The Use of the B Formal Method for the Design and the Validation of the Transaction Mechanism for Smart Card Applications

Denis Sabatier[1] and Pierre Lartigue[2]

[1] Steria Méditerranée - Equipe AtelierB 530, rue F.Joliot 13791 Aix-en-Provence
denis.sabatier@steria.fr
[2] Gemplus Research Lab - Av Pic de Bertagne 13881 Gemenos
pierre.lartigue@gemplus.com

Abstract. This document describes an industrial application of the B method in smart card applications. In smart card memory, data modification may be interrupted due to a card withdrawal or a power loss, the EEPROM memory may result in an unstable state and the values subsequently read, may be erroneous. The transaction mechanism provides a secure means for modifying data located in the EEPROM. As the security in smart card application is paramount, the use of the B formal method brings high confidence and provides mathematical proofs that the design of the transaction mechanism fulfills the security requirements

1. Introduction

The EEPROM memory is used for permanently storing data in smart cards. In normal operation mode, when the card is pulled out of the terminal and the power is turned off, the information stored in the EEPROM is preserved.

Due to its electronic characterization and to its physical constraints, a modification of data is not performed in a single and atomic operation. Instead, this process may take up to a few milliseconds. This lapse of time is required for electronically charging the memory cells and reaching a steady state that enables permanent retention of the information. If the power is turned off, or the card is unexpectedly pulled out while a memory cell is being written, the electronic charge may not be sufficient for retaining the information in a durable way. Values obtained from subsequent accesses to the memory may be non-deterministic.

The transaction is a software mechanism that prevents errors from these misuses; it enables modification of data in a secure way: the data are all correctly modified or their values are left unchanged.

This insures the coherence among data simultaneously modified within a transaction.

J. Wing, J. Woodcock, J. Davies (Eds.): FM'99, Vol. I, LNCS 1708, pp. 348-368, 1999.

A transaction can be initiated, terminated or canceled at any time by the application. Nevertheless, in case of a card pull-out, a transaction must always be terminated in a way which provides data integrity.

Overview of the Transaction Mechanism

The transaction mechanism provides a means to update several data in an "atomic" way. This feature is of great importance to keep the coherence among data which are simultaneously modified.

Within a transaction, it is required that every data value is either correctly modified or left unchanged: if a card withdrawal occurs while a transaction is activated, the system shall preserve each data initial value.

In the example depicted bellow, the modification of data shall change the state of the memory from "State0" to "State1" in the absence of errors.

In case of an uncompleted modification process, the memory may be changed to an unwanted state "State2". Then, the transaction shall restore the initial state "State0".

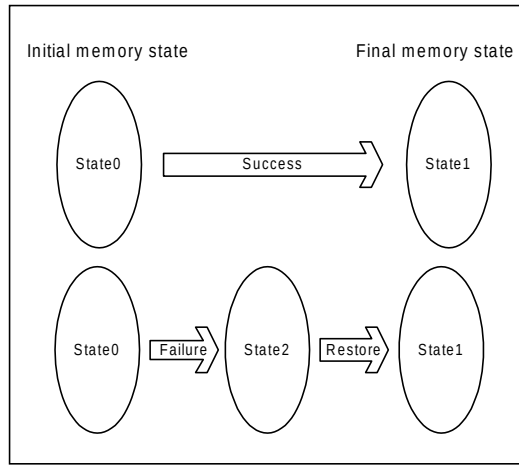

Fig. 1. Overall transaction mechanism

The Backup Level

Prior to any modification or update, the data must be copied (duplicated) in a special location of the EEPROM called the "backup zone". The process of copying

the data in the backup zone is performed by the backup mechanism and may itself be interrupted if the card is withdrawn or if the power is turned off.

Once the copy is performed, data values can be modified freely, without risking a card pull-out.

If several data are modified within a transaction, their copies are piled-up in the backup zone. In case of a card pull-out, their values shall be restored by the system by the "rollback" process at the next card insertion.

Once a transaction is terminated, and if no error has occurred, the data stored in the backup zone are discarded. This is performed by the "commit" process.

Then, for security reasons, the backup zone shall be cleared.

