5th International Eurographics Workshop on Design, Specification, and Verification of Interactive Systems (DSV-IS'98), Cosener's House, Abingdon, UK, June, 3-5 1998, pp 333-352.

A Uniform Approach for Specification and Design of Interactive Systems: the B Method

Yamine AIT-AMEUR, Patrick GIRARD, Francis JAMBON

LISI / ENSMA
BP 109, Téléport 2
F-86960 Futuroscope cedex, France
Tel.: +33 5 49 49 80 63, Fax: +33 5 49 49 80 64
E-mail: {yamine, girard, jambon}@ensma.fr
Web: http://www.lisi.ensma.fr/cao.html

Abstract : We have experienced the B Method on a case study which was defined by the French working group on formalisms for interactive systems, i.e. a Post-It® Notes like collaborative application. This experience showed that the B approach allows to cover the description, the formal specification, and the design of each component of basic architecture models, i.e., the five components of the Arch model. Moreover, it has shown that the proposed approach is capable to formally handle large case studies and generate proof obligations which, when proved –automatically– allows to assert the correctness of the development, and the checking of several user requirements.

Keywords : B method, specification refinement, software architecture, interaction properties verification, case study, specification of interactive system.

1. Introduction

The past four editions of DSV-IS Workshops have largely focused on formal specification for Interactive Systems. The first approaches attempted to define new semi-formal notations for parts of interactive system design, such as UAN [Hix & Hartson 1993] or MAD [Scapin & Pierret-Golbreich 1990] without defining specific models. The expressive power of these notations is clearly correlated to the domain they are planned to cover, but their formal semantics remains undefined. Conversely, the major researchers' efforts during the past few years have been to endeavour the use of well-known formalisms in interactive system design, such as the Z-notation [Johnson 1995], Petri nets [Accot, Chatty, & Palanque 1996], or functional approaches, whose semantics is clearly defined. However, most of these approaches lead on interactive models such as the York Interactors [Duke & Harrison 1993b], Interactive Cooperative Objects [Palanque 1992] or the CNUCE model [Paterno' 1994 ; Paterno' & Faconti 1992].

Our goal in this work is to explore a third approach: we suggest using a well-defined formal method in interactive design context, without defining or using any interactive model. We propose the B Method [Abrial 1996] which is a model description oriented formalism, such as VDM [Bjorner 1987] or Z [Spivey 1988]. The choice of this method was motivated by the existence of a complete software development tool supporting it [Steria méditerranée 1997]. We tried it out on a case study which has been defined by the French working group on formalisms for highly interactive systems [Palanque & Girard 1996]: a Post-It® Notes like collaborative application.

This paper is organised as follows. The first section discusses some formal approaches that have been studied in the design of interactive systems context. We particularly focus on the scope and the goal of these approaches (specification, validation, or verification) and on the tools that support these methods. The second section describes the case study we have chosen. The third section presents the B method and illustrates it with one abstract machine of the functional core of our case study. Then, the fourth section details the B expression of specific interactive features from the case study. Last, we conclude this paper giving briefly our results either from the formalisation or the verification point of views.

2. formal approaches overview

As we stated in the introduction, a lot of ad hoc formalisms have been defined for Interactive System description or verification.

MAD (*Méthode Analytique de Description des Tâches*) [Scapin & Pierret-Golbreich 1990] is first and foremost a method for task analysis. The typical user of the MAD method is a human factors specialist. The goal of a MAD study is to collect the set of user's tasks –computerized or not. Recently, the method has been extended to the logical presentation level of user interfaces by the MAD*/SSI models [Gamboa Rodrígez & Scapin 1997]. Supported by a set of tools, IMAD*/ALACIE, the method now bridges the gap from tasks to interaction. UAN (*User Action Notation*) [Hix & Hartson 1993] proposed another approach: it focus on the description of the interaction level of graphic user interfaces. For example, a basic action of a UAN description is pointing an object with the mouse. In XUAN [Gray, England, & McGowan 1994], the temporal relationships of UAN have been extended, and a subset on XUAN descriptions, eXUAN [McGowan 1995] may be partially executable. UAN may express the same temporal relationships as MAD, but does not provide any methodology to extract them. The well-known GOMS model (Goals, Operators, Methods, Selections) [Card, Moran, & Newell 1983] is meant for the predictive evaluation of interaction between the final system an expert user. The GOMS' tasks are ranked among their duration and complexity. Duration is also one of the main results of a GOMS evaluation.

On the opposite, strongly correlated to Interactive Model definition, several approaches have used well-known formal systems or notation in order to address validation and verification issues. Classical Petri net formalism has been adapted to interactive needs in the ICO model [Palanque 1992], and model oriented Z notation has been used in the York Interactors [Duke & Harrison 1993a ; Duke & Harrison 1993b]. LOTOS has been used in the CNUCE model [Paterno' 1994 ; Paterno' & Faconti 1992]. The last edition of DSV-IS Workshop discussed the actual use of formalisms. Two major needs emerged: verification and validation

[Fields, Merriam, & Andy 1997], also called 'verification of implementations against their specifications' and 'verification of specification properties' [Campos & Harrison 1997]. In fact, most of the results focused on the second problem, that is ensuring the correctness of specification. In the review of Campos and Harrison, a framework for the classification of user interface properties has been defined. The current tendency seems today to attempt to correlate different formalisms to cover the aspects of interactive systems. In fact, while the major part of these approaches addresses the validation of specification, using tools appears to be difficult: they do not exist, or are limited in use. When using different formalisms, articulating them together is also very difficult.

In opposite to these options, we will explore the way of using a unique formal notation, suitable for proving, and, so doing, to fulfil verification requirements: the B method. It provides a uniform setting to describe all the parts of an interactive system.

