Embedding Ergonomic Rules as Generic Requirements in a Formal Development Process of Interactive Software

Philippe Palanque, Christelle Farenc & Rémi Bastide
LIHS, University Toulouse 1, Place Anatole France, 31042 Toulouse Cedex, France.
{farenc,palanque,bastide}@univ-tlse1.fr

Abstract: This paper presents a formal framework for the development of interactive software that bridges the gap between ergonomic knowledge and software design. It builds upon previous work on formal notations and proposes an integrated development process from requirements to model-based execution. It also embeds ergonomic knowledge in requirements, and proposes a way to formally represent them and to prove their fulfilment over a detailed formal specification of the interactive software.

Keywords: formal notations, development process, ergonomic rules, model-based systems.

1 Introduction

The usability and reliability requirements in the design of Interactive Systems are usually investigated in separate and limited ways. There is a lack of structured methods and related tools that can drive the work of designers and developers to take into account these two requirements in a complementary way. We believe that formal specification techniques can help to take into account the reliability requirement, while task modelling and ergonomic rules are a concrete way to cope with usability.

For several years we have been working on the building and the use of formal specification techniques for the design and the implementation of interactive applications (Bastide & Palanque, 1990). A formal specification technique called ICO (Palanque & Bastide, 1995) is now available and we are now focusing on the case tool supporting it (Bastide & Palanque, 1995b). This work is directly dealing with the reliability requirement, which cannot be met, as far as interactive systems are concerned, without the mathematical tools underlying formal specification techniques.

It is now widely agreed that usability is also a key issue in the design of interactive systems. One of the tools for meeting usability requirement is to take into account users' activities while designing the system. Task modelling (through task analysis) is a way to represent these activities and we have shown in previous work that activity can be captured using formal notations. This can be done with formal specification techniques or by translating task models built using dedicated notations such as the User Action Notation (Hartson & Hix, 1989) to formal notations (Palanque et al., 1995).

Using formal notations for task modelling provides a number of benefits that are otherwise very hard to reach. For instance, we have shown in (Palanque et al., 1997) the possibility to assess complexity and performance of a given system with respect to its potential use (described in the task models).

The benefits increase significantly when formal notations are used for modelling both the activity and the system. In this case the gap between task models and system models can be bridged and properties (such as compatibility and conformance) between the two kind of models can be proven. Compatibility (i.e. the set of actions represented in the task model is included in the set of actions offered by the system model) deals with what a system offers to the user. Conformance (i.e. the temporal relationship between the actions described in the task model is compatible with the temporal evolution described in the system model) deals with how this is offered to the user.

Another way for increasing the usability of interactive software is the use of Human Factors knowledge expressed in terms of ergonomic rules. Ergonomic rules come from a number of different
sources: recommendation papers (Smith, 1988), design standards (Organization, 1992), style guides which are specific to a particular environment (Corporation, 1993), design guides (Scapin, 1986; Vanderdonckt, 1994) and algorithms for ergonomic design (Bodart & Vanderdonckt, 1994).

An ergonomic rule can be considered as a principle that has to be taken into account for the building or the evaluation of user interfaces (UI) in order to respect cognitive and sensory-motor capabilities of users. These principles can have a general scope, or can be tailored to a specific ‘context of use’ dependent on tasks, user models, user environment, location, organizational aspects, etc.

Most of times, ergonomic rules are used during the evaluation phase of the development process of interactive application. Sometimes, however, they are used in the design phase, either explicitly as in most of the User Interface Development Environments (UIDE) or explicitly in research tools such as TRIDENT (Bodart et al., 1994).

However, during the design phase, ergonomic rules are often taken into account only implicitly (i.e. hidden in a supporting tool) or as generic guidelines and thus do not receive the same treatment as other requirements of the application. This specific use of ergonomic rules make cumbersome both design and validation phase when requirements are to be studied.

In this paper we focus on bridging the gap between ergonomic rules and the actual design of interactive systems. Section 2 proposes a design process that integrates the various components introduced above in order to take into account both usability and reliability requirements. Section 3 shows how to represent explicitly ergonomic rules as requirements by means of a temporal logic. The claim here is not to represent them exhaustively but to show that some of them can be represented this way, and that this representation can then be exploited afterwards in the design process. Section 4 exemplifies the design process presented in Section 2 on a case study. The last section shows how the ergonomic rules represented as requirements are used in the validation phase.

