FORMAL VERIFICATION AND VALIDATION OF INTERACTIVE SYSTEMS SPECIFICATIONS

From Informal Specitications to Formal Validation

Yamine AÏT-AMEUR [1], Benoit BREHOLÉE [2], Patrick GIRARD [1],
Laurent GUITTET [1] and Francis JAMBON [3]
1) LISI / ENSMA, BP 40109, Téléport 2, 86961 Futuroscope cedex, France
2) ONERA-CERT-DTIM, 2 Avenue Edouard Belin, BP 4025, 31055 Toulouse cedex, France
3) CLIPS-IMAG, BP 53, 291 avenue de la bibliothèque, 38041 Grenoble cedex 9, France

E-mail: {yamine, girard, guittet}@ensma.fr, breholee@cert.fr, francis.jambon@imag.fr

Abstract: This paper proposes a development process for interactive systems based both on verification and validation methods. Our approach is formal and use at first the B Method. We show in this paper how formal B specifications can be derived from informal requirements in the informal notation UAN. Then, these B specifications are validated using the data oriented specification language EXPRESS. Several scenarios can be tested against these EXPRESS specifications.

Key words: B Method, EXPRESS, UAN, interaction properties, verification, validation, formal specification of interactive systems.

1. INTRODUCTION

Graphical user interfaces relying mostly on software, are being more and more used for safety-critical interactive systems –for example aircraft glass cockpits– the failure of which can cause injury or death to human beings. Consequently, as well as hardware, the software of these interactive systems needs a high level of dependability. Besides, on the one hand, the design process must insure the reliability of the system features in order to prevent disastrous breakdowns. On the other hand, the usability of the interactive system must be carefully carried out to avoid user misunderstanding that can trigger similar disastrous effects. So, the software dependability of these safety-critical interactive systems rely as well on safety as on usability properties. Our work focuses on the use of formal techniques in order to increase the quality of HCI software and of all the processes resulting from the development, verification, design and validation activities.

In past workshops and conferences, we presented our approach through papers dealing with formal specifications of HCI software [4], formal verification of HCI software [5], test based validation of existing applications [19]**,** This paper addresses another topic not tackled yet by our approach: design and formal validation of formal specifications with respect to informal requirements. This work completes the whole development process of a HCI software. Indeed, our approach uses the B formal technique for representing, verifying and refining specifications [4, 5, 19], test based validation of existing applications [19], secure code generation [18] and integration of formal approaches [12].

This paper starts from the translation of the requirements in the UAN notation [15] and shows how B specifications can be derived from. Then, the EXPRESS formal data modeling language [11] is put into practice for the validation of the derived B specifications. We show how the B specifications can be translated to EXPRESS code which allows validation.

This paper is structured as follows. Section 2 reviews the different notations and formal techniques that have been experienced on HCI. Next section gives the informal requirements of a case study and its representation in the UAN notation. Section 4 presents the B technique and the specifications of the case study in B. Section 5 is related to validation. It presents the formal data modeling technique EXPRESS which allows the validation of the B specifications. We show how an automatic translation from B to EXPRESS can be performed and how this technique is applied to our case study. The result is a set of EXPRESS entities that are checked against various scenarios. Last, we conclude on the whole suggested approach.

2. NOTATIONS AND TECHNIQUES IN HCI: A BRIEF STATE OF THE ART

2.1 Notations & Formal techniques

In order to express HCI software requirements, several notations were suggested. As examples, MAD (for “Méthode Analytique de Description”)

[27] and HTA (for Hierarchical Task Analysis) [29] use a hierarchical decomposition of user tasks. On the other side, a notation like UAN [15] and its extension XUAN [13] allow the description of not only the interface feedback, but of the user behaviors as well. UAN specifications record the state of the interface and tasks are described as state evolutions. This state orientation of UAN facilitates translation to state based formal techniques –B for example.

Several techniques were used in the HCI area. These techniques differ from some point of views: semantics –algebraic or state based– verification –incremental proof or fully automatic proof– etc. Some of these techniques can be summarized in the following.

On the one hand, the first techniques are state based. They were based on automata through statecharts [31] and ATN [30] [14], Petri Nets [2] [23]. They have been extended to support temporal logics to allow automatic model checking like in CTL* [25], XTL [7] and SMV [8] [22], or with the Lustre language [26]. The previous techniques support code generation and automatic proving. Other techniques supporting state based semantics and incremental proving and refinement like Z [20], VDM [21] or B [5] were suggested.

On the second hand algebraic techniques have been applied with LOTOS [24, 25] for describing HCI software. The proofs are achieved by rewriting and refinement is performed by transformation. Other techniques based on higher order type systems have been experienced.

All these techniques cover a limited part of the development of an HCI. Our approach does not use only one technique, but it suggests to use several techniques which cooperate, choosing each technique where it has proved to be most efficient.

2.2 Our approach

Our approach uses the B technique. B supports formal specifications, refinement from specifications to code and property verification through the proof of the generated proof obligations. Specifications are derived from the informal UAN notation and are validated using the EXPRESS data modeling language.