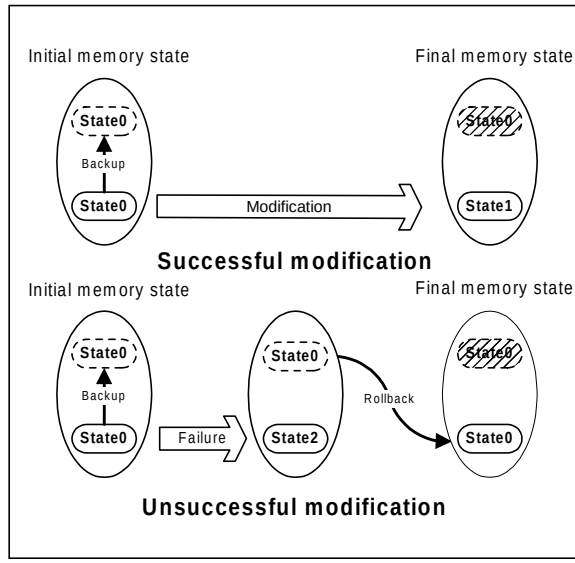

Fig. 2. Transaction and backup mechanisms

The above figure illustrates the use of the backup mechanism in a transaction:

- The data "State0" are first duplicated in the memory,
- The successful data modifications change the subset of memory "State0" to "State1"; the duplicated data then become useless, and they shall be discarded.
- Conversely, in case of a modification failure ("State2" is obtained instead of "State1"), the duplicated data are restored to retrieve the original state of the memory ("State0").

2. The Functional Requirements and the Security Properties

The use of formal methods in industrial applications requires first to identify, state and clarify the functional and security requirements together with the environment assumptions. This task may involve re-expressing the requirements in an informal text (French, English, etc..). This task is preliminary to the formalisation step. Many omissions and inconsistencies may already be found at this early stage of the process.

For the sake of conciseness, only the most important requirements of the transaction mechanism are listed below. Each requirement is labeled to facilitate the traceability of the formal representation.

Functional Requirements

FUN1	The transaction mechanism shall insure that a modification of data in the EEPROM is always performed in a safe way: the data is either correctly modified or its value is left unchanged.
FUN2	At any time of the transaction process, the card may be withdrawn and the power may be turned off.
FUN3	At any time, the application may abort a transaction. The state of the memory shall be brought back to its original state.
FUN4	Nested transactions are not supported: a transaction must be terminated before a new one is started.

Security Requirements

SEC1	To avoid risks of data disclosure, the transaction mechanism shall not permanently retain information, and the backup zone shall be cleared after use.
SEC2	The transaction mechanism shall be resistant to card pull-out.

Environment Assumptions

ENV1	The backup mechanism must not rely on any specific hardware device nor on chip features. It must be generic enough to be implemented on any type of component.
ENV2	Any unpredictable memory behavior due to defective aging cells and over-stressed modifications is out of the scope of the backup mechanism.

3. The Formal Modeling

The formal modeling technique that we use for our application does not consist in directly constructing the software algorithm. Instead, we shall obtain it in a rigorous way, from the main requirements using the refinement mechanism. We consider that the system dynamics are expressed through asynchronous events.
Each event is characterized by two components:

- a trigger that we call "guard", specifying some (but not all) necessary conditions for the event to occur,
- the action which modifies the global data of the system when the guard is valid.

The development is made in several refinements: each step must refine its preceding level. The transition to the next level is characterized by a transformation of the state abstract data to more concrete data, and of course, by the refinement of the events.

The events in refinements can be obtained in several ways:

1. A concrete event directly refines its abstract counterpart (possible strengthening of the guard and translation of the action section to take into account the new data space),
2. An abstract event is split into several concrete refining events. Combined together, the new concrete events shall refine the abstract one.
3. Addition of new concrete events which do not have abstract counterparts. However, the new concrete events must refine the abstract virtual event "skip".

Usually, the model development begins with an abstract model featuring very few events. At the beginning of the model development, we attempt to add events using modalities (2) and (3).

These events provide an implementation of the algorithm.

As a card pull-out may occur at any time during the transaction process, it appears very suitable and effective to represent the system by events.

4. The Abstract Model

Scope

The abstract model provides a high level representation of the functional requirements and the security properties. It aims at stating in a non-ambiguous way the main requirements which must be fulfilled throughout the whole software development life cycle.

The scope of the formal model is to provide a comprehensive representation of *one transaction*. We assume that the backup zone is cleared before a transaction begins. It

ends with the data either correctly updated or left unchanged. In both cases, the backup zone must be cleared before exiting from the transaction process.

Should this model be used in a more complex application, the whole transaction process can be re-started several times, but its validity and robustness is demonstrated once.