2 Development Process

Early software development models (including waterfall and V models) focussed mainly on the identification and the clear separation of the various phases of the development of software systems. However, the way they represent the process (i.e. in a linear and mainly one-way structure) is very limited and not able to deal with prototyping issues. Boehm (1988) introduced the spiral development model to deal explicitly with prototyping and iteration. Prototyping is an essential issue in the development of interactive systems and thus this model has been widely adopted.

However, this model is incomplete with respect to the various models that have to be built during the development of an interactive system. For instance, it does not refer to task modelling and usability evaluation, that are now recognized as critical for the design of usable interactive systems. Research has been also conducted in this field, and the star model (Hartson & Hix, 1989) explicitly introduces task analysis and usability evaluation as phases of the development process. However, most of the phases are generally conducted without computer support for representing and analysing the produced models. This is not critical when dealing with ‘classical’ interactive systems, but each manual operation may be source of a fatal error when safety critical systems are concerned.

The development process of a software system is highly iterative (as for the spiral model) and presents various phases (definition of informal requirements, specification, design, coding, user testing) that require human creativity and intervention. An important aspect is that each phase produces several models corresponding to the top-down process of taking into account information in a stepwise refinement manner. When dealing with interactive systems it is now widely accepted that user information has to be taken into account and that this must be done through task analysis and task modelling. In this way, user goals have to be analysed as part of the specification phase, while task analysis is conducted during the design phase.

Concerning the integration of formal methods, we argue that, although the phases involving creativity remains unchanged, the use of formal methods provides benefits in the design process.

If formal methods are used during the design process, the coding phase can be at least partly automated for instance by means of code generation. This automation can also be done by the interpretation at run-time of the models built in the earlier phases, following a model-based approach (Wilson et al., 1993; Szekely et al., 1993). We have previously investigated the pros and cons of these two approaches in (Bastide & Palanque, 1995a).

The advantage of using formal notations to support the design is the potential for mathematical verification they provide. A formal model (whether it describes a task or a system) may be automatically checked for several important properties such as deadlock freedom or termination. If, at a given stage,
the model fails to comply with such properties, it has to be reworked and corrected until the problem is solved. This process is illustrated in Figure 1.

![Figure 1: Design process of the various models within a phase using formal notations.](image)

In order to cope with all the issues and thus deal both with safety and usability requirements that are crucial for safety critical interactive applications, we propose an iterative development process based on formal notations.

### 2.1 A New Development Process

The solution we propose to this end relies on four principles:

1. We propose a formal notation that spans both system modelling and task modelling. This brings to task modelling the advantages of formal approaches, the most important of which are conciseness, consistency and lack of ambiguity. This also makes task models amenable to mathematical verification.

2. The fact that both task and system models are constructed within the same formal framework enables us to consistently merge task and system models. This, in turn, allows checking that task and system models comply with each other, and moreover enables us to perform quantitative analysis on the task/system merger in order to check whether the models comply with preplanned objectives in terms of complexity and timing.

3. We represent requirements using a dedicated formal notation different from the one used for task and system modelling. This allows taking advantage of each notation and, as we provide bridges between them, it is possible to check their mutual conformity. In order to take full advantage of the benefits of the use of formal specification techniques, it is necessary to follow a rigorous development process rooted on mathematically founded notations. Temporal logics belong to the category of declarative languages. This class of languages provides abstract specifications, as they are adequate for describing ‘what’ a system does. This is significantly different from procedural languages such as process algebra (Hoare, 1985) or Petri nets (Murata, 1989) that cannot describe ‘what’ a system does without describing at the same time ‘how’ this is done. This is a reason why we have decided to use two different formalisms for the modelling of single system: a temporal logic for the description of the properties of the system and a dialect of Petri nets for the description of its behaviour.

4. We propose a development process supporting the use of formal notations for requirements, task, and system modelling. This process (see Figure 2) instantiates the development process described in Figure 1 as a sub-process to support formal modelling and extends it to explicitly represent cross verification between the three models.

![Figure 2: The design life cycle using both tasks and system models.](image)

The development process in Figure 2 might start by some initial requirements description, by some rough model of the system (which may originate from the existing situation, e.g. analysis of the paper documents in an information system analysis) or with an initial task model. The initial model is then submitted to the formal sub-life-cycle of Figure 1. The aim of the formal analysis is to check whether the model is ‘defect free’ or not. This ‘defect free’ characteristic varies according to the model under consideration. For instance, in the system model it means absence of deadlocks, in the task model it
means termination and in the requirement model it means the absence of conflicts.

