Using Formal Methods in Safety-Critical Interactive System Design: from Architecture-based Approaches to Tool-based Development

Patrick GIRARD, Mickaël BARON

LISI/ENSMA
1 rue Clément Ader, Téléport 2, BP 40109
86961 Futuroscope Cedex
France
{girard,baron}@ensma.fr

Francis JAMBON

CLIPS-IMAG
271 rue de la bibliothèque, BP 53
38041 Grenoble cedex 9
France
Francis.Jambon@imag.fr

Abstract

Although formal methods are increasingly used by researchers in HCI, their usage in actual interactive developments has not been put in practice. In this article, we describe our experience with a specific formal method –the B method– from two viewpoints. On the one hand, we demonstrate how it is possible to use formal methods on real development, from specification to actual code. Our case study concerns a real-time functional core. Doing so, we notice that some HCI concepts, such as architecture models, may have to be adapted or recreated. On the other hand, we show how it is possible to make formal methods easier to use by the way of a complete integration into HCI tools. We conclude in eliciting the lessons learned from this experience.

1 Introduction

In this contribution, we would like to introduce our experience in designing interactive systems with the help of formal methods –issued from software engineering. Doing so, we try to elicit the points that make all the difficulty of actually realizing our goals. Our approach is based on the use of the B formal method (Abrial, 1996) in order to address usability as well as security in critical interactive applications, such as avionics or chemical plants supervision. We do not intend to focus on the use of this particular formal method. Adversely, we only use this experience to illustrate the gap between software engineering and usability engineering practices while designing real-world interactive systems.

In the next part –part 2– of this article we give a short list of formal approaches that have already been used in usability engineering, and we give several points that explain their poor usage in that field. In part 3, we relate our first attempts in applying the B method in interactive systems design. We particularly focus on architectural problems, which might constitute a solid bridge between software engineering and usability engineering practices. In part 4, we show how actual usability engineering tools might incorporate secure development methods by the way of leaning on the formal semantics of software engineering tools. Last we conclude on discussing the lessons learned in these experiences.

2 Formal approaches in HCI

Formal specification techniques become regularly used in the HCI area. On the one hand, user-centered design leans on semi-formal but easy to use notations, such as MAD (Scapin & Pierret-Golbreich, 1990) and UAN (Hix & Hartson, 1993) for requirements or specifications, or GOMS (Card, et al., 1983) for evaluation. These techniques have an ability to express relevant user interactions but they lack clear semantics. So, neither dependability nor usability properties can be formally proved.

On the other hand, adaptation of well-defined approaches, combined with interactive models, brings partial but positive results. They are, for example, the interactors and related approaches (Duke & Harrison, 1993; Paternò, 1994), model-oriented approaches (Duke & Harrison, 1993), algebraic notations (Paternò & Faconti, 1992), Petri nets (Palanque, 1992), Temporal Logic (Abowd, et al., 1995; Brun, 1997). Thanks to these techniques, some safety as well as usability requirements may be proved.

However, these formal techniques are used in a limited way in the development process, mainly because of these three points:

- Few of them can lean on usable tools, which allow real scale developments. Case studies –mostly at the specification level only– have been demonstrated, but no actual application has been completely designed with these methods.
- Formal notations are currently out of the scope of interactive systems designers. Their usage by non-specialists it everything but easy.
- Formal studies are currently disconnected from usual usability engineering tools. No commercial tool and very few research ones really incorporates semantically well defined approaches.

In this paper, we relate our studies one model-oriented approach –the B method– whose one great advantage is to be well instrumented. But we do not allege it is the best nor the perfect formal method to be used. Our claim is that this model-oriented technique that uses proof obligations can be used with profit in an usability engineering context, more, it might be used together with model checking techniques, where automatic proofs of properties can be performed.

3 The B method and interactive systems design and development

This section presents the different steps that have been made in the attempt to use the B method in the design and development of interactive systems. Starting from the reduced aspect of verifying software specifications, we show how it has been possible to reach the implementation step in a complete formal development. Then, we focus on architecture problems. Last, we conclude in analyzing the difficulty of this extreme approach.

