SUIDT: A task model based GUI-Builder

Mickaël Baron, Patrick Girard

SUIDT: A task model based GUI-Builder

Mickaël Baron, Patrick Girard
Laboratoire d'Informatique Scientifique et Industrielle, ENSMA,
1 rue Clément Ader, 86961 Futuroscope Chasseneuil
http://www.lisi.ensma.fr/ihm
{baron, girard}@ensma.fr

ABSTRACT

User interface design tools use different approaches. Interface builders are easy to use. They elect "presentation" aspects but are not interested in user task analysis. Model Based Systems describe models for building applications but they are often difficult to use.

The approach presented in this contribution is the collaboration of these two points of view for interface construction. It co-ordinates the "easy-to-do" interface builders for interface graphics and "hard to use" model based systems for task-based interface design.

We present a development tool for end-users called SUIDT (Safe User Interface Design Tool). It uses visual programming techniques (as interface builders use) to build every application model and allows building the final application with respect to all models (as model-based systems). In addition, it enforces the respect of rules, and allows switching from design to test and vice versa, all along the development process.

KEYWORDS

Model-Based Systems, Visual Interface Design, Visual Programming, Task Model, B language, CTT, Formal Methods.

INTRODUCTION

Tools that help developers in realizing quality applications are today numerous. Two approaches can be opposed.

Interface builders (usually commercial products) enforce the "presentation" aspect, and allow realizing the functional core according to the interface design. A major drawback of these tools is the lack of task model between the functional core and the interface. Guaranteeing that the interface respects the rules given by the functional core guarantees in no way the purposes of users can be reached.

On the contrary, Model Based Systems (MBS) enforce the specification aspects of models (functional core, task model, interface model) to build and often generate interactive applications. The main obstacle to these approaches generalization is the difficulty for using them. Indeed, MBS use is not adapted to end-users and, generally, the necessity to learn specific languages makes the development process hard. The main characteristic of "end-users" in this context is that they do not have extensive programming skills.

Our approach consists in getting together the two ways. On the one hand, we lean on linked models whose semantics can be clearly defined. On the other hand, we provide the user with highly interactive tools that allow for model manipulation with respect to their semantics. Moreover, all along the design process, we permit the user to switch between design phase and test phase, with no loosing of testing context.

A first research led us to the GenBuild system [1] and [2], see Figure 1.

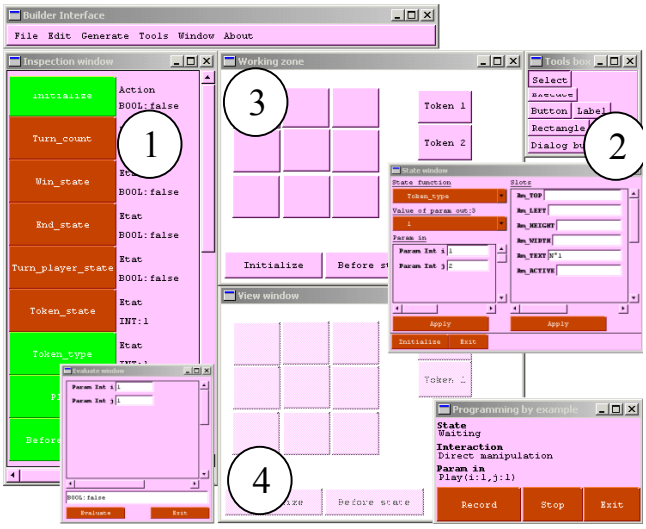

Figure 1. GenBuild System

This tool generates automatically an interactive application from a functional core. This interactive environment is made up of a set of buttons able to call each functional core primitive, using an automatically generated dialogue box to enter each action parameter (part 1 in Figure 1).

The originality of the approach lies in the fact that the automatically generated application is interfaced with an interface generator, which permits building a true interactive application and allows keeping the connection with the functional core at the same time (part 2).

The designer of the application can go alternatively from one creation step (part 3) of interface to a test step (part 4), and vice versa, without loosing his/her context of testing. We started from a functional core definition in C++ to which we added expressions such as intervals of values. They check in the final application the correctness of functional core function calling.

Nevertheless there are two major limitations on GenBuild. On the one hand, the system lacks user viewpoint: no task model is taken into account. On the opposite, it leans on semantically poor descriptions of functional core (an ad-hoc formalism). To avoid this problem, we might have designed new formalisms. On the other hand we decided to use well defined formalisms and to put them together through different models that represent the different aspects of interactive applications.

