This paper aims at clarifying the articulation between the task models and system models encountered in CHI design practices. We demonstrate how the use of a formal task model may enhance the design of interactive systems, by providing quantitative results on which designers may base their decisions. We also demonstrate that it is possible to describe both task and system models within the same formal framework. This enables us firstly to formally prove that task and system models comply with each other, and secondly to perform quantitative analysis on the combination of task and system models. The approach is illustrated by a toy example which, despite its small size, allows us to develop both task and device models and to perform several iterations of the design process. The device and tasks are modelled using the Interactive Cooperative Objects (ICO) formalism, which is based on Petri nets and on the object-oriented approach. The formality of Petri nets allows for axiomatic validation of isolated and interacting subsystems.
Synergistic Modelling of Tasks, Users and Systems Using Formal Specification Techniques

Philippe Palanque† and Rémi Bastide*

This paper aims at clarifying the articulation between the task models and system models encountered in CHI design practices. We demonstrate how the use of a formal task model may enhance the design of interactive systems, by providing quantitative results on which designers may base their decisions. We also demonstrate that it is possible to describe both task and system models within the same formal framework. This enables us firstly to formally prove that task and system models comply with each other, and secondly to perform quantitative analysis on the combination of task and system models. The approach is illustrated by a toy example which, despite its small size, allows us to develop both task and device models and to perform several iterations of the design process. The device and tasks are modelled using the Interactive Cooperative Objects (ICO) formalism, which is based on Petri nets and on the object-oriented approach. The formality of Petri nets allows for axiomatic validation of isolated and interacting subsystems.

Keywords : interactive systems design, task modelling, performance evaluation, formal specification, Petri nets.

1 Introduction

Although the use of task models is gaining acceptance in the design of interactive applications [18,21], it is far from being a widespread practice. A reason for this might be the lack of agreement on how task models interact or relate with the models used in the design of the system itself (e.g. data models, object models or dynamic models). The extent to which task models may help in designing models of the behaviour of the system itself is also unclear.

This discrepancy between "anthropocentric - (people) oriented requirements specifications and computer-centric design specification" [12] has been dealt with in a series of communications in the journal "Interacting with Computers" [4,3,11]. Task analysis has been identified as a key principle in user interface design since [17]. However the very notion of task still has a number of different definitions according to the authors' perspective. Tasks can be considered at a very abstract, device-independent level [6] or on the contrary may be defined in a more concrete way, as a sequence of interaction with a given device [37]. The approach presented in this paper adopts the latter view of tasks. Moreover, our work gives much more emphasis on task modelling than on task analysis: Our work aims at providing a formal notation able to represent the results provided by a classical task analysis technique such as [10] in a concise and non ambiguous way amenable to formal verification. This paper also fits into the framework for HCI design presented by [5], stating that the overall complexity of an interactive system is partitioned between user tasks and machine tasks, and that a careful design can migrate the complexity from the task of the user to the system.

We advocate a design practice where task and system models are tightly integrated, and produced together within an iterative process. The design starts either with some initial 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 which provides the end-user requirements. Task models and system models are then evolved incrementally. After each modification the complexity of task models is quantitatively assessed. The system designers propose modifications in the system models, in order to allow building simpler and more efficient task models. Task models are built in accordance with the new system, and analysed once again. This iteration will continue until satisfactory task models are produced.

Our approach can be related to that of [35], where the logical notation of Z is used to represent both people and computer systems. By comparison to Z, our approach benefits from the executability of Petri nets and their
diagrammatic representation, and puts the emphasis on the dynamic behaviour of humans and systems rather than on their logical description. Petri nets, for example, allow for a compact and readable description of concurrency and synchronisation, which is lacking in other formal approaches such as Z.

Our proposal is significantly different from others such as [54] where models of the system are deduced or generated from task models. In our approach task and system models are not deduced from one another, but on the contrary are evolved in synergy and are checked for compliance with each other.

We extend the scope of our formal notation to encompass 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.

The fact that both task and system model are constructed within the same formal framework enables us to consistently merge task and system models. This in turn allows us to check that task and system models comply with each other (the precise signification of that is detailed later on), and moreover enables us to perform quantitative analysis on the task/system merger in order to check whether the models comply with pre-planned objectives in terms of complexity and timing.

This design process 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. Our approach to this end is to use a sound and well established formal notation to support both task and system modelling. We have chosen high-level Petri nets [65] because of their mathematical foundation, and of the huge amount of work devoted to their formal analysis. We use an object structured dialect of Petri nets (namely the ICO formalism [26]) because the object oriented features of this formalism (classification, encapsulation, inheritance) allows us to cleanly cope with the complexity of modern interactive systems. 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 invariant), 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.

