F. Jambon, "From Formal Specifications to Secure Implementations," in *Computer-Aided Design of User Interfaces (CADUI'2002, Valenciennes, France)*, C. Kolski and J. Vanderdonckt, Eds. Kluwer Academics, 2002, pp. [to be published].

From Formal Specifications to Secure Implementations

Francis Jambon
LISI-ENSMA
BP 40109, 86961 Futuroscope cedex, France
e-mail: Francis.Jambon@ensma.fr
http://www.lisi.ensma.fr/members/jambon/

Abstract: This paper proposes a new tool-supported technique for the complete development of safety-critical interactive systems from the specification to the implementation step. Safety as well as usability properties are continuously guaranteed during the development process. This technique relies on formal specifications of the requirements and so uses the model-oriented formal method B and a new ad-hoc software architecture model –CAV– which is an hybrid of MVC and PAC models. At the implementation step, this technique uses automatic code generation. Moreover, links from secure generated code to native non-secure libraries are clarified. This development process is illustrated by a fully implemented case study.

Key words: Safety-critical interactive system, formal specification, B method, development process, usability property.

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. Usually, safety-critical systems act in the real-word, i.e., they control the most of hardware –for example an aircraft engine. This represents the major cause of their safety-critical characteristic. As a consequence, most of the functional core of these systems are asynchronous, i.e., the system state –for example the engine rotation speed– can be altered independently of the user actions: a typical characteristic of process-control systems.

The verification of safety as well as of usability properties of software can be achieved by tests or proofs. For asynchronous systems, the former technique is rather difficult to set up: the needed number of test sequences to get a full cover may be not computable. That explains why the latter technique –proofs– is a relevant one for the verification of safety-critical interactive systems. In previous works, the adaptation of well-defined approaches, combined with interactive models, brought partial but positive results. For example, we can find the Interactors and related approaches [1, 2], model-oriented approaches [1], algebraic notations [3], Petri nets [4] or Temporal Logic [5]. These techniques enable designers to prove some safety as well as usability requirements. However, these formal techniques are used in the development process in restricted conditions due to two constraints:

– They mostly rely on ad-hoc specification models –e.g. Interactors– and do not use well-known software architecture models such as MVC or PAC. The consequence is that these unusual models make the specification task to most interactive systems designers uneasy to achieve, and the refinement to implementation difficult to carry out.
– Few of them can preserve formal semantics of the requirements from the specification to the implementation steps. Most of them can prove usability properties at the specification level only. So, it cannot be proved that the final software is exactly what has been specified.

Our development process uses the B formal method [6]. On the one hand, compared to VDM and Z, the B method permits the definition of a constructive process to build whole applications, respecting of all the rules by the use of a semi-automatic tool [7]. Our first results [8] focus on low-level interaction mechanisms, such as mouse and windows control and use the Arch software architecture model [9]. Recent results detail the use of the new CAV software architecture model for the specification of process-control interactive systems [10]. We so demonstrated that the B method might be used with profit in interactive system specification. This paper shows that the B method and the CAV software architecture model can be used to specify and **thoroughly to implement** a safety-critical interactive

system. Moreover, a tool –Atelier B– is used to prove automatically all the proof obligations and to generate C code. This work may be considered as a new step towards the definition of an actual tool-supported interactive development process based on formal approaches.

The paper is organized as follows: in section 2, our case study –the battery control panel– is presented. Section 3 focuses on the specifications and explains how the safety and usability requirements can be formally checked. The B method and the CAV architecture model are also briefly described. Then, in section 4, the implantation of the case study is detailed, and at last, there is a discussion about used programming philosophies.

2. CASE STUDY

The case study is a control panel for a set of three rechargeable batteries. The aim of this system is to produce continuous electric power supply. This system may be a part of a more complex safety-critical system as avionics. The operator is in charge of the selection of the switched battery.

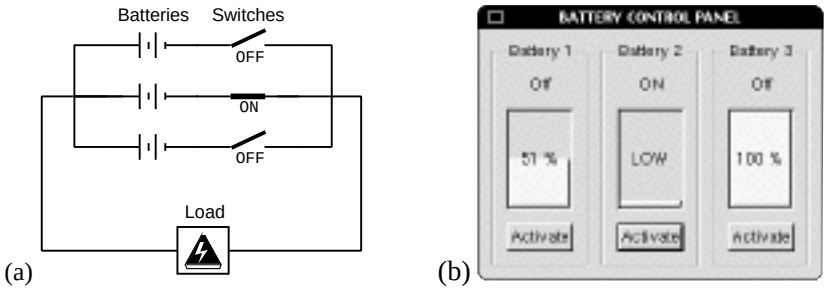

Figure 1. Electric diagram (a) and interface layout (b) of the electric power supply

The functional core of the system is a set of three batteries and three switches. As shown on figure 1-a, the load can be supplied by any battery. In order to prevent short-cuts between batteries, only one switch must be in position ON at the same time. Moreover, to ensure power supply continuity, at least one switch must be in position ON. For the same reason, an empty battery cannot be switched on. These three requirements of the functional core deals with the safety of the system.