The summary of this paper is the following. In section 2, we discuss the interest of a formal functional core with a concrete example: franc/euro exchange application. Section 3 discusses the first task model called **abstract task model**, by describing the formalism. The **concrete task model** and the interface are described in section 4. In section 5 we describe the SUIDT environment tool. Then, section 6 briefly compares our work to some products from research laboratories. Finally, we give some perspectives of our work.

A FORMAL FUNCTIONAL CORE

The lack of a true formal description prevents any reasoning on the functional core. The aim of our method is to be able to lean on a really safe part, which can be developed independently from any interactive perspective, and on which full reasoning may be conducted. Software engineering methods, and more precisely formal language studies, explore these problems. We decided to find a good candidate for our purpose among existing languages. One of our major prerequisite was that the formal language had to be instrumented by professional tools. As we wrote above, we started our present work from the GenBuild system, with a pseudo formalism, which corresponds in fact to pre/post-conditions; this led us to choose as formalism a model-based method, whose semantics can be mostly represented with invariants and pre/post-conditions.

Among the increasing number of formal methods, VDM, Z and B are most employed. 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. Z [3] method uses set theory notations. Like VDM [4] it is based on preconditions and postconditions. Moreover, VDM allows the generation of a set of proof obligations which simplify the use of the method regarding to Z. B [5], based on the weakest precondition calculus, allow the description, in high abstraction level language, of the different components of a given system.

The better choice seems to be the B language, which, associated to the "Atelier B" [6] development environment, allows the complete design of applications, from the specification step to the implementation one. B method relevance is to ensure the respect of properties expressed during the specifications all along the development process. The technique of proof obligations that is used in B is the demonstration of theorems, which once proved, ensures the maintenance of the properties.

We are going to present now to illustrate our approach the franc/euro exchange application and its functional core.

FRANC/EURO EXCHANGE APPLICATION

The target application we use all along this article is the franc/euro exchange. It is a small tool that makes conversions from francs to euros and vice versa. The user enters the value he/she wants to convert he/she chooses the direction of conversion and makes the conversion, and last, he/she can read the result. There are many kinds of interfaces for this application (that is the reason we chose it) see Figure 2. We can find calculator-type interfaces or very simple ones.

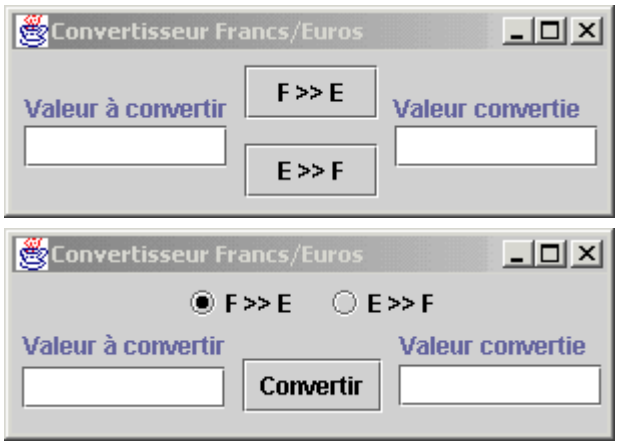

Figure 2. Interface of exchange application (a) & (b)

FRANC/EURO EXCHANGE FUNCTIONAL CORE

The functional core of franc/euro exchange is very simple. Two primitives of modification (*franc_euro_exchange* and *euro_franc_exchange)* set the direction of conversion. Two other ones realize the conversion (*input_value* and *convert_value)*. The last two primitives allow the conversion: *franc_euro_output_value* and *euro_franc_output_value*. There are many other ways to conceive the functional core. This is not important for our purpose. It is only to notice that it is possible to conceive at least two different interfaces with this functional core, Figure 2 (a and b).

The developer of the functional core is a programming specialist. He/she knows classical languages and perfectly uses formal methods. The programming specialist provides a functional core to a different developer. This one is called

"end-user" and knows what are the expectations of the specification (ergonomic characteristics…) but he/she has a little experience in classical languages. Many characteristics of "end-users" as are described in end-user programming works [7].

ABSTRACT TASK MODEL

At this step, we have a formal functional core. Nevertheless, calling core functions must follow an accurate scenario. From the specification, the designer must develop an analysis of the activity [8]. It must be independent from any idea of interface to develop and it must not be composed of insinuations on interaction device to implement.

Then the end-user defines the user's needs in terms of task with help of formalism. It is an abstract task model that can put together potential scenarios to use the functional core. This is why we base our abstract task model on a formalism yet employed which is called CTT (ConcurTaskTrees) [9]. Moreover CTT is supported by CTTE (ConcurTaskTree Environment) tool [9]. The next part is going to give a short CTT definition.

