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.

Interactive System Safety and Usability Enforced with the Development Process

Francis JAMBON, Patrick GIRARD and Yamine AÏT-AMEUR[1]

Laboratory of Applied Computer Science
LISI/ENSMA, BP 40109
86961 Futuroscope cedex, France
jambon@ensma.fr, girard@ensma.fr, yamine@supaero.fr
http://www.lisi.ensma.fr/ihm/index-en.html

Abstract. This paper introduces a new technique for the verification of both safety and usability requirements for safety-critical interactive systems. This technique uses the model-oriented formal method B and makes use of an hybrid version of the MVC and PAC software architecture models. Our claim is that this technique –that uses proofs obligations– can ensure both usability and safety requirements, from the specification step of the development process, to the implementation. This technique is illustrated by a case study: a simplified user interface for a Full Authority Digital Engine Control (FADEC) of a single turbojet engine aircraft.

1. Introduction

Formal specification techniques become regularly used in the area of computer science for the development of systems that require a high level of dependability. Aircraft embedded systems, the failure of which may cause injury or death to human beings belong to this class.

On the one hand, user-centered design leads to semi-formal but easy to use notations, such as MAD [1] and UAN [2] for requirements or specifications, or GOMS [3] for evaluation. These techniques could 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, gives partial but positive results. Among them, we find the interactors and related approaches [4, 5], model-oriented approaches [4], algebraic notations [6], Petri nets [7] or temporal logic [8, 9]. Thanks to these techniques, some safety as well as usability requirements may be proved.

Nevertheless, theses formal techniques are used in the development process in a limited way because of two constraints:

[1] Yamine AÏT-AMEUR is now Professor at ENSAE-SUPAERO, ONERA-DTIM, 10 av. Edouard Belin, BP 4032, 31055 Toulouse cedex, France.

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.

- Formal techniques mostly depend on ad hoc specification models –e.g. interactors– and do not concern well-known software architecture models as Arch, MVC or PAC. As a consequence, these unusual models make the specification task hard to use by most user interfaces designers.
- Few of these formal techniques can preserve formal semantics of the requirements from the specification to the implementation steps. Most of them can prove ergonomic properties at the specification level only. So, it cannot be proved that the final software is exactly what has been specified.

This article focuses on the B method [10, 11]. On the one hand, compared to VDM and Z, it makes possible the definition of a constructive process to build whole applications, with the respect of all the rules by the use of a semi-automatic tool [12]. On the other hand, the interactive system can be specified with respect to well-known software architecture models as Arch [13]. In this paper, we will show how the B method can be used to specify a critical system, the FADEC user interface case study, and how dependability as well as user interface honesty can be proved.
This work may be considered as a new step towards the definition of an actual interactive development method based on formal approaches. Our first results [13, 14] focus on low-level interaction mechanisms, such as mouse and window control. We showed that the B method might be used with profit in interactive development. Our aim in this article is to apply the method on critical systems for two main reasons. First, we believe that critical systems are applications of primary importance for safe methods. In addition, critical systems introduce special needs in terms of flow of control. So, we focus on two main points: (1) how can the specification of both critical systems and the B method influence software architecture –e.g. how B method constraints can be interpreted into well known HCI approaches– and (2) what are the benefits of using the B method for the specification process of critical systems.
The paper is organized as follows: in section 2, the B method is presented, and some previous results in applying formal approaches in HCI context are briefly summarized. In section 3, a study upon architecture models suitable for both critical systems and the B method is detailed. Last, the fourth section describes the specification of the case study and explains how the safety and usability requirements can be formally checked.

2. The B Method and Interaction Properties [14]

The B method allows the description of different modules, i.e., abstract machines that are combined with programming in the large operators. This combination enables designers to build incrementally and correctly –once all the proof obligations are proved– complex systems. Moreover, the utmost interest in this method, in our case, is the semi-automatic tool it is supported by.

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.