The operator interprets the system state via the positions –ON or OFF– of the switches and the levels of the batteries –in percentage of maximum. The operator can select the switched battery via three push-buttons. Figure 1-b shows the user interface layout. When a switched battery level is below 10% of its initial capacity, the percentage is replaced by a LOW warning message.

The user interface must ensure conformity, i.e., the interface state must display exactly the functional core state. This last requirement deals with the usability of the system.

This case study is an elementary safety-critical process-control system: the operator can control side-effects on hardware –the switches– whereas the hardware state –the batteries levels– is altered asynchronously. Both safety and usability properties have to be ensured. We will reach such results thanks to the formal method B.

3. SPECIFICATIONS

This first required step of our design process consists in modeling the battery control panel requirements with the B language. Three kinds of requirements must be fulfilled:

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

3.1 The Control-Abstraction-View Model

Classic software architecture models as PAC or MVC are not compliant with the 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 refers to [10] for a more detailed description of the model. Briefly explained, the model uses the external strategy of MVC: the outputs of the system are devoted to a specific module –the View– while inputs are concerned by another one –the Control– that also manages symmetrical inputs from the reactive system, that is directed by the third module –the Abstraction. Like PAC, the Control module synchronizes and activates both the View and the Abstraction modules in response to both user and functional core system events, assuming though its role of dialogue controller (center of figure 2).

3.2 Specifications Base

In the B method [6], abstract machines represent the specifications. The typical B abstract machine starts with the keyword MACHINE and ends

with the keyword END. A set of clauses can be defined in between. In our case study, these clauses appear in the following order:

– INCLUDES is a programming in the large clause that allows to import instances of other machines. Every component of the imported machine becomes usable in the current machine. The included machine(s) can be renamed by a prefix. This clause allows modularity capabilities.
– VARIABLES is the clause where all the attributes of the described model are represented. Variables may be abstract or concrete, i.e. the latter can be directly implemented, the former not.
– INVARIANT clause describes the properties of the attributes defined in the VARIABLES clause. The logical expressions described in this clause remain true in the whole machine and they represent assertions that are always valid.
– OPERATIONS clause 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 operation. The substitutions used in our specifications and their semantics are described below.

Applying our CAV software architecture model to the case study is straightforward: each module corresponds to a B abstract machine that encapsulates few attributes in its VARIABLE clause. The *AbsrBatt* abstract machine models the Abstraction of the system –the batteries– and the variables of this abstract machine are the switches positions and the batteries levels. These variables are altered through two operations: one to switch on a battery and one to update the levels:

```
MACHINE
  AbstBatt
ABSTRACT_VARIABLES
  switch_batt1, level_batt1, ...
OPERATIONS
  changeBattery (batt) = ...
  updateLevels (level1, level2, level3) = ...
END
```

The *ViewBatt* abstract machine models the View –the interface layout– and the variables of this abstract machine represent the state of each widget: the batteries levels, the switches positions, the states –enabled or disabled– of the warning messages LOW and pushbuttons (fig. 1-b). Operations of the machine update the switches positions and the batteries levels:

```
MACHINE
  ViewBatt
ABSTRACT_VARIABLES
  display_level1, display_low1, display_on1, display_button1, ...
OPERATIONS
  showOn (batt) = ...
  showLevels (level1, level2, level3) = ...
END
```

The *CtrlBatt* abstract machine models the Control –the dialogue controller– of the user interface. This abstract machine includes instances of the View and of the Abstraction. The variables of the controller refers to the validity of the events from the real word, i.e., if a button is disabled an event is not supposed to be generated by this button. Operations of the controller are events coming from the batteries or the user: the update of the batteries levels and the activation of one of the pushbuttons.

```
MACHINE
  CtrlBatt
INCLUDES
  abst.AbstBatt, view.ViewBatt
ABSTRACT_VARIABLES
  evt_valid1, ...
OPERATIONS
  evtLevels ( level1, level2, level3 ) = ...
  evtButton ( nb ) = ...
END
```

The above specifications are incomplete: the INVARIANT clauses and the operations bodies have not been specified yet. Now we are going to complete some of them while specifying the usability and safety properties of the system.