3. The case study

Our case study is a cooperative version of a Post-It® Note software. It appears on the screen as a Block which can be iconified, but cannot be moved. Clickable areas allow direct manipulation of the block, for iconification, and close (quit). It is possible to enter text into the Post-It® Note on the block, which always exists. Last, it is possible to detach the upper Post-It® Note from the block, and then to drag it anyway. The Post-It® Notes themselves have a similar behaviour, that differs in four ways: (1) the detach area is restricted to a drag command area, (2) the close area becomes a kill area, (3) a resize area is defined in the lower-right corner, corresponding to a standard resize behaviour, and (4) a send behaviour is defined. This behaviour consists in emitting the Post-It® to a receiver which is visualized as a special icon. The interactive trigger is a drop (mouse up event) onto the icon.

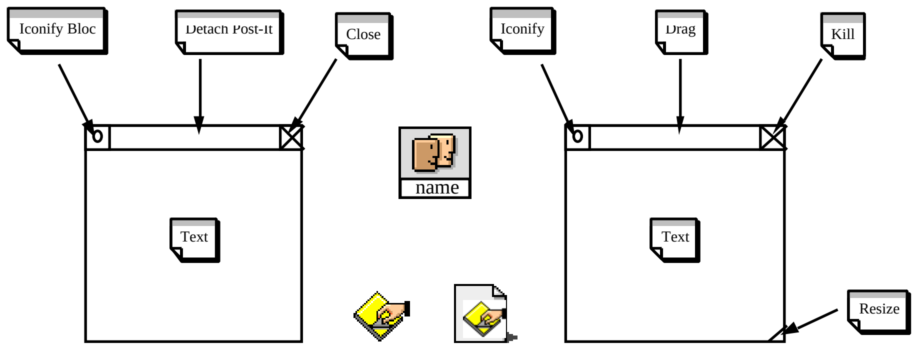

Figure 1: From the left to the right, The Post-It® block, the three icons (Post-It® block, User, and Post-It®), and the Post-It® itself, together with their active areas.

In this work, we study more particularly the interactive behaviours of the block and the Post-It® Notes. We focus on direct manipulation of these two classes of objects, and we do not expand the "receivers" point of view. We leave the text inputs, that are restricted to "string input". On the opposite, we focus on the mouse actions and their relations with the representation and the behaviour of objets.

4. The B-Method

Among the increasing number of formal methods that have been described during the last decade, model oriented methods, such as VDM, Z or B, seem to have a good place. These methods are based on model description. They consist in defining a model by the variable attributes which characterize the described system, the invariants that must be satisfied and the different operations that alter these variables. Starting from this observation, Z method uses set theory notations and allows to encode the specifications in a structure named schema. Like VDM, it is based on preconditions and postconditions [Hoare 1969 ; Hoare, et al. 1987]. Moreover, VDM allows the generation of a set of proof obligations which simplify the use of the method regarding to Z. In opposite, B is based on the weakest precondition technique of Dijkstra [Dijkstra 1976]. Starting from this method, J.R. Abrial has defined a logical calculus, named the generalized substitutions calculus. Notice that our choice is B. This choice is motivated by the fact that B is supported by tools which allow a complete formal development. Moreover, since it is based on the weakest precondition calculus, B helps to prove the termination.

4.1 The abstract machine notation

The abstract machine notation is the basic mechanism of the B method. J.R. Abrial defined three kinds of machines identified by the keywords MACHINE, REFINEMENT and IMPLEMENTATION:

- MACHINES represent the upper level of a program development. They represent the higher level of abstraction. They describe the formal specification of a system.
- REFINEMENT machines are intermediate steps. They are less abstract than the MACHINES. They are used to refine the MACHINES into structures that are approaching more and more the programming language level.
- Finally, IMPLEMENTATIONS are the last development level. They represent the algorithms that implement the specification described at the MACHINE level. When proved, IMPLEMENTATIONS can be translated into a program written in a given programming language.

Note that the development is considered to be correct only when every refinement is proved to be correct with respect to the semantics of the B language. Gluing invariants between the different MACHINES of a development are defined and sets of proof obligations are generated. They are used to prove the development correctness.

4.2 Description of abstract machines

Several important clauses have been described by J.R. Abrial for the definition of abstract machines. Depending on the clauses and on their abstraction level, these clauses can be used at different levels of the program development. In this paper, a subset of these clauses has been used for the design of our specifications. We will only review these clauses. A whole description can be found in the B-Book [Abrial 1996]. The following table shows the syntax of the machines we are using in our case study. Other syntax possibilities are offered in B, and we do not intend to review them in this paper, in order to keep its length short enough.

```
MACHINE
   EXTENDS
   SETS
   CONSTANTS
   PROPERTIES
   VARIABLES
   DEFINITIONS
   INVARIANT
   INITIALISATION
   OPERATIONS
END
```

Briefly, these clauses mean:

- EXTENDS is a clause that allows to import instances of other machines. Every component of the imported machine becomes usable in the current machine. This clause allows modularity capabilities.
- SETS defines the sets that are manipulated by the specification. These sets can be built by extension, comprehension or with any set operator applied to basic sets.
- CONSTANTS defines all the constants that are used in the machine. Notice that the constants described can have any type (naturals, elements of sets, constant functions and so on).
- PROPERTIES are logical expressions that are satisfied by the constants described in the previous clause.
- VARIABLES is the clause where all the attributes of the described model are represented. In the methodology of B, we find in this clause all the selector functions which allow accessing the different properties represented by the described attributes.
- DEFINITIONS is a set of definitions introduced by the user. They are rewritten everywhere they are used in a machine. It allows simplification of machine notation.
- INVARIANT clause describes the properties of the attributes defined in the clause VARIABLES. The logical expressions described in this clause remain true in the whole machine and the represent assertions that are always valid.
- INITIALISATION clause allows to give initial values to the variables of the corresponding clause. Note that the initial values must satisfy the invariant.