A higher-level design loop is then initiated, where (for each iteration) task models are built in accordance with the current system model and both the complexity and the performance of these models is quantitatively analysed. The system designers propose modifications in the system models, in order to allow simpler and more efficient task models to be built.

Formal analysis that looks for defects aims at checking whether designers build the system right, qualitative analysis aims at checking whether they build the right system i.e. a system corresponding to users’ needs as expressed in both task and requirements models. In our development process, qualitative analysis aims at checking that the system model is conformant to both task model and requirement model. So, each time a new system model is produced, the qualitative analysis checks whether or not this model still meets the requirements and is still consistent with the task model. If not, one or several models are modified and the verification process is carried out again.

This design loop may be undertaken successfully only if the analysis of the task models yields precise and quantitative results, that may be checked against pre-planned objectives. We have devised a set of metrics that can be applied to Petri net models. Those metrics are based on well established Petri net analysis techniques (e.g. the checking of place and transition invariants), but also on weightings applied to the various components of the net (places, transitions, arcs, inscriptions, functions), that will allow us to quantitatively assess the complexity of the analysed tasks (Palanque et al., 1997).

3 Ergonomic Rules as Requirements

Not all the ergonomic rules can be taken into account as generic requirements; Some are too specific or too close to some kind of interaction technique (Farenc et al., 1996).

For these reasons we have only selected some ergonomic rules that are both generic to interaction techniques and application domains. However, we are currently investigating most of the documents gathering ergonomic rules in order to classify them, to select the ones that can represented as generic requirements and to formally describe them.

We have selected rules belonging to two different classes: dynamic behaviour and interface objects behaviour.

Dynamic Behaviour

Rule 1: Any command must have a visible result.

Rule 2: When the result of a process activated by a command is displayed in more than 2 seconds and less than 6 seconds the mouse pointer must become a waiting pointer (hourglass, clock).

Rule 3: When the result of a process activated by a command is displayed in more than 6 seconds a progress indicator box must be displayed.

Rule 4: When a command might lead to loss or modification of data or to a long process a message box requiring confirmation of the command must be displayed (warning message).

Interface Objects Behaviour

Rule 5: Any user action on an Interactive Object must be graphically displayed.

Rule 6: Any message box must remain displayed until an explicit user action.

Rule 7: Action or warning message box display must be associated with a sound feedback.

3.1 Formal Description of Ergonomic Rules

In order to represent ergonomic rules in a precise and non-ambiguous way we have decided to use a specific temporal logic called ACTL. Temporal logics (Emerson & Srinivasan, 1988) allow for the description of qualitative temporal aspects of systems. In order to describe these temporal aspects, specific temporal operators have been added with respect to other modal logics. While modal logics permit the description of properties over a given state, temporal logics aims at describing them over an execution path, i.e. a set of states. For this purpose linear time temporal logics propose basically four operators: always (G), eventually (F), next (X) and until (U). Branching time temporal logics allow describing and reasoning about non-deterministic systems amongst which interactive systems belongs to (Dix, 1990). Non determinism is thus dealt with by means of representation of several alternative futures in the evolution of a system. For this propose, ACTL (that belongs to the category of branching time temporal logics) provides two other operators:

\[ \forall \] for all the possible paths
\[ \exists \] it exist at least one possible path
One of the problems in using ‘classical’ temporal logics is the difficulty to relate the properties described in terms of states with the behaviour the system described in terms of actions (that cause state changes). Several logics have been built in order to allow describing properties both in terms of states and in terms of actions. ACTL (DeNicola & Vaandrager, 1990) is one of these, chosen for its expressiveness and its potential for automated model checking (DeNicola et al., 1993).

The use of a formal notation for representing ergonomic rules presents a double advantage:

- First, it is possible to describe in a concise and non-ambiguous way ergonomic knowledge and thus making both its use and understanding easier,
- Second, it is possible to use verification tools for checking whether or not a given system is conformant with a set of selected rules. Some tools (Löwgren & Nordqvist, 1992; Kolski & Millot, 1991) used ergonomic rules in order to verify this conformance, but this verification is realized too late, i.e. during the evaluation phase of the development process when the interface presentation is developed. The use of ergonomic rules as requirements provides a verification before the development of the interface presentation and consequently minimize the number of iteration in the development process.