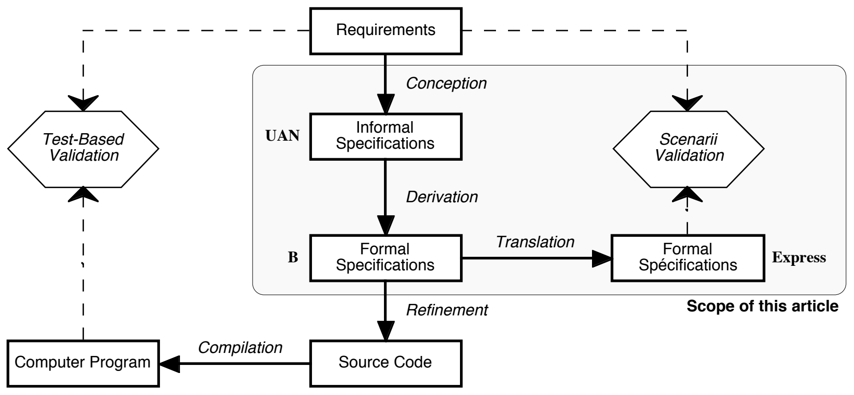

Figure 1: Scope of this article in the approach we suggest for handling the development and validation of HCI

Formal specifications, property verification and refinement from specification to code have been have been presented in [4, 5, 19] respectively. This paper presents the last point: deriving specifications from semi-formal notations and their validation in EXPRESS. This paper completes the whole developed approach described in figure 1.

3. CASE STUDY AND THE USER ACTION NOTATION

3.1 The case study: the *Rangeslider*

An usual slider –with a single cursor– is a graphical toolkit widget used by interface designers to allow the specification of a value in an interval. The *Rangeslider* [3] used by Spotfire™ (http://www.spotfire.com) is an enhanced version of this classical slider, i.e., it supplies two cursors –see fig. 2– in order to allow users to select not only a single value, but a range of values. This new widget is used by interface designers to implement easy-to-use zoom or filtering functions. A *Rangerslider* user can interact with the widget by the way of three different kinds of actions:

– **Move one cursor:** the user moves one of the two cursors, to the left or to the right. As a consequence, the area of the center zone expands or reduces. The moved cursor cannot cover over the other cursor nor exceed the widget length.

- **Move the center zone:** the user moves the center zone, and at the same time both cursors come after it. So the area of the center zone remains unchanged. No cursor can exceed the widget length.
- **Select a value in outer zones:** the user clicks in one of the outer zones –MinZone or MaxZone– and the closest cursor moves at the selected point. As a consequence, the area of the center zone expands or reduces.

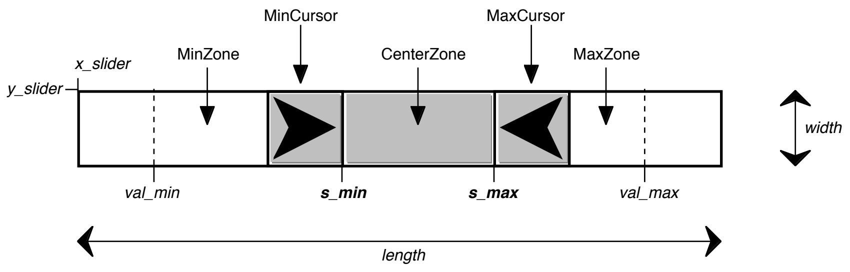

Figure 2: RangeSlider scheme —with variables names used both in the UAN, B and EXPRESS specifications.

3.2 The User Action Notation

The User Action Notation is an interaction-design notation. Hix et al. suggest that *"the UAN is intended to be written primarily by someone designing the interaction component of an interface, and to be read by all developers, particularly those designing and implementing the user interface software"* [15]. The UAN is user- and task-oriented. A UAN specification describes, at the physical level, the user actions and their corresponding interface feedback and state changes.

A typical UAN specification in a three-columns table which must be read from left to right and from top to bottom. The first column is dedicated to the user actions –mouse-clicks, keystrokes, etc.– the second one to the interface feedback –icon highlighting, widget display, etc.– and the third one to the interface state –name of the selected icon, position of a slider, etc. User actions as well as interface feedback have a specific representation.

User actions represent user interactions with devices as mouse or keyboard. Among the UAN user actions, we use in §3.3 the following:

– **[X]**	context of the object X
– **~[X]**	move cursor in the context of object X
– **~[x,y in X]**	move cursor to (arbitrary) point within object X
– **Mv**	depress the mouse button
– **M^**	release the mouse button
– *****	the action can be repeated 0 or more times

Interface feedback represents the system actions that are observable. Among the UAN interface feedback, we use:

– **X !**	highlight object X
– **X -!**	unhighlight object X
– **X > ~**	object X follows (in dragged by) cursor
– **display(X)**	display object X
– **erase(X)**	erase object X
– **redisplay(X)**	equivalent to erase(icon) and then display(icon)
– **@x,y**	at point x,y (e.g. to display X)
– **condition : action**	the *action* is triggered if *condition* is true
– **(…)**	grouping mechanism

In the example table 1 below, in order to select a file, the user moves the mouse pointer to the file icon "`file🗎`" and depresses the mouse button. Then the file icon is highlighted and all other icons are unhighlighted. The interface state –the name of the selected file– is updated when the user depresses the mouse button.