· OPERATIONS clause is the last clause of a machine. It defines all the operations (functions and procedures) that constitute the abstract data type represented by the machine. Depending on the nature of the machine, the OPERATIONS clause authorizes particular generalized substitutions to specify each operations. The substitutions used in our specifications and their semantics is described below.

4.3 Semantics of generalized substitutions.

The calculus of explicit substitutions is the semantics of the abstract machine notation. It is based on the weakest precondition approach of Dijkstra. Formally, several substitutions are defined in B. If we consider a substitution S and a predicate P representing a postcondition, then [S]P represents the weakest precondition that establishes S after execution. The substitutions of the abstract machine notation are inductively defined by the following equations. Notice that we did not give the whole substitutions, we restricted ourselves to the ones used for our development. The reader can refer to the literature [Abrial 1996 ; Lano 1996] for a more complete description:

```
      [SKIP]P    <==> P
      [S1 || S2]P   <==> [S1]P and [S2]P
      [PRE  E  THEN S  END ]P     <==> E  and  [S]P
      [ANY v WHERE E THEN S END]P    <==>   v (P =>[S]P)
      [x:=E]P    <==> P(x/E)
```

The lst one represents the predicate P where all the free occurrences of x are replaced by the expression E.

4.4 An example: Post-It®-mess

Based on the definitions, and for the case study previously exposed, a first abstract machine is defined. It is related to the management of messages written on a given Post-It®. This simple machine is used to illustrate the abstract machine notation. In fact, this machine represents the heart of our application, i.e., the functional core. In our case study, it only contains the message written onto the Post-It®. Additionally, a state variable has been defined, which represents different state possibilities for the Post-It® itself (read, destroyed, stored, etc.). Notice that most choice in this specification do not relate to the initial requirements of our case study (which was very unprecise). We give comments over this machine where necessary:

```
MACHINE POSTIT_MESS
SETS
   POST_MESS;
   MESS_STATUS  = { read, unread, destroyed, stored}
CONSTANTS
   max_post_mess
PROPERTIES
   max_post_mess=card(POST_MESS)
```

The set POST_MESS represents the set of all the possible messages to be written. This set behaves as a type for messages. The properties clause asserts that its cardinal is given by the constant max_post_mess. Finally, the set MESS_STATUS enumerates the possible states of a message which are interesting for the application we are currently specifying.

```
VARIABLES
   the_post_it_mess, message_status, post_it_mess_creation,
the_message
INVARIANT
   the_post_it_mess       <: POST_MESS      &
   the_message : the_post_it_mess       -->   STRING      &
   message_status : the_post_it_mess       -->   MESS_STATUS   &
   post_it_mess_creation: the_post_it_mess       -->   BOOL
INITIALISATION
   the_post_it_mess :={}     ||
   message_status   :={}     ||
   the_message   :={}     ||
   post_it_mess_creation  :={}
```

Variables define the model; the *Invariant* clause allows the typing of these variables and finally, the *Initialisation* clause gives their initial values. In this machine, the_post_it_mess is the set of the effectively created messages. It is included (<:) in the set of all the messages. Three functions the_message, message_status and post_it_mess_creation are functions which return the string corresponding to the message content, the status of the message and a Boolean attesting that the message is effectively created. Notice that these functions represent the properties of a message. Moreover, this is a specification based on the definition of selectors.

Below, a set of operations are defined using the PRE THEN substitution which was described before.

```
OPERATIONS
```

The create_post_it_mess operation allows the creation of message containing a String mess, with an unread status.

```
   pp <-- create_post_it_mess(mess)=
      PRE
         the_post_it_mess /= POST_MESS &
```

Because of the limit, we must ensure that we can create another Post-It.

```
         mess : STRING
      THEN
      ANY
         tt
      WHERE
         tt : POST_MESS - the_post_it_mess
      THEN
         the_post_it_mess := the_post_it_mess \/ {tt}  ||
         the_message(tt)  := mess              ||
         message_status(tt)  := unread            ||
         post_it_mess_creation(tt) :=    TRUE              ||
         pp := tt
      END
   END ;
```

The read_message operation allows the reading of a message represented by pp. Its precondition says that the message is created (since it belongs to the set the_post_it_mess) with an unread status and updates its status when reading is achieved.

```
   read_message (pp)=
      PRE
         pp : the_post_it_mess&
         message_status(pp=  unread
      THEN
         message_status(pp:=    read
      END
```

The destroy_message operation allows the destruction of a message represented by pp. Its precondition says that the message is created (since it belongs to the set the_post_it_mess) with an unread or read status. This means that an eventually unread message can be destroyed. The status is then updated to become destroyed.

```
   destroy_message (pp) =
      PRE
         pp : the_post_it_mess &
         (message_status(pp)=unread or  message_status(pp)=read
)
      THEN
         message_status(pp):=destroyed
      END ;
```

The store_message operation allows the storage of a message represented by pp. Its precondition says that the message is created (since it belongs to the set the_post_it_mess) with an unread or read status. This means that an eventually unread message can be stored. The status is then updated to become stored.

```
   store_post_it(pp) =
      PRE
         pp : the_post_it_mess &
         (message_status(pp)=unread or message_status(pp)=read )
      THEN
         message_status(pp):=stored
      END
END
```

Some choice realized here should be discussed. They are independent from any interactive activity, and can be clearly explained. Hence, it is not possible to read twice a Post-It®: reading it lead it to the read status, while being unread belongs to the precondition of "read_post_it". As we will see later, the B method will ensure these choice in the whole application. Notice that all the abstract machines presented in the case study of this paper are designed following this one. It can be used as a template for the description of the other machines.

4.5 Strengths of B

The B Method is based on sound and well known semantics since it is based on predicate logic and on the weakest precondition calculus. But one of the major advantages of this approach is the uniform description of the whole development. Indeed, we will show in the reminder of this paper that the same notation is kept to describe every part that constitutes an interactive system. Moreover, this method gives a technique for proof obligation generation, proving, and refining to code.