3.1 Using B for HCI specifications

In (Aït-Ameur, et al., 1998b; Aït-Ameur, et al., 1998a), the authors use for the first time the B method for the verification of interactive systems. Lying on a pure interactive case study –see below– these works suggest formally based solutions which allow solving difficulties that are inherent to interactive systems specification, such as reachability, observability or reliability. The case study is a co-operative version of a 3M™ Post-It® Note software. With this case, it is possible to address highly interactive problems due to the direct manipulation style, such as drag and drop, iconfication & de-iconification, resizing, and so on. A special attention is paid on mouse interaction and window management.

This use of the B method on a non-trivial case study has illustrated the capability of B to handle different aspects of the software life cycle in the area of interactive systems. The described approach demonstrates:

- Complete formalization: the approach is completely formalized and most of the proof obligations are automatically proved. The other ones need only few steps of manual proof.
- Property checking: it is possible to check properties on specifications, thanks to the weakening of preconditions.
- Reverse engineering aspects can be handled with this approach and the specifications of already existing programs can be used to develop new ones. Therefore, reusability issues appear.
- Incremental design: the specifications are incrementally built. Indeed, programming in the large operator allows to compose abstract machines and therefore to build more complex specifications. Yet, this process needs to follow a given methodology issued from the area of interactive system design.

One can object that this case study is situated at a too low level for the interactive viewpoint. Properties such as keeping the mouse pointer into the screen are not relevant in current graphical systems where this is ensured –or supposed to be ensured– by the windowing system. In fact, this emphasizes the problem of using formal methods in actual interactive environments. Is it acceptable to use formal techniques when we lean on graphical layers that are not formally defined? One solution, as described in this work, might be to make a reengineering analysis of such tools (Jambon, 2003).

The first step reached by this study is the one of a complete specification of an interactive system, with respect to some interactive properties. As many works in the field of formal methods in interactive systems design, it is possible to concentrate on some properties, but two drawbacks can be given:

- Because of the strong relation to the coding activities, interactive properties are not related to the user activity.
- Formally ensuring that specification are consistent, and respect properties, does not ensure that the actual code will respect specification, without a link between implementation and specification.

One of our major goals in exploring the usage of formal methods in the context of interactive systems design and development was to ensure that other people than pure formal methods specialists could use the method. So, with help of B tools, we tried to realize the whole development of an interactive application, from high-level specifications to running code. We first propose a architecture model to assist the designer (§3.2), and then define heuristics to implement this model (§3.3).

3.2 Formal development versus software architecture models

3.2.1 Case study

The case study is here a control panel for a set of three rechargeable batteries. It is an elementary safety-critical process-control system: the operator is in charge of selecting the live battery –via the switches– whereas the hardware state –the batteries levels– is updated asynchronously (figure 1). Consequently, the functional core of this case study –the batteries– is a real-time system.

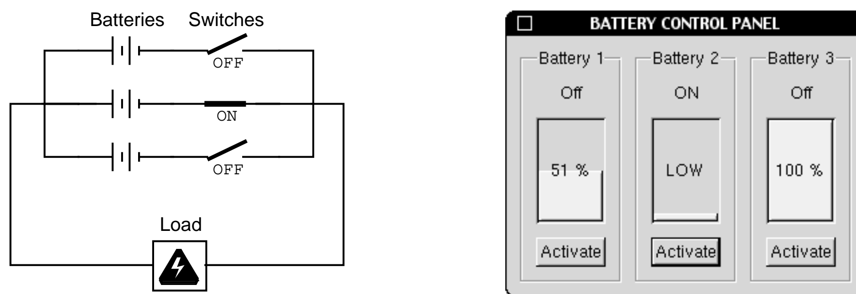

Figure 1: Case study electric diagram and user interface.

Both safety and usability properties have to be ensured. This required first step of the design process consists in modelling the battery control panel requirements with the B language. Three kinds of requirements must be fulfilled:

- The system must be safe, i.e., the system must avoid shortcuts and it must not be possible to switch on an empty battery.
- The system must be honest, i.e., the user interface widgets must display exactly the batteries levels and switches positions.
- The system must be insistent, i.e., the system must warn the operator when a battery is going to be empty.

