection (PTS) which sets the different parameters of the T=1. After reaching an agreement, the reader initiates the session by sending a first block, switches to a receive mode and starts a timer to control the liveliness of the card. A frame is made of a prologue field (mandatory), a data field (optional) and a checksum (mandatory). The prologue field consists of three bytes, the node address, the protocol control byte and the length of the data field. The protocol control byte (PCB) defines the three types of blocks:

- an information block (Iblock) is used to convey information or positive acknowledgements,
- a receive ready block (Rblock) is used to convey positive or negative acknowledgements,
- a supervisory block (Sblock) is used to exchange control information between the card and the reader. The data field may be present according to the controlling function.

2.1. Error free mode

The first block sent by the reader shall be an Iblock or a Sblock. An acknowledgement must be received before sending the next block. Each Iblock is labelled with the sender sequence number N(s) and is incremented after each sending. N(s) uses a one-bit counter. After sending an I[N(i)], the sender must receive a $R[N(i+1)_{mod2}]$ for a positive acknowledgement and a R[N(i)] to send again the last Iblock. The supervisory blocks do not need a sequence number. It is always a pair of i-request/i-response. When the card or the reader sends a message whose size is greater than the size of the input buffer, the sender slices the message into several Iblocks. The M bit of the prologue indicates that the Iblock will be followed by another one at least. Receiving an Iblock with M=0, means that the receiver is requested to transmit.

The semantic of an Iblock is defined by those two bits: I[N(s),M]. Figure 1 shows two exchanges in an error free protocol transmission. The reader sends a message that must be followed by another one at least. The card acknowledges by sending an Rblock with N(s+1). When the reader sends I (x,0), the card is allowed to send its own message.

When the protocol layer of the card receives an Iblock, it sends it to the application layer without waiting for the whole message, allowing the application layer to process the command. If the card needs more time than allowed by the

reader watchdog, it requests the reader for more execution time by sending a Sblock [Wtx-Request (TimeRequested)]. Such a Sblock must be acknowledged by a Sblock [Wtx-Response (TimeObtained)].

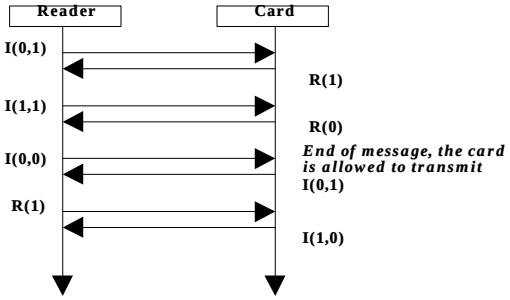

Fig. 1 Exchange of Iblock

The card or the reader can adjust the size of their input buffers by sending a Sblock [Ifs-Request (RequestedSize)]. The answer shall be a Sblock [Ifs-Response (ObtainedSize)]. The two other supervisory commands are Abort to cancel the current message, and Resync to resynchronise the protocol.

2.2. Error handling

The receiver detects transmission errors and sequence errors. Both sides have to handle the following errors:

- parity error (prologue element only) or checksum error,
- invalid PCB,
- length of the information field incompatible with the size of the input buffer,
- loss of synchronisation: less characters than expected have been received,
- failure to receive the response of a supervisory request.

The reader watchdog is the means to detect a mute card or the loss of a block. The fault treatment consists in retransmitting the block. If the retransmission is unsuccessful three consecutive times, a resynchronisation is initiated. If three consecutive resynchronisations fail, the reader resets the card in order to restart the protocol.

2.3. The state machine

The protocol can be represented by a simple state machine, where actions can be authorised only in given state. After receiving a message, the system can evolve into another state by sending a message or in some cases remaining mute. This matrix represents the rules of the protocol. The following table shows a part of the state machine, the smart card side, for the receipt of a Chained Iblock. This first level of formalisation has been a means to discover errors in the informal specifications but it has not been sufficient.