4.5.1 Proof obligations (PO)

The calculus of generalized substitutions outlined above is applied for each abstract machine defined in the development. Rewriting techniques are applied to achieve this calculus and they lead to a set of proof obligations which need to be proved in order to have a sound and consistent specification and development.

4.5.2 Proofs and proving

When the proof obligations are generated, they have to be proved. For this purpose, a set of proof rules are provided and the developer can achieve the proof of the PO's if they are provable. Moreover, the tools implementing the B Method have an automatic prover which allows to prove a major part of these PO's. The remaining PO's are proved "by hand" using the interactive prover. This approach allows to check the correctness of the specifications with respect to the user needs. Indeed, some of the PO's are definitely not provable if the abstract machine is not well defined. For example, in the previous abstract machine, we cannot store a message which is destroyed. Finally, in our case study, the 17 PO of the POSTIT_MESS abstract machine have all been proved automatically using the "Atelier B" tool [Steria méditerranée 1997].

4.5.3 Refinement: from the specifications to the code

As stated above, the B Method allows not only to support specifications through abstract machines, but it allows the support of refinement and implementations as well. Indeed, it is possible to set the whole development in a common language, with a common semantics and a common proof technique. This is very important since it provides a uniform approach for the description of program developments.

5. Experimenting the B-Method

In this section, we explain how the interactive viewpoint can be taken into account during a B design. We base our modular decomposition on the ARCH model [UIMS 1992]. In the B terminology, the architectural modules are called "abstract machines". The figure 2 shows the architectural decomposition, with dependencies between machines.

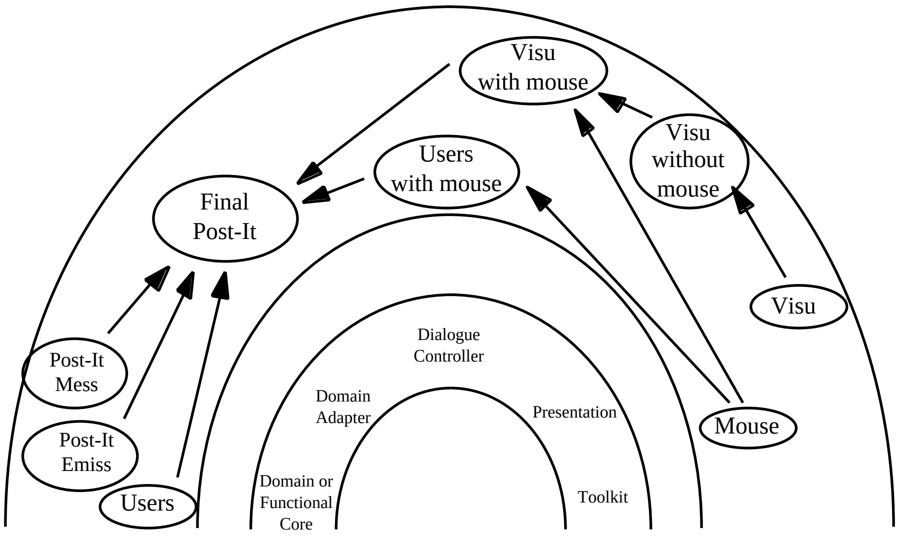

Figure 2: architectural ARCH-like decomposition of cooperative Post-It® Notes application, with the reference to the ARCH model

We illustrated the previous section with a machine (Post-It-Mess) which belongs to the functional core. This is the normal use of the B-method. In the following subsections, we explain how B can be used to take into account more specific interactive features. Section 5.1. explains the non-interactive aspects of Post-It® visualization. It includes visual states (iconified or not, greyed, hidden, and so on) and is splitted into two "abstract machines", for B convenience (Visu and Visu-without-mouse). The links with the graphical toolbox is not detailed here. Section 5.2. exposes the necessary reverse engineering of the mouse. It is obvious that the existence of a B mouse "abstract machine" would have skipped this phase. Section 5.3. details the integration of mouse and visualization in order to define the dialogue controller of the Post-It®. Section 5.4. shows how functional core requirements and dialogue requirements can mix to build the whole application. Positioning this machine conforming to the ARCH model is not straightforward. It is concerned with domain adaptation, but also with dialogue control. So, using "slinky" facilities of the ARCH model, we propose including it at the boundary of the domain adapter and the dialogue controller. Lastly, we expose in section 5.5. the practical use of B-Tools, and demonstrate the proving process. Notice that names in figure 2 do not match actual names in the B development, for graphical space reasons. We will ensure the correspondence as needed.

5.1 The Post-It® Visualization: the POSTIT_VISU_WITHOUT_INT machine

Two machines were specified for the visualization of the windows. The first one is the POSTIT_VISUALIZATION abstract machine (*visu* in fig. 2) which manages the state of the windows from display point of view. Indeed, a set named SCREEN_STATE={displayed, hidden, iconified, eliminated} allows recording all the possible states of the windows. Moreover, this abstract machine contains operations like update_hidden which specifies the refreshment of the screen. The second machine (*visu without mouse* in fig 2), extends the previous one and allows to manage the windows without dealing with any kind of interaction. It is a high level abstraction of a subset of the toolkit.

```
MACHINE POSTIT_VISU_WITHOUT_INT
EXTENDS
   POSTIT_VISUALIZATION
SETS
   POST_VISU;
   COLOUR = {yellow, green, white, black, red,grey};
   BLINK = {blinking, non_blinking};
   SOUNDS = { anormal_sound, emit_sound, no_sound };
   REACTION = {greyed, enabled, disabled} ;
   BLOCK_VISU = {block_iconified, block_open}
```

