The Use of Formal Methods for Smart Cards, a Comparison between B and SDL to Model the T= 1 Protocol

Jean-Louis Lanet, Pierre Lartigue
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; pierre.lartigue}@ccmail.edt.fr

Abstract: In order to obtain high confidence in the software embedded into a smart card, we evaluated different techniques like model checking and theorem proving. Nevertheless due to the low cost of smart cards and mechanical constraints, the amount of memory available on chips is small. The code generated by the tools must be compact enough to fit the constraints. In this paper we compare different code generators with a case study of a protocol dedicated to smart cards. We show that under some conditions, the model checking tools are able to generate code with an acceptable overhead for smart cards. Our work on the B method is in progress. The invariants are more difficult to express and to prove but we pointed out some ambiguities and errors contained in the standard.

1. Introduction

The use of formal methods in the software development for smart cards is a way to improve 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. If properties on specifications can be proved, the best way to gain confidence into the embedded software is to generate the source code automatically. But the constraints on the size of the memory are very tough. And before investing in a new technology, we have to evaluate the ability of the commercial tools to generate an application that fits our constraints.

The chosen 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 [2]. 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 a set of rules and several scenarios to explain some cases. But in fact some 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. For this application

we have several benchmarks of implementations for different chips regarding the size of the RAM and EEPROM memory.

Other results were expected like the easiness to express and to obtain a proof (if obtainable) and the possibility to generate the test suites associated to a model. We chose formal methods which are recommended by the ITSEC for the levels considered. The choice of SDL and B was due to the fact that they are supported by industrial tools, Atelier B from Stéria for the B method and ObjectGéode from Vérilog or SDT from Telelogic for SDL. The model checking technique is particularly well adapted to prove dynamic properties and it seems interesting to verify if it is possible to handle dynamic properties with B without adding dedicated variables or if mixing both techniques is conceivable.

In the rest of this paper we present the protocol in section 2 and then we describe the SDL model used to check the dynamic properties in section 3. The formal model of the B specification is introduced in section 4. Our conclusions are presented in section 5.

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) 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 a 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 emit again the last Iblock. The supervisory blocks do not need a sequence number. It is always a pair of i-request/i-response. When an emitter sends a message whose size is greater than the size of the input buffer, the sender slices the message into several Iblocks. With the bit *more* of the prologue, it indicates that the Iblock will be followed by at

least another one. Receiving a Iblock with M=0, means that the receiver is requested to transmit.

The semantic of a Iblock is defined by those two bits : I[N(s),M]. The following scheme shows two exchanges in an error free protocol transmission. When the protocol layer of the card receives a Iblock, it sends it to the application layer without waiting for the whole message. The application layer processes the command. Sometimes the card needs more time than allowed by the reader timer. It requests the reader for more execution time by sending a Sblock [Wtx-Request (TimeRequested)] which must be acknowledged by a Sblock [Wtx-Response (TimeObtained)].

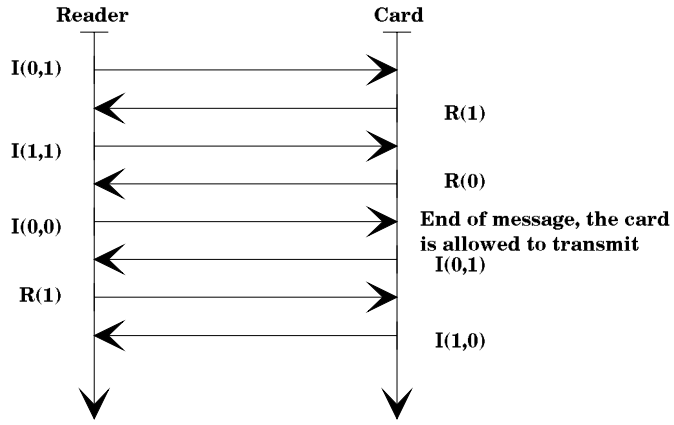

Fig. 1 Exchange of Ibloc

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 the Abort to cancel the current message, and the 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 watch dog 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.

3. The SDL Model

Our model includes a communication link, a reader and a card. We model the protocol layer and the application layer. The link can modify the content of a block, by generating an error during the reception. In order to limit the size of the reachability graph of our model, both side applications attempt to emit only one message. It can or cannot be sliced into several blocks. The number of possible blocks is limited too.