Variables / Invariants

The following section introduces the main entities which are used in the model of the abstract machine:

The function "Memory0" represents the EEPROM memory of the smart card. It is a total function from the set of addresses to the set of values (at this stage of the formal representation, we do not take into consideration whether a memory element is a byte, a short integer, etc...).

$$Memory0 \in DATAZONE \longrightarrow DATUM$$

The partial function "NewState0" identifies the subset of elements to be overridden in the memory during the transaction. The variable "NewState0" denotes the final status of the transaction system. It is an abstract representation of the anticipated EEPROM memory state when the transaction is terminated. The "NewState0" variable is only used for formally identifying the expected final memory status, and for formally demonstrating that it will be reached by the system. The variable "NewState0" is an abstract variable called a "prophecy variable":

$$NewState0 \in DATAZONE \nrightarrow DATUM$$

Events

The event "Transaction" describes, from an abstract point of view, the functional properties of the system. It behaves in two different ways as described by the "CHOICE" statement (see below):

- the total function "Memory0" is correctly updated by the partial function "NewState0", **or**
- the function "Memory0" is left unchanged ("skip" statement).

```
Transaction ≙
    CHOICE
        Memory0 := Memory0 <+ NewState0
    OR
        skip
    END;
```

Remark 1. *In the above expression, "Memory0" is overridden by the function "NewState0": Memory0 := Memory0 <+ NewState0.*

5. The First Refinement

Scope

In this refinement, the transaction mechanism is performed in several successive steps, each of them shall be modeled by an event. We introduce the backup zone for storing copies of the data which are modified in the transaction.

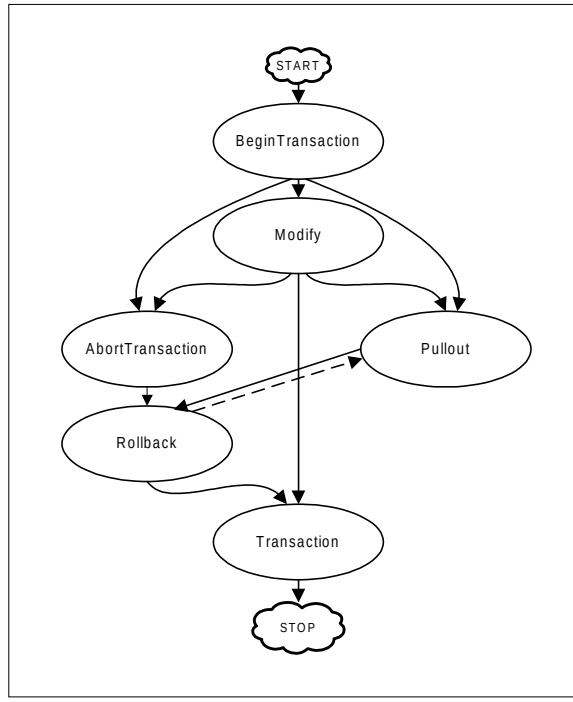

Fig. 3. State diagram of the transaction abstract model.

It is assumed that the backup zone is empty before starting the transaction and it must be demonstrated that it is also empty when the transaction is finished (aborted or normally terminated).

The modeling technique enables refinement of the abstract model by "zooming" in the event "Transaction". This process is also elicited as "time stretching"; it allows the introduction of the events "BeginTransaction", "Modify", "Rollback", "Pullout" and

"AbortTransaction". The transaction mechanism can be represented as a state machine (see above).

Variables and Invariants

In this section, we introduce the major variables and entities of the refinement model:
The variables „NewState1“ and „Memory1“ are identical to the abstract level.
The variable „CardState1“ represents the state of the transaction mechanism:

$$CardState1 \in \{START, BACKUP, MODIFIED, WITHDRAWN, INIROLLBACK, ROLLBACK, STOP\}$$

The variable "BackMemory1" is a partial function which indicates the data which are duplicated in the backup zone:

$$BackMemory1 \in DATAZONE \rightarrowtail DATUM$$

Before starting a transaction, it is assumed that the backup zone is cleared and when a transaction is terminated, the backup zone must also be cleared:

$$CardState1 \in \{START, STOP\} \Rightarrow BackMemory1 = \emptyset \quad (1)$$

The variable "MemoryInit1" is an abstract variable, which indicates the state of the memory, before the transaction is started. It can be considered as a snapshot of the memory just before the transaction is started:

$$MemoryInit1 \in DATAZONE \rightarrow DATUM$$

At any time, except at the end of a transaction, the initial state of the memory "MemoryInit1" can be restored using the backup data "BackMemory1". One should notice that, when the transaction is terminated, the backup zone is cleared *(BackMemory1=Ø);* it is then impossible to retrieve "MemoryInit1" from "Memory1":

$$CardState1 \neq STOP \Rightarrow MemoryInit1 = Memory1 \lhd BackMemory1 \quad (2)$$