Table 1: UAN specification of the task "select icon"

TASK: **select file**		
USER ACTIONS	**INTERFACE FEEDBACK**	**INTERFACE STATE**

`~[file🗎] M^`	`file🗎 !`
	`▯ file🗎' ≠ file🗎 : file🗎' -!`	

`Mv`		`selected = file`

3.3 Specification of the Range-Slider in the UAN

The three tables below –table 2 to table 4– are the UAN specifications of the three user interactions described in §3.1. In fact, a full UAN specification must comprise five tables –one table for each user interaction. However the two pairs of UAN tables for cursor and outer zones are so similar that one table of each pair has been omitted. In these tables, *Rangeslider* is the name of the whole slider object and Δx is the spatial increment on the abscissa axis.

3.3.1 Move one cursor (MinCursor)

In order to move the left cursor –MinCursor– the user must move the mouse button in the context of the MinCursor object. Then, he can depress the mouse button and drag the cursor. The MinCursor follows the mouse pointer and the center zone must be redisplayed. At each increment, the value of the `s_min` variable is updated.

Table 2: UAN specification of the task "move MinCursor"

TASK: **move MinCursor**		
USER ACTIONS	**INTERFACE FEEDBACK**	**INTERFACE STATE**
~[MinCursor] Mv	MinCursor !	
~[x,y in RangeSlider]*	0<x<s_max : MinCursor > ~ redisplay(CenterZone)	s_min=s_min+Δx
M^	MinCursor -!	

3.3.2 Move the center zone (CenterZone)

In order to move the center zone –CenterZone– the user must move the mouse button in the context of the CenterZone object. Then, he can depress the mouse button and drag the zone. The zone follows the mouse pointer and both cursors must be redisplayed. At each increment, the value of the `s_min` and `s_max` variables are updated.

Table 3: UAN specification of the task "move CenterZone"

TASK: **move CenterZone**		
USER ACTIONS	**INTERFACE FEEDBACK**	**INTERFACE STATE**
~[CenterZone] Mv	CenterZone !	

~[x,y in RangeSlider]*	s_min<x<s_max :
	CenterZone > ~
	redisplay(MinCursor)
	
	redisplay(MaxCursor)
		s_min=s_min+Δx
		s_max=s_max+Δx

M^	CenterZone -!	

3.3.3 Select a value in an outer zone (MinZone)

In order to select a value in an outer zone –MinZone– the user must move the mouse button in the context of the MinZone object. Then, he depresses the mouse button. At this point, the left cursor –MinCursor– as well as the center zone must be redisplayed at the new position.

Table 4: UAN specification of the task "select a value in MinZone"

TASK: **select value in MinZone**

USER ACTIONS	**INTERFACE FEEDBACK**	**INTERFACE STATE**
`~[x,y in MinZone] Mv^`	`redisplay(MinCursor)@x,y` `redisplay(CenterZone)`	`s_min=s_min+Δx`

These UAN tables, together with the figure 2 are the high level and informal specifications of the Rangeslider. These specifications are very useful for interface designers because they express in a rather short and precise way the behavior of the RangeSlider. However, the UAN is a notation to express requirements but cannot be used to prove or test the features of the interaction object we analyze. As an example, we cannot be sure that the cursor will never cover over the other cursor nor exceed the widget length. So we must now use formal methods to prove this kind of properties.

4. THE B TECHNIQUE AND FORMAL SPECIFICATIONS

4.1 The B formal technique

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 invariant 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 post-conditions [16, 17]. 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 [10]. Starting from this method, J.R. Abrial [1] 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.

4.2 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 [1]. Briefly, these clauses mean:

– 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. It represents the variables of the model of the specification. 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.
– 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.

4.3 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.

- **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.
- **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 interactively 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 our case study the 38 of the 40 generated proof obligations have been proved automatically using the "Atelier B" tool [9].
- **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. The refinement makes it possible to introduce design and implementation details while refining. Finally, refinement allows not only to derive code but also to make the proofs and the proving process easier thanks to incremental design decisions introductions.

4.4 Deriving the B range slider specification from the UAN notation

The definition of the UAN specification helps to derive a formal B specification since it encodes the notions of state and of operations.

The following abstract machine describes what a set of range sliders is. It describes the set of all the sliders to be `SLIDERS` and two constants describing the length and the width of the screen. The `PROPERTIES` clause types these two constants and gives their corresponding values.

```
MACHINE the_slider

SETS
  SLIDERS
```

```
CONSTANTS
  screen_width,
  screen_height

PROPERTIES
  screen_width : NAT  ∧
  screen_width = 800 ∧
  screen_height : NAT ∧
  screen_height = 600
```

The model of this abstract machine is given by the attributes defined in the VARIABLES clause. The set `sliders` describes the set of the actually described range sliders. The other variables allow to access the attributes of a given range slider.