For instance, using ACTL, Rule 1 can be formally expressed as follows:

Let Commands be the set of commands that an application can perform, CVisible the set of system commands with a visible result, and UCommands the set of command that can be initiated by the user, such that CVisible ⊆ Commands:

∀ c ∈ UCommands, ∃ c’ ∈ CVisible,
∀ G[execute(c)][∀X<execute(c’)>true)

This can be read as follows:
For all the possible futures (∀ operator), it is always true (G operator) that if the user command c is executed, then for all the possible futures (∀ operator) in the next state (X operator) a system command c’ (with visible result) is executed.

4 Case Study: The ATM

This section’s aim is to demonstrate how to model a software prototype of an Automated Teller Machine (ATM), using the Interactive Cooperative Objects (ICO) formalism. The ATM offers a keyboard for entering the pin number and devices for entering the card, getting the card back and getting the cash. The ATM allows the user to change the amount selected if it is greater than the one allowed by its credit card. The entering of the pin number (number of trials allowed) is not described as it do not add to the specification.

4.1 Modelling the ATM

The ATM is modelled as an ICO class, featuring attributes, services, a formal behavioural specification (the Object Control Structure, or ObCS) and a complete presentation part. The description of the class, specifying the first three components, may be seen in Figure 3. In this software prototype, the interface consists of user interface widgets such as push buttons, whereas the final system would use specialized physical devices.

According to the Arch architecture, the dialogue is modelled by a high level Petri net (HLPN). The main difference with basic Petri nets is that this formalism allows coping with data structure and data values in the models (as for example the precondition in transition T7). In this kind of Petri nets, tokens are references to other objects of the system. This formalism provides a concise, yet formal and complete specification for the control structure of the application as each feature is formally described in the mathematical foundation of Petri nets. The modelling power of the formalism allows describing a lot of constraints, otherwise hard to describe in natural language.

Class ATM
 Attributes
 a, b: Real;  
c: Card;  
r: {CANCEL, RETRY};
 Methods
 Ok <a, b: real>; Boolean;  
Avail <c: Card>; Real;  
Visible commands
 DisplayCardIn, DisplayPin, DisplayAmount,  
DisplayCashCardIn, DisplayWindow,  
DisplayCardRemoved, DisplayCashRemoved
 Sound commands
 Alert
 Services
 InsertCard <c: Card>;  
Select <a: Real>;  
GetCash; GetCard;
 ObCS (see Figure 4)  
Presentation

Class ATM
Activation function

<table>
<thead>
<tr>
<th>Widget</th>
<th>User’s actions</th>
<th>Service</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pushbutton InsertCard</td>
<td>Click</td>
<td>InsertCard</td>
</tr>
<tr>
<td>Pushbutton £20</td>
<td>Click</td>
<td>Select</td>
</tr>
<tr>
<td>Pushbutton £40</td>
<td>Click</td>
<td>Select</td>
</tr>
<tr>
<td>Pushbutton £160</td>
<td>Click</td>
<td>Select</td>
</tr>
<tr>
<td>Pushbutton Card</td>
<td>Click</td>
<td>GetCard</td>
</tr>
<tr>
<td>Pushbutton Cancel</td>
<td>Click</td>
<td>User Cancel</td>
</tr>
<tr>
<td>Pushbutton Retry</td>
<td>Click</td>
<td>User Retry</td>
</tr>
</tbody>
</table>

Rendering

<table>
<thead>
<tr>
<th>ObCS’s transitions</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>InsertCard</td>
<td>DisplayCardIn</td>
</tr>
<tr>
<td>Enter Pin</td>
<td>DisplayPin</td>
</tr>
<tr>
<td>Select</td>
<td>DisplayAmount</td>
</tr>
<tr>
<td>T4</td>
<td>DisplayCashCardAvail</td>
</tr>
<tr>
<td>T5</td>
<td>DialogWindowAlert</td>
</tr>
<tr>
<td>T8</td>
<td>DisplayCardRemoved</td>
</tr>
<tr>
<td>T9</td>
<td>DisplayCashRemoved</td>
</tr>
<tr>
<td>Others</td>
<td>No specific Rendering</td>
</tr>
</tbody>
</table>

Layout: Main and Dialog Windows

Figure 3: The ICO class ATM.

A transition of a HLPN may occur if each of its input places holds at least one token, so that each variable labelling an input arc may be bound to an object. When a transition occurs, the objects bound to input variables are removed from the input places and their values are processed by the transition’s action that may also generate or delete objects. The new or modified objects are finally set into the output places, according to the variables labelling the arcs and thus fully stating the flow of references of objects in the net.