From the above expressions (1) and (2), it can be deduced that "MemoryInit1" equals "Memory1" when the transaction is started:

$$CardState1 = START \Rightarrow Memory1 = MemoryInit1$$

The process of modifying the data is considered complete when the EEPROM memory at the beginning of the transaction "MemoryInit", has been overridden by the data "NewState1":

$$CardState1 = MODIFIED \Rightarrow (Memory1 = MemoryInit1 \lhd NewState1)$$

Events

A transaction is initiated by the event "BeginTransaction" and it can only be triggered when the system state is "START". This operation is an abstract operation; its only purpose is to switch the system state from "START" to "BACKUP".

The event "Modify" is an abstract event which indicates that the modifications have been performed and the memory is completely updated. The way the modifications have been realized is not described at this level of the formalisation:

```
Modify ≙
   SELECT CardState1 = BACKUP
   THEN
         Memory1 := Memory1⩤NewState1
         || BackMemory1 := (dom(NewState1) Memory1)
         || CardState1 := MODIFIED
   END;
```

The event "AbortTransaction" may be triggered by the application at any time before or after the modification of data is performed (*CardState1∈{BACKUP,MODIFIED}).* This corresponds to a willful action of the programmer, and the memory must be changed back to its original state (i.e before the transaction was started). Then the system must start the restore/rollback process :

```
AbortTransaction ≙
    SELECT CardState1 ∈ {BACKUP,MODIFIED}
    THEN
         CardState1 := INIROLLBACK
    END;
```

The event "Rollback" restores the initial state of the memory by overriding "Memory1" with the backed up data "BackMemory1":

```
Rollback ≙
    SELECT CardState1 ∈ {INIROLLBACK, WITHDRAWN}
     THEN
         Memory1 := Memory1⩤BackMemory1 ||
         CardState1 := ROLLBACK
    END;
```

The "Pullout" event is an abstract description of a physical and mechanical phenomenon. Its only goal is to make sure that the card pull-out event does not break the invariant statements of the backup mechanism. It will not be implemented, and it is not part of the backup algorithm.

The security requirements of the transaction mechanism are thus fulfilled by demonstrating that the "Pullout" event does not hamper the backup process. The backup mechanism is then formally proved to be resistant to card withdrawals that may occur at any time.

At this level of the refinement process, the "Pullout" event only indicates that the transaction system must proceed with rolling back the data located in the backup zone. This is indicated by switching the state variable CardState1 to "WITHDRAWN: the only possible event to be elected now becomes "ROLLBACK":

```
Pullout ≙
    SELECT CardState1 ≠ STOP
    THEN
        CardState1 := WITHDRAWN
    END;
```

The "Transaction" event has been refined by several new events as described above. It only consists now in erasing the backup data "BackMemory1" and switching the state variable " CardState1" to STOP:

```
Transaction =

    SELECT CardState1 ∈ {ROLLBACK,MODIFIED}
    THEN
        BackMemory1 := ∅ ||
        CardState1 := STOP
    END;
```

6. The Second Refinement

Scope

In the first refinement, the modification of the data was performed in a single event called "Modify".

In reality, a transaction may be performed by several partial data modifications. The goal of this refinement is to introduce two new events "BuffModify" and "BuffBackup" which represent iterative buffer modifications.

The transaction is complete when all partial buffer modifications have been performed correctly as described in the abstract model.

The action performed by the abstract specification of the "Modify" event, is now replaced by an undefined number of iterations of the events "BuffBackup" and "BuffModify".

One should notice that the card is likely to be withdrawn at any time, and in particular before or after the events "BuffBackup" and "BuffModify". This implies that the "Pullout" event may be triggered at any time during the transaction.