In fact we used two models of the protocol. The first one, that has been verified was not designed to be implemented. It was a functional model. Both tools (Objectgéode and SDT) are well adapted to the simulation and verification. Both use the concept of observer. Some ambiguous points of the standard have been brought to the fore.

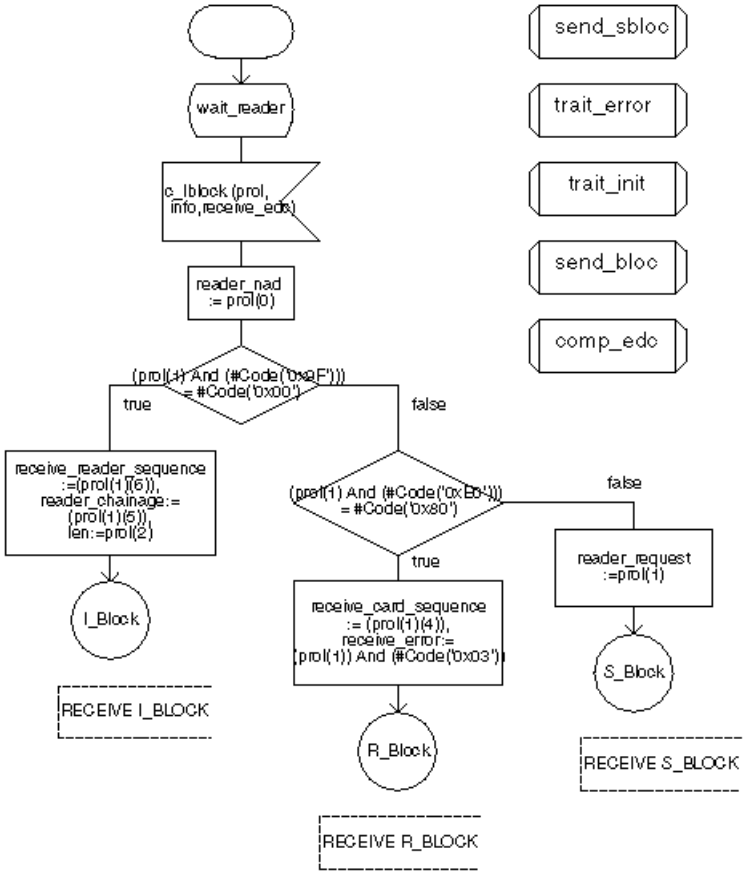

Fig. 2 Analysis of the Bloc Received

The second model was a *refinement* of the functional model and we only modelled the card side. We took into account the implementation constraints. This

part was more difficult, because the final code is very sensitive to the architecture of the model. Using a functional model as a starting point of the protocol final design made easier part code factorization. But on the other hand it was more difficult to simulate with the exact input flow.

Using this methodology, we have to prove rigorously again the properties on the second model, because they are no correlation between the two models. We did not do it because we focused only on the code generation problem.

In figure 2 we give a part of the SDL model. The first scheme shows the smart card side analysing the prologue field of the incoming message. According to the value of the PCB, the system can process three different treatments.