CTT FORMALISM

CTT [10] is defined by its authors as a notation for task model specifications to overcome limitations of notations previously used to design interactive applications. Its main purpose is to be an easy-to-use notation, which can support the design of real industrial applications. The CTT task model is based on three major points.

- this user action oriented approach, is based on hierarchical structure of tasks represented by a tree-like structure;
- it requires identification of temporal relationships, by Lotos, among tasks at the same level;
- it allows the identification of the objects associated to each task and of the actions which allow them to communicate with each other.

In addition CTT formalism includes four categories of tasks depending on the allocation of their performance.

User tasks are performed entirely by the user, they require cognitive or physical activities without interacting with the system.

Application tasks are completely executed by the system.

Interaction tasks are performed by user interactions with the system.

Abstract tasks are tasks that need to be refined by the three previous cases

CTT formalism is adequate to describe our abstract task model because it denotes a chain of tasks made by the user. We specify the possible scenarios of the application from the abstract task model. We have an abstract view of the working of the application with no interface element. This abstract description is sufficient but end-users must have in mind a first idea of one interface.

In order to define our abstract task model we employ a limited version of the CTT formalism. Nevertheless, the semantic of CTT formalism is not altered to preserve its formal properties. Our approach uses fully CTT formalism for graphic part (hierarchical structure of tasks and temporal relationships). But we limit this formalism for the object elements and pre-condition. Objects are entities that are manipulated to perform tasks and pre-conditions are predicates. Thus we associate to the lower level tasks only elements from the application domain (i.e. the functions of the functional core). Likewise, pre-conditions evaluate attributes from the functional core.

Now we are going to describe the hierarchical decomposition of tasks into smaller ones in our franc / euro exchange application to illustrate our approach.

EXCHANGE APPLICATION ABSTRACT TASK MODEL

This specification of the Figure 3 respects the scenario defined in the functional core part.

The user begins to input the value to convert. The choice of conversion direction is realized during the conversion. If the user wants to convert something into francs, he/she chooses "convert to franc", and vice versa. Finally when the conversion is made the result is given. The Franc/Euro exchange task is interactive. As long as the exit task is not selected, the iterative task is executed. The user associates functions of the functional core with the lower level tasks of the abstract task model: *input_value* function with *Input_Value_to_Convert* task, *franc_euro_exchange* and *convert_value* function with *Convert in Euro* task.

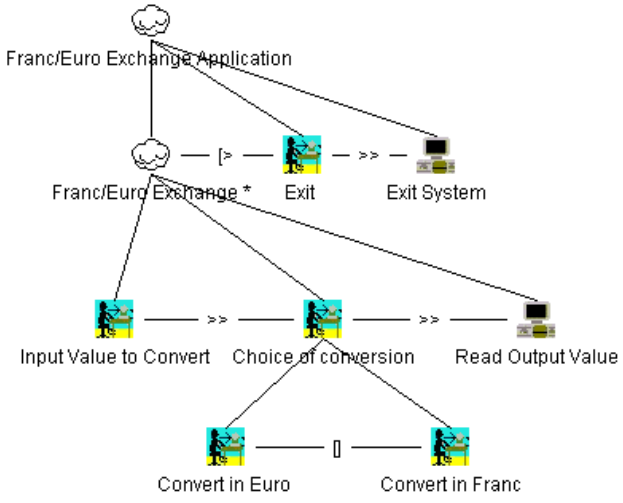

Figure 3. Exchange Application's abstract task model

We have defined an abstract task model which allows us to

model the functional core specification logic. Then it is possible to have several abstract task models for one functional core. But for each new abstract task model the developer must represent in his mind the interface that is not still created.

The next part will show what we call a concrete task model associated to the interface.

CONCRETE TASK MODEL

The concrete task model is linked to the interface model, the functional core. It describes the application actions (feedback) and the user interaction of the interface. In this part we explain our concrete model, the refinement notion with the abstract task model and the formalism used.

WHAT IS AN CONCRETE TASK MODEL?

One of the main limitations of the CTT formalism does not take into account the level of interaction i.e. the interaction type (click on a button, move mouse cursor on a frame). The concrete task model allows then to define this level of interaction. It is a refinement of the abstract task model from the point of view of the application and the interaction tasks. This model implements the application actions and the interaction ones.

At the beginning of the abstract task model development, the end-user has a blurred vision of the interface, he/she do not know exactly what his/her interface might look like. He/she bases on the abstract task model that is an "abstract" model. Then the end-user can build the application interface.