3.3 Usability Properties

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 must be specified in the INVARIANT clause of the View:

```
INVARIANT
  ((display_low1=TRUE) ⇔ (display_on1=TRUE   display_level1≤10))   ...
```

The operations of the View must be specified to fulfill this invariant. For the *showLevels* operation, the specification below means that for any new value of the variables, they must satisfy the invariant, whatever the way they are computed:

```
OPERATIONS
  showLevels (level1, level2, level3) =
  ...
  ANY low1, low2, low3 WHERE
     ((display_on1=TRUE   level1≤10) ⇔ (low1=TRUE))   ...
  THEN
     display_low1 := low1 || ...
  END || ...
```

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 and has not been described in the paper for the explanation would be too long.

3.4 Safety Properties

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 the operations.

In the Abstraction abstract machine this safety requirement is specified in the PRE...THEN... substitution of the *changeBattery* operation. This substitution means that the operation is defined only if the parameter is not an empty battery. At this point, the specification sets the safety requirements up. As a consequence, the B language semantics ensure that the operation will never be called with an empty battery as parameter.

```
OPERATIONS
  changeBattery (batt) =
  PRE
     batt   1..3
     ((batt=1) ⇒ (level_batt1≠0)) ∧ ...
  THEN
     ...
  END
```

In fact, we have delegated to the Control abstract machine –that includes the Abstraction– the 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 call this technique the *delegation of safety*. This generates two consequences:

1. The operator cannot be aware of 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 proofs 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 an 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 and so even if the user interface is defective.

```
MACHINE
  CtrlBatt
INVARIANT
  ((abst.level_batt1 = 0)    (view.display_button1 = FALSE))
  (¬(abst.level_batt1 = 0)    (view.display_button1 = TRUE))
  ((abst.level_batt1 = 0) ⇔ (evt_valid1 = FALSE))
  (¬(abst.level_batt1 = 0) ⇔ (evt_valid1 = TRUE))   ...
OPERATIONS
  evtButton ( nb ) =
  PRE
     nb   1..3
  THEN
     SELECT
        ((nb=1)   (evt_valid1=TRUE))   ...
     THEN
        abst.changeBattery(nb) || view.showOn(nb)
     END
  END ; ...
END
```

The total specification of the abstract machines is too long to be reproduced in this paper, but the two latter requirements are characteristic examples of the specification problems we had to deal with. Now we focus on the second step of the development process: the implementation.

4. IMPLEMENTATION

This section details the second step of our development process, i.e., the implementation of the control panel from the formal specifications carried out in the previous section. The final objective of our study is to design a working program. So, on the one hand we must implement secure code from the formal specifications but on the other hand we need to link this code to existing software libraries because it is not realistic –if ever possible– to specify a complete system with formal methods.

4.1 Implementation of the CAV Model

The final program must be a set of software modules in which some of them are formally specified and implemented, and some other are developed with classic software engineering methods. In order to dissociate these two antagonists 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 2. We now focus on these three types of modules: secure code, interface and native modules.

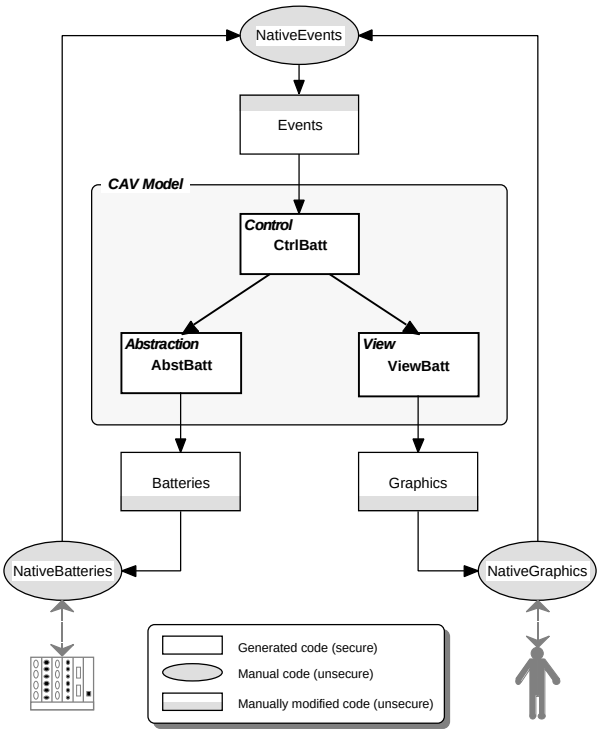

Figure 2. The CAV software architecture with interfaces and native modules

4.2 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 machines operations do respect the invariant as well as the abstract machines operations they refine. Providing the proof obligations are actually proved, the implementation machines respect the safety and usability requirements. So, the code is **secure** whether the specifications are