2 The ICO Formalism

2.1 Introduction

A lot of work has been more or less recently done in the field of formal approaches for interactive systems [20, 24] but really little of it has been directed towards the use of formal methods for the actual construction of interactive systems. Among these our work is very similar to the one in [54] but mainly differs in the way the building of models is undertaken (they consider task models as the starting point for the design of the system models with no iteration) and on the formalism used (they use FullLOTOS instead of Object Petri nets). More precisely, our formal approach is based on the use of the ICO formalism [26]. This formalism uses concepts borrowed from the object-oriented approach (classification, encapsulation, inheritance, client/server relationship) to describe the structural or static aspects of systems, and uses high-level Petri nets to describe their dynamic or behavioural aspects. Most of the work on Objects and Petri nets has be dedicated to the embedding of object oriented concepts in Petri nets, i.e. to represent tokens in Petri nets as object classes. Using such approaches, the behaviour of a system is modelled as a single Petri net and the data structure of the system as classes flowing as token in the Petri net. The main problem with such approaches is that the Petri net describing the behaviour of the system is usually huge and thus hard to build, to understand and to modify. Using the ICO formalism, the system is modelled as a set of classes cooperating together. The behaviour of each class is modelled by an object Petri net (tokens can be object classes) and the cooperation between classes is done according to a client-server protocol formally defined in terms of Petri nets. Therefore, the behaviour of the system is fully formally expressed in Petri nets. Section 2.2 will present more in details the ICO formalism and section 2.3 will present the client server protocol that defines the communication among classes. As this formalism is aimed at modelling interactive systems, section 2.4 describes how input (based on events) and outputs (based on states) are represented in the models.

2.2 Informal presentation of the formalism

Interactive Cooperative Objects (ICO) are an object-oriented formalism dedicated to the modelling of interactive systems. ICO integrate the Object-Oriented (OO) modelling techniques such as OMT [36] and Petri Nets. The notions coming from the OO domain (such as encapsulation, inheritance, composition, association and use relationship) are used to form a structural model of the system under design, while Petri nets are used to describe the dynamic or behavioural aspects of the system. In classical OO approaches, dynamic aspects are usually modelled by some form of state machine or StateCharts. We have selected Petri nets for this purpose because of their ability to model concurrency, synchronisation and indeterminism in a clean and elegant way, and because of the huge theoretical work that has been devoted to them which provides techniques and tools for formal analysis.

ICOs were originally devised for the modelling and implementation of event-driven interfaces. An ICO model of
A system is made up of several communicating objects, where the behaviour and communication protocol of the objects are described by Petri nets.

When we extend the use of the formalism to encompass task modelling as well, the task models themselves are encapsulated by object classes, thus providing the task description with a clearly defined state and behaviour description. The "task object" communicates with system objects in the same syntactical way as system objects communicate with one another [32].

When two objects communicate, one is in the position of a "client", requesting the execution of a service and waiting for a result, while the other is in the position a "server" whose role is to execute the service. In our modelling approach, task objects (which model the behaviour of a user interacting with the system) will most often be in the position of clients (thus modelling the fact that the user applies for a service offered by the system), whereas the system objects will most often be in the position of servers (because in modern interfaces where the software is event-driven, the system is usually passive, waiting for the user to trigger an action).

The fact that the communication protocol as well as the objects' inner behaviour are both modelled in terms of Petri nets allows us to have a precise semantics for the behaviour of a system modelled by a set of communicating objects. We will make use of this feature in the section "Syntactic Consistency". In the case study presented in this paper we only have one class for describing the system but the communication will be exemplified by the showing the communication between the class representing the system model and the task modelling the use of that system model.

The interface part of a ICO class gives the list of services provided by the instances of this class, and their signature. The behaviour of the instances is given by a Petri net. This behaviour is related to the interface definition in the following way:

For each service given in the interface, the ObCS net contains one Service Input Port and one Service Output Port that are special purpose places: The service input port is meant to receive the service invocations and their input parameters, while the service output port is the channel through which the service results will be provided by the object. A service input port can only have output arcs in the ObCS, and conversely a service output port can only have incoming arcs.

The processing associated to a service is modelled by one or several macro-transitions related to the service’s input and output ports. The transitions connected to the service input port are called Accept Transitions, while the transitions connected to the output port are called Return transitions.