The following picture depicts the different states of the transaction system:

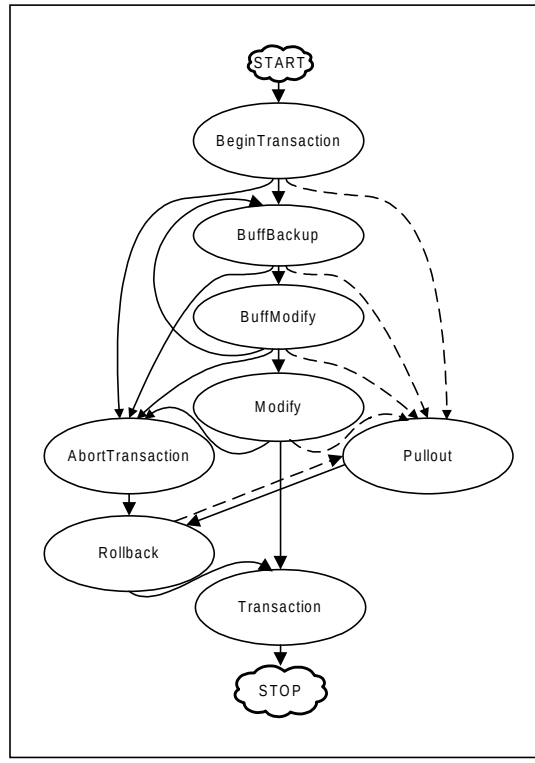

Fig. 4. State diagram of the transaction refinement

Variables and Invariants

The variables "NewState2", "Memory2", "MemoryInit2", "BackMemory2" are identical to their counterparts in the previous level.
Besides, the value "BUFFBACKUP" is added to the possible states of "CardState2":

```
CardState2 ∈ {START, BACKUP, MODIFIED, WITHDRAWN,
INIROLLBACK, ROLLBACK, STOP, BUFFBACKUP}
```

The Variable "NewMemory2" is a function representing a buffer which is updated at each iteration of the sequence defined by the event "BuffBackup" followed by "BuffModify":

```
NewMemory2 ∈ DATAZONE ⇸ DATUM
```

Events

The event "BuffBackup" updates the buffer "NewMemory2". At this level of the abstract model, no particular constraint is imposed on the buffer: it can be of any size and may take any value represented by the partial function "l_new_memory".

The expression of "BackMemory2" shown below indicates that the subset of memory to be overridden by "NewMemory2" is entirely backed up in the backup zone.

One should also notice that if an element is backed up several times, it is only required that the backup zone stores its oldest values. This requirement is of great importance to be able to retrieve the original state of the memory. It is expressed by the expression: BackMemory2:=(dom(l_new_memory) Memory2)⩤ BackMemory2

```
BuffBackup ≙
     SELECT CardState2 = BACKUP
     THEN
          ANY l_new_memory
          WHERE
              l_new_memory : DATAZONE ⇸ DATUM
          THEN
           BackMemory2:=(dom(l_new_memory) Memory2)⩤
                                                BackMemory2
           || CardState2 := BUFFBACKUP
           || NewMemory2 := l_new_memory
          END
     END;
```

The sequencing of the events "BuffBackup" and "BuffModify", is imposed by their guards; it insures that the memory buffer is backed up before being modified by "NewMemory2".

The expression of "BuffModify" is straightforward as the modification of the memory is performed by overriding "Memory2" by the buffer "NewMemory2":

```
BuffModify ≙
     SELECT (CardState2 = BUFFBACKUP)
     THEN
          Memory2 := Memory2 ⩤ NewMemory2
          || CardState2 := BACKUP
     END;
```

Remark 2. In the above expression,"CardState2" is changed to "BACKUP". This implies that, in absence of card withdrawal, the only eligible future events will be "BuffBackup" or "Modify" (see above and after).

The event "Modify" is now refined by an abstract event. Its guard provides the conditions of termination of the memory modification *Memory2 = MemoryInit2 ⩤ NewState2*:

```
Modify ≙
     SELECT CardState2 = BACKUP  ∧
            (Memory2 = MemoryInit2 ⩤ NewState2)
     THEN
            CardState2 := MODIFIED
     END;
```

We must notice that the model does not provide any evidence that the sequence of events "BuffBackup" and "BuffModify" will terminate and that the final state described by "NewState2" is reached.

Instead, the model only indicates that for a transaction to terminate, the complete set of modifications defined by "NewState2" shall be performed by several buffer modifications.

It is the application's responsibility to demonstrate that the transaction events are correctly used, and that the memory modifications predicted by the abstract variable "NewState" are indeed realised.

7. The 3rd Refinement

Scope

The 3rd refinement describes the High Level design of the transaction system: the backup zone is now described as a stack.

For the sake of simplicity, if a data value is required to be backed up several times within a transaction, it is duplicated in the backup zone and piled up in the stack: the system does not perform any checks to reduce the amount of duplicated data.

Similarly, the rollback mechanism consists in restoring the backed up buffers in the reverse order to insure that their oldest values are restored in the memory.