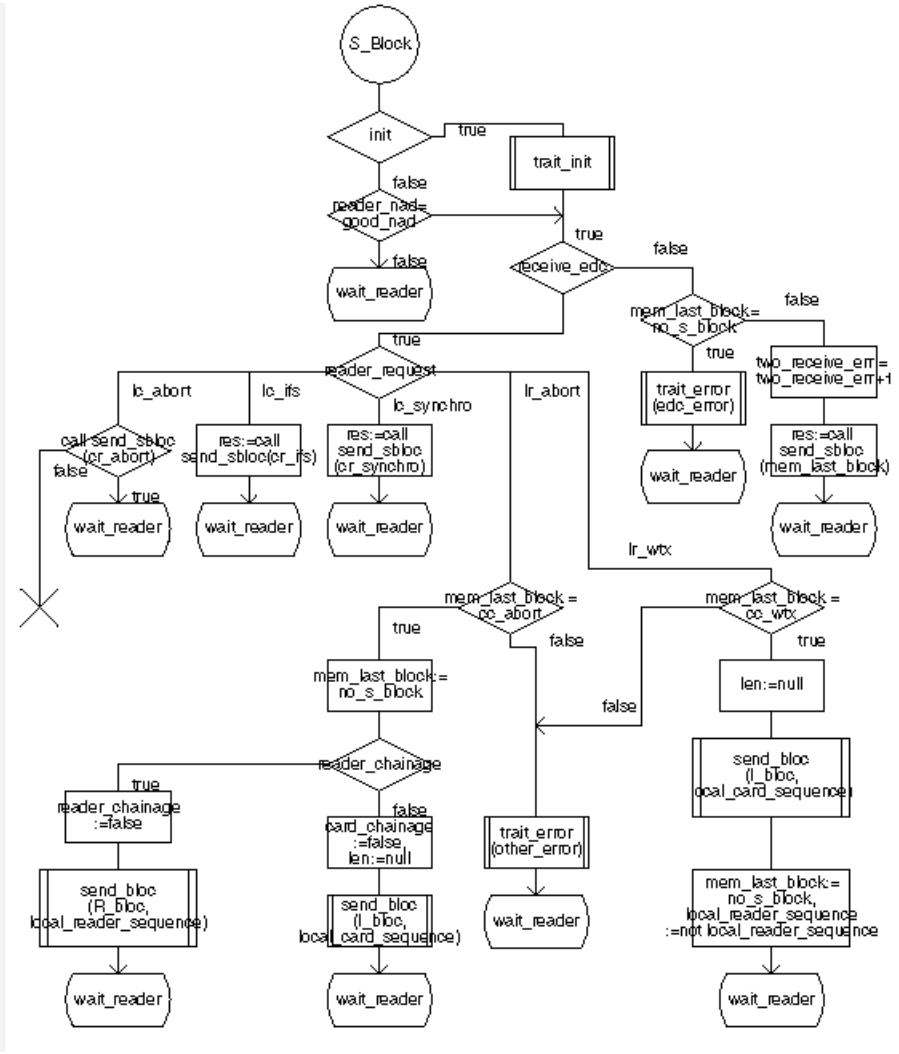

Fig. 3 Treatment of an Sbloc Event on the Smart Card Side

Figure 3 shows how to handle a Sblock. The procedure Send_Sbloc, answers to the reader according to the request and the number of errors already detected. The procedure Trait_error handles several cases. For example when a Sblock response arrives while no request has been previously sent.

3.1. The use of observer to prove the behavioural properties

The properties to prove are described in the specifications as rules of the protocol. Most of them are dynamic properties that can be easily expressed by an Observer [3]. Observers are based on automata and are therefore more convenient to use than temporal logic expression.

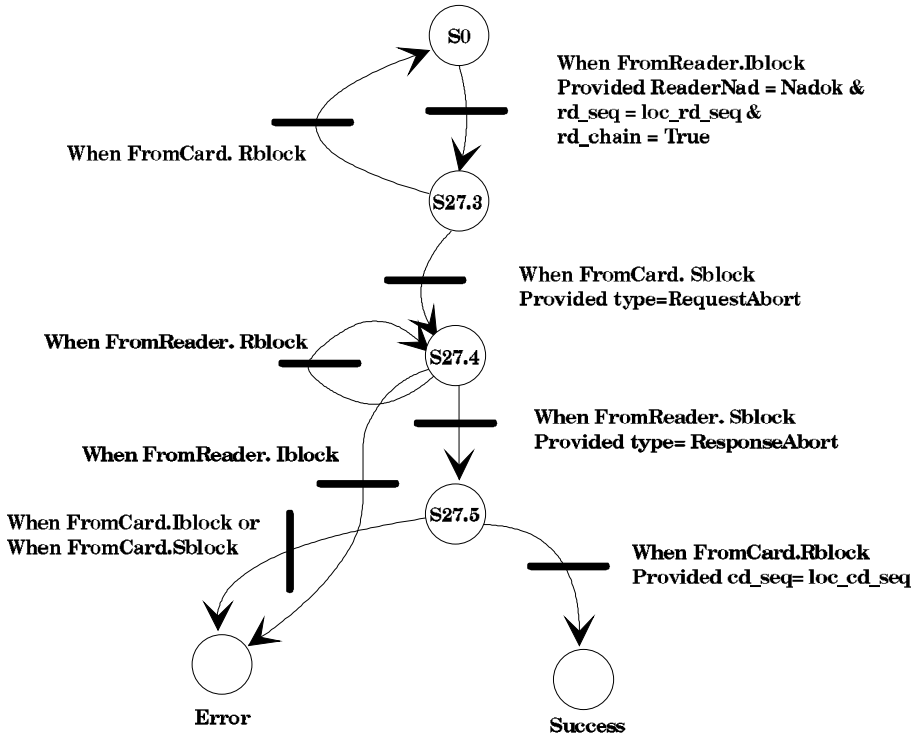

Fig. 4 Observer of the Property defined by Scenario 27.