State \ Event	Iblock Chained	
	? I (1,1)	? I(0,1)
s0: Initial State	s0 / !Rblock(R)	s1 / !Rblock(N) or s2 / !Sblock(abortReq) or s1 / !Rblock(R)
s1: Reader Chaining	s1 / !Rblock(N)	s1 / !Rblock(N) or s2 / !Sblock(abortReq) or s1 / !Rblock(R)
s2:Wait Ab.Resp	s2 / !Sblock(abortReq)	s2 / !Sblock(abortReq)

Table 1: Protocol Partial State Machine for smart card side

The whole state machine is a 18*10 complete matrix describing the protocol. For example, if the card is in the initial state (s0) and receives a chained Iblock, the card can response with a positive or negative Rblock according to the receiving status or it can request an abort if the transmitted message is not valid (i.e. a command for dumping a too large memory block). The invariants must reflect this state machine. They must express which message is expected in a given state and which message can be sent. Another property is the correct evolution law of the N counters of the reader and smart card.

3. The B model

3.1. Dynamic properties

For a protocol, most of the invariants refer to dynamic properties which are difficult to express in a simple way with B. For example, the rule 2.2 states that the card can not receive a chained Iblock if it has sent a chained Iblock previously. The next state after sending a chained Iblock can only be a positive or negative Rblock. If we combine it with another rule (rule 9, abort request) we obtain table 1.

One of the approaches [4] is to transform the B model in another formal specification. Then a model checker is applied in order to prove dynamic properties. We believe that this approach introduces an important break in the proof. There are no guarantee that the model verified with the model checker is true to the B model. Another approach is to develop the underlying theory as presented in [6], [7] to incorporate dynamic properties.

A pragmatic approach is to use historical variables to express the state. This approach is possible due to the fact that the only temporal operator we need for this protocol is the **next** operator. The properties stand for relations between the current state and the previous one avoiding us to maintain a large amount of historical variables. We use *cd_PndIbloc*, *cd_PndRbloc* and *cd_PndSbloc* variables to store the information concerning the last block sent. For example, the *cd_PndIbloc*

variable can contain the value *Chained, Unchained, NoIbloc*. We can express the following invariant: if the card has previously sent a chained Iblock it can only receive a positive (*Next*) or negative (*Reply*) Rblock.

(*cd_PndIbloc* = *Chained*) ⇒ (*cd_bloc* = *Next*) ∨ (*cd_bloc* = *Reply*)

This simple approach is not sufficient because there are no means to prove the correct update of the historical variables in the different operations. The following operation *cd_rec1_Ibloc_SndIbloc* (third refinement step) sends the Iblock given by the application (operation icd.get_msg_from_appli) and updates the *Cd_PndIbloc* variable.

cd_rec1_Ibloc_SndIBloc =
SELECT
((*EndProtocol* = **FALSE**) ∧ (*cd_amoi*=**TRUE**) ∧
(*cd_bloc* = *UnChained*) ∧ (*cd_PndSbloc* = *NoSbloc*)∧
(*cd_PndIbloc* = *NoIbloc*))
THEN
cd_PndIbloc ← **icd.get_msg_from_appli**;
cd_ssendIbloc(*cd_PndIbloc*);
cd_amoi:= **FALSE**;
cd_I1amoi:= **FALSE**
END;

It is necessary to check **manually** that when a given block is sent, its historical variables are updated. In fact, there is here a possibility to make errors that can not be detected. But we believe that this manual check introduces less errors than translating the specification in another formalism and using a model checker.

3.2. Specification of the protocol

We used the same approach as [2]: at each refinement step we introduce new events and new invariants. At the abstract level, the reader initiates the protocol by sending a message. Each message can be sliced into several information blocks. The card answers by sending a message. At the abstract level we only express typing invariants.

Hereafter, we present a part of the abstract level related to the receipt of an unchained Ibloc on the smart card side. At the initialisation step we express that the first receipt of a message is done by the smart card. Several operations are not defined at this level e.g. *cd_analyse*, but must be present (empty operations). The *rcv* variable indicates which side has the control and *type* variable gives the type of the message received. When the smart card receives an unchained Ibloc it can send a *Sblock*, a *ChainedIBloc* or an *UnchainedIbloc* and gives back the control to the reader.