3.2.2 Control-Abstraction-View software architecture model

Our first idea for designing such a system was to use a well-known multi-agent model, such as MVC (Goldberg, 1984) or PAC (Coutaz, 1987), because acceptability of formal methods is greatly influenced by using domain standard methods. The interactive system specifications must however stay in the boundaries of the B language constraints. We selected three kinds of constraints that relate to our purpose. These main constraints are:

- Modularity in the B language is obtained from the inclusion of abstract machine instances –via the INCLUDES clause– and, according to the language semantics, all these inclusions must build up a tree.
- The substitutions used in the operations of abstract machines are achieved in parallel. So, two substitutions –or operations– used in the same operation cannot rely on the side effects of each other.
- Interface with the external world, i.e. the user actions as well as the updates of system state, must be enclosed in the set of operations of a single abstract machine.

Classic software architecture models such as PAC or MVC are not compliant with these drastic B language constraints. That is why we proposed a new hybrid model from MVC and PAC to solve this problem. The design of

this new software architecture model –CAV– cannot be detailed here. The reader should refer to (Jambon, et al., 2001) for a more detailed description of the model design (see figure 2).

Briefly speaking, the CAV software architecture model uses the external strategy of MVC: the outputs of the system are devoted to a specific abstract machine –the View– while inputs are concerned by another one –the Control– that also manages symmetrical inputs from the reactive system which is directed by the third abstract machine –the Abstraction. The Control machine synchronizes and activates both View and Abstraction machines in response to both user and reactive system events.

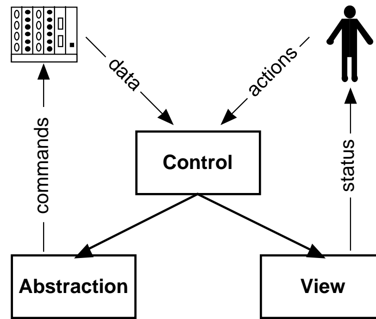

Figure 2: The three components of the Control-Abstraction-View software architecture model.

3.2.3 Specification

Among the usability properties, the system is in charge of warning the user if a battery is going to be empty. This usability requirement has to be specified as: if the battery switch is in position ON and the level is below or equal 10%, a warning message must be shown. This is specified in the INVARIANT clause of the View. As a consequence, the operations of the View must be specified to fulfill this invariant whatever the way they are computed. This insistence property specification is restricted to the View abstract machine. So, it is fairly easy to handle. On the contrary, the Conformity property requires the Control mediation between Abstraction and View. Its specification is similar to the specification of safety below.

Among the safety requirements, we detail now the prevention of user error: the operator must not be able to switch on an empty battery. At first, this safety requirement deals with the functional core of the system, i.e., it must be specified in the Abstraction. Moreover, this requirement is not a static but a dynamic property: the battery can become empty while switched on, but an empty battery must not be switched on. This requirement is not static predicate, so, it cannot be specified in the invariant clause of the abstract machine. In the B language semantics, this category of requirement must be specified in a precondition substitution of operations.

In fact, we delegated to the Control abstract machine –that includes the Abstraction– this safety requirements, i.e. the Control is in charge of the verification of the semantic validity of the parameters when it calls the operation of the Abstraction abstract machine. We name this technique the delegation of safety. This generates two consequences: (1) The operator cannot be aware of the fact that a battery could not be switched on ; (2) An action on a pushbutton can be generated with a empty battery number as parameter, so some required proof obligations cannot be proved.

The first consequence is easy to set up. We have to improve the interface layout and to update the state of the button: enabled or disabled. Of course, if a button is disabled, it is well known that this button cannot emit any action event. This assertion may seem to be sufficient to solve the second consequence above. That is not exact: the B semantics cannot ensure that a disabled button cannot emit events because the graphic toolkit is not formally specified. So, the Control abstract machine must filter the input events with the button states specified in the View abstract machine. This is required by the formal specification. The benefit of this consequence is that our system is safe whether the user interface is defective.

3.3 From formal specifications to implementation

The final program must be a set of software modules in which some of them are formally specified and implemented, and some others are developed with classic software engineering methods. In order to dissociate these two antagonist types of modules, interfaces have been inserted in between. So, at the implementation step, the CAV architecture supports some add-ons as shown on figure 3. We now focus on these three types of modules: secure code, interface and native modules.