Let's take the franc/euro exchange application to illustrate it. Concerning "Read Output Value" task the developer does not precise the way the task will be realized but its abstract purpose. For example, the conversion result can be either displayed on the interface or printed.

The abstract task model calls a set of interaction tasks and of application tasks. The concrete task model must completely refine all these tasks. Let us begin our explanation by showing Figure 4 which represents the widget objects.

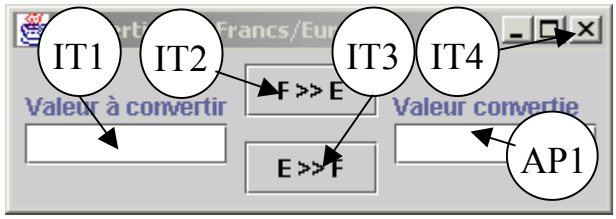

Figure 4. Interaction and Application tasks

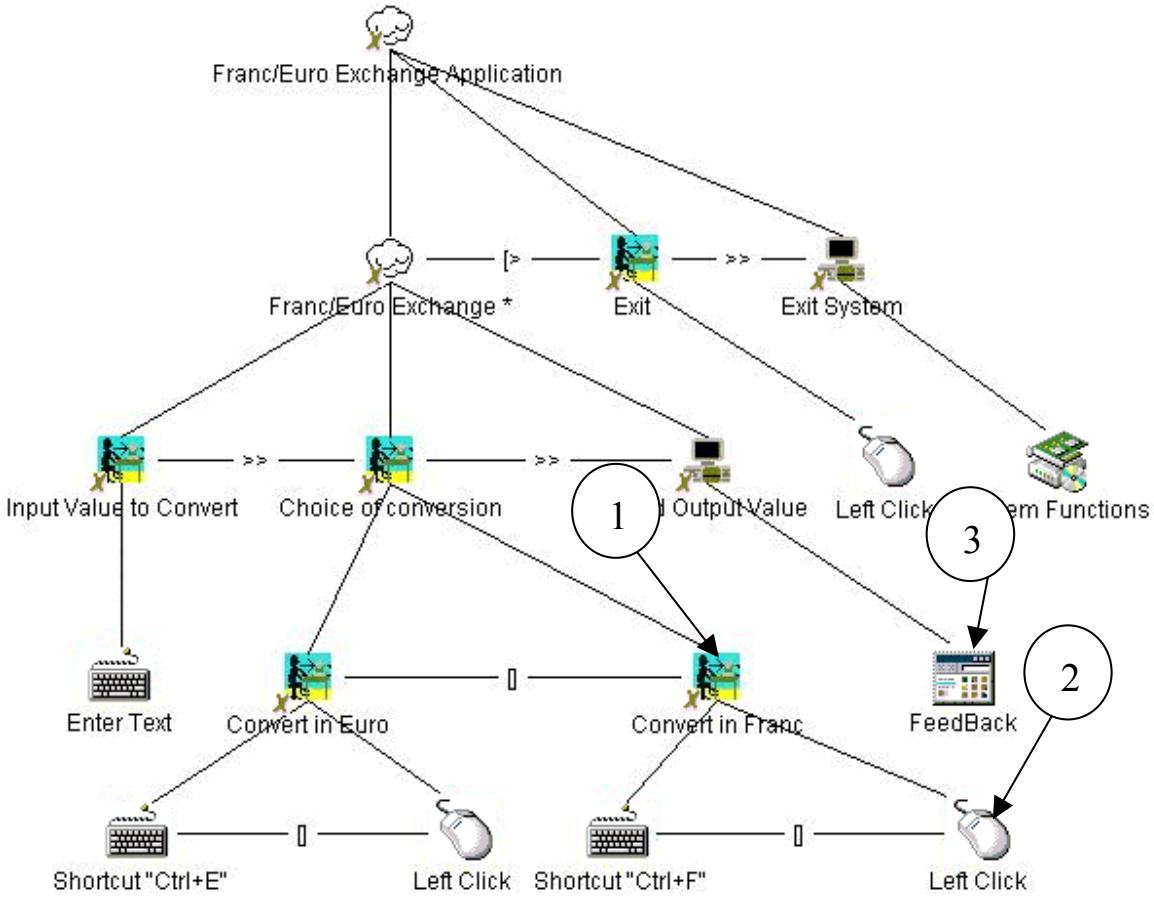

Figure 5. Exchange Application's concrete task model

The designations 1,2,3,4 and 5 of the Figure 4 represent the interaction and application tasks implementation of the abstract task model. The interaction task n°1 corresponds to a text edit. The exchange application user enters a value which is sent to a primitive of the functional core *enter_amount*. The interaction tasks n°2 and n°3 make the conversion by calling up either the *franc_euro_exchange* or *euro_franc_exchange* primitives. Finally *convert_value* primitive is to be called. The application task n°5 displays the result by calling one of both primitives (*franc_euro_output_value*, *euro_franc_output_value*). Finally application task n°4 allows to leave the application.