MACHINE
tt
SETS
liste_acteur ={*rd,cd*}
SEES

```
   context
ABSTRACT_VARIABLES
   type, rcv
INVARIANT
   type ∈ TYPE ∧  rcv ∈ liste_acteur
INITIALISATION
  type:∈ TYPE ||
  rcv:= cd

OPERATIONS
  cd_rec1_Ibloc = SELECT
        ((rcv = cd) ∧ (type=IblocUnChained))
       THEN
          skip
  END;
  cd_rec1_Ibloc_SndSBloc = BEGIN
       rcv := rd ||
       type:= Sbloc
  END;
  cd_rec1_Ibloc_SndIBloc = BEGIN
        rcv := rd ||
       type:∈ {IblocChained}∪{IblocUnChained}
  END;
  cd_analyse = skip ;
END
```

At the first refinement step, we introduce a new entity: the communication medium. The invariants state that the control of the protocol is under the control of one side or none. Hereafter is the first refinement limited to the smart card side. The typing invariants are not included here. The gluing invariants express firstly the relations between *type* and the input buffer *canal_bloc*, and secondly the relations between *rcv* and *cd_amoi*, *pr_rd_to_cd*. The *cd_amoi* variable is the first phasing variable. Such variables will not be implemented and will remain abstract variables. The *pr_rd_to_cd* variable represents the arrival of a message in the input buffer. The last invariant expresses the end of the exchange.

The behaviour of the protocol is refined in the initialisation phase. Now, the *cd_analyse* operation is partially defined. It aims at verifying the contents of the buffer and managing the error treatment.

```
REFINEMENT
  tt1
REFINES
  tt
SEES
  context,context2
INCLUDES
  icd.Appli(1), ird.Appli(2), Canal
ABSTRACT_VARIABLES
 cd_amoi, cd_bloc, cd_knowsReaderIsFinished,  EndProtocol
INVARIANT
  ((type=IblocChained)  ⇒ (canal_bloc = Chained)) ∧
```

```
((type=IblocUnChained) ⇒ (canal_bloc = UnChained)) ∧
(type=Rbloc ⇒ (canal_bloc ∈ {Reply}∪{Next})) ∧
((pr_cd_to_rd = TRUE) ⇒ (pr_rd_to_cd = FALSE)) ∧
((pr_rd_to_cd = TRUE) ⇒ (pr_cd_to_rd = FALSE)) ∧
((cd_amoi = TRUE ∧ type=IblocChained) ⇒ (cd_bloc = Chained)) ∧
((cd_amoi = TRUE ∧ type=IblocUnChained) ⇒ (cd_bloc = UnChained)) ∧
((cd_amoi = TRUE ∧ type=Rbloc) ⇒
        ((cd_bloc∈READY) ∧ (cd_bloc ∈ {Reply}∪{Next}))) ∧
(cd_amoi = TRUE ⇒ rd_amoi = FALSE ) ∧
(rd_amoi = TRUE ⇒ cd_amoi = FALSE ) ∧
((pr_cd_to_rd = TRUE) ⇒ ((cd_amoi = FALSE) ∧ rd_amoi = FALSE)) ∧
(cd_amoi = TRUE ⇒ ((pr_rd_to_cd = FALSE)∧ (pr_cd_to_rd = FALSE))) ∧
((rcv = cd) ⇔ ((cd_amoi = TRUE) ∨ (pr_rd_to_cd = TRUE))) ∧
(((cd_amoi=TRUE) ∧ (cd_bloc=UnChained)
 ⇒ (cd_knowsReaderIsFinished=TRUE)) ∧
 ((cd_bloc=Chained) ⇒ (cd_knowsReaderIsFinished=FALSE)) ∧
 ((EndProtocol=TRUE) ⇒ ((cd_knowsReaderIsFinished = TRUE)
        ∧ (rd_knowsCardIsFinished=TRUE) ∧ (rd_amoi=TRUE)))

INITIALISATION
 cd_amoi:= FALSE;
 EndProtocol:= FALSE ;
   cd_knowsReaderIsFinished:= FALSE ;
 VAR blk IN
     blk ← ird.get_sbloc_from_appli;
     IF ((blk ∈ SUPERVISOR) ∧ (blk≠NoSbloc)) THEN
       rd_ssendSbloc(blk)
     ELSE
       blk ← ird.get_msg_from_appli;
       rd_ssendIbloc(blk)
     END
  END

OPERATIONS

 cd_analyse = SELECT
     EndProtocol = FALSE ∧ pr_rd_to_cd = TRUE THEN
  cd_bloc:= canal_bloc;
 cd_amoi:= TRUE;
  raz_rd_to_cd;
  ANY l_cd_knowsReaderIsFinished WHERE
     l_cd_knowsReaderIsFinished ∈ BOOL ∧
     (((cd_amoi=TRUE) ∧ (cd_bloc=UnChained))
      ⇒ l_cd_knowsReaderIsFinished = TRUE) ∧
     ((cd_bloc=Chained) ⇒ (l_cd_knowsReaderIsFinished=FALSE))
  THEN
     cd_knowsReaderIsFinished:= l_cd_knowsReaderIsFinished ||
     EndProtocol:= bool((rd_knowsCardIsFinished = TRUE)
        ∧ (l_cd_knowsReaderIsFinished=TRUE) ∧ (cd_amoi=TRUE))
  END
 END;
```

```
cd_rec1_Ibloc = SELECT
    ((EndProtocol = FALSE) ∧ (cd_amoi = TRUE) ∧ (cd_bloc = UnChained))
 THEN
    cd_SAppli:∈ SUPERVISOR
 END;

 cd_rec1_Ibloc_SndSBloc = SELECT
    ((EndProtocol = FALSE) ∧ (cd_bloc = UnChained) ∧
    (cd_amoi = TRUE) ∧ (cd_SAppli≠NoSbloc))
 THEN
    cd_ssendSbloc(cd_SAppli) || cd_amoi:= FALSE
 END;

 cd_rec1_Ibloc_SndIBloc = SELECT
    ((EndProtocol = FALSE) ∧ (cd_bloc = UnChained) ∧
    (cd_amoi = TRUE) ∧ (cd_SAppli=NoSbloc))
 THEN
    ANY sup WHERE
      sup ∈ INFORMATION ∧ sup ∈ {Chained}∪{UnChained}
    THEN
      cd_ssendIbloc(sup)
    END ||
    cd_amoi:= FALSE
 END;
```