Informally, as described in figure 2, each range slider is characterized by:
– `x_slider` and `y_slider` are the coordinates of the up left corner of the window describing the range slider,
– `width` and `length` are respectively the width and the length of a given range slider,
– `val_min` and `val_max` are the minimal and maximal values associated to a range slider,
– and finally, `s_min` and `s_max` are the current low and up values of the described range slider.

```
VARIABLES
  sliders, x_slider, y_slider, width, length, val_min, val_max,
s_min, s_max
```

All these variables are typed in the `INVARIANT` clause. This clause contains the properties that are always satisfied by the variables of the model. These properties shall be maintained by the operations that affect these variables. Two kinds of properties are described:
– typing properties that give types to the variables. The set `sliders` is declared as a subset of the set `SLIDERS`. Then, all the other variables are accessing functions and they are typed by their signature,
– safety properties which ensure a set of critical properties and model consistance. They are described in first order logic and are maintained by the B prover. They assert that the low (resp. Up) value of a slider shall be greater (resp. Lower) or equal to the minimal (resp. maximal) value of the range slider. Moreover, it states that the whole range slider is contained in the screen dimensions. This last assertion ensures visibility and reachability properties.

In the B language, these properties are described by:

```
INVARIANT
sliders □ SLIDERS □
  x_slider  □ sliders-->NAT □
  y_slider  □ sliders-->NAT □
  width     □ sliders-->NAT □
  length  □ sliders-->NAT □
  val_min   □ sliders-->NAT □
  val_max   □ sliders-->NAT □
  s_min     □ sliders-->NAT □
  s_max     □ sliders-->NAT □
/* Safety properties of the slider */
  □ sl.(sl:sliders =>(val_min(sl) >= 0)) □
  □ sl.(sl:sliders =>(val_min(sl) <= s_min(sl))) □
  □ sl.(sl:sliders =>(s_min(sl)<s_max(sl))) □
  □ sl.(sl:sliders =>(s_max(sl) <= val_max(sl))) □
  □ sl.(sl:sliders =>(val_max(sl) <= length(sl))) □
  □ sl.(sl:sliders
     =>(x_slider(sl) □ 1..screen_width)) □
  □ sl.(sl:sliders
     =>(y_slider(sl) □ 1..screen_height)) □
  □ sl.(sl:sliders
       =>(x_slider(sl)+length(sl) □ 1..screen_width)) □
  □ sl.(sl:sliders
     =>(y_slider(sl)+width(sl) □ 1..screen_height))
```

The initialization clause initializes all the variables of the model to the empty set. One can be astonished why functions are initialized to the empty set. In B, accessing functions are considered as subsets of Cartesian products, therefore, they can be initialized to an empty set.

```
INITIALISATION
  sliders := {} ||
  x_slider := {} ||
  y_slider := {} ||
  length := {} ||
  width := {} ||
  val_min := {} ||
  val_max := {} ||
  s_min := {} ||
  s_max := {}
```

The first operation allows to create a range slider with `XX`, `YY` as coordinates of its left up corner. Its length and width are respectively given by the parameters `LENGTH` and `WIDTH`. Finally, `VMIN` and `VMAX` parameters indicates the minimal and maximal values of the range. The slider is created with `VMIN` and `VMAX` as initial minimal and maximal values. A preconditon ensures that the parameters are correctly typed and the invariant is maintained. It ensures that the creation of a range slider is correctly performed.

```
OPERATIONS
  create(XX,YY,LENGTH,WIDTH,VMIN,VMAX)=
  PRE
     sliders ≠ SLIDERS □
     XX □ NAT □ YY □ NAT □
     WIDTH □ NAT □ LENGTH □ NAT □
     VMIN □ NAT □ VMAX □ NAT □
     VMIN >= 0 □
     VMIN < VMAX □
     VMAX <= LENGTH □
     XX □ 1..screen_width □
     YY □ 1..screen_height □
     XX+LENGTH □ 1..screen_width □
     YY+WIDTH □ 1..screen_height
  THEN
     ANY
        sl
     WHERE
        sl □ SLIDERS - sliders
     THEN
        sliders := sliders □ {sl}   ||
        x_slider(sl):=XX          ||
        y_slider(sl):=YY          ||
        length(sl):=LENGTH        ||
        width(sl):=WIDTH          ||
        val_min(sl):=VMIN         ||
        val_max(sl):=VMAX         ||
        s_min(sl):=VMIN           ||
        s_max(sl):=VMAX
     END
  END;
```

In order to keep this paper in a reasonable length, we show only one operation that manipulates the range slider. It allows to move the left value of the range slider to the left. In B this operation is described by:

```
  move_left_slider(one_slider, new_left_min_value)=
  PRE
     one_slider □ sliders □
     new_left_min_value □ NAT □
     new_left_min_value > val_min(one_slider) □
     new_left_min_value < s_max(one_slider)
  THEN
     s_min(one_slider) := new_left_min_value
  END;
....
END
```

Other operations related to the range slider have been described in this abstract machine. Moreover, the whole application is represented by several abstract machines not presented in this paper. Indeed, abstract machines related to the mouse management, to the direct manipulation and so on have been described. Finally, notice that the abstract machine described in B and presented in this paper has shown that it is possible to:

– ensure that a range slider remains in the screen limits,
– ensure that the low and up values of a range slider respect the definition of a range,
– move the low value, of a range slider to the left in order to decrease its left value, by running the corresponding operation.