Variables and Invariants

The backup zone is described as a stack and the buffers are piled-up and numbered using increasing numbers. The variable "Top3" represents the number of backed up buffers:

```
Top3 ∈ NATURAL
```

The variable "SavedBuff3" is a total function which associates each buffer to its corresponding number. A buffer is described as a subset of the memory (a partial function from "DATAZONE" to "DATUM") :

```
SavedBuff3 ∈ 1..Top3 ⟶ (DATAZONE ⇸ DATUM)
```

The variable "Add3" identifies the set of all backed up elements. It is an abstract variable. It is used as an intermediate variable to facilitate and simplify the predicate expressions of the invariants:

```
Add3 ⊆ DATAZONE ∧
Add3=dom(union(SavedBuff3[1..top3])
```

The variable "Mark3" indicates for each backed up element belonging to "Add3" , the corresponding buffer number of its first occurrence in the backup zone:

```
Mark3 ∈ Add3 ⟶ 1..Top3 ∧
(∀ad | (ad ∈ Add3 ⇒
Mark3(ad)=min({tt| tt ∈ 1..Top3 ∧ ad∈ dom(SavedBuff3(tt))})
```

"Mark3" is an abstract variable which is used for gluing the variable "BackMemory2" as defined in the previous level to the variable "SavedBuff3". The following statement indicates that the backed up data identified by "BackMemory2" are also marked in the backup zone by the "Mark3" variable. In case of a rollback, these values shall be restored in the memory:

```
Add3 = dom(BackMemory2)∧
∀ad|(ad ∈ Add3 ⇒
          BackMemory2(ad) = (SavedBuff3(Mark3(ad)))(ad))
```

The following figure provides a graphic representation of the entities listed above. In this example, the element "el1" is backed up twice (in the buffers n°2 and n°3 respectively). The function "Mark3" indicates that the first occurrence of the element "el1" in the backup zone is in the buffer n°2. The variable "Top3" is equal to 3:

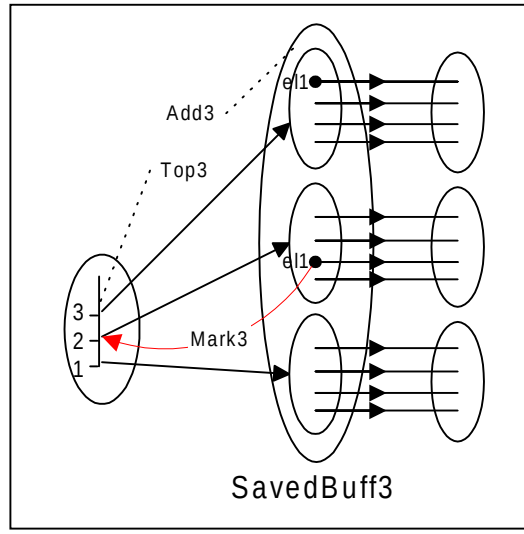

Fig. 5. Abstract representation of the backup zone

Event

The events defined in this refinement have similar meanings to those of the previous level but their expressions are modified to take into account the new variables defined above. The BuffBackup event illustrates these changes:

– the function "NewMemory3" is updated by some partial function "l_new_memory" and the variable "CardState3" is switched to "BUFFBACKUP",
– the buffer of data *(dom(l_new_memory) Memory3)* which will be modified by "NewMemory3" is appended to the backup zone stack "SavedBuff3". Its index is set to "Top3+1". The global expression is: *SavedBuff3 := SavedBuff3 ∪ {Top3+1↦(dom(l_new_memory) Memory3)}.*
– the variable "Top3" representing the index of the last buffer is incremented, *Top3:=Top3+1*,
– the elements of memory which are not already backed up in the backup zone *((dom(l_new_memory)-Add3)* are tagged with the value "Top3+1" and added to "Mark3" *(Mark3:=Mark3∪(dom(l_new_memory)-Add3)*{Top3+1}.*

The expression of the event "BuffBackup" is shown below:

```
BuffBackup ≙
     SELECT CardState3 = BACKUP
     THEN
            ANY l_new_memory
```

```
            WHERE
               l_new_memory ∈ DATAZONE ⇸ DATUM
            THEN
               CardState3 := BUFFBACKUP
             ‖ NewMemory3 := l_new_memory
             ‖ SavedBuff3 := SavedBuff3 ∪
               {Top3+1↦(dom(l_new_memory)  Memory3)}
             ‖ Top3 := Top3+1
             ‖ Mark3 := Mark3 ∪
               (dom(l_new_memory)-Add3)*{Top3+1}
             ‖ Add3 := Add3 ∪ dom(l_new_memory)
            END
     END
END
```

8. The 4th Refinement

Scope

The 4th refinement is used for technical and simplification purposes. In the 3rd refinement, several abstract variables were introduced for gluing the variables with those of the 2nd level. These variables are not required anymore in the 4th refinement ("Mark3" and "Add3") and they are simply removed.