The previous sets describe all the values of the possible states of a window starting from colours, blinking, emitting sounds, reaction and the state of the whole block of Post-It®'s.

```
CONSTANTS
*  max_post_visu,
   max_post_it_wide,
   max_post_it_high,
PROPERTIES
*  max_post_visu=card(POST_VISU)   &
*  max_post_it_wide=300   &
*  max_post_it_high=250   &
VARIABLES
   the_post_it_visu,
   get_new_post ,
   block_state,
   post_it_window_status,
```

These variables describe the accessors to the different components of a window. The the_post_it_visu is the set of all the windows effectively created, and get_new_post, block_state and post_it_window_status are the function which respectively imports a window from the POSTIT_VISUALIZATION machine, gives the state of the block and computes the status of the window from the one on the imported window.

```
INVARIANT
   the_post_it_visu <: POST_VISU &
   get_new_post  :  the_post_it_visu --> post_new &
   block_state   :  block_VISU &
   post_it_window_status  :  the_post_it_visu -->SCREEN_STATE &
   (!xx. (xx : the_post_it_visu    =>
      (x_post_it_position(xx) : 1..max_post_it_wide &
      y_post_it_position(xx) : 1..max_post_it_high )))
```

The previous logical expression is an important invariant. It states that all (!) the windows of the set the_post_it_visu have their upper-left corner in the screen. This invariant is conserved and proved for all the operations.

We will now describe the "move_window_position" action, and every elements it uses. In this machine, move_window_position is not strictly constrained. For example, it is possible to partially quit the visualization window while moving a Post-It®. This constraint will be put later on. We only take into account the constants, variables and invariants that are needed by move_window_position.

```
OPERATIONS
   move_window_position(pp, aa, bb)=
   PRE
      aa : NAT &
      bb : NAT &
      aa : 1..max_post_it_wide &
      bb : 1..max_post_it_high &
      pp : the_post_it_visu   &
      block_state = block_open   &
      (post_it_window_status(pp)= displayed  or
      post_it_window_status(pp)= hidden )
```

The previous preconditions say that the new coordinates of the upper-left corner of a window (aa,bb) must belong to the screen, that the window pp is effectively created (belongs to the_post_it_visu set), that the block of Post-Its is effectively open and that the window can be displayed or hidden.

From these previous precondition, we can infer the following properties:

- a window is never moved if it is iconified,
- a window is never moved if the block is not open,
- a window will never have its upper-left corner outside from the screen. This is very important for the window manipulation.

```
   THEN
   ANY
      ppold
   WHERE
      ppold : the_post_it_visu &
      ppold = pp
   THEN
      x_post_it_position(pp):= aa    ||
      y_post_it_position(pp):= bb    ||
      update_hidden(get_new_post(pp))
   END
END;
```

Here, a parallel substitution is used. It says that after executing this operation the new coordinates of the upper-left corner become (aa, bb) and that the operation update_hidden is called. This last operation is called in order to update the status of the windows that become hidden or displayed after the current window has been moved.

5.2 Reverse engineering of the mouse: the POSTIT_MOUSE machine (*mouse* in fig 2)

In order to allow the interaction using a mouse, thanks to reverse engineering techniques, we have extracted a set of specifications which describe the behaviour and the actions performed on the mouse. Let us briefly summarize them.

```
MACHINE POSTIT_MOUSE

SETS
```

```
   POST_MOUSE;
   MOUSE_STATE ={up, down , clicked}
```

The mouse can be in three states. Without loss of generality, we have voluntarily omitted the other states because they are not used in our application. Notice that they could have been included.

```
CONSTANTS
   max_post_mouse,
   x_mouse_position_default,
   y_mouse_position_default,
   max_mouse_position_wide,
   max_mouse_position_high
PROPERTIES
   max_post_mouse =card(POST_MOUSE) &
   x_mouse_position_default=20 &
   y_mouse_position_default=20 &
   max_mouse_position_wide = 300 &
   max_mouse_position_high =250
```

These constants define the size of the screen for the window together with the default position for a mouse.

```
VARIABLES
   the_post_it_mouse,x_post_it_mouse_position,y_post_it_mouse_p
osition,    post_it_mouse_state,post_it_mouse_creation
INVARIANT
   the_post_it_mouse<: POST_MOUSE &
   x_post_it_mouse_position :   the_post_it_mouse --> NAT  &
   y_post_it_mouse_position :   the_post_it_mouse --> NAT &
   post_it_mouse_state:   the_post_it_mouse --> MOUSE_STATE &
   post_it_mouse_creation  : the_post_it_mouse --> BOOL
```

For a given mouse in the set the_post_it_mouse, the previous accessors define a set of functions which allow to retrieve the coordinates of a mouse, its state and its creation. Several operations on the mouse (creation, moving, clicking, and so on ...) are defined below.

```
OPERATIONS
   pp <--create_mouse_position=
      ...
   Move_mouse_with_drag(pp, aa,bb)=
      ...
   move_mouse(pp, aa, bb)=
      ...
   mouse_up(pp)=
      ...
   mouse_down(pp)=
      ...
   mouse_clicked (pp)=
      ...
END
```

This reverse engineering task to re-design the mouse specification is a crucial phase in our work. It allows to use the B language in a uniform manner even for already designed programs. However, this steps of development needs to be achieved for other parts of an interactive application (e.g. keyboard).