Our concrete task model is close to UAN (User Action Notation) [11] formalism. UAN represents the user interface by a hierarchy of asynchronous tasks. But the problem of this formalism is that it decomposes an interaction task in the lowest level of detail records: user actions, corresponding interface feedback and state change. Finally it is a textual notation.

The choice of the concrete task model formalism was guided by the type of users : the end-users. For them it is more practical to use a graphical notation and a superior level of description.

In the next part we will present the concrete model formalism.

CONCRETE TASK MODEL FORMALISM

The formalism we use is also based on the CTT formalism. The CTT formalism allows us to describe the refinement of the interaction and application tasks from abstract task model.

CTT formalism is employed but we do not use the same categories. The task categories of CTT formalism do not allow to describe the decomposition of interaction and application tasks. So we propose new categories which depend either to interaction task, or to application task. We add to the interaction tasks several categories which describe the interactions on the widgets used to implement a graphic user interface. For instance, in our application, there is an interaction “click with left button of mouse” on “Convert in Franc” button. This interaction and the widget is a new category of the concrete task model. Figure 5 tag n°1 shows the *Convert in Franc* task refinement and tag n°2 describes one solution to refine interaction task. Next it is necessary to link the new categories (from interaction categories) with the objects of functional core. These objects of functional core are the functions available from the refined abstract task. So we have described the user interaction on interface (with call of objects of functional core) but it is missing the modification of interface (feedback).

It can be the same reasoning for the application tasks where only two other tasks are added: system tasks and feedback tasks. System tasks call objets from an operating system (printing for example). Feedback tasks are used to modify elements of graphic user interface after an interaction. For example tag n°3 shows the display of the new computing value of conversion.

These enhancements do not modify the semantics of CTT formalism but just give the possibility to create a single task from abstract task (tag n°3).

Now, we are going to present the development tool which uses the functional core to create the abstract, concrete task model and the user interface. Also, we will show the way to link each models between them.

“SUIDT” DEVELOPMENT TOOL

We built an interactive end-user programming environment called SUIDT (Safe User Interface Design Tool) which regroups functional core, abstract, concrete task model and user interface.

We obtain a structure of our development environment that shows the links between models, see Figure 6. The programming specialist programs the functional core and the end-user designs abstract , concrete task model and application interface by using visual programming.

When the development is finished, we can generate a final application that guarantees the purpose of users can be reached.

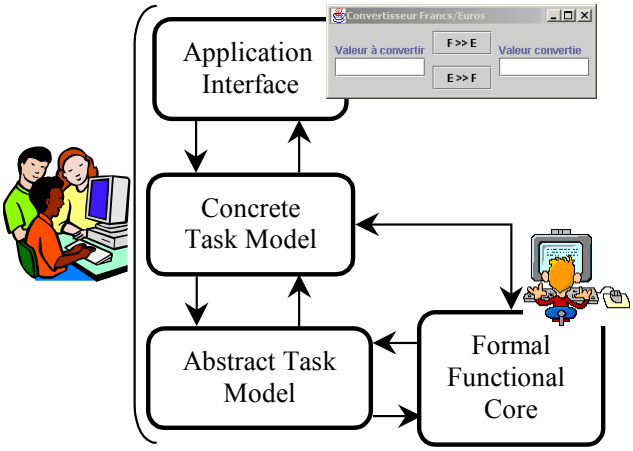

Figure 6. Development Structure

AN INTERFACE GENERATOR AND FUNCTIONAL CORE ANIMATOR

The GenBuild method can fit over directly to a functional core expressed with B method. The specification analysis with a sufficient level of details permits generating automatically an interface manipulation for model manipulation. The interface displays the formal functional core and allows to check whether it works or not. It also ensures visually that invariants are established.

In order to generate this development interface, we must recover all signatures of operations, made up operation name, and in/out parameters. They respect pre- and post-conditions. These conditions permit to know the type of

parameters and their conditions. The development interface will display the functional core by associating the functions to graphical objects. Some text zones associated to buttons put up the information connected with operations, notably pre- and post- conditions. It is a way to help the future designer in his/her development by displaying the information of the functional core.