The formal expression of the invariant clause and of the events is identical to the previous model, purged of the predicates and the useless variables.

Variables and Invariants

The variables at the 4th refinement are all identical to the variables of the 3rd refinements, except for "Mark3" and "Add3" which are simply removed. They are labeled "CardState4", "NewMemory4" and the gluing is straightforward:

```
(CardState4 = CardState3) ∧
(NewMemory4 = NewMemory3) etc..
```

Proof

As the refinement only consists in a simplification of the previous model, the resulting proof obligations are either obvious or automatically demonstrated by the Prover.

9. The 5th Refinement

Scope

The 5th refinement consists in modeling the modification and the constraints on the global variables resulting from a card pull-out.

As a card pull-out may occur at any time during the backup process, we might think it necessary for each event to model its counterpart when the card is withdrawn.

It can be easily stated that the only case where a card withdrawal may alter the transaction mechanism is when the backup memory or the data memory are modified. An erroneous modification in the backup zone might prevent the system from being able to restore the initial state of the memory. Similarly, an erroneous modification of the memory may cause the system not to reach its final expected condition.

As a consequence, two new events are introduced to provide a formal expression of the card withdrawal during the processes of backing up and buffer modification: "BuffBackupPullout" and "BuffModifyPullout".

The demonstration of the proof obligation generated by these new events provide evidence that the global invariants of the transaction system are not broken by the "Pullout" events.

They are introduced in the model in place of the "Pullout" event when the system state is either "BACKUP" or "BUFFBACKUP". The guard of "Pullout" is then simply restricted (or strengthened).

When the card is plugged back in the terminal after a pull-out, it is assumed that:

- *only the last buffer of the backup zone may be erroneous (the only case where the last buffer may be erroneous is when the card has been pulled-out while the backup was in progress),*
- *all buffers but the last one are necessarily correct.*

As a consequence, *all valid buffers* in the backup zone require to be rolled back after a card pull-out.

At this level of abstraction, it is assumed that:

- the last buffer is considered invalid if the card is withdrawn while the backup is in progress (the other buffers are all valid),
- all buffers in the backup zone are valid if the card is withdrawn at any other time.

Variables and Invariants

The variable "LstBufVal5" is a simple Boolean value which indicates the last buffer validity:

```
LstBufVal5 ∈ BOOL
```

All variables except "Top5" and "SavedBuff5", are identical to the 4^{th} refinement and are labeled "CardState5", "NewMemory5" etc...
In normal operating mode (no card pull-out), the variables "Top5" and "SavedBuff5" are identical to their counterparts in the previous refinement, and the last buffer is assumed to be correct. This is stated as follows:

```
(LstBufVal5 = TRUE
 ⇒(Top5 = Top4 ∧SavedBuff5 = SavedBuff4))
```

In case of a card pull-out occurring while the backup is in progress, the last buffer is declared erroneous *(LstBufVal5 = FALSE)*. It comes out that all buffers of the backup zone except the last one, match with the previous refinement *(1..Top4) SavedBuff5=SavedBuff4*. The last erroneous buffer is simply added on top of the backup stack *(Top5=Top4+1)*. This is expressed as follows:

```
LstBufVal5 = FALSE
 ⇒ ((Top5=Top4+1)∧(1..Top4) SavedBuff5=SavedBuff4)
```

Events

The event "BuffBackupPullout" is similar to its counterpart "BuffBackup" except that the stack "SavedBuffer5" is updated by a buffer "`l_back`" which is not related to "NewMemory5": $SavedBuff5 := SavedBuff5 \cup \{Top5+1 \mapsto l\_back\}$.

The Boolean "LstBufVal5" is set FALSE:

```
BuffBackupPullout ≙
     SELECT CardState5 = BACKUP
     THEN
            ANY l_back
            WHERE
               l_back ∈ DATAZONE ⇸ DATUM
            THEN
               CardState5 := BUFFBACKUP
             || SavedBuff5:=SavedBuff5∪{Top5+1↦l_back}
             || Top5 := Top5+1
             || LstBufVal5 := FALSE
            END
     END
```

Similarly, the event "BuffModifyPullout" is identical to the event "BuffModify" except that the buffer which is appended to the memory is not equal to "NewMemory5" (it can be of any kind).