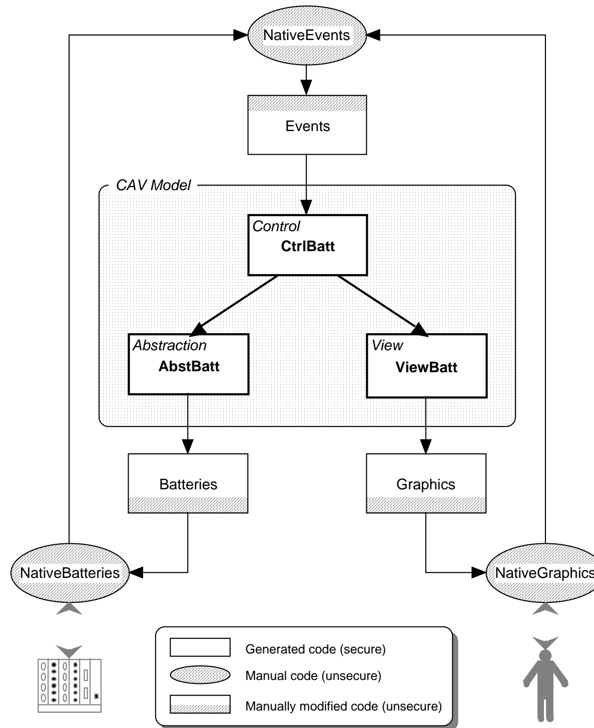

Figure 3: The CAV software architecture with interface and native modules.

3.3.1 Secure Code

The core of the interactive system has been specified in three B abstract machines. These machines specify the minimum requirements of the system but do not give any implantation solution. To do so, the B method uses implementation machines that refine abstract machines. The implementation machines are programmed in BØ pseudo-code that shares the same syntax with the B language, and is close to a generic imperative programming language. In implementation machines, the substitutions are executed in sequence. BØ pseudo-code can be automatically translated into C code.

As implementation machines refine abstract machines, they must implement all the operations of the abstract machines. Moreover, the B method and semantics ensure that the side effects on variables of the implementation machine operations do respect the invariant as well as the abstract machine operations they refine. Providing the proof obligations are actually proved, the implementation machines respect the safety and usability requirements. So, the code is secure providing the specifications are adequate.

3.3.2 Native Code and Interfaces

A working program cannot be fully developed with formal methods because most of graphic widgets and hardware drivers libraries are not yet developed with formal methods. As a consequence, the battery control panel uses three native modules:

- The NativeGraphics software module controls the graphic layout of the user interface. It uses the GTk library.
- The NativeBatteries software module simulates the batteries with lightweight processes. It uses the POSIX thread library.
- The NativeEvents software module is in charge of merging the events coming from the user or the hardware and formats them to the data structure used by the BØ translator.

These three modules are not secure. However, the modules can be tested with a reduced set of test sequences because the procedures of these modules are only called by the secure code that does respect the formal specification. For example, the bar graph widget of NativeGraphics module is to be tested with values from 0 to 100 only because the secure modules are proved to use values from 0 to 100 only. Abnormal states do not have to be tested. The interfaces module roles are to make a syntactic filtering and translation between native modules and secure code:

- The Events software module receives integer data and translates them to 1..3 or 0..100 types. This module is secure because it as been specified and fully implemented in BØ but is called by non-secure modules.
- The Graphics and Batteries modules are specified in B and the skeleton of the modules is implemented in BØ and then manually modified to call the native modules NativeBatteries and NativeGraphics respectively.

3.3.3 Programming Philosophy

At last, the project outcome is a set of C source files. Some of these files are automatically generated from the BØ implementation, while others are partially generated or manually designed. The formal specification and implantation require about one thousand non-obvious proof obligations to be actually proved. All these proof obligations can be proved thanks to the automatic prover in a few dozen of minutes with a standard workstation.

The core of the system is formally specified and developed. The programming philosophy used is called the offensive programming, i.e., the programmer does not have to question about the validity of the operations calls. The B method and semantics ensure that any operation is called with respect to the specifications. Most of the dialogue controller as well as the logic of the View and the Abstraction are designed with this philosophy. As a consequence, most of the dialog control of the system is secure.

On the opposite, the events coming from the real-word –user or hardware– have to be syntactically and semantically filtered. This programming philosophy is defensive. On the one hand, the syntactic filtering is done by the Event module that casts the parameter types –from integer to intervals. On the other hand, the semantic filtering is achieved by the Control module, which can refuse events coming from disabled buttons. So, the system is resistant to graphic library bugs or transient errors with sensors. This filtering is required by the proof obligations that force upon the operation calls to be done with valid parameters.