The Vérilog tool provides the ability to specify an observer like a SDL process. Most of the scenarios of the ISO standard has been modelled by an observer. The observer is external to the specification so we used one observer per property to prove. Observers have access to each variable of the system.

3.2. Code generation

The first difficulty we had to face was the use of simple types like byte or boolean. Those two types are very often used for the development of smart card operating system but they are not part of the SDL standard. Both tools provide their own proprietary implementation. Those proprietary types do not allow the use of

standard operations in an optimised way. In a smart card, most operations are logical byte or bit operations. The Objectgéode viewpoint is to use an integer standard type and to transform it during code generation to map it onto a byte.

The second problem is related to the difficulties to link the new code and the already developed one. In a smart card most of the code, in particular low level routines, are in the ROM area. A good knowledge of the tool is necessary in order to obtain an optimised code. For the data, the buffer must be shared between the new code and the low level routines. With the Vérilog tool it is easy to share such variables.

Both tools are able to link the generated code with a real time operating system. But the smart card does not need such possibility, the scheduler is proprietary and well optimised. Only the Vérilog tool was able to generate the code without any kernel. The size of the code generated by SDT always includes the kernel.

Another difficulty arises when the compilation failed. It is difficult to get a precise diagnosis. The used compilers were a Texas Instrument one for the ARM7 and the KEIL compiler for 8051. The generated code is not always accepted by the compiler. In the SDL model we have minimised the number of static variables. But in order to obtain a better visibility all boolean variables are coded using bytes. For the variables of the model, we need 25 bytes and the size of the I/O buffer is limited to 20 bytes.

A Gemplus team has translated a 8051 optimised code for the ARM7 processor. The code of the Gemplus T=1 offers several possibilities (different sizes of the code), whereas the SDL generated code has minor simplifications and offers a standard service to the application layer. Moreover, the Gemplus code for the ARM7 has not been optimised.

	SDT		ObjectGéode		Gemplus	
	ARM7	*8051*	*ARM7*	*8051*	*ARM7*	*8051*
Rom	5300	10400	2780	2700	3200-2000	2400-1800
Ram	230	320	50	120	120-60	80-40
Stack	130		60			15

Tab. 1 Size of the Memory for the Different Implementations.

The overhead generated by the SDT tool is due to the impossibility to generate the code without including the real time kernel. The ObjectGéode tool generates an acceptable overhead in the case of the 8051, and almost no overhead for the ARM7. The automatically generated code has been optimised but it seems that more optimisations can be done. Nevertheless it needs a good knowledge about code generator tools.

4. The B model

We used the same approach as [4] considering in the abstract level that the transmission is atomic. At each refinement step we introduce new events and new invariants. At the abstract level, a message is sent by the reader in one shot. The card answers by sending a message. The property we want to prove is that both sides have received a message. At the first refinement step, we introduce the fact that the messages are sliced into several information blocks and that the control of the protocol is alternatively given from one side to the other. The new invariants concern this command response mode. The third step introduces new messages (supervisory bloc) that can arise between two exchanges of the protocol. The next refinement introduces a new event : the tick of the clock. This clock is used on the reader side to detect an unresponsive card. We add a new entity the communication medium at the sixth refinement. The protocol is under the control of one side or none. The last refinement considers the occurrence of errors.

The project is still in progress, we have not reached the last refinement. As a consequence our model does not take into account the time and the errors. We have refined our model up to the third step. In order to generate the smart card code we introduced the communication medium. At this step we separated the two parts of the protocol in order to proceed to the implementation.

4.1. Specification of the protocol

In order to obtain an optimised implementation of the protocol, it is necessary to avoid the use of the SETS statement. It is easier to map an abstract constant to the smallest entity. So we use a specific abstract machine for the context. We define several sets like BLOC that contain the three types of block INFORMATION, SUPERVISOR and READY. The last one contains only the Next Rbloc, the Reply Rbloc has to be introduced in the error level.