For the whole developed abstract machine, 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 40 proof obligations are generated for this application. We had to prove only 2 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.

All these properties are safety properties. In the next section we address the problem of the validation of such formal specifications that is not supported by the B formal technique.

5. THE EXPRESS TECHNIQUE AND VALIDATION OF SPECIFICATIONS

Describing data models is a major concern of the data management and knowledge management areas. Several formalisms, models and techniques allow to represent data and/or knowledge. For example, OMT, UML and so on are used to represent information systems while KIF, KADS and so on are techniques for knowledge representation. The definition of such formalisms for representing information systems requires a set of concepts usually needed. These concepts are related to structure, description and behavior. Structure defines the organization of the data in the information system. For example classification or object orientation or relational databases are ways of structuring information.

Description is related to the properties of the structured information. It is defined by the set of the properties that a given description has. Attributes in classes or in relational tables are ways to describe different classes or tables. Behavior gives the information on how the data behave. Behavior is obtained either by giving a function which shows how data evolve (constructive approach) or by giving constraints on the data to restrict the behavior to the licit data.

These three concepts require the definition of a language or any other formalism which allows to handle them. It is highly suitable that this language is processable. EXPRESS is a formal data modeling language that handles these three kinds of knowledge. A data model in EXPRESS is represented by a set of schemas that may refer to each other. Each schema contains two parts. The first part is a set of entities that are structured in an object oriented approach with multiple inheritance. The second part is a procedural part which contains procedures, functions and global rules.

5.1 The EXPRESS data modeling language

The EXPRESS language has been designed in the context of the STEP (STandard for the Exchange of Product model and data) international project which aims at describing formal models for exchanging product data. Moreover, this language can be used for the specification of several applications in computer science areas.

This section focuses on the constructions we use in the reminder of the paper. More details about the definition of this language can be found in [6, 28]**.** The EXPRESS language focuses on the definition of entities –types– which represent the objects –classes– we want to describe. EXPRESS is type oriented: entity types are defined at compile time and there is no concept of meta-class. Each entity is described by a set of characteristics called attributes. These attributes are characterized by a domain and constraints on these domains. An important aspect of these entities is that they are hierarchically structured allowing multiple inheritance as in several object oriented languages. This part of the specification describes the structural and the descriptive parts of the domain knowledge.

On the other hand, it is possible to describe processes on the data defined in the entities by introducing functions and procedures. These functions are used to define constraints, pre-conditions and post-conditions on the data. They are also used to specify how the values of some properties that may be derived from the values of other properties. This part of the specification describes the procedural part of the domain knowledge.

Finally, in EXPRESS, entities –data– and functions –processes– are embedded in a structure called a SCHEMA. These schemes may reference each other allowing a kind of modularity and therefore specification in the large possibilities.

5.1.1 Entity definition

In EXPRESS, entities are defined in terms of attributes. Indeed, a set of attributes (may be an empty set) is assigned to each entity. Each attribute has a domain (where it takes its values) defining a data type. It can be either a simple domain –integer, string– or a structured domain –lists, sets, bags– (hard encoded in EXPRESS) or an entity type meaning that an attribute is of type of another entity. We syntactically write:

```
SCHEMA foo1;
ENTITY A;                     ENTITY B;
att_A :INTEGER;                 att_1:REAL;
INVERSE                           att_2:LIST [0:?] OF STRING;
att_I□: B FOR att_3□;             att_3:A;
END_ENTITY;                     END_ENTITY;
END_SCHEMA;
```

Informally, the entity *B* has three attributes: an integer, a list of string and a pointer to another entity *A* which has only one integer attribute. *att_I* is an inverse attribute of entity A corresponding to the inverse link defined by attribute *att_3* in entity B.

Semantically, an entity has a model. In the EXPRESS community, the model is named a physical file. The model consists of a set of entity instances with explicit instance identity. The attribute values are either literal values of the EXPRESS simple or structured built-in types or references to other entity instances. Instead of entering into deep semantic details, we give an example of a model (physical file) which can be associated to the previous entity definitions.

Let us consider a particular representation, named instance, of the entity B, where *att_1* evaluates to 4, *att_2* is the list ('hello', 'bye') and *att_3* points the particular instance of the entity A where its *att_A* attribute evaluates to 3. Then, the model (physical file) associated to these particular instances of the entities A and B is described by

```
# 1=A(3, #2) ;
# 2=B(4,('hello','bye') ,# 1);
```

5.1.2 5.1.2. Derived attributes

As it is the case for several object oriented languages, it is possible to have derived attributes in the entity definitions. For example, we can derive in entity *B2* (assumed to be defined in the same schema *Foo1*) a Boolean attribute stating whether or not the length of the *att_2* list is equal to the integer attribute *att_A* defined in entity A by writing:

```
ENTITY B2;
att_1:REAL□;
att_2:LIST[0:?] OF STRING;
att_3: A□;
DERIVE
  att_4 :BOOLEAN:= (SELF.att_3\A.att_A=SIZEOF(SELF.att_2))□;
END_ENTITY;
```

where:

- **SELF** is an EXPRESS keyword representing a variable which designates the current entity,
- **.** is the dot notation to access the attribute of an entity,
- \ character allows to access super-type,
- and **SIZEOF** is an EXPRESS built-in function which gives the length of an aggregate data type.