2.1. The Abstract Machine Notation

The abstract machine notation is the basic mechanism of the B method. J.-R. Abrial defined three kinds of machines identified by the keywords MACHINE, REFINEMENT and IMPLEMENTATION. The first one represents the high level of specification. It expresses formal specification in a high abstract level language. The second one defines the different intermediate steps of refinement and finally the third one reaches the implementation level. Do note that the development is considered to be correct only when every refinement is proved to be correct with respect to the semantics of the B language. Gluing invariant between the different machines of a development are defined and sets of proof obligations are generated. They are used to prove the development correctness.
A theorem prover including set theory, predicates logic and the possibility to define other theories by the user, achieves the proof of these proof obligations. The proving phase is achieved either automatically, by the theorem prover, or by the user with the interactive theorem prover. The model checking method, which is known to be often overwhelmed by the number of states that are needed to be computed is not used in the present version of the B tool [12].

2.2. Description of Abstract Machines

J.-R. Abrial described a set of relevant clauses for the definition of abstract machines. Depending on the clauses and on their abstraction level, they can be used at different levels of the program development. In this paper, a subset of these clauses has been used for the design of our specifications. We will only detail these clauses. A whole description can be found in the B-Book [10]. The typical B machine starts with the keyword MACHINE and ends with the other keyword END. A set of clauses can be defined in between. In our case, 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. This clause allows modularity capabilities.
- USES has the same modularity capabilities as INCLUDES except that the OPERATIONS of the used machines are hidden. So, the imported machine instances cannot be modified.
- SETS defines the sets that are manipulated by the specification. These sets can be built by extension, comprehension or with any set operator applied to basic sets.
- VARIABLES is the clause where all the attributes of the described model are represented. In the methodology of B, we find in this clause all the selector functions which allow accessing the different properties represented by the described attributes.
- INVARIANT clause describes the properties of the attributes defined in the clause VARIABLES. The logical expressions described in this clause remain true in the whole machine and they represent assertions that are always valid.
- INITIALISATION clause allows giving initial values to the VARIABLES of the corresponding clause. Do note that the initial values must satisfy the INVARIANT clause predicate.

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.

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

Other syntax possibilities are offered in B, and we do not intend to review them in this article, in order to keep its length short enough.

2.3. Semantics of Generalized Substitutions.

The calculus of explicit substitutions is the semantics of the abstract machine notation and is based on the weakest precondition approach of Dijkstra [15]. Formally, several substitutions are defined in B. If we consider a substitution S and a predicate P representing a postcondition, then [S]P represents the weakest precondition that establishes P after the execution of S. The substitutions of the abstract machine notation are inductively defined by the following equations. Do notice that we restricted ourselves to the substitutions used for our development. The reader can refer to the literature [10, 11] for a more complete description:

[SKIP]P   P **(1)**

[S1 || S2]P   [S1]P   [S2]P **(2)**

[PRE E THEN S END]P   E   [S]P **(3)**

[ANY a WHERE E THEN S END]P   a (E   [S]P) **(4)**

[SELECT P1 THEN S1 WHEN P2 THEN S2 ELSE S3 END]P **(5)**
(P1   [S1]P)   (P2   [S2]P)   ((¬P1   ¬P2)   [S3]P)

[x:=E]P   P(x/E) **(6)**

The substitution (6) represents the predicate P where all the free occurrences of x are replaced by the expression E. Do notice that when a given substitution is used, the B checker generates the corresponding proof obligation, i.e., the logical expression on the right hand side of the operator "   ". This calculus propagates a precondition that must be implied by the precondition set by the user. If not, then the user proves the precondition or modifies it. For example, if E is the substitution [x+1] and P the predicate x   2, the weakest precondition is x   1.

2.4. Interaction Properties

Proving interaction properties can be achieved by the way of *model checking* or *theorem proving* [16]. Theorem proving is a deductive approach to the verification of

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.