MACHINE
context
ABSTRACT_CONSTANTS
BLOC, INFORMATION, SUPERVISOR, READY, SRESP,SREQ,
Next, Chained, UnChained, ReqWtx, RespWtx, ReqIfs , RespIfs, ReqAbort, RespAbort ,
ReqResync, RespResync, Sreply, ErrProtocol

PROPERTIES
$BLOC \subseteq \mathbf{NAT} \wedge INFORMATION \subseteq \mathbf{NAT} \wedge SUPERVISOR \subseteq \mathbf{NAT} \wedge READY \subseteq \mathbf{NAT} \wedge$
$SREQ \subseteq \mathbf{NAT} \wedge SRESP \subseteq \mathbf{NAT} \wedge$
$Chained \in INFORMATION \wedge UnChained \in INFORMATION \wedge$
$ReqWtx \in SREQ \wedge RespWtx \in SRESP \wedge ReqIfs \in SREQ \wedge RespIfs \in SRESP \wedge$
$ReqAbort \in SREQ \wedge RespAbort \in SRESP \wedge ReqResync \in SREQ \wedge RespResync \in SRESP$
$\wedge Next \in READY \wedge (READY \cap INFORMATION \cap SUPERVISOR = \varnothing) \wedge$
$SUPERVISOR = SRESP \cup SREQ \wedge SReply \in SREQ \rightarrow\!\!\!\!\!\!\rightarrow SRESP \wedge$
$(\forall req.(req \in SREQ \Rightarrow (SReply(req) \in SUPERVISOR))) \wedge$
$(SReply = \{ReqWtx \mapsto RespWtx, ReqIfs \mapsto RespIfs , ReqAbort \mapsto RespAbort,$
$ReqResync \mapsto RespResync\}) \wedge (BLOC = SUPERVISOR \cup INFORMATION \cup READY)$
END

The set BLOC contains either an INFORMATION block or a SUPERVISOR block or a READY block. The SUPERVISOR set contains two sets of requests (SREQ) and responses (SREP). Those sets are linked with a bijection.

At the abstract level we only define a few operations that are refined until the second level. We split the specification into a large amount of operations in order to help the prover. The first operation analyses the contents of the incoming buffer and sets different variables according to the message header. This operation performs a task equivalent to the SDL process given by figure 2.

rd_analyse =
  SELECT
    (*EndProtocol* = **FALSE**) ∧ *pr_cd_to_rd* = **TRUE** ∧ (*rd_amoi* = **FALSE**)
  THEN
    rd_bloc := *canal_bloc*;
    rd_amoi := **TRUE**;
    raz_cd_to_rd;
    ANY *l_rd_knowsCardIsFinished* **WHERE**
      l_rd_knowsCardIsFinished ∈ **BOOL** ∧
      (((*rd_amoi*=**TRUE**) ∧ (*rd_bloc*= *UnChained*))
      ⇒ (*l_rd_knowsCardIsFinished* = **TRUE**)) ∧
      ((*rd_bloc*=*Chained*) ⇒ (*l_rd_knowsCardIsFinished*=**FALSE**)) ∧
      ((*rd_PndSbloc* ∈ *SREQ*) ∧ (*rd_amoi* = **TRUE**)
    THEN
      rd_knowsCardIsFinished := *l_rd_knowsCardIsFinished* ||
      EndProtocol := **bool**((*cd_knowsReaderIsFinished* = **TRUE**)
        ∧ (*l_rd_knowsCardIsFinished*=**TRUE**) ∧ (*rd_amoi*=**TRUE**))
    END
  END;

We show hereafter the different operations linked with the arrival of a Sblock. This part is equivalent to the SDL process given by figure 3, except that the error treatment is not taken into account.