Proofs

The demonstration of the proof obligations provides evidence that the events "BuffBackupPullout" and "BuffModifyPullout" fulfill the invariant predicates. They must preserve the following statements:

- all buffers of the backup zone except the last one are identical to the previous refinement,
- the last buffer is erroneous if the card is pulled-out while the backup is in progress; it shall not be restored by the rollback process.

10. The 6th Refinement

The 6th refinement is used for technical and simplification purposes.

11. The 7th Refinement

The 7th refinement provides an implementation of the backup zone. Each buffer of the backup zone consists of a header and a set of values. The header provides the necessary information for restoring number of data values in the memory: the start address of the buffer zone and the length (number of data) of the buffer.

A supplementary checksum is added to the Header. It is assumed that the checksum is able to determine in a non-ambiguous way the validity of the buffer.

The details of the computation of the checksum are out of the scope of this article. The formal representation only states that the validity of the last buffer determined by the variable "LstBuffVal5" (see 5th refinement) can be substituted by the computation of a checksum value. Nevertheless, we must notice that the checksum calculation must prevent misinterpretation caused by simultaneous errors occurring together in the buffer data and in the checksum itself.

12. The 8th,9th, and 10th Refinements

The last three refinements of the development provide the complete algorithm of the transaction system. Several pointers are introduced for managing the backup and the rollback processes.

For the sake of efficiency, these pointers are located in the RAM memory. This implies that their values are lost when the card is pulled out. This is modelled by erasing their values in the pull-out events (setting their values to zero for example). The formal model provides evidence that the algorithm is resistant to card pull-out, and the pointer values can be retrieved by a well designed scanning of the backup zone, and by analysing the header structures: checksum, size of the buffer data etc..

13. The Last Refinement / The Implementation Level

The last refinement is called the implementation. It provides the complete backup algorithm and is translated into the C language.

The process of code generation needs to transform an event based architecture into a sequential program which can be run by a smart card processor.

This can be obtained by first transforming the SELECT statements of the events into a PRE statement; the events must now be considered as pre-conditioned operations.

The sequencing of these operations is performed by a "scheduler" which is responsible for calling the operations in a way that complies with the pre-conditions. The scheduler is defined in a separate B machine, and its correctness is verified by the demonstration of the Proof Obligations.

The actual version of the "B0 checker" and "C" translator AtelierB tools, do 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 keep in mind that no proof obligation can be generated during this process.

14. Code Generation

The design of the transaction mechanism using the B formal method has been carried out in parallel with a traditional design using the C language.

The generation of the source code from the B model has been performed manually and its effectiveness in terms of code size, memory and stack is strictly comparable to the traditional design:

- The same number of variables and pointers on the backup stack has been used in both designs,
- In order match the code size of the original design, some optimizations have been required for translating the event based model: all operations of the transaction model have been in-lined in the "scheduler" calls to reduce the processor stack and improve the execution speed.

15. Conclusion

The use of the B method for the design of the transaction mechanism has been greatly beneficial for the following reasons:

- formal description of the functional and security requirements, together with the environment assumptions,
- formal validation of the detailed design and the pseudo-code algorithm,
- reduction of the validation task load,
- 100% of the Proof Obligations of the project (over 1500 in total) have been demonstrated in interactive mode with no added lemmas.

The transcription of C source code from the implementation level has been performed manually as no automatic translator is available for smart Card applications.

The results in terms of code size, memory and stack is strickly comparable to the original design.

The design of an automatic translator is one of our concerns:

- It shall be able to generate a source code which is compact enough to be used in the smart card applications,
- It must be flexible enough to meet the peculiarities of compilers and chips which are used in the industry,
- It should also take into account the programming recommandation to improve the security of the design.

Acknowledgements

Many Thanks to J.R Abrial for his contribution to the design of the B model, and to Savy Kong for his commitment and his enthusiasm in the project during the summer of '98.

References

1. Abrial J-R, *The B-Book: Assigning programs to meanings.* Edited by Cambridge University Press (1996)
2. Abrial J-R and Mussat L, *Introducing Dynamic Constraints in B.* D.Bert (Editor): Proceedings of the Second International B Conference B'98: Recent Advances in the Development and Use of the B Method, April 1998, Springer.
3. 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)