interactive properties. Unless powerful theorem provers are available, proofs must be made "by hand". Consequently, they are hard to find, and their reliability depends on the mathematical skills of the designer. Whereas model checking is based on the complete verification of a finite state machine, and may be fully automated. However, one of the main drawbacks of model checking is that the solution may not be computed due to the high number of states [16]. The last sessions of EHCI as well as DSV-IS show a wide range of examples of these two methods of verification.
For instance, *model checking* is used by Palanque et al. who model user and system by the way of object-oriented Petri nets –ICO– [17]. They argue that automated proofs can be done to ensure first there is no cycle in the task model, second a specific task must precede another specific task (*enter_pin_code* and *get_cash* in the ATM example) and third the final functional core state is the final user task (*get_cash* and *get_card*). These proofs are relative to reachability. Furthermore, Lauridsen uses the RAISE formalism to show that an interactive application –functional core, dialogue control and logical interaction– can be built using translations from the functional core adapter specification [18]. Then, Lauridsen shows that his refinement method can prove interaction properties as predictability, observability, honesty, and substitutivity.
In the meantime, Paternó and Mezzanotte check that unexpected interaction trajectories expressed in a temporal logic –ACTL– cannot be performed by the user. The system –a subset of an air traffic control application– is modeled by interactors specified with LOTOS [19]. Brun et al. use the translation from a semi-formal task-oriented notation –MAD– [1] to a temporal logic –XTL– [8] and prove reachability [20].
Our approach in this article –with the B method– deals with the first method, i.e., *theorem proving*. Yet, the method does not suffer from the main drawbacks of theorem proving methods, i.e., proving all the system “by hand”. In our former studies [13, 14], about 95% of the proofs obligations, regarding *visibility* or *reachability*, were automatically proved thanks to the “Atelier B” tool. Our present work –the FADEC user interface specification– has been successfully and fully automatically proved. All proof obligations, regarding *safety* and *honesty*, have been distributed in the separate modules of the system specification, as we will see later on. Moreover, since the specification is incrementally built, the proofs are also incrementally built. Indeed, compositionality in B ensures that the proofs of the whole system are built using the ones of the subsystems. This technique simplifies considerably the interaction property verifications. And then, this incremental conception of applications asserts that the proofs needed at the low-level B-machines of the application, i.e. the functional core, are true at the higher levels, i.e. the presentation. So, the *reliability* is checked by construction.

3. The FADEC Case Study and its Software Architecture

A FADEC (Full Authority Digital Engine Control) is an electronic system that controls all the crucial parameters of aircraft power plants. One of the system roles is

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.

to lower the cognitive load of pilots while they operate turbojet engines, and to reduce the occurrence of pilot errors.
Our case study focuses on the startup and the shutdown procedures of a single turbojet engine aircraft. In our scenario, the pilot controls the engine ignition –off, start, run– and the fuel valve –closed, open. The engine states can be perceived via the fuel pressure –low, normal, high– and the engine temperature –low, normal, high. The system interface is composed of lights and push buttons. The interface layout adopts the *dark cockpit philosophy* which minimizes distracting annunciation for pilots, i.e. only abnormal or transition states are visible. So, the normal parameters of the engine during flight do not light up any interface lights.
In this section, we start with an analysis of constraints imposed by the B language over architecture design. Secondly, we explain why "pure" MVC and PAC models fail against B requirements. Lastly, we describe our hybrid model, named CAV.

3.1. Rules for B Architecture Design

Our case study belongs to the safety-critical interactive-system category. More precisely, as in common interactive systems, the user controls the software system, but, as a reactive system, a third part, that evolves independently from both the software system and the user, must also control the system. In first approximation, the software may be modeled as a unique view that displays some functional core under a specific interactive control. Our first idea for designing such a system was to use a well-known multi-agent model, such as MVC or PAC, 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:

1. 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 form a tree.
2. 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. So, they are not allowed on the abstract machines specifications.
3. 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.

3.2. Classical Multi-Agent Architecture Models

As we explained in the upper section, our first impulse was to apply directly a classical multi-agent approach to our problem. Nevertheless, we discovered rapidly that none of them could be used without modification. In this section, we briefly describe MVC and PAC architecture models and relate how they are inappropriate.

MVC is an acronym for "Model, View, Controller". It is the default architecture model for the Smalltalk language [21], and became the first agent-based model for

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.