rd_rec_SblocResp =
  SELECT
    ((*EndProtocol* = **FALSE**) ∧ (*rd_amoi* = **TRUE**) ∧ (*rd_S2amoi*=**FALSE**) ∧ (*rd_bloc* ∈
    SRESP) ∧ (*rd_PndSbloc*≠*NoSbloc*))
THEN
    rd_PndSbloc ← **ird.get_sbloc_from_appli**;
    rd_S2amoi := **TRUE**
  END;

rd_rec_SblocResp_SndSbloc =
  SELECT
    ((*EndProtocol* = **FALSE**) ∧ (*rd_amoi* = **TRUE**) ∧ (*rd_S2amoi*=**TRUE**) ∧ (*rd_bloc* ∈
    SRESP) ∧ (*rd_PndSbloc*≠*NoSbloc*))
  THEN
    rd_ssendSbloc(*rd_PndSbloc*);
    rd_amoi := **FALSE**;
    rd_S2amoi := **FALSE**
  END;

rd_rec_SblocResp_SndIbloc =
  SELECT
    ((*EndProtocol* = **FALSE**) ∧ (*rd_amoi* = **TRUE**) ∧ (*rd_S2amoi*=**TRUE**) ∧ (*rd_bloc* ∈ *SRESP*) ∧ (*rd_PndSbloc*=*NoSbloc*) ∧ (*rd_knows_cd_is_chaining*≠*Chained*))
  THEN
    rd_SndBuf , *rd_PndIbloc* ← **ird.get_msg_from_appli**;
    rd_ssendIbloc(*rd_PndIbloc*,((0..*rd_SndSize*)◁*rd_SndBuf*));
    rd_amoi := **FALSE**;
    rd_S2amoi := **FALSE**
  END;

rd_rec_SblocResp_SndRbloc =
  SELECT
    ((*EndProtocol* = **FALSE**) ∧ (*rd_amoi* = **TRUE**) ∧ (*rd_S2amoi*=**TRUE**) ∧ (*rd_bloc* ∈ *SRESP*) ∧ (*rd_PndSbloc*=*NoSbloc*)∧ (*rd_knows_cd_is_chaining*=*Chained*))
  THEN
    rd_amoi := **FALSE**;
    rd_ssendRbloc;
    rd_S2amoi := **FALSE**
  END;

rd_SblocReq = **SELECT**
    ((*EndProtocol* = **FALSE**) ∧ (*rd_amoi* = **TRUE**) ∧ (*rd_bloc* ∈ *SREQ*) ∧ (*rd_bloc*≠ *NoSbloc*))
 THEN
    rd_ssendSbloc(*SReply*(*rd_bloc*));
    IF *rd_bloc* = *ReqAbort* **THEN**
      ird.cancel_Ibloc_to_appli;
    END;
    rd_PndSbloc := *SReply*(*rd_bloc*);
    rd_amoi:=**FALSE**
END;

4.2. Proof of the invariants

The use of event based software architecture refines progressively abstract operations by several simple operations. As explained in [5], the guards expressed in each SELECT clause must be reinforced in the refinement machine.

At this refinement level (errors are not introduced) it is possible to translate the rules without adding any unnecessary variables. The rule 1 expressed in the ISO standard does not need to be proved. Instead of this, it is inherent in the initialisation process.

Rule 2.1 states that an unchained Iblock sent by A is acknowledged by B with another Iblock chained or not. Contrary to what the standard states, such exchange does not indicate the readiness to receive the next Iblock from A in the case of a chained Ibloc. In the following invariant we state that if the card has sent an unchained Ibloc, the next received block cannot be a Next Rblock.

(((*cd_amoi*=**TRUE**) ∧ (*cd_PndIbloc* = *UnChained*))
⇒ (*cd_bloc* ≠ *Next*))

The rule 2.2 states that a chained Iblock is acknowledged by a Rblock. The following invariant states that if the card has sent a chained Iblock, the next block to be received can be :

- a Next Rbloc if no supervisory request has been sent,
- a supervisory request if no request is pending,
- a supervisory response if the response corresponds to the request sent,
- a chained or unchained Iblock if the previous sent response was an abort response (scenario 28 of the standard).

$$
\begin{aligned}
&(((cd\_PndIbloc = Chained) \wedge (cd\_knows\_rd\_is\_chaining = NoIbloc) \wedge \\
&(cd\_amoi=\textbf{TRUE})) \\
&\Rightarrow \\
&(((cd\_bloc = Next) \wedge (cd\_PndSbloc = NoSbloc)) \cup \\
&((cd\_bloc \in SREQ) \wedge (cd\_PndSbloc = NoSbloc)) \cup \\
&((cd\_bloc \in SRESP) \wedge (cd\_bloc \in SReply(cd\_PndSbloc))) \cup \\
&((cd\_bloc \in \{Chained\} \cup \{UnChained\}) \wedge (cd\_PndSbloc = RespAbort))))
\end{aligned}
$$