5.3 The interaction: Post-it-visu-with-int

The following abstract machine, named POSTIT_VISU_WITH_INT_MOUSE (*visu with mouse* in fig 2), allows the use of mouse interaction on the windows. Therefore, this machine extends the two machines corresponding to windows and mouse.

```
MACHINE POSTIT_VISU_WITH_INT_MOUSE

EXTENDS
   POSTIT_MOUSE , POSTIT_VISU_WITHOUT_INT
SETS
   POSTIT_VISU_WITH_MOUSE
CONSTANTS
   max_post_it_visu_with_mouse
PROPERTIES
   max_post_it_visu_with_mouse = card(POSTIT_VISU_WITH_MOUSE) &
   screen_wide = max_mouse_position_wide &
   screen_high = max_mouse_position_high &
   max_post_it_wide=max_mouse_position_wide &
   max_post_it_high=max_mouse_position_high
```

The previous properties express that the screen for the mouse and for the windows are of the same dimensions. These properties are mandatory to prove the correctness of the development from the positions point of views.

```
VARIABLES
   the_post_it_visu_with_mouse,
   post_it_visu_with_mouse_creation,
   get_the_mouse,
   get_the_post_it_visu
INVARIANT
   the_post_it_visu_with_mouse <: POSTIT_VISU_WITH_MOUSE &
   post_it_visu_with_mouse_creation:the_post_it_visu_with_mouse
-->BOOL &
   get_the_mouse : the_post_it_visu_with_mouse --
>the_post_it_mouse &
   get_the_post_it_visu :the_post_it_visu_with_mouse-->
the_post_it_visu
```

The set the_post_it_visu_with_mouse records all the windows manipulated by a mouse. The selectors define a set of functions which allow to extract the properties of a given window which is manipulated by a mouse. It gives the window using the get_the_post_it_visu function and the mouse using the get_the_mouse function.

```
INITIALISATION
   the_post_it_visu_with_mouse     :={} ||
   get_the_mouse    :={} ||
   get_the_post_it_visu      :={} ||
   post_it_visu_with_mouse_creation     :={}
OPERATIONS
   pp<-- create_post_it_and_mouse(pp_visu, pp_mouse) =
      ...
```

Let us give the details of the action move_window_with_mouse which allows to move a window combining the interaction of the mouse. This action has a set of preconditions which needs to be valid before the action is performed. They are commented below.

```
   move_window_with_mouse(pp, aa, bb)=
   PRE
      aa : NAT &
      bb : NAT &
      aa : 1..max_post_it_wide &
      bb : 1..max_post_it_high &
      pp : the_post_it_visu_with_mouse   &
```

The new coordinates of the upper-left corner of the window must define a point in the limits of the screen.

```
      block_state = block_open   &
```

The block of post'its must be open, otherwise the window of a post'it cannot be moved.

```
      post_it_visu_with_mouse_creation(pp)=TRUE &
      post_it_visu_creation(get_the_post_it_visu(pp))=TRUE &
      post_it_mouse_creation(get_the_mouse(pp))=TRUE &
```

The different elements manipulated by the operation (mouse, a window, a window with mouse interaction) must be already created.

```
      post_it_window_status(get_the_post_it_visu(pp))=displayed
&
      post_it_mouse_state(get_the_mouse(pp))= down &
```

The mouse must be in a state down and the window must be displayed.

```
      x_post_it_position(get_the_post_it_visu(pp))+5 :
         1..max_mouse_position_wide &
      y_post_it_position(get_the_post_it_visu(pp))+5 :
         1..max_mouse_position_high &
      x_post_it_window(get_the_post_it_visu(pp))-5 :
         1..max_mouse_position_wide &
```

The coordinates delimiting the moving zone of the window must belong to the screen i.e. must appear in the screen.

```
      x_post_it_mouse_position(get_the_mouse(pp)):
         x_post_it_position(get_the_post_it_visu(pp))+5..
         x_post_it_window(get_the_post_it_visu(pp))-5  &
      y_post_it_mouse_position(get_the_mouse(pp)):
         y_post_it_position(get_the_post_it_visu(pp))..
         y_post_it_position(get_the_post_it_visu(pp))+5
```

The mouse position must be in the moving zone delimited previously.

```
   THEN
      move_window_position(get_the_post_it_visu(pp),aa,bb)  ||
      Move_mouse_with_drag(get_the_mouse(pp),aa, bb)
```

The window is moved by calling the move_window_position action present in the POSTIT_VISU_WITHOUT_INT and the mouse is also moved bt the action move_mouse_with_drag of the machine POSTIT_MOUSE. Notice, that this action is one of the actions which combines the mouse toolkit and the window manager tool kit.

```
   END
```

The previous abstract machine gives supplementary preconditions on the objects. These preconditions imply the ones of the corresponding operations in the machines imported by extension. Therefore, the prover is capable to complete the proof of correctness. This is an important issue of the B method which shows that B is not only capable to structure the development of specifications, but it allows the structuration and the modularization of the proofs. Finally, this mechanism of weakest precondition conserves the coherence of the whole specification.

This previous machine has shown that it is not possible to :

- move a window with a mouse whose position is not in the moving zone,
- move a window if there is no mouse,
- move a window if it is eliminated. Note that this property is inherited form the imported machine.
- …

5.4 Complementary machines: POSTIT_EMISS and POSTIT_USERS

Two other machines are necessary to have a complete development of the specifications.

- the POSTIT_EMISS abstract machine defines a simple protocol allowing to emit, receive and create messages. These messages are intended to be linked to a Post-It window in the final application. In this machine, the messages have the state emitted, received, non_emitted and so on. Operations like emit_post, receive_post and create_post are described in this machine.
- the POSTIT_USERS abstract machine is devoted to the management of the users. Each user is defined by an icon of constant dimension, which can be moved on the screen.

Notice that we have restricted these two machine to simple operations. The management of the protocol of emission and users is not a main part of this application.

5.5 The final application: POSTIT_FINAL

After having defined all the different components of our application, extracted from figure 2, we are capable to describe the whole application. This application is an extension of all the machines described above. It defines the set of Post-Its and variables –selectors– which extract the objects coming from each sub-machine, i.e., extended machines.

```
MACHINE POSTIT_FINAL

EXTENDS
   POSTIT_EMISS, POSTIT_VISU_WITH_INT_MOUSE,
POSTIT_MESS,POSTIT_USERS
SETS
   POST
```

```
CONSTANTS
   max_post
PROPERTIES
   max_post=max_post_emiss &
   max_post=max_post_visu  &
   max_post=max_post_mess
VARIABLES
   the_post_it,
   get_the_post_it_emiss,
   get_the_post_it_visu_with_mouse,
   get_the_post_it_mess,
   post_it_creation
```

The set the_post_it is the set of all the effectively created Post-Its. The other functions are used to access the different components of the objects. We omit describing the invariants of this machine.