adequate. As an illustration, we reproduce below the implementation of the Control operation *evtButton* specified in the previous section. Note that the SELECT...WHEN... non determinist specification is implemented by a determinist classic CASE...EITHER....OR... instruction.

```
IMPLEMENTATION
  CtrlBatt_i
REFINES
  CtrlBatt
IMPORTS
  abst.AbstBatt, view.ViewBatt
CONCRETE_VARIABLES
  evt_valid1, ...
OPERATIONS
  evtButton ( nb ) =
  BEGIN
     CASE nb OF
        EITHER 1 THEN IF (evt_valid1=TRUE) THEN
           abst.changeBattery(1) ; view.showOn(1) END
        ...
     END
  END ; ...
END
```

4.3 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 do 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 modules 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.

4.4 Programming Philosophies

At last, the project outcome is a set of C source files. Some of these files are automatically generated from the BØ implementations, 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 do 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*. One the one hand the syntactic filtering is done by the Event module that casts the parameters types –from integer to interval. On the other hand, the semantic filtering is achieved by the Control module that can refuse events coming from disabled buttons. So, the system is resistant to graphic libraries bugs or transient errors with sensors. This filtering is required by the proof obligations that force upon the operations 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.

CONCLUSION

A previous work tends to show that the B method can specifies WIMP interactive systems [8] or process-control interactive systems [10]. This

paper now shows that we successfully achieved the complete development of a safety-critical process-control interactive system from formal specifications. We proved that the system respects the requirements that deals with safety and usability. Moreover, this case study shows how the Control-Abstraction-View software architecture model can be successfully implemented. Some points may be enlightened:

– Safety and usability requirements may be specified by invariant or operation preconditions of B abstract machines.
– Incremental development from specifications to implementation of an interactive system can be achieved in a reasonable laps of time.
– Links between secure and insecure code are mandatory in the design of working programs and require interfaces.

This work is a new step toward an actual method for specifying, designing and implementing interactive systems with the formal method B and the use of an industrial tool. The next step is to overcome the lack of secure graphic libraries. That is the reason why we are currently developing a formal graphical widget library.

REFERENCES

1. Duke, D.J. and Harrison, M.D. Abstract Interaction Objects. *Computer Graphics Forum*. 12, 3 (1993), pp. 25-36.
2. Paternò, F. A Theory of User-Interaction Objects. *Journal of Visual Languages and Computing*. 5, 3 (1994), pp. 227-249.
3. Paternò, F. and Faconti, G.P. On the LOTOS use to describe graphical interaction in . Cambridge University Press, 1992. pp. 155-173.
4. Palanque, P. *Modélisation par Objets Coopératifs Interactifs d'interfaces homme-machine dirigées par l'utilisateur*. PhD Université de Toulouse I, Toulouse, 1992, 320 p.
5. Brun, P. XTL: a temporal logic for the formal development of interactive systems in *Formal Methods for Human-Computer Interaction, edited by P. Palanque and F. Paternò.* Springer-Verlag, 1997. pp. 121-139.
6. Abrial, J.-R. *The B Book: Assigning Programs to Meanings*. Cambridge University Press, 1996, 779 p.
7. ClearSy. Atelier B - version 3.5, 1997.
8. Aït-Ameur, Y., Girard, P. and Jambon, F. Using the B formal approach for incremental specification design of interactive systems in *Engineering for Human-Computer Interaction, edited by S. Chatty and P. Dewan.* Kluwer Academic Publishers, 1998. Vol. 22, pp. 91-108.
9. Bass, l., Pellegrino, R., Reed, S., Sheppard, S. and Szezur, M. The Arch Model : Seeheim revisited, in *Proc. User Interface Developper's Workshop* 1991).
10. Jambon, F., Girard, P. and Aït-Ameur, Y. Interactive System Safety and Usability enforced with the development process, in *Proc. Engineering for Human-Computer Interaction (EHCI'01)* (Toronto, Canada, May 11-13, 2001), PREceedings, pp. 61-76.