HCI. This model splits the responsibility for user interface into autonomous agents that communicate by messages, and are divided into three functional perspectives: *Model* stands for application data, and their access. It is the only object that is allowed to communicate with other *Model* objects. *View* is in charge of graphical outputs. It gives the external representation of the domain data, using *Model* objects services to extract the data to be presented. It also presents the perceivable behavior of the agent. This separation between *Model* and *View* allows a *Model* to own several *Views*. Lastly, *Controller* is responsible for inputs, and for making perceivable the behavior of the agent. It also manages the interactions with the other *Controllers*.
When the user gives some input, the associated *Controller* triggers a *Model* function. Then, the *Model* sends a message to all its *Views* to inform them for a change. Each *View* may request for the new state of the *Model*, and can refresh the graphical representation if needed.
The main problem with MVC is the double link between the *View* and the *Model*. This point violates the first rule we identified, which concerns B abstract machine inclusion order. More precisely, the *Model* cannot access the *View* –for sending it a message– if the *View* must ask the *Model* for data to be visualized.

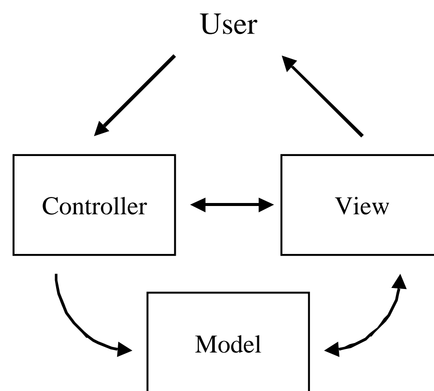

Fig. 1. The three components of the Model-View-Controller software architecture model

The **PAC** model, for "Presentation, Abstraction, Control" was proposed in 1987 by J. Coutaz [22]. Opposed to MVC, it is absolutely independent from languages. Agents are made of facets, which express complementary and strongly coupled computational services. The *Presentation* facet gives the perceivable input and output behavior of the object. The *Abstraction* facet stands for the functional core of the object. As the application itself is a PAC agent, no more component represents the functional core. There is no application interface component. The *Control* facet insures coherency between *Presentation* and *Abstraction*. It solves conflicts, synchronizes the facets, and refreshes the states. It is also takes charge of communication with other agents –their *Control* facets. Lastly, it controls the formalism transformations between abstract and concrete representations of data.

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.

The PAC model gives another dimension as interactive objects are organized in hierarchies. Communication among the *Control* facets in the hierarchy is precisely defined. Modification to an object may lead its *Control* facet to signal this modification to the *Control* facet of parent object, which may in turn communicate this modification to its siblings. This allows all the parts of the application to correctly refresh. We believe that this precise point –i.e. the honesty property– may be directly addressed in a B development. It is the basis for the refinement steps we intend to conduct.

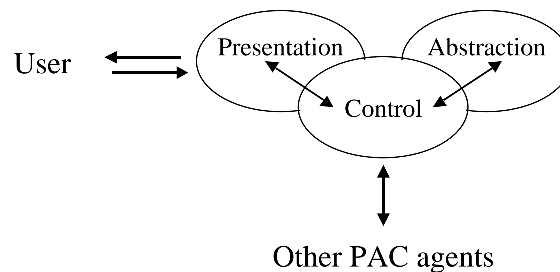

Fig. 2. The three facets of the Presentation-Abstraction-Control software architecture model

PAC solves the problem of MVC, because the *Control* facet is the only responsible for synchronizing the other two facets. Unfortunately, PAC does not respect the third rule on a unique entry point. In classical interactive systems, the unique entry point is the user. So, the *Presentation* facet may be considered as the unique entry point of the program. But, in safety critical systems, the reactive system itself is another external entity that must be taken into account. For that purpose, the presentation facet does not seem to be a good candidate.

3.3. The Hybrid CAV Model (Control-Abstraction-View)

We propose an hybrid model from MVC and PAC to solve this problem. The 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 aircraft events, though assuming its role of control.
To limit exchanges between control and the two other components, a direct link is established between the *View* and the *Abstraction*, to allow the former to extract data from the latter. This point is particularly important in the B implementation of this model, because of the second rule we mentioned upper, that enforces to pay particular attention to synchronization problems between machines. This last point is mainly discussed in the following section.

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.