Behaviour of the System

The current state of the ObCS of the ATM (Figure 4) is fully determined by the distribution and the value of tokens (black dots in the places) in the various places of the net. The current marking corresponds to the presentation in Figure 3, where card, pin number and amount have been entered. The amount entered is 100£. According to the current state in the ObCS, only transitions T8 and T9 are available. This is rendered in the user interface by showing the corresponding buttons (this is stated in the activation function Figure 3) are available. This shows how it is possible to describe with the ICO formalism multi-threaded dialogues. Indeed, if those two services have required several actions from the user, the user would have been able to handle them concurrently. As the transition T4 deposits one token in both places P7 and P8 this describes a production of parallelism in the model.

Figure 4: ObCS of the ATM Class.

At the opposite the transition T10 (that is enabled only when the user performs both GetCash and GetCard services) describes a reduction of parallelism in the model. This is usually called a synchronization (of several flows of control).

When the transition T10 is fired, the system comes back into its initial state i.e. one basic token in the No Card place and no token in all the other places of the model.
Link with the Presentation Part of the UI
The activation function is used for presenting, at each state of the interaction, the legal actions to the user. Thus, the active or inactive state of the widgets is fully determined by the possible occurrence of the transitions they relate to in the ObCS. The fact that a transition appears in the activation function (i.e., is triggered by the action of the user on a widget) is graphically represented in the formalism using greyed out circle with broken arrows above the transition (see for instance T9).

The rendering function determines, when a transition is fired, if some presentation action must be executed in order to present information to the user. Those presentation actions can be related to transitions or to places in the ObCS. In the current case study we only relate presentation actions to transitions i.e. to user actions. In general they are often related to places i.e. to the current state of the system, as rendering is fundamentally a state-based process.

The model-based environment executing the ICOs specification at runtime automatically does these rendering activities. This environment is under development and the algorithms used have been presented in (Bastide & Palanque, 1995b).

5 Requirements Validation
The aim of this section is to show how the use of ACTL for requirement description and the ICO formalism for system and tasks description makes it possible to qualitatively assess that they are mutually conformant.

Of course, the result of this qualitative assessment can result in modifying one or several models including requirements.

ACTL formulas can be checked with system models at different levels:

- Classically, automated tools for verification are model checkers. In the case of ACTL model checking is performed over a Labelled Transition Systems such as an automaton.

- Over the ICO specification by checking rendering and activation functions.

For space reasons we focus on the principles and thus only present here the validation of two requirements over the specification presented in the previous section.

Rule 1 (see Section 3) can be easily proven over the structure of the ICO specification. Indeed, there is always a rendering performed by the system, as for each possible user action of the ObCS, there is a presentation function executed before and after the action occurs.

Rule 7 is also easy to prove, as each transition that opens a dialogue window (in this case only transition T5) is associated to the method Alert in the rendering function.

Model checking techniques are usually exploited in order to prove generic behavioural properties related to liveness and safety. For the case study presented here one such property could be re-initializability. Using model checking, ACTL properties are proven over a labelled transition system (LTS). The marking graph of the ObCS of Figure 4 is an LTS and, even without the help of an automated verification tool, its shape clearly shows the re-initializability of the ATM. Indeed, whatever the state the system is in, it always exist an execution path that leads to the initial state. For less trivial system tool support is usually mandatory.

Figure 5: The set of possible states corresponding to the ObCS in Figure 4.

6 Conclusions
This paper has presented a formal framework for the design of interactive software. Its aim is to integrate knowledge in the HCI field usually treated in an independent way. For instance, we propose a way to bridge the gap between ergonomic knowledge and software design through formal notation and verification techniques. The framework we propose requires tool support in order to ensure efficiency. The PetShop (Petri nets Workshop) environment
This workshop provides tools for the editing of the various graphical models (tasks and systems) and for the temporal logics specifications (ergonomic rules and generic requirements). Besides, tools for computer supported verification (both qualitative and quantitative) and for the run time execution of models are key parts of the environment.

Acknowledgement

The work is mainly funded by the Esprit project MEFISTO 24963 on Modelling Evaluating and Formalizing Interactive Systems using Tasks and Objects.

References


Author Index

Bastide, Rémi, 1
Farenc, Christelle, 1
Palanque, Philippe, 1
<table>
<thead>
<tr>
<th>Keyword</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>development process</td>
<td>1</td>
</tr>
<tr>
<td>ergonomic rules</td>
<td>1</td>
</tr>
<tr>
<td>formal notations</td>
<td>1</td>
</tr>
<tr>
<td>model-based systems</td>
<td>1</td>
</tr>
</tbody>
</table>