Figure 7 represents the tool which animates the functional core. Tag 1 illustrates the function name. The function parameters are displayed on tag 2 and for every parameter the tool decomposes the display in three portions: the *parameter name*, its *type* and its *value*.

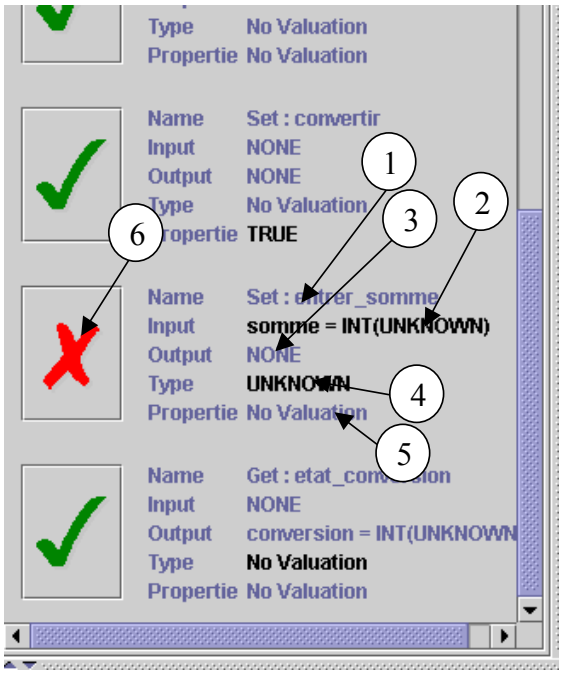

Figure 7. Interface of Functional Core

Likewise tag 3 returns the output parameter information. Finally tags 4 and 5 allow to know the predicate of type and properties. The tool evaluates for each parameter value the predicates of field 4 and 5. So the function is executed only if these two predicates are true and only when the end-user clicks on the button 6.

Therefore, he is going to use a usable functional core and check it to see if it really works. For that, the end-user may interact with the graphical function of the functional core by pushing on the buttons associated to the functional core functions. The end-user must respect pre- and post-conditions when he/she calls functions. A set of dialog boxes is used to enter parameters. For example, enter_amout function needs a real-type parameter.

The advantage of the approach is that we have a functional core we can test and use during the application design process. Once tested, the functional core corresponds to the expectations of the specification. It only remains to define an interface and especially an abstract task model that guarantees users goal are reached.

ABSTRACT TASK MODEL BUILDER

The second module of our environment builds the abstract task model of the application. The developer is the end-user who just knows how the functional core works. With that kind of tool, he/she is given a way to program rapidly and intuitively. For this, user-friendly programming techniques such as programming by demonstration or visual programming are used. Then end-user designs his/her abstract task model in a working zone called "Abstract Task Model View", Figure 8. The abstract task model design is an incremental development composed of two parts.

The first part consists in creating the graph, i.e. to make a hierarchical decomposition of tasks. The end-user makes her/his abstract task model by putting on the working zone the types of tasks (user tasks, interaction tasks…) by giving them a specific name. Then he/she defines the temporal relationships among tasks at the same level. Finally the end-user links up the abstract task model the functional core. He/she associates functions of functional core with every task.

The second part consists in testing the abstract task model to verify if it works to the functional core specification logic. Checking tools of abstract task models is similar to the CTTE's ones. But it also allows testing interactively the model. So, the end-user examines the progress of the abstract task model according to scenarios and he/she can choose the parameter values. But the simulation of the task model in CTTE only allows checking enabled tasks. On the opposite SUIDT modifies the state of the functional core when a task is executed because there is a connection between these two models. When objets (functions) of functional core are called, the end-user must give (by the way of a dialog box) additional information (in parameters). Thus with our approach, end-user tests both models in the same time.

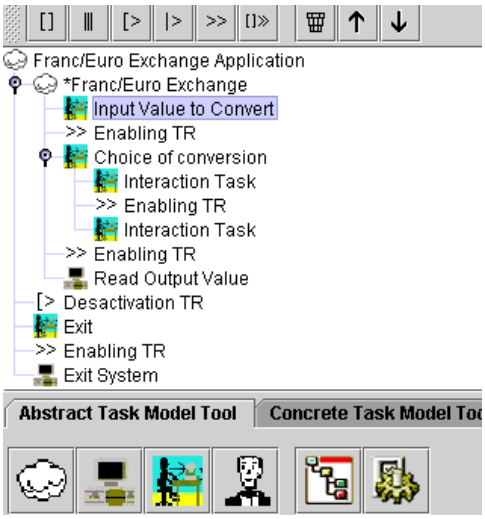

Figure 8. Task Model Tool of SUIDT

We are going to build an interface that dresses this abstract

task model.