The second step introduces the treatment of the supervisory block that can arise between two exchanges of the protocol. In this refinement we introduce most of protocol rules that are not linked with error treatments.

The third refinement introduces new events:

- the tick of the clock. This clock is used on the reader side to detect an unresponsive card.
- occurrence of errors. The opposite side can reply with a negative acknowledgement. The error treatment consists in re-emitting the last message until reaching a limit of consecutive errors.

The last refinement is linked with the management of the input/output buffer. The related supervisory blocks are only introduced here.

3.3. Hypothesis

This model needs some hypothesis on the environment. The first one states that the error detection mechanisms are perfect. An erroneous message is always detected and no erroneous message (but well constructed) can be emitted. The protocol prologue (the three first bytes) is always either correct or detected as an erroneous message.

The second hypothesis concerns the application response time. The application layer must always react correctly within a given response time or if not, it must require process time by sending an Sblock request. If this hypothesis is wrong there is a possibility that both sides emit on the same line concurrently.

3.4. General properties

As stated in [8], there are different kinds of properties that can be verified. Some of them, known as general properties are not dependent on the definition of the protocol like:

- deadlock, both sides are waiting each other,
- unspecified message arrival, a message occurs in a state where no treatment is specified,
- livelock, both sides are exchanging useless messages and they can not escape from this infinite loop,
- unreachable state or transition, some actions can not be activated.

Those general properties are interesting but not sufficient. In paragraph 3.4 we express service properties while we formalise here the unspecified message arrival and the non-determinism. Those properties must only be checked at the last refinement when the guards are the most restrictive. At the other levels, it is unnecessary to check such invariants because the guards are not complete (necessary but not sufficient).

The condition of "Unspecified Message Arrival" requires that at least one guarded operation can be elected when an event occurs; this must be ensured for any event. This constraint can easily be modelled by adding a special lemma in an ASSERTION clause. Considering that each of n operations of the B model is guarded by corresponding predicate G_n, the constraint is expressed as:

bool $(G_1 \vee G_2 \vee .. \vee G_n)$ = **TRUE**

As an example, rule 3 states that the smart card can request a certain amount of time (INF) to process a command by sending SRequestWtx (INF). This request must be acknowledged by SResponseWtx with an identical parameter. A temporary deadlock occurred because no operation took into account the case when the returning parameter was different. Effectively, if this parameter is different the standard does not state anything. We modified the model and considered such a case as an error, and this is treated in the next refinement.