There is no need to use the defensive programming philosophy in native modules. The procedures of these modules are called only by secure modules, so the parameters must be valid anytime. Neither verification nor filtering is necessary. The programming philosophy looks like the offensive philosophy except that the native modules are not formally specified but must be tested, so we name this philosophy half-offensive. As a consequence the development of high-quality native code can be performed with a reduced programming effort.

3.4 Formal method in interactive system design: what kind of user ?

As we write upper, one of our first goals was to ensure that other people than pure formal method specialists could use the method. Did we succeed? We must admit that this goal is not fully reached today. In our first attempts on the Post-It® case study, even if the B tool automatically demonstrated most proofs, it remained some of them to be demonstrated by hand. This task cannot be made by non B specialists.

In the second case, for the battery case study, we obtained a fully automated process with the B tool. But it required to pay strong attention on condition writing, more, despite of the smallness of the study, the number of generated proof obligation let us think that a much more example might overcharge the tool. However, we demonstrate that,

thanks to a new software architecture model –CAV– and thanks to some specification heuristics, it is possible to develop a working case study.

4 Incorporating formal methods in HCI tools

Another way to allow cooperation between SE and HCI is to lean on formal semantics while building a tool for HCI. We describe in this section such an approach, and show how it can bring different solutions. In section 4.1, we shortly review the area of HCI tools, mainly GUI-Builders[1] and Model-Based tools[2]. Section 4.2 describes the fundamentals of our proposal: connecting directly and interactively a GUI-Builder to a functional core, by the way of formal semantics. Section 4.3 relates how to incorporate task-based analysis in this process.

4.1 HCI tools at a glance

Human computer interaction tools for building interactive software are numerous. On the one hand, GUI-Builders and tools like Visual Basic® or JBuilder® do not support any kind of external model. Moreover they do not provide any way to handle any kind of formal method. On the other hand, Model-Based tools (Puerta, 1996) deal with models, but are not usable for actual software development. MBS are the evolution of primitive User Interface Management Systems (UIMS). (Szekely, 1996) gives a generic overview of architectural description of MBS components.

First, the model is the most important component of MBS environments. It represents all the different views of the interactive application and may be decomposed into sub-models such as the domain, the dialog, and/or the presentation models. Three abstraction layers are identified. (1) The higher layer is made of the *domain and task models*. (2) The intermediate layer, the *abstract specification layer*, may involve abstract interaction objets and abstract data. Finally, (3) the lower layer describes the *concrete specification*. Interaction objets are concrete widgets that come from toolkits.

The second category of components of MBS environments is a set of tools able to manage the different models. *Modeling tools* help the designer to edit the models as MASTERMIND (Szekely, et al., 1995), or more user-centered, such as forms in MECANO (Puerta, 1996). *Automatic design tools* are able to complete and/or to concretize some abstract specifications in order to produce new and more concrete specifications as JANUS (Balzert, et al., 1996). *Implementation tools* allow direct production of code from the models as MOBI-D (Puerta & Eisenstein, 1998). *Validation tools* are the really added value of MBS compared from non MBS approaches. Reasoning upon model is easy, and enforcing properties through interactive systems is much more easy when external models are available. A new tendency incorporates formal approaches into MBS in order to get the benefits of formal validation. PETSHOP (Navarre, 2001) and our suggested approach push forward this tendency.

4.2 A semantic link between the functional core and the HCI tool

The basic idea of our approach is to build HCI tools that lean on formal semantics to ensure that properties are maintained all along the development process. At the same time, we do not expect the user to become a formal method specialist.

Our first step was to demonstrate how it is possible to build a tool that ensures a semantic formal link. We start from a formally developed functional core. We assume that this functional core, which has been specified with the B method, delivers services through an API. It is possible to automatically link such a functional core to a tool that exploits function signatures and formal specifications to help building interactive software.