CONCRETE TASK MODEL BUILDER

This builder allows to make the concrete task model via an abstract task model, a functional core and a graphical user interface. This last element is built in the same time with the concrete task model. The SUIDT environment proposes a small GUI-Builder to make interface application. Actually there are only three widgets in this GUI-Builder (*button*, *textfield* and *label*) which are sufficient to design the *frank / euro exchange* application interface but we plan for the future development to expand this tool box.

The concrete task model working zone is the same as the abstract task model. But the tree-like structure of abstract task model cannot be modified directly. Therefore the designer must come back to abstract task model builder to change it. He/she builds his concrete task model in the same way as the abstract task model. The concrete task model building consists in refining (or deepening) the tree-like structure of abstract task model. But we refine only the application and interaction tasks. Thus the designer selects a task from abstract task model and puts down tasks to concrete task model hierarchical structure.

The end-user designer creates categories for the interaction tasks before to build the concrete task model. It is the SUIDT GUI-builder which allows to add or suppress these categories of interaction tasks. A new category is built in associating widgets from current user interface with an interaction on this interface application for example "Convert in Franc" button with "click with left button of mouse".

Then the new categories of the tasks interactions and the categories of the tasks applications (feedback task and system task) are placed on the concrete task model by the end-user developer who links each tasks by a temporal relation.

Finally he/she programs the content of the task by following the approach of GenBuild. It consists in associating attributes from interface with attributes from functional core and inversely. Several examples:

- A category of interaction task: a click on "Convert in Franc" button executes *euo_franc_exchange and convert_value* function of functional core, tag 2 of **Figure 5**.
- A category of application task: e*uro_franc_output_value* function is called to modify the textfield content which displays the convert value, tag 3 of **Figure 5**.

No conventional programming is required, the end-user developer uses visual programming techniques but at this time we can only manipulate few attributes of the interface.

RELATED WORKS

In this last section, we situate our work compared to the literature. Our approach is close model based systems. Indeed, we develop a set of models (task model, interface model, function model) that finally generate an interactive application. In MBS [12] there may be modeling tools that assist the developer in the construction of the models. Their goal is to mask all or a part of the complexity of the complex modeling language. Among the available tools, we find simple editors to develop textual specifications of the model (MASTERMIND [13]), forms for the creation of elements of the model (Mobi-D [14]). In our case, only the functional core is developed in a textual way.

We find also graphic specialized editors, CTTE or VTMB [15]. These tools can only create and evaluate task models. SUIDT looks like CTTE for usage of the CTT formalism. But CTTE and VTMB do not allow to associate to tasks the references from objects of models (functional core, task model) because the task model is not connected with the others models. Instead SUIDT allows to design fully a final application by taking into account rules of the functional core and purposes of the user (abstract, concrete task model and user interface).

The simulation of the task model of CTTE and VTMB tools controls the task model of all kinds of incoherence (loops without end) and checks the expectations of the specification. SUIDT checks also the errors from abstract and concrete task model, but our tool ensures that the calls of the functional core and the interface are coherent.

Our work is incremental by following a linear cycle (functional core, abstract task model, concrete model and graphic interface). We can test (on the condition of having a minimum of information for all the models) the application that is in progress as MASTERMIND [13] and PetSHOP [16]. But the important point of our method is that the end-user may alternate the stages of design and test without losing the execution context.

In [17] Pribeanu presents design heuristics aiming to progressively derive the presentation from task model and application domain model. Our approach is close, in fact the development of the application begins with the application domain model (functional core) and the abstract task model without introducing a view of the interface. When these two models are built, the presentation can be established.

CONCLUSIONS and FUTURE WORK

We presented in this paper a new Computer-Aided Design for User Interfaces (CAD-UI) tool that leans on several well-defined formalisms.

The main idea is to start from a purely formalized functional core, using a strong formal language, to build an interactive application in a highly interactive way, with respect to the user viewpoint. It is built by a programming specialist. A first task model is called abstract task model. It is a task model that can put together potential scenarios to use functional core. It defines a hierarchical structure of

application in using functions of functional core. A second task model is called concrete task model. It refines tasks from abstract task model. Thus concrete task model implements the interaction and application tasks.