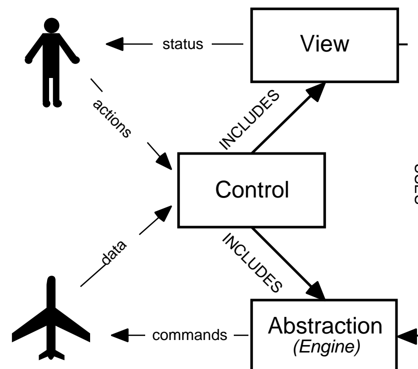

Fig. 3. The three components of the Control-Abstraction-View software architecture model

4. The FADEC User Interface Specification in B

In this section, we detail the development process we used for the case study, and we focus on HCI requirement satisfaction. The first step consists in modeling the FADEC, using the CAV architecture model we described in the previous section. The second step concentrates on safety requirements, that concerns inputs. The third step pays attention to honesty property, mainly outputs. Lastly, we illustrate the expressive power of the B method with iterative resolution of side effects on our specification.

Two kinds of requirements must be fulfilled:

- The system must be safe, i.e. the pilot must not be able to damage the engine. For example, the fuel pressure must be *normal* in order to begin the startup sequence.
- The system must be honest, i.e. the user interface lights must reflect the exact engine parameters –pressure and temperature– at any moment.

4.1. Modeling the FADEC User Interface

Applying our architecture model to the FADEC case study is straightforward. Each B machine encapsulates few attributes.
The *Engine* abstract machine –the *Abstraction*– models the functional core of the FADEC, e.g. the engine control parameters in the VARIABLE clause, and the variation sets of them in the SETS and INVARIANT clauses. The SETS are defined in respect to the logical description of the system, and a unique “SetProbeData” is defined for both the fuel pressure and the engine temperature:

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.

MACHINE
Engine
SETS
SetIgnition = {*off*, *start*, *run*} ;
SetFuelValve = {*open*, *closed*} ;
SetProbeData = {*low*, *normal*, *high*}
VARIABLES
Ignition , *FuelValve* , *FuelPress* , *EngineTemp*
INVARIANT
Ignition   *SetIgnition*
FuelValve   *SetFuelValve*
FuelPress   *SetProbeData*
EngineTemp   *SetProbeData*   ...

The *View* abstract machine models what the pilot can perceive from the user interface, e.g. the lights and their status (TRUE for on, and FALSE for off). Because of the dark cockpit philosophy, we chose to use two lights for reflecting either ignition state, pressure or temperature that have three different states, and only one for fuel valve status which has only two different states. Moreover, the *View* abstract machine uses an instance of the *Engine* abstract machine in order to be aware of its sets of variables. It is expressed by a USES clause:

MACHINE
View
USES
engine.Engine
VARIABLES
IgnitionOff, *IgnitionStart*, *FuelValveClosed*,
EngineTempLow, *EngineTempHigh*, *FuelPressLow*, *FuelPressHigh*,
StartButtonEnabled
INVARIANT
IgnitionOff, *IgnitionStart*   BOOL×BOOL
FuelValveClosed   BOOL
EngineTempLow, *EngineTempHigh*   BOOL×BOOL
FuelPressLow, *FuelPressHigh*   BOOL×BOOL
StartButtonEnabled   BOOL   ...

The *Control* abstract machine is the centralized control of the system. So, it does not need to define any functional core nor presentation variables which are already defined in the *Engine* and *View* abstract machines respectively. On the other hand, it must include the sets, the variables and the operations of both instances of *Engine* and *View*:

MACHINE
Control
INCLUDES
engine.Engine, *view.View*

The sets, the variables and some of the invariants of the three abstract machines are now precisely defined. We can focus on the INVARIANT clauses that ensure safety.

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.