In figure 4, we can see a screen copy of the SUIDT (Safe User Interface Development Tool) environment (Baron & Girard, 2002; Baron & Girard, 2004; Baron, 2003). On the left, the animator (tag 1) consists in fully generated interface that allows to interactively run the functional core. Every function of the functional core is usable through button activation. When parameters are required, a dialog box appears to allow the user to enter them. Functions are

[1] Graphical User Interface Builders
[2] currently called MBS for Model Based System

textually described, and current state of the functional core can be estimated through the result of all functions. It is important to notice that all that part is fully automatically generated. It allows the user to "play" with his/her functional core, and to be aware of functional core state. In the right part of the figure (tag 2), we can see the GUI-Builder view, where widgets can be dropped to build the concrete user interface. In the center, as in any GUI-Builder, we can see a property window, which allows the user to finalize the presentation. Below this window, the last window permits associating events to functions from the functional core.

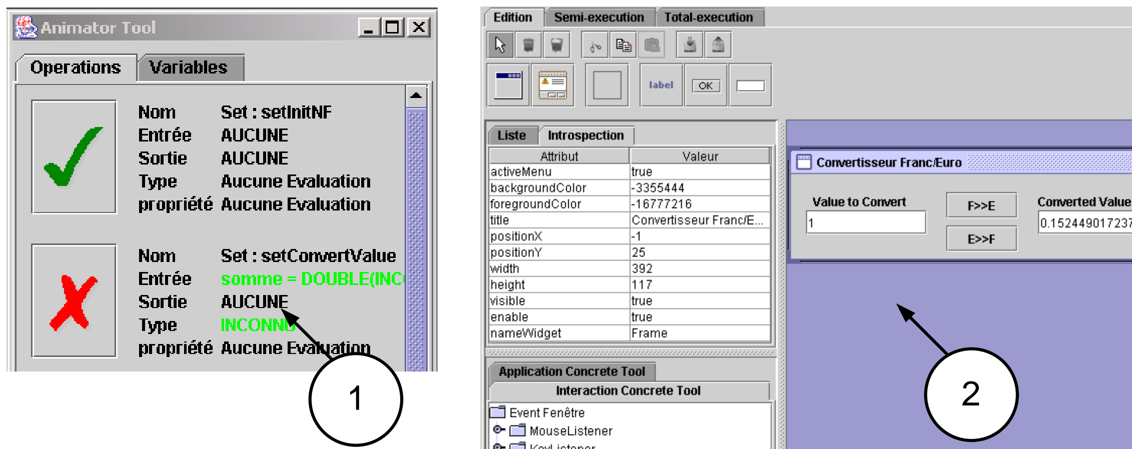

Figure 4: the SUIDT environment.

The great two originalities at this point are: first, at any time, we can switch from design mode to test mode where functional core can be called from either the presentation or the animator (the context of the functional core remains consistent); second, the system leans on formal specifications from the functional core to ensure that calls are correct. This study demonstrates that it is possible to incorporate formal approaches in interactive tools. The benefit is not very important at this stage, because interactive model is poor: we assume that the link between widgets and functional core is direct. In the next part, we show how it is possible to enhance this model.

4.3 Linking task based analysis and formal semantics

The second step of our study consists in focusing on user validation. We incorporated task-based analysis into our system by the way of two level task models (abstract and concrete task models) based on the CTT formalism (Paternò, 2001). These two levels correspond to the intermediate and to the lower levels of the generic MBS architecture. They allow to take into account the user needs and to realize a successive validation in two steps.

4.3.1 Functional Validation

The intermediate level of a task model allows to validate the user point of view on the functional core features. In fact, in SUIDT, this validation level corresponds to higher level models and allows modelling the goals of the user over the interactive application. More concretely, SUIDT links together the domain model and the task model. The dynamics of the task model is based on precedence constraints (temporal operators), on the guards (playing the role of pre-conditions) of the functional core functions and on the task post-conditions which permit to modify the state of the functional core expressed at the leaves of the task model. At this intermediate level of the task model, two main results are obtained. On the one hand, it is possible to test the functionalities of the functional core in order to evaluate if the user needs are satisfied at any abstraction level of specification. On the other hand, it is possible to record scenarios for further tests of integration.

4.3.2 User Interface Validation

In a second step, a graphical interface, designed thanks to a classical GUI-Builder, is associated with the intermediate task model level in order to obtain a lower level model. This model is a refinement of the state of the previous task model, where every interactive or application task of the CTT is described in terms of concrete interaction or application objects. Refinement of a state consists in adding other state variables related to interaction. Since we deal with a single environment, it becomes possible now to relate these interactive or application tasks of the CTT to the concrete GUI level.