Let us describe the use of the emission of a Post-It. The user creates a Post-It, drag it on the icon corresponding to the user and then it is emitted.

```
OPERATIONS
   pp <-- create_post_it (pp_emiss, pp_visu,pp_mess )=
      ...
   emit_post_it(receiver,icon_receiver,sender,pp)=
   PRE
      receiver : NAT1 &
      sender : NAT1 &
```

The sender and the receiver are valid. They are defined as a positive integer.

```
      icon_receiver : the_post_it_users &
      post_it_user_creation(icon_receiver)=TRUE &
```

The icon corresponding to the receiver must be created. This precondition invokes an operation of the POSTIT_USERS abstract machine.

```
      x_post_it_user_position(icon_receiver):
1..max_post_it_wide &
      y_post_it_user_position(icon_receiver):
1..max_post_it_high &
```

The icon is effectively visible in the screen i.e. it is not outside the limits of the screen.

```
      pp : the_post_it &
      the_post_it /= POST &
      post_it_creation(pp)=TRUE  &
```

The Post-It, the corresponding mouse, the corresponding window associated to the mouse and the corresponding window without mouse are effectively created.

```
      post_it_mouse_creation(get_the_mouse(
      get_the_post_it_visu_with_mouse(pp)))=TRUE &
      post_it_visu_with_mouse_creation(
      get_the_post_it_visu_with_mouse(pp))=TRUE &
      post_it_visu_creation(get_the_post_it_visu(
      get_the_post_it_visu_with_mouse(pp)))=TRUE &
      user_adr_sender(get_the_post_it_emiss(pp))=sender &
```

The user which sends the Post-It is the one who created this post-it.

```
      message_status(get_the_post_it_mess(pp))=unread &
```

The message contained in this Post-It must be unread.

```
      emission_status(get_the_post_it_emiss(pp))=non_emitted &
```

The status of the message to be sent is not emitted.

```
      block_state=block_open &
```

No Post-It can be sent if the block of Post-Its is not open.

```
      post_it_window_status(get_the_post_it_visu(
         get_the_post_it_visu_with_mouse (pp)))=displayed &
```

The current Post-It to be sent must be displayed.

```
      post_it_mouse_state(get_the_mouse(
         get_the_post_it_visu_with_mouse (pp)))=down &
```

The associated mouse must be in a down state.

```
      x_post_it_position(get_the_post_it_visu(
         get_the_post_it_visu_with_mouse (pp)))+5 :
            1..max_mouse_position_wide   &
      x_post_it_window(get_the_post_it_visu(
         get_the_post_it_visu_with_mouse (pp)))-5 :
            1..max_mouse_position_wide   &
      y_post_it_position(get_the_post_it_visu(
         get_the_post_it_visu_with_mouse (pp)))+5 :
            1..max_mouse_position_high    &
      x_post_it_mouse_position(get_the_mouse(
         get_the_post_it_visu_with_mouse (pp))) :
            x_post_it_position(get_the_post_it_visu(
         get_the_post_it_visu_with_mouse (pp)))+5
            x_post_it_window(get_the_post_it_visu(
         get_the_post_it_visu_with_mouse (pp)))-5    &
      y_post_it_mouse_position(get_the_mouse(
         get_the_post_it_visu_with_mouse (pp))):
            y_post_it_position(get_the_post_it_visu(
         get_the_post_it_visu_with_mouse (pp)))
            y_post_it_position(get_the_post_it_visu(
         get_the_post_it_visu_with_mouse (pp)))+5
```

The positions of the Post-It and of the mouse must be in the screen. Moreover, the moving zone of the Post-It must be visible in the screen.

```
   THEN
   ANY
      pp_emiss
   WHERE
      pp_emiss= get_the_post_it_emiss(pp)
   THEN
      emit_post(pp_emiss, receiver,sender) ||
      eliminate(get_the_post_it_visu_with_mouse(
         get_the_post_it_visu(pp)))||
      move_window_with_mouse(
         get_the_post_it_visu_with_mouse (pp),
         x_post_it_user_position(icon_receiver),
         y_post_it_user_position(icon_receiver)
         )
   END
   END
   ;
```

The Post-It is emitted and the window is moved using the move to the position of the receiver identified by the icon_receiver user. The message becomes emitted, the window Post-It is eliminated and the mouse stays on the position of the receiver.

All the other operations have been described in this machine. The complete case study is available on our web site. We cannot insert it in whole in the present paper.

5.6 Proofs

For all the development, the proof obligations have been generated. They all have been automatically proved. However, this specification has not been built at the first attempt. We had to enrich the preconditions and to remove other preconditions. Indeed, the prover behaves following:

- preconditions are not complete, therefore the proof cannot be achieved,
- preconditions are contradictory, then the user has to make new choices and to check the requirements.

Finally, about 160 proof obligations are generated for this application. We had to prove only 4 proof obligations using the interactive prover, i.e., "by hand". This shows that when the application is well specified following sound software engineering concepts, the proof phase can be considerably reduced.

6. Conclusion and future work

In this paper, we expose the use of the B method in the area of HCI. We relate it to other approaches in this area: it can be considered as a third approach between the definition of ad hoc new formalisms and the use of well defined formal notations through HCI models. It consists in giving a model-independent method that can be customized to specific HCI models.

The great strength of the B method is its ability to refinement. That is the possibility to finally derive concrete programs form abstract specifications. While doing that, one part of the main objective of formal method can be achieved: it is to say the *verification* that implementation satisfies its specification. This work has shown, that interaction properties can be expressed *a priori* at any level of the chosen architecture model —ARCH in our example. We show in this work that interactive specific properties, like visualization or dialogue properties can be expressed, and, so doing, ensured. For example, ensuring that a Post-It® cannot leave its visualization screen is possible. This is made via formal proofs established by the use of tools.