The condition of "determinism" requires that one and only one operation can be elected at a time when an event occurs. This requirement must be fulfilled to avoid any deterministic software which would react in a non predictable manner. This constraint can be modelled in the ASSERTION clause saying that if the operation Op_1 is elected (corresponding guard G_1 is valid), then none of the other guarded operations can be elected.

Considering that each of n operations of the B model is guarded by the corresponding predicate G_n, the constraint is expressed as:

G_1 => **not** $(G_2 \vee .. \vee G_n)$

Similarly, the same reasoning must be applied to the other operations. We obtain N predicates of the following expression:

$Gj \Rightarrow$ **not** $(G_1 \vee G_2 \vee .. \vee G_{j-1} \vee G_{j+1} \vee G_n)$

To help the theorem prover to solve it, we transform this expression into:

bool(**not** $(G_j) \vee$ **not** $(G_1 \vee G_2 \vee .. \vee G_{j-1} \vee G_{j+1} \vee G_n)$) = **TRUE**

Which is equivalent to:

$$\mathbf{bool}(G_j \wedge (G_1 \vee G_2 \vee .. \vee G_{j-1} \vee G_{j+1} \vee G_n)) = \mathbf{FALSE}$$

As a consequence:

$$(\mathbf{bool}(G_j \wedge G_1) = \mathbf{FALSE}) \wedge (\mathbf{bool}(G_j \wedge G_2) = \mathbf{FALSE}) \wedge (\mathbf{bool}(G_j \wedge G_n) = \mathbf{FALSE})$$

Thus the condition of "determinism" is ensured when each of the following predicates can be stated:

$$\mathbf{bool}(G_i \wedge G_j) = \mathbf{FALSE}$$

3.5. Service properties

The use of event based software architecture refines progressively abstract operations by several simple operations. As explained in [2], the guards expressed in each SELECT clause must be reinforced in the refinement machine.

At the abstract level and at the first level, the invariants are linked with the end of the protocol. At the second level we introduced the protocol rules. Rule 1 expressed in the ISO standard does not need to be proved. Instead of this, it is inherent in the initialisation process.

At the first refinement level we introduce invariants representing the state machine. Rule 2.2 expresses that a chained Iblock is acknowledged by a Rblock and a chained Iblock can not be received if a message is chaining on the same side. We split this rule into two invariants.

$$(((cd\_PndIbloc = Chained) \wedge (cd\_error = \mathbf{FALSE}) \wedge (cd\_amoi = \mathbf{TRUE}))$$
$$\Rightarrow (cd\_bloc = Next))$$

The second invariant states that if the card receives an acknowledgement of the reader (*cd_bloc = Next*) it can send another block or the last one. The *cd_amoi* variable represents the input/output buffer.

$$(((cd\_bloc = Next) \wedge (cd\_error = \mathbf{FALSE}) \wedge (cd\_amoi = \mathbf{FALSE}))$$
$$\Rightarrow (cd\_PndIbloc = Unchained) \vee (cd\_PndIbloc = Chained))$$

This invariant must be refined at the second level introducing the Sblock according to rule 9. The invariants are a transcription of the rules and they must express the rule exactly. One of the most ambiguous rules of the standard is rule 9: *The abortion of a chain can be initiated by either the sender or receiver of a chain sending a Sblock Abort Request. The Sblock Abort Request shall be answered by a Sblock Abort Response after which a Rblock may be sent depending on whether it is necessary to give back the right to send*. The scenario of figure 3 is given as an example of possible behaviour.

The final invariants of rule 2.2 and 9 must state that it is possible to receive a given type of block only if the system was in a particular state and only a subset of possible messages can be sent from this state.

$$(((cd\_PndIbloc = Chained) \wedge (cd\_knows\_rd\_is\_chaining = NoIbloc) \wedge (cd\_amoi = \mathbf{TRUE}))$$
$$\Rightarrow$$
$$(((cd\_bloc = Next) \wedge (cd\_PndSbloc = NoSbloc)) \vee$$

((*cd_bloc* ∈ *SREQ*) ∧ (*cd_PndSbloc* = *NoSbloc*)) ∨
((*cd_bloc* ∈ *SRESP*) ∧ (*cd_bloc* ∈ *Sreply* (*cd_PndSbloc*))) ∨
((*cd_bloc* ∈ {*Chained*}∪{*UnChained*}) ∧ (*cd_PndSbloc* = *RespAbort*))))