Figure 1 illustrates an excerpt of a ICO class definition. Only one service is described, aService, taking an integer as input parameter and returning a string. The associated behaviour is described in the ObCS: the subnet comprised between aService accept and return transitions (not detailed) is meant to compute a return string r according to the input parameter p.

```plaintext
Class aServer { // (excerpt)
  Methods
    string aService(int p);
  ObCS
```

Class aServer { // (excerpt)
  Methods
    string aService(int p);
  ObCS
The interface and ObCS together fully define the behaviour of instances: A service request will begin executing when one of its associated accept transitions is enabled by the current marking of the ObCS. Conversely, the execution of a service is modelled by the occurrence of the associated macro-transitions, which states the side-effect of the service execution on the object.

2.3 Communication between objects

The ObCS are meant to describe the "server" behaviour of objects (i.e. what their synchronisation constraints are) but also their "client" behaviour (i.e. how they request services from other Cooperative Objects in the system).

The communication between Cooperative Objects is syntactically expressed using the conventional dot notation, by actions in the ObCS transitions.

Such as service invocation is illustrated in Figure 2. The class described here acts as a client of the aServer class described in Figure 1. The ObCS descriptions includes the definition of the places type: for example, place PB is defined to hold references to instances of the aServer class, while the tokens contained in place PC will be 3-tuples holding an integer, a reference to an instance of aServer and a string.

Class aClient { // (excerpt)
Methods
// ...
ObCS // Definition of the places' type
PA : <int>
PB : <aServer>
PC : <int, aServer, string>

\[ r = s.aService( p ); \]
\[ \langle p, s, r \rangle \]
The variables on the arcs act as formal parameters for the transition. The action of the transition is to call the service \texttt{aService} on the object bound to variable \(x\), providing as a parameter the integer bound to variable \(p\). A transition whose action is the invocation of another Cooperative Object is called an \textit{invocation transition}. The default semantics for such a call is the \textit{synchronous rendezvous}, whose operational semantics can be found in [26].

The semantics of the synchronous rendezvous is given within the framework of high-level Petri nets, by enhancing the ObCS nets of both the client and the server of the rendezvous. Although the designer of the net only sees the ObCS descriptions as given in Figure 1 and Figure 2, the nets, before being actually executed in the running system, are expanded according to this formal client-server protocol:

2.4 Modelling the interaction between user and system

The ICO formalism includes special features to describe the interaction between the user and the system. Special services are distinguished in the interface of an ICO class as \textit{user services} i.e. services that can be triggered interactively by the user through the use of some input device, or some conventional user interface element. The interaction in the direction user \(\rightarrow\) system is modelled by the \textit{Activation function}, defined as follows:

- The presentation of the user interface is defined by a structured set of widgets (\texttt{Wid}) which can be constructed using the kind of graphical presentation editor found in most UIMS. Each widget is able to react to a predefined set of events (called \texttt{Evt}), triggered by the user.
- The activation function (ACT) associates to each couple (widget, user action) one and only one of the services offered by the ICO class.

\[
ACT: (Wid \times Evt) \rightarrow Serv
\]

Conversely, interaction in the direction system \(\rightarrow\) user is modelled by the \textit{Rendering Function}. While the activation function is event-driven, the rendering function is state-driven. This fact that output from the system is state based has been studied in [12] and characterised as the \textit{predictability property} of interactive systems, stating that the portion of the system state that is relevant to the user must be visible at any stage of interaction.

Thus the ICO formalism models the rendering function as follows:

- The widgets in Wid offer a predefined set of output primitives \texttt{WidOut} (such as enabling/disabling, text output, ...). The implementation of these primitives heavily depends on the programming environment.
- The rendering function Rend associates to each reachable state of the system a set of output primitives dedicated to the rendering of this state

\[
REND: \text{Marking} \rightarrow P(\text{WidOut})
\]

Petri nets do not present an explicit enumeration of all the reachable states, but on the contrary, model state by state variables (called \textit{places}) and a distribution of tokens in these places, which is called the \textit{marking} of the net. This local modelling of states allows to partition the rendering functions between the places of the ObCS nets.

3 A Simple Case Study

To demonstrate our approach, we will present a very simple case study, where the system to be designed is hardware rather than software. However, the kind of problems studied in this case study will be encountered in software design if time-driven evolution is present. Our current work addresses scalability issues, in order to make this approach viable for real life software systems, but those concerns are not presented here for space reasons and can be found in [33].

This case study aims at exemplifying the use of both tasks and system models for the design of an interactive system. As the efficiency of