4.2. Safety Requirements

The first requirement of the FADEC is safety. For instance, the start mode must not be used if the fuel pressure is not *normal*. This property must always be satisfied. In B, this requirement may be enforced with an INVARIANT clause that applies on the variables *Ignition* and *FuelPress*. The *Engine* abstract machine which represents the system functional core is responsible for it, with the following B expression:

$$\neg (Ignition = start \wedge FuelPress \neq normal)$$

We do not pay attention to what action is done. We only focus on the fact that never abnormal fuel pressure may be observed when startup is processing. In the semantics of B, the invariant must equal true at the initialization of the abstract machine, at the beginning and at the end of any operation. Note that the substitutions of the initialization as well as the operations are assumed to be executed in parallel.
Of course, the startup operation of the engine must satisfy this invariant, so the operation is guarded by an ad-hoc precondition PRE that ensures the operation will never be used if the fuel pressure is different from *normal*:

```
startup =
        PRE     FuelPress = normal
        THEN    Ignition := start
        END ;
```

For HCI, what is interesting now is: *Is the user able to make an error?* Whatever the user does, the B specification of the engine machine ensures that it will not be possible to violate the invariant. What about user actions and user interface state now? In our architecture model, the *Control* abstract machine is responsible for user inputs and for functional core actions activation. As a consequence, it must include an instance of the *Engine* abstract machine, and must use its operations within their specifications. Using B allows propagating the conditions. A new guard can/must be set for the operation used when the pilot presses the startup button. We do not give the engine the responsibility for controls, we propagate this semantic control to the user interface. One basic solution is to guard the activation of the startup button in the *Control* abstract machine by the precondition *engine.FuelPress = normal*:

```
start_button_pressed =
        PRE     engine.FuelPress = normal
        THEN    engine.startup ||
                …
        END ;
```

This basic solution suffers from two main drawbacks:

- The redundancy of preconditions is needed by modular decomposition of the B abstract machines
- The pilot's action on the system is blocked without proactive feedback, i.e. the pilot can press the startup button and nothing happens.

A more clever design is to delegate the safety requirements of the functional core to the user interface. In this new design, the user interface startup button is disabled

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.

while the startup sequence cannot be initiated for safety reasons. Now the user interface is in charge with the safety requirements of the system. Two modifications are needed:

- The *startup* operation of the *Control* abstract machine is now guarded by the state of the startup button:

view.StartButtonEnabled = **TRUE**

- The invariant must ensure that the startup button is disabled when the fuel pressure is not normal and enabled otherwise:

((*engine.FuelPress* = *normal*   *view.StartButtonEnabled* = **TRUE**) (*engine.FuelPress*   *normal*   *view.StartButtonEnabled* = **FALSE**))

The B semantics –and the *Atelier B* tool– checks for the validity of these assertions, and ensures for the compatibility of all abstract machines operations. Our software is now assumed not to allow the pilot doing anything wrong that can damage the turbojet engine.

4.3. Usability Requirements

Honesty is a well known property in user interfaces [23]. In safety critical systems, system honesty is crucial because user actions depend on the user capacity to evaluate the state of the system correctly. This point assumes the displayed state **is** the state of the actual system. Ensuring user interface honesty requires the specification to prove that the system state –represented by *Engine* variables– is always reflected in the pilot interface –represented by *View* variables. Like safety requirements, this requirement stands for an *always true* invariant.
It seems conspicuous that the honesty property must be stipulated in the INVARIANT clause of the *View* abstract machine. However, updates of the *Engine* and *View* variables are achieved in parallel by the operations of the *Control* abstract machine, because of the B semantics constraints –quoted in the §3.1. As a result, it is impossible to get the *Engine* variables update **before** the *View* variables update. As a consequence, the honesty property must be stipulated in the INVARIANT clause of the *Control* abstract machine only. For example, the light *FuelValveClosed* must be *on* only when the fuel valve is closed. We can express it by an exhaustive invariant that gives the two right solutions:

((*engine.FuelValve* = *closed*   *view.FuelValveClosed* = **TRUE**) (*engine.FuelValve*   *closed*   *view.FuelValveClosed* = **FALSE**))

Our software architecture model assumes that the *Control* abstract machine really acts. So, the operation of the *Control* abstract machine, which is used when the pilot presses the close button, i.e. the *close_fuel_button_pressed* action, must update both the *Engine* state and the *View* state:

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.

```
close_fuel_button_pressed =
    BEGIN
            engine.close_fuel ||
            view.update_ui(     engine.Ignition, closed,
                                engine.FuelPress, engine.EngineTemp,
                                view.StartButtonEnabled      )
    END ;
```