The derived (computed) attributes do not physically appear in the model (physical file).

5.1.3 5.1.3. Constraining entities

It is possible to limit the allowed population (elements) of the models to those instances that satisfy some stated constraints. They are introduced thanks to the WHERE clause of EXPRESS that provides for instance invariant, and to the global RULE clause that provides for model invariant.

Let us assume that the allowed values for *att_A* are [1..10] and that exactly two instances shall have an attribute value 1, we may write (QUERY is a built-in iterator on class):

```
ENTITY A□;                     RULE  Card FOR A□;
att_A: integer;                WHERE
WHERE                        SIZEOF( QUERY( inst<*A |
(SELF.att_A>=1) and            (inst.att_A=2))) =2
(SELF.att_A<=10) ;           END_RULE;
END_ENTITY;
```

Derivations and constraints are the only places where functions may occur. They provide the two high level abstraction mechanisms identified as necessary in data driven active databases. Therefore, it is possible to formally specify a global class of problems. Moreover, derivations and constraints are inherited. These features define a set inclusion semantics to EXPRESS inheritance mechanism.

5.1.4 Functions and procedures

As we have seen previously on examples, functions can be used to associate rules to data. These rules may be either derivation or (local or global) constraints. Syntactically, a function is declared in EXPRESS as:

```
FUNCTION F (x : typ_1; y : typ_2) :typ_3;
( *Function_Body;*)
End_Function;
```

The previous declaration has only presented a function interface. It has two parameters of types *typ_1* and *typ_2*. The result is of type *typ_3*. We note that these types can be either built-in EXPRESS types, or defined types or defined entities.

function_body represents the body of the defined function. Assignment, sequence and control structures (if statement loops and recursion) can be used in this function body. These features give powerful expression possibilities to the language. Indeed, we get the same expression possibility as other recursive specification languages.

5.2 Translation of B specifications to EXPRESS

The translation from B specifications to EXPRESS code is based on the semantics of generalized substitutions on which B is built. The idea consists in:

– representing the state variables of the model by an EXPRESS entity. This entity describes a state in the underlying transition system. According to the B semantics, this transition system describes the semantic model of the developed application,
– representing the invariant properties by global EXPRESS rules. Indeed, the properties that are described in the `INVARIANT` B clause are global properties that need to be satisfied at each state,
– and finally, representing operations by entities expressing the initial and the final states with local rules that express the relationship between the initial and the final states.

All the objects that are defined in an abstract machine are translated into EXPRESS. Each abstract machine corresponds to one EXPRESS schema.

5.3 The case study in EXPRESS

The following EXPRESS entity defines the model associated to the abstract machine described in B. It is obtained by a translation of all the variables that are described in the `VARIABLES` B clause.

```
SCHEMA The_Slider;

ENTITY Slider;
  x_slider,y_slider  :INTEGER;
  width, length        :INTEGER;
  val_min, val_max  :INTEGER;
  s_min, s_max      :INTEGER;
END_ENTITY;
```

For a given range slider, the previous entity describes the `x_slider` and `y_slider` representing its coordinates, its width and length, its minimal and maximal values and finally its low and up values. The instanciation of this entity allows to create a range slider. This entity preserves the identifiers introduced in the B abstract machine. Moreover, it encodes all the invariant properties which are related to typing of the variables.

The other invariant clauses that are related to the universally quantified properties, which express safety properties, are represented by a global EXPRESS rule. This rule expresses that all the instances of the entity `slider` satisfy the expressed logical properties. It states that the set of all the instances of a range slider satisfying these properties is exactly the set of all instances of a range slider. It is given by:

```
RULE coord FOR (Slider);
LOCAL
  sliders_ok, ens_sliders :
  SET OF Slider := [] ;
END_LOCAL;
  sliders_ok := QUERY(s <* Slider | ((s.val_min >=0) AND
       (s.val_min <= s.s_min) AND
       (s.s_min < s.s_max) AND
       (s.s_max <= s.val_max) AND
       (s.val_max <= s.length) AND
       (s.x_slider >= 0) AND
       (s.x_slider + s.length < 800) AND
       (s.y_slider >= 0) AND
       (s.y_slider + s.width < 600)));
  ens_sliders := QUERY(s <* Slider | true);
WHERE
  sliders_ok = ens_sliders ;
END_RULE;
```

Finally, operations are also transformed into an EXPRESS entity. The translation principle is based on the semantics of B. Indeed, the entity slider expresses the state of the described system (state based formal semantics). So, an operation, acting on a state, transforms an initial state `Ei` to a final state `Ef`.