This work exposes preliminary results. We do not claim that the B method is able to solve every problem we may encouteer in DSV-IS. Nevertheless, it seems to be a promising way of investigations. We have now to work in three directions: first, defining a specific method to help designers to build interactive applications is essential; second, and probably linked to the first point, it seems important to relate our process to HCI models, such as these we noticed upper. Correlations with other formalisms should be very important too, in order to help to *validate* our B-specification against users needs. This step, which is obviously out of the scope of formal method *senso strictu*, supposes translations between B-formal semantics and understandable users' needs representation. Last, there is a strong need for building reusable abstract machines once for all in the HCI area. The possibility to reuse these machines would considerably reduce testing and verifying costs.

7. References

[Abrial 1996] Abrial J.-R. *The B Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[Accot, Chatty, & Palanque 1996] Accot J., Chatty S., & Palanque P. A formal description of low level interaction and its application to multimodal interactive systems. *Third International Eurographics Workshop on Design, Specification, and Verification of Interactive Systems, Namur, Belgium*, 5-7 June 1996. p. 92-104.

[Bjorner 1987] Bjorner D. VDM a Formal Method at Work. *VDM Europe Symposium'87*, 1987.

[Campos & Harrison 1997] Campos J.C. & Harrison M.D. Formally Verifying Interactive Systems: A Review. *Eurographics Workshop on Design, Specification, Verification of Interactive Systems, Granada, Spain*, June 4-6 1997. p. 109-124.

[Card, Moran, & Newell 1983] Card S., Moran T., & Newell A. *The psychology of Human-Computer Interaction*. Lawrence Erlbaum Associates, 1983.

[Dijkstra 1976] Dijkstra E. *A Discipline of Programming*. Englewood Cliff (NJ), USA : Prentice Hall, 1976.

[Duke & Harrison 1993a] Duke D.J. & Harrison M.D. Abstract *Interaction Objects*. Computer Graphics Forum. 1993a. vol. 12,n° 3, p. 25-36.

[Duke & Harrison 1993b] Duke D.J. & Harrison M.D. *Towards a Theory of Interactors*. Amodeus Esprit Basic Research Project 7040, 1993b System Modelling/WP6.

[Fields, Merriam, & Andy 1997] Fields B., Merriam N., & Andy D. DMVIS: Design, Modelling and Validation of Interactive Systems. *Eurographics Workshop on Design, Specification, Verification of Interactive Systems, Granada, Spain*, June 4-6 1997. p. 29-44.

[Gamboa Rodrígez & Scapin 1997] Gamboa Rodrígez F. & Scapin D.L. Editing MAD* task description for specifying user interfaces, at both semantic and presentation levels. *Eurographics Workshop on Design, Specification and Verification of Interactive Systems (DSV-IS'97), Granada, Spain,* June 4-6 1997.

[Gray, England, & McGowan 1994] Gray P., England D., & McGowan S. *XUAN: Enhancing the UAN to capture temporal relation among actions*. Department of Computing Science, University of Glasgow, February 1994. Department research report IS-94-02.

[Hix & Hartson 1993] Hix D. & Hartson H.R. *Developping user interfaces: Ensuring usability through product & process*. Newyork, USA : John Wiley & Sons, inc., 1993.

[Hoare 1969] Hoare C.A.R. *An Axiomatic Basis for Computer Programming*. CACM. 1969. vol. 12,n° 10, p. 576-583.

[Hoare et al. 1987] Hoare C.A.R., Hayes I.J., Jifeng H., Morgan C.C., Sanders A.W., Sorensen I.H., Spivey J.M., & Sufrin B.A. *Laws of Programming*. CACM. 1987. vol. 30,n° 8.

[Johnson 1995] Johnson C.W. Using Z to support the design of interactive, safety-critical systems. *IEE/BCS Software Engineering Journal*, March 1995. vol. 10, n° 2, p. 49-60.

[Lano 1996] Lano K. *The B Language Method: A guide to practical Formal Development*. Springer, 1996.

[McGowan 1995] McGowan S. *From UAN to eXUAN: Specifying simulations of the temporal properties of interaction*. University of Glasgow, Department of Computing Science, March 1995. Technical report TR-1995-5.

[Palanque 1992] Palanque P. *Modélisation par Objets Coopératifs Interactifs d'interfaces homme-machine dirigées par l'utilisateur*. PhD : Toulouse I, 1992.

[Palanque & Girard 1996] Palanque P. & Girard P. *Groupe de travail GP3-FLASHI (Formalismes et Langages Appliqués aux Systèmes Hautement Interactifs)*. GDR-PRC Communication Homme-Machine, 1996. Rapport d'activité.

[Paterno' 1994] Paterno' F. *A Theory of User-Interaction Objects*. Journal of Visual Languages and Computing. 1994. vol. 5,n° 3, p. 227-249.

[Paterno' & Faconti 1992] Paterno' F. & Faconti G.P. *On the LOTOS use to describe graphical interaction*. Cambridge University Press, 1992. p. 155-173.

[Scapin & Pierret-Golbreich 1990] Scapin D.L. & Pierret-Golbreich C. *Towards a method for task description : MAD*. Work with display units 89, Elsevier Science Publishers, North-Holland, 1990.

[Spivey 1988] Spivey J.M. *The Z notation: A Reference Manual*. Prentice Hall, 1988.

[Steria méditerranée 1997] Steria méditerranée. *Atelier B version 3.0*. 1997.

[UIMS 1992] UIMS. The UIMS Workshop Tool Developers: A Metamodel for the Runtime Architecture of an Interactive System. *SIGCHI Bulletin*, 1992. vol. 24, n° 1.