Another consequence is that the operation of the *View* abstract machine must properly update the variable:

```
update_ui (ignition, fuel_valve, fuel_press, engine_temp, start_button_enabled) =
    PRE
            ignition    SetIgnition
            fuel_valve    SetFuelValve
            engine_temp    SetProbeData
            fuel_press    SetProbeData
            start_button_enabled    BOOL
    THEN
            ANY fvc WHERE
                    fvc    BOOL
                    ( (fuel_valve = closed)     (fvc = TRUE) )
                    ( (fuel_valve    closed)     (fvc = FALSE) )
            THEN
                    FuelValveClosed := fvc
            END || ...
    END
```

4.4. Specificity of Asynchronous Systems

In critical systems such as the FADEC, some parts of the system change without any interaction with the user. For example, a probe whose control is obviously outside of the system updates the fuel pressure. A real-time kernel is in charge of interrogating every probe and sending responses to the whole system. In our analysis, the real-time kernel is out of our scope. Nevertheless, its entry point into our system must be defined. We propose to manage the aircraft in a way that is symmetric to the user. As stated in figure 1, it is controlled by the *Engine* abstract machine. When events are initiated by the aircraft power plant, their entry point is the *Control* abstract machine. So doing, the *Control* is completely in charge of updating the internal state –functional core– and the external state of the application –view. More, it can also ensure that the state of interaction is also correct.

Because we were focusing on HCI, we did not really pay attention to this side of the application. In our sense, one of the most interesting result of our study is the following: the B method helped us to discover a hazardous but important side-effect. The automatic prover detected a problem with fuel pressure invariant as soon as we introduced the action that updates this pressure into the functional core: on the one hand, the fuel pressure must be normal during the startup sequence, otherwise, the pilot cannot press the start button. On the other hand, if the fuel pressure falls down

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.

when the *Engine* abstract machine is in start mode, the turbojet engine must stop. We did not take this case into account. Fortunately, the B semantics does not allow this. Therefore, the *Engine* abstract machine must be enhanced:

```
update_fuel_press (fuel_data) =
        PRE         fuel_data     SetProbeData
        THEN
                    FuelPress := fuel_data ||
                    SELECT Ignition = start    fuel_data    normal
                    THEN     Ignition := off
                    END
        END ;
```

As a result, the *Control* abstract machine must update the *View* accordingly:

```
fuel_press_event (data) =
        PRE data     SetProbeData
        THEN
                    SELECT engine.Ignition = start     data     normal THEN
                                engine.update_fuel_press (data)          ||
                                view.update_ui(     off, engine.FuelValve,
                                                    data, engine.EngineTemp,
                                                    FALSE    )
                    ELSE
                                engine.update_fuel_press (data) ||
                                view.update_ui(     engine.Ignition, engine.FuelValve,
                                                    data, engine.EngineTemp,
                                                    TRUE     )
                    END
        END ;
```

5. Conclusion

A previous work [13, 14] show that the B language can be used to specify WIMP interactive systems and ensure usability properties. This work shows that the B method can enforce safety and usability in a process-control interactive system with asynchronous behavior. Moreover, this study covers specification and design topics: we define a new software architecture model, that allows an actual instantiation with the B method, and we describe an empiric method for modeling safety-critical interactive systems in B. Four points may be enlightened:

- safety and usability are ensured by using invariant that are relatively easy to find from the non formal description of the system behavior,
- incremental development can be achieved with this method, which is particularly suitable in HCI domain,
- using the B tool is very helpful for ensuring specification completeness, as we discovered during our analysis,
- the modules and then the whole specification may be completely validated thanks to the prover, in a whole automated way.

At the end of the last step, the specification must be considered as safe respect with to the requirements –safety and usability.

This work is a second step towards a real method for specifying, designing and implementing interactive systems with the B method. The perspectives are numerous. First, we need to use the B refinement theory to implement the specifications we realized. This step, which has already been initiated, will lead us to pay a particular attention to the connections with user interface toolkits. Then, a more exhaustive study must be accomplished to evaluate what properties may be enforced using the B language, and what properties cannot. This will then allow us to design a safety critical system with the collaboration of a set formal methods to avoid limitations among each methods.