The operation `move_left_slider` considers two states: the initial state `Ei` and the final state `Ef` and its input parameter, namely `new_left_min_value`. The description of this entity is given by:

```
ENTITY Move_Left_Slider;
  -- states
  Ei, Ef : Slider ;
  -- input parameter
  new_left_min_value : INTEGER;
```

The next part completes the description of an operation by an entity. It translates the precondition part (expressed by the B keyword `PRE`), the effect of the operation by expressing the change of `s_min` in the final state and finally it states the unchanged attributes in final state. The result gives the following `WHERE` rules.

```
WHERE
-- Translation of preconditions
       pre1: new_left_min_value >= Ei.val_min ;
       pre2: new_left_min_value < Ei.s_max ;
-- Translation of perations
     ope1: Ef.s_min = new_left_min_value;
-- Translation of unchanged state variables
       cst1: Ef.x_slider = Ei.x_slider ;
       cst2: Ef.y_slider = Ei.y_slider ;
       cst3: Ef.width = Ei.width ;
       cst4: Ef.length = Ei.length ;
       cst5: Ef.val_min = Ei.val_min ;
       cst6: Ef.val_max = Ei.val_max ;
       cst7: Ef.s_max = Ei.s_max ;
END_ENTITY;
....
END_SCHEMA;
```

This approach shows that it is possible to automatically translate B specifications into EXPRESS data modeling specifications. This translation will allow to give data models that represent specification tests.

5.4 Validation scenarios

In order to describe tests of B specifications –recall that validation and test are not supported by B– we need to describe instantiations of the EXPRESS data model.

As an illustration consider two rangesliders that are described by the same coordinates (`x_slider` =20 and `y_slider`=30), the same length and width (equal to `length`=100 and `width` = 10), the same minimal and maximal values (equal to `val_min`= 40 and `val_max`= 80) and the same up value (equals to `s_max` = 60). Consider that the first range slider RS1 corresponding to the initial state has a low value (equals to `s_min`= 50) and the second range slider RS2 has a low value (equals to `s_min` = 45). In fact this situation corresponds to a moving of the left value of a range slider. It can be expressed as `move_left_slider(RS,45)`. Here we consider that the range sliders `RS1` and `RS2` corresponds respectively to the range sliders of the initial and final states. In EXPRESS, this situation corresponds to the description of the three following instances:

```
#1=SLIDER (20, 30, 10, 100, 40 , 80, 50 , 60)▯;
#2=SLIDER (20, 30, 10, 100, 40 , 80, 45 , 60)▯;
#3=MOVE_LEFT_SLIDER(#1, #2, 45)▯;
```

The previous set of instances represent a test case for the `move_left_slider` operation. The method can be generalized to other operations and to compositions of these operations that allow the description of a wide range of user scenarios. The test sequences can then be produced using the UAN specifications described in §3.3. Thanks to these specifications, a wide coverage can be achieved.

6. CONCLUSION

This paper shows a formal technique that allows to derive, verify and validate formal B specifications of HCI software. The informal requirements are expressed using the semi-formal notation UAN which is used as the basis for writing formal specifications. This process is proved helpful for writing formal specifications. Indeed, the direct derivation of these specifications from informal requirements is a hard task. This approach bridges the gap between user oriented specifications which feed the formalization process, the B formal development and verification techniques.

As a second step this paper addresses a crucial issue related to formal validation of formal specifications. It suggests to use a data oriented modeling language, namely EXPRESS, which allows to represent validation scenarios. This approach increases the efficiency of the HCI software development process since validation is not performed at the programming language level but at higher and abstract specifications. This approach allows to validate scenarios of application earlier in the development process. The result increases the efficiency of the development and decreases its cost.

Finally, to end the whole development process we suggest there is a need for taking into account user tasks descriptions and user tasks validations. This topic has not been addressed in this paper but it will be tackled in future developments. Indeed, we think that task representations and validations are possible within the framework we have developed.

REFERENCES

1. Abrial, J.-R. *The B Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.
2. Accott, J., Chatty, S. et Palanque, P. A formal description of low level interaction and its application to multimodal interactive systems. In *Proceedings of Eurographics Workshop on Design, Specification, and Verification of Interactive Systems (DSV-IS'96)* (5-7 June, Namur, Belgium), Springer-Verlag, 1996, pp. 92-104.
3. Ahlberg, C. et Truve, S. Tight Coupling: Guiding User Actions in a Direct Manipulation Retrieval System. In *Proceedings of HCI'95 Conference on People and Computers X* ,1995, pp. 305-321.
4. Aït-Ameur, Y., Girard, P. et Jambon, F. A Uniform approach for the Specification and Design of Interactive Systems: the B method. In *Proceedings of Eurographics Workshop on Design, Specification, and Verification of Interactive Systems (DSV-IS'98)* (3-5 June, Abingdon, UK), 1998, pp. 333-352.
5. Aït-Ameur, Y., Girard, P. et Jambon, F. Using the B formal approach for incremental specification design of interactive systems. In *Proceedings of Engineering for Human-Computer Interaction* (14-18 September, Kluwer Academic Publishers, 1998, pp. 91-108.
6. Bouazza, M. *Le langage EXPRESS*. Hermès, Paris, 1995.
7. Brun, P. XTL: a temporal logic for the formal development of interactive systems. Palanque, P. et Paternò, F. (Ed.). In *Formal Methods for Human-Computer Interaction*, Springer-Verlag, 1997, pp. 121-139.
8. Clarke, E.M., Emerson, E.A. et Sistla, A.P. Automatic Verification of Finite-State Concurrent Systems using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems*. 2, 8 (1986), pp. 244-263.
9. ClearSy. *Atelier B - version 3.5*. 1997.
10. Dijkstra, E. *A Discipline of Programming*. Prentice Hall, Englewood Cliff (NJ), USA, 1976.
11. EXPRESS. *The EXPRESS language reference manual*. ISO, 1994 ISO 10303-11.
12. Girard, P., Baron1, M. et Jambon, F. Integrating formal approaches in Human-Computer Interaction. In *Proceedings of INTERACT 2003 - Bringing the Bits togETHer - Ninth IFIP TC13 International Conference on Human-Computer Interaction - Workshop <<Closing*