This rule is not completely symmetric because of the abort. It is completed by rule 9. This rule details the different behaviours of both sides when an abort is requested depending which side intitiated the request and if this side was chaining or not. The following invariants complete rule 2.1 according to rule 9.

$$
\begin{aligned}
&(((rd\_PndIbloc = Chained) \wedge (rd\_knows\_cd\_is\_chaining = UnChained) \wedge \\
&(rd\_amoi=\textbf{TRUE})) \\
&\Rightarrow \\
&(((rd\_bloc = Next) \wedge (rd\_PndSbloc = NoSbloc)) \cup \\
&((rd\_bloc \in SREQ) \wedge (rd\_PndSbloc = NoSbloc)) \cup \\
&((rd\_bloc = SReply(rd\_PndSbloc)))))
\end{aligned}
$$

$$
\begin{aligned}
&(((rd\_knows\_cd\_is\_chaining = Chained) \wedge (rd\_amoi=\textbf{TRUE}) \wedge \\
&(rd\_PndIbloc = NoIbloc)) \\
&\Rightarrow \\
&(((rd\_bloc \in \{Chained\} \cup \{UnChained\}) \wedge (rd\_PndSbloc = NoSbloc)) \cup \\
&((rd\_bloc \in SREQ) \wedge (rd\_PndSbloc = NoSbloc)) \cup \\
&((rd\_bloc \in SRESP) \wedge (rd\_bloc \in SReply(rd\_PndSbloc)))))
\end{aligned}
$$

$$
\begin{aligned}
&(((cd\_knows\_rd\_is\_chaining = Chained) \wedge (cd\_amoi=\textbf{TRUE}) \wedge \\
&(cd\_PndIbloc = NoIbloc)) \\
&\Rightarrow \\
&(((cd\_bloc \in \{Chained\} \cup \{UnChained\}) \wedge (cd\_PndSbloc = NoSbloc)) \cup \\
&((cd\_bloc \in SREQ) \wedge (cd\_PndSbloc = NoSbloc)) \cup \\
&((cd\_bloc \in SRESP) \wedge (cd\_bloc \in SReply(cd\_PndSbloc)))))
\end{aligned}
$$

To express these properties we do not need more variables than the SDL process. Both information *cd_knows_rd_is_chaining* and *cd_PndIbloc* are part of the implementation of the protocol.

4.3. Deadlock freeness and determinism

The condition of "deadlock freeness" 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 predicat G_n, the constraint is expressed as:

a) $bool(G_1) \cup bool(G_2) \cup .. \cup bool(G_n) = TRUE$

As an example of deadlock, 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 deadlock occured 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 consider such a case as an error, and this will be treated at 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 predicat G_n, the constraint is expressed as:

$G_1 => not(G_2 \cup .. \cup G_n)$

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

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

The above expression is equivalent to:

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

Which is equivalent to:

$G_j \cap (G_1 \cup G_2 \cup .. \cup G_{j-1} \cup G_{j+1} \cup G_n) = FALSE$

As a consequence:
$G_j \cap G_1 =$ FALSE &
$G_j \cap G_2 =$ FALSE & ...
$G_j \cap G_n =$ FALSE

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

b) $G_i \cap G_j =$ FALSE

From a practical point of view, we must be aware that proof obligations resulting, from the expressions a) or b), would probably be difficult to demonstrate

because of their complexity due to their relatively important size. It is necessary to check those properties at each level of refinements.