Acknowledgements

We thank Hervé BRANLARD and Joël MITARD for their documentation about operational FADEC procedures, and Sylvie FERRÉS for the English review.

References

1. Scapin, D.L. and Pierret-Golbreich, C. Towards a method for task description : MAD in *Working with display units, edited by L. Berliguet and D. Berthelette.* Elsevier Science Publishers, North-Holland, 1990. pp. 371-380.
2. Hix, D. and Hartson, H.R. *Developping user interfaces: Ensuring usability through product & process*. John Wiley & Sons, inc., Newyork, USA, 1993.
3. Card, S., Moran, T. and Newell, A. *The Psychology of Human-Computer Interaction*. Lawrence Erlbaum Associates, 1983, 280 p.
4. Duke, D.J. and Harrison, M.D. Abstract Interaction Objects. *Computer Graphics Forum*. 12, 3 (1993), pp. 25-36.
5. Paternò, F. A Theory of User-Interaction Objects. *Journal of Visual Languages and Computing*. 5, 3 (1994), pp. 227-249.
6. Paternò, F. and Faconti, G.P. On the LOTOS use to describe graphical interaction in . Cambridge University Press, 1992. pp. 155-173.
7. Palanque, P. *Modélisation par Objets Coopératifs Interactifs d'interfaces homme-machine dirigées par l'utilisateur*. PhD Université de Toulouse I, Toulouse, 1992, 320 p.
8. 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.
9. Abowd, G.D., Wang, H.-M. and Monk, A.F. A Formal Technique for Automated Dialogue Development, in *Proc. DIS'95, Design of Interactive Systems* (Ann Arbor, Michigan, August 23-25, 1995), ACM Press, pp. 219-226.
10. Abrial, J.-R. *The B Book: Assigning Programs to Meanings*. Cambridge University Press, 1996, 779 p.
11. Lano, K. *The B Language Method: A guide to practical Formal Development*. Springer, 1996.
12. Steria Méditerranée. Atelier B in 1997.

13. Aït-Ameur, Y., Girard, P. and Jambon, F. A Uniform approach for the Specification and Design of Interactive Systems: the B method, in *Proc. Eurographics Workshop on Design, Specification, and Verification of Interactive Systems (DSV-IS'98)* (Abingdon, UK, 3-5 June, 1998), Proceedings, pp. 333-352.
14. 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.
15. Dijkstra, E. *A Discipline of Programming*. Prentice Hall, Englewood Cliff (NJ), USA, 1976.
16. Campos, J.C. and Harrison, M.D. Formally Verifying Interactive Systems: A Review, in *Proc. Eurographics Workshop on Design, Specification and Verification of Interactive Systems (DSV-IS'97)* (Granada, Spain, 4-6 June, 1997), Springer-Verlag, pp. 109-124.
17. Palanque, P., Bastide, R. and Sengès, V. Validating interactive system design through the verification of formal task and system models, 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. 189-212.
18. Lauridsen, O. Systematic methods for user interface design, 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. 169-188.
19. Paternò, F. and Mezzanotte, M. Formal verification of undesired behaviours in the CERD case study, 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. 213-226.
20. Brun, P. and Jambon, F. Utilisation des spécifications formelles dans le processus de conception des Interfaces Homme-Machine, in *Proc. Journées Francophones sur l'Ingénierie de l'Interaction Homme-Machine (IHM'97)* (Poitiers-Futuroscope, 10-12 septembre, 1997), Cépaduès Éditions, pp. 23-29.
21. Goldberg, A. *Smalltalk-80: The Interactive Programming Environment*. Addison-Wesley, 1984.
22. Coutaz, J. PAC, an Implementation Model for the User Interface, in *Proc. IFIP TC13 Human-Computer Interaction (INTERACT'87)* (Stuttgart, September, 1987), North-Holland, pp. 431-436.
23. Gram, C. and Cockton, G. *Design Principles for Interactive Software*. Chapman & Hall, 1996, 248 p.