(((*cd_bloc* = *Chained*) ∧ (*cd_error* = **FALSE**) ∧ (*cd_amoi*=**FALSE**))
=>
((*cd_ PndIbloc* = *NoIbloc*) ∧ (*cd_PndSbloc* = *NoSbloc*)) ∨
((*cd_PndSbloc* ∈ *SREQ*) ∧ (*cd_PndSbloc* ≠ *NoSbloc*) ∧ (*cd_ PndSbloc* ≠ *ReqWtx*))))

And those invariants can lead to different implementations of the protocol. One of them is given by figure 3: it is possible for the card to systematically give back the right to send to the reader or to give back the control after sending its own message.

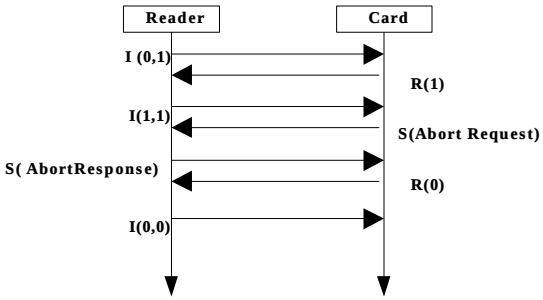

Fig.2 Informative Scenario 27

Due to the fact that the invariants represent the state machine, the only unnecessary variables are the phasing variables and the historical variables. But such variables are not implemented, they remain abstract variables. The choice of splitting the specifications into several small operations (68) leads to a high ratio of automatically proved PO (83%).

	Obvious	number of PO	number of Unproved	%Pr
Total	5065	198	83	17

4. Conclusions

We have shown that the B Method is suitable to formalise protocols and to express different kinds of properties. Some general properties have been proved at the model last refinement while service properties have to be proved during the refinement process. The B Method does not increase the complexity of the model and all the service properties have been proved.

The use of a formal specification pointed out some ambiguities of the protocol. The statement of rule 2.1 is not accurate if the card or the reader answers with a chained Iblock. We had to define the protocol behaviour when the answer of SRequestWtx is different from the answer expected. Some other points have been noticed, and for the ISO standard next release, this formal specification will help to

clarify the protocol.

We have specified this protocol with another formal method: SDL. If we compare both methods, it appears that we spent less time using SDL to specify the model and the properties and naturally to verify them. Moreover, the C code generator of the Atelier B is less efficient than the one of the Verilog tool. But we did not point out the problem of the SRequestWtx. And if we add too much protocol details (an implementable version) the size of the behavioural tree is too large to check properties.

In our approach there are two points where the proof can be bypassed. The first one concerns the update of the historical variables and the second the transformation of the SELECT statement of each operation by PRE statement [3] in order to generate the code. This checking must be performed manually. But we believe that the use of only one formalism to prove the correctness of the embedded code is more advisable.

References

1. ITSEC, *Information Technology Security Evaluation Criteria*. Version 1.2, ed. 1993.

2. Abrial J-R. and Mussat L., *Specification and Design of a Transmission Protocol by Successive Refinements Using B.* in Mathematical Methods in Program Development, Edited by M. Broy and B. Schieder, Springer Verlag (1997)

3. Lanet J.-L. and Lartigue P., *The Use of Formal Methods for Smart Cards, a Comparison between B and SDL to Model the T=1 Protocol*, Comparing Systems Specification Techniques, pp 3-16, Nantes, 26-27 March, 1998

4 Julliand J., Legeard B., Machicoane T., Parreaux B., Tatibouët B., *Specification of an Integrated Circuit Card Protocol Application using the B method and linear Temporal Logic*, 2nd B Conference, Montpellier, 1998.

5. ISO 7816-3. *ISO/IEC 7816-3*. Draft, September 1995.

6. Abrial J-R and Mussat L., *Introducing Dynamic Constraints in B*, to appear in 2nd B Conference, Montpellier, 1998.

7. Butler M., Walden M., *Distributed System Development in B*, TUCS Technical Report n°53, Oct-96.

8.Groz R., *Vérification de propriétés logiques des protocoles et systèmes répartis par observation de simulations*. PhD Thesis, University of Rennes 1, 1989.