As result of the SUIDT approach, every model is an executable model since the functional core plays a central role. In fact, every model is related to the functional core, which allows to run an actual program each time the task model is run. Thus, the running context (during test phase) is not re-initialized or lost when the designer switches to the design mode. So, it is easier for him/her to validate the prototype. Moreover, as for tools like MASTERMIND or PETSHOP, our application allows to test the program under construction. In figure 5, we can see a view of the simulation tool. On the tag 1, the simulator tool permits to animate the task model in concordance with the user interface (tag 2). The designer can directly test the interactive application either by interaction on user interface (tag 2) or by animation of the task model (tag 1).

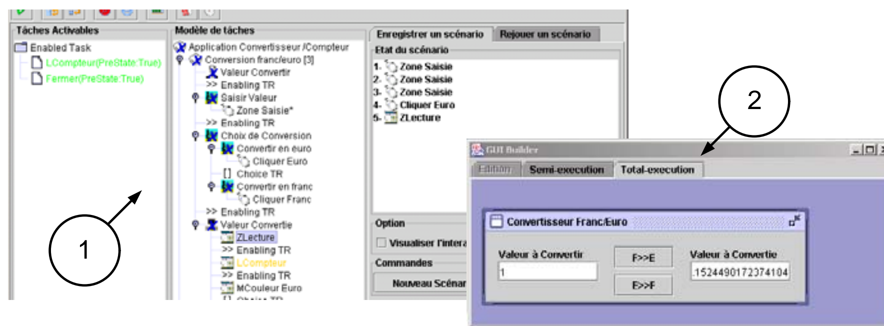

Figure 5: task-based aspects of SUIDT.

5 Conclusion and lessons learned

Our studies bring partial solutions in the field of formal methods use in interactive systems design. Two approaches was studied and gave complementary findings. The first one use a new software architecture model –CAV– and some heuristics to obtained a fully automated process with the B tool. This approach do not hide the complexity of the formal development but gives patterns to lower it. On the opposite, the second solution embed in a GUI-builder like tool the complexity of the formal development, and so, allows interactive systems designers to use the method in a blink mode. So, no particular mathematical skills are required.

On the one hand, we demonstrate how formal methods can really be usable in interactive systems design. In the meantime, their usage is restricted to specialists that come to grips with mathematical fundamental because automatic proving is not always possible in real developments. On the other hand, we do not propose a global method to build interactive systems with such tools. In both approaches we assumed that functional cores have to be designed first. In many cases, this is not the best way to work because in some cases, early task analysis may turn up new needs. Modifying the functional core and consequently its formal specifications to rebuild a new solution might be difficult.

One of the strongest questions that have been raised by these studies is: what kind of user for formal methods in usability engineering ? One the one hand, manipulating formal methods themselves is often hard. Complete formal development is very difficult, and formal tools such as “Atelier B” are not really able to manage real scaled applications. On the other hand, manipulating formal methods through GUI-Builder like tools seems very interesting. But where is the place for formal development ? And who might make it ? All these points are to be studied, and solutions to be bring by further work.

References

Abowd, G.D., Wang, H.-M. & Monk, A.F. (1995). A Formal Technique for Automated Dialogue Development, *Proceedings of the DIS'95, Design of Interactive Systems*, 219-226. ACM Press.

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

Aït-Ameur, Y., Girard, P. & Jambon, F. (1998a). A Uniform approach for the Specification and Design of Interactive Systems: the B method, *Proceedings of the Eurographics Workshop on Design, Specification, and Verification of Interactive Systems (DSV-IS'98)*, 333-352.

Aït-Ameur, Y., Girard, P. & Jambon, F. (1998b). Using the B formal approach for incremental specification design of interactive systems, *Proceedings of the Engineering for Human-Computer Interaction*, 91-108. Kluwer Academic Publishers.

Balzert, H., Hofmann, F., Kruschinski, V. & Niemann, C. (1996). The JANUS Application Development Environment-Generating more than the User Interface, *Proceedings of the Computer-Aided Design of User iterface (CADUI'96)*, 183-206. Presse Universitaire de Namur.