The end-user designs the abstract task model and the concrete task model with our SUIDT environment. It allows editing, testing and generating interactive applications. The development of SUIDT is still in progress. This environment has been implemented with JAVA programming language and B language.

Several directions can be explored from that approach.

_ We have taken on limited example to experiment SUIDT feasibility. We need to evaluate this approach against different kinds of applications – such as process control or database applications.

_ The task models of SUIDT and CTTE are based on the semantics of CTT. We have showed that SUIDT do not modify the CTT semantic, and even if our approach with two task models (abstract and concrete task model) does not correspond exactly to that used by CTTE tool, it will be interesting to study the possibility to exchange data between these two tools.

_ The generation of dialog control will authorize the automatic construction of final application.

_ Last we will be interested in the generation of documentation from the task model.

REFERENCE

1. Baron, M. and Girard, P. Construction interactive d'application à partir du noyau fonctionnel, in *Proc. Ergonomie et informatique avancées (Ergo-IHM'2000)* (Biarritz, France, 3-6 octobre 2000, 2000), ESTIA, pp. 85-93.
2. Baron, M. and Girard, P. Bringing Robustness to End-User Programming, in *Proc. 2001 IEEE Symposia on Human-Centric Computing Languages and Environments* (Stresa, Italy, September 5-7 2001, 2001), Entergraphica, pp. 142-149.
3. Spivey, J.M. *The Z notation: A Reference Manual*. Prentice Hall Int., 1988.
4. Bjorner, D. VDM a Formal Method at Work, in *Proc. VDM Europe Symposium'87* 1987), Springer-Verlag, pp. .
5. Abrial, J.-R. *The B Book: Assigning Programs to Meanings*. Cambridge University Press, 1996, 779 p.
6. ClearSy. Atelier B - version 3.5 in 1997.
7. Lieberman, H. *Your Wish is my command*. Morgan Kaufmann, 2001, 416 p.
8. Scapin, D. and Bastien, J.-M.C. Analyse des tâches et aide ergonomique à la conception : l'approche MAD* (chapitre 3) in *Analyse et conception de l'I.H.M. / Interaction Homme-Machine pour les S.I. vol.1, edited by C. Kolski.* Hermès Science, 2001. Vol. 1,
9. Paternò, F. *Model-Based Design and Evaluation of Interactive Applications*. Springer, 2001, 208 p.
10. Paternò, F., Mancini, C. and Meniconi, S. ConcurTaskTrees: A Diagrammatic Notation for Specifying Task Models, in *Proc. IFIP TC13 human-computer interaction conference (INTERACT'97)* (Sydney, Australia, 1997), pp. 362-369.
11. Hix, D. and Hartson, H.R. *Developping user interfaces: Ensuring usability through product & process*. John Wiley & Sons, inc., Newyork, USA, 1993.
12. Szekely, P. Retrospective and challenge for Model Based Interface Development in *Eurographics Workshop on Design, Specification, and Verification of Interactive Systems (DSV-IS'96), edited by F. Bodart and J. Vanderdonckt.* Springer-Verlag, 1996. pp. 1-27.
13. Szekely, P., Sukaviriya, P., Castells, P., Muthukumarasamy, J. and E. Salcher. Declarative interface models for user interface construction tools : the MASTERMIND approach, in *Proc. IFIP TC2/WG2.7 Working Conference on Engineering for Human-Computer Interaction (EHCI'95)* (Grand Targhee Resort (Yellowstone Park), USA, 14-18 August, 1995), Chapman & Hall, pp. 120-150.
14. Puerta, A. and Eisenstein, J. Interactively Mapping Task Model to Interfaces in Mobi-D, in *Proc. Eurographics Workshop on Design, Specification and Validation of Interactive Systems (DSV-IS'98)* (Abingdon, UK, 3-5 June, 1998), Proceedings, pp. 261-274.
15. Biere, M., Bomsdorf, B. and Szwillus, G. The Visual Task Model Builder, in *Proc. Third Conference on Computer-Aided Design of User Interfaces (CADUI'99)* (Louvain-la-neuve, Belgique, 21-23 October, 1999), Kluwer Academic Publishers, pp. 245-256.
16. Ousmane, S. *Spécification comportementale de composants CORBA*. PhD Université de Toulouse 1, Toulouse, 2001, 201 p.
17. Pribeanu, C. and Vanderdonckt, J. Exploring Design Heuristics for User Interface Derivation From Task and Domain Models, in *Proc. Computer-Aided Design of User Interfaces (CADUI'2002)* (Valenciennes, France, May 15-17, 2002), Kluwer Academics, pp. 103-110.