4.4. Code generation

The process of code generation needs to transform an event based software architecture (provided by the last refinement level of our B model) into a sequential program which can be run by a smart card processor. At first, this needs to transform the SELECT statements of each operation by PRE statements, and secondly to "simulate" the behaviour of a "simple" scheduler whose task would be to call, the newly transformed pre-conditioned operations, in a pre-defined and sequential order. The following diagram depicts the process to be used for code generation:

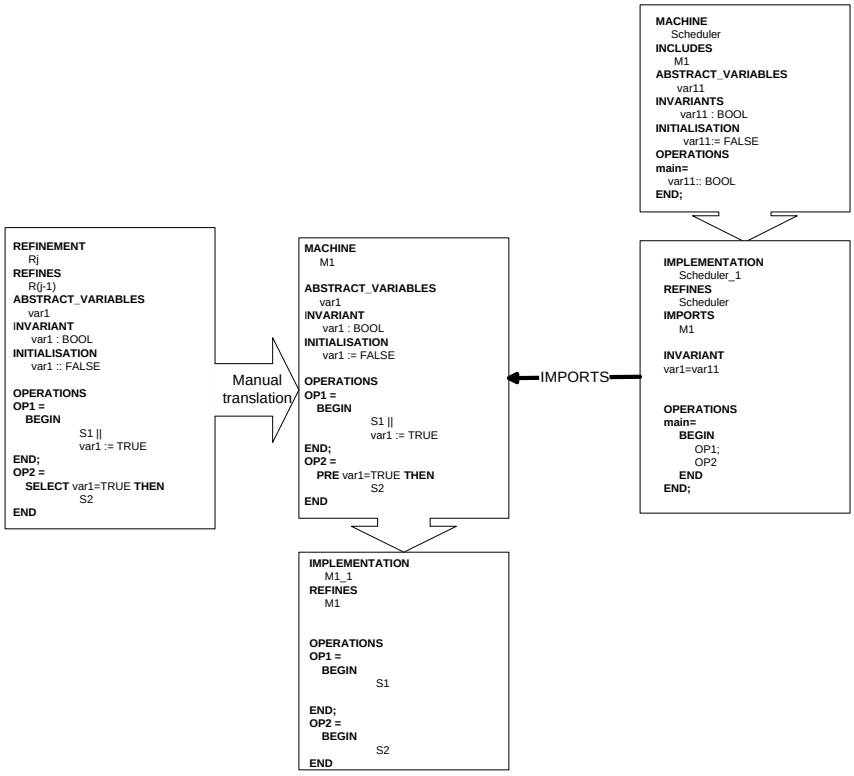

Fig. 5 Transformation of Guards.

Indeed, the actual version of the "B0 checker" and "C" translator AtelierB tools, does not support the SELECT clauses in the implementation.The transformation of the SELECT clauses into PRE statements must be done in a specific B abstract machine. This must be performed manually or by a script command; the traceability between the original model and the transformed one can be easily demonstrated. Although we must bear in mind that no proof obligation can be generated along this process.

5. Conclusions

The main goal of this study was to check the possibility to generate automatically codes that fit the constraints of the smart card by CASE tools supporting formal methods. This work is still in progress concerning the B method. Only a part of the protocol has been modelled. Nevertheless we show that for the model checking tools and in particular ObjectGéode it is possible to generate code for small embedded systems without an excessive overhead. Our B model must be more refined in order to include the errors processing to complete the comparison.

The use of a formal specification pointed out some ambiguities of the protocol. The statement of rule 2.1 is not accurate in the case when the card or the reader answers with a chained Iblock. We had to define the behaviour of the protocol when the answer of SRequestWtx is different from the answer expected. Some other points have been noticed, and for the next release of the ISO standard this formal specification will help to clarify the protocol. Sometimes, customers ask for a specific implementation of the protocol. In every case we have to check if such a requirement is possible. Having a model of the protocol allows us to make quickly a prototype and check the possibility of implementing such requirements.

Finally we can point out that the time overhead needed for the specification of the protocol is compensated with the time saved during the test phase and the documentation. Generating automatically the test cases using a model of the system is probably a key point to reduce development time. This will be the next step of this study. But if some works have already been done for the SDL tools, the atelier B tool does not include such a functionality. We have to look for some academic works adapted to the B method.

Acknowledgements

Thanks to Steria Méditerrannée and in particular Denis Sabatier and Clément Roques for their contribution to the design of the B model.

References

1. ITSEC, *Information Technology Security Evaluation Criteria*. Version 1.2, ed. 1993.
2. ISO 7816-3. *ISO/IEC 7816-3*. Draft, September 1995.
3. Groz, *Vérification de propriétés logiques des protocoles et systèmes répartis par observation de simulations*. Thèse de l'université de Rennes (1989)
4. 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)
5. Abrial J-R and Mussat L., *Introducing Dynamic Constraints in B*, to appear in 2nd B Conference, Montpellier, 1998.