the Gaps: Software Engineering and Human-Computer Interaction>> (September 1-5, Zurich, Switzerland), 2003.

13. Gray, P., England, D. et 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.
14. Guittet, L. *Contribution à l'Ingénierie des Interfaces Homme-Machine - Théorie des Interacteurs et Architecture H4 dans le système NODAOO*. Doctorat d'Université (PhD Thesis) : Université de Poitiers, 1995.
15. Hix, D. et Hartson, H.R. *Developping user interfaces: Ensuring usability through product & process*. John Wiley & Sons, inc., Newyork, USA, 1993.
16. Hoare, C.A.R. An Axiomatic Basis for Computer Programming. *CACM*. 12, 10 (1969), pp. 576-583.
17. Hoare, C.A.R., Hayes, I.J., Jifeng, H., Morgan, C.C., Sanders, A.W., Sorensen, I.H., Spivey, J.M. et Sufrin, B.A. Laws of Programming. *CACM*. 30, 8 (1987).
18. Jambon, F. From Formal Specifications to Secure Implementations. In *Proceedings of Computer-Aided Design of User Interfaces (CADUI'2002)* (May 15-17, Valenciennes, France), Kluwer Academics, 2002, pp. 43-54.
19. Jambon, F., Girard, P. et Boisdron, Y. Dialogue Validation from Task Analysis. In *Proceedings of Eurographics Workshop on Design, Specification, and Verification of Interactive Systems (DSV-IS'99)* (2-4 June, Universidade do Minho, Braga, Portugal), Springer-Verlag, 1999, pp. 205-224.
20. Johnson, C.W. Using Z to support the design of interactive, safety-critical systems. *IEE/BCS Software Engineering Journal*. 10, 2 (March 1995), pp. 49-60.
21. Marshall, L.S. *A Formal Description Method for User Interface*. Ph.D Thesis : University of Manchester, 1986.
22. McMillian, K. *The SMV System*. Carnegie Mellon University, 1992.
23. Palanque, P. *Modélisation par Objets Coopératifs Interactifs d'interfaces homme-machine dirigées par l'utilisateur*. Doctorat d'Université (PhD Thesis) : Université de Toulouse I, 1992.
24. Paternò, F. et Faconti, G.P. On the LOTOS use to describe graphical interaction. In Cambridge University Press, 1992, pp. 155-173.
25. Paternò, F. et Mezzanotte, M. Formal verification of undesired behaviours in the CERD case study. In *Proceedings of IFIP TC2/WG2.7 Working Conference on Engineering for Human-Computer Interaction (EHCI'95)* (14-18 August, Grand Targhee Resort (Yellowstone Park), USA), Chapman & Hall, 1995, pp. 213-226.
26. Roché, P. *Modélisation et validation d'interface homme-machine*. Doctorat d'Université (PhD Thesis) : École Nationale Supérieure de l'Aéronautique et de l'Espace, 1998.
27. Scapin, D.L. et Pierret-Golbreich, C. Towards a method for task description : MAD. Berliguet, L. et Berthelette, D. (Ed.). In *Working with display units*, Elsevier Science Publishers, North-Holland, 1990, pp. 371-380.
28. Schenck, D. et Wilson, P. *Information Modelling The EXPRESS Way*. Oxford University Press, 1994.
29. Shepherd, A. Analysis and training in information technology tasks. Diaper, D. (Ed.). In *Task Analysis for Human-Computer Interaction*, Ellis Horwood, Chichester, USA, 1989, pp. 15-55.
30. Waserman, A. User Software Engineering and the design of Interactive Systems. In *Proceedings of 5th IEEE International Conference on Software Engineering* ,IEEE society press, 1981, pp. 387-393.

31. Wellner, P. StateMaster : a UIMS based on Statecharts for prototyping and target implementation. In *Proceedings of Human Factors in Computing Systems (CHI'89)* (30 April - 4 May, Austin, USA), ACM/SIGCHI, 1989, pp. 177-182.