Baron, M. & Girard, P. (2002). SUIDT : A task model based GUI-Builder, *Proceedings of the TAMODIA : Task MOdels and DIAgrams for user interface design*, 64-71. Inforec Printing House.

Baron, M. & Girard, P. (2004). SUIDT : Safe User Interface Design Tool (Demo), *Proceedings of the International Conference on Intelligent User Interfaces Computer-Aided Design of User Interfaces*, 350-351. ACM Press.

Baron, M. (2003). *Vers une approche sûre du développement des Interfaces Homme-Machine (Thesis).* Doctoral dissertation, Université de Poitiers.

Brun, P. (1997). XTL: a temporal logic for the formal development of interactive systems. In P. Palanque & F. Paternò (Eds.), *Formal Methods for Human-Computer Interaction* (pp. 121-139). Springer-Verlag.

Card, S., Moran, T. & Newell, A. (1983). *The Psychology of Human-Computer Interaction*. Lawrence Erlbaum Associates.

Coutaz, J. (1987). PAC, an Implementation Model for the User Interface, *Proceedings of the IFIP TC13 Human-Computer Interaction (INTERACT'87)*, 431-436. North-Holland.

Duke, D.J. & Harrison, M.D. (1993). *Abstract Interaction Objects*.

Goldberg, A. (1984). *Smalltalk-80: The Interactive Programming Environment*. Addison-Wesley.

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

Jambon, F. (2003). Premiers pas vers la rétro-conception en langage B d'une bibliothèque de composants graphiques, *Proceedings of the 15e conférence francophone sur l'Interaction Homme-Machine (IHM'03)*, 118-125. ACM press.

Jambon, F., Girard, P. & Aït-Ameur, Y. (2001). Interactive System Safety and Usability enforced with the development process, *Proceedings of the Engineering for Human-Computer Interaction (8th IFIP International Conference, EHCI'01, Toronto, Canada, May 2001)*, 39-55. Springer.

Navarre, D. (2001). *Contribution à l'ingénierie en Interaction Homme-Machine*. Doctoral dissertation, Université Toulouse 3.

Palanque, P. (1992). *Modélisation par Objets Coopératifs Interactifs d'interfaces homme-machine dirigées par l'utilisateur*. Doctoral dissertation, Université de Toulouse I.

Paternò, F. (1994). *A Theory of User-Interaction Objects*.

Paternò, F. (2001). *Model-Based Design and Evaluation of Interactive Applications*. Springer.

Paternò, F. & Faconti, G.P. (1992). On the LOTOS use to describe graphical interaction. In (pp. 155-173). Cambridge University Press.

Paternò, F., Mori, G. & Galimberti, R. (2001). CTTE: An Environment for Analysis and Development of Task Models of Cooperative Applications, *Proceedings of the ACM CHI 2001*, ACM Press.

Puerta, A. (1996). The MECANO project : comprehensive and integrated support for Model-Based Interface development, *Proceedings of the Computer-Aided Design of User interface (CADUI'96)*, 19-35. Presse Universitaire de Namur.

Puerta, A. & Eisenstein, J. (1998). Interactively Mapping Task Model to Interfaces in Mobi-D, *Proceedings of the Eurographics Workshop on Design, Specification and Validation of Interactive Systems (DSV-IS'98)*, 261-274.

Puerta, A.R., Cheng, E., Ou, T. & Min, J. (1999). MOBILE : User-Centered Interface Building, *Proceedings of the* 426-433. ACM/SIGCHI.

Scapin, D.L. & Pierret-Golbreich, C. (1990). Towards a method for task description : MAD. In L. Berliguet & D. Berthelette (Eds.), *Working with display units* (pp. 371-380). Elsevier Science Publishers, North-Holland.

Szekely, P., Sukaviriya, P., Castells, P., Muthukumarasamy, J. & E. Salcher. (1995). Declarative interface models for user interface construction tools : the MASTERMIND approach, *Proceedings of the IFIP TC2/WG2.7 Working Conference on Engineering for Human-Computer Interaction (EHCI'95)*, 120-150. Chapman & Hall.

Szekely, P. (1996). Retrospective and challenge for Model Based Interface Development. In F. Bodart & J. Vanderdonckt (Eds.), *Eurographics Workshop on Design, Specification, and Verification of Interactive Systems (DSV-IS'96)* (pp. 1-27). Namur, Belgium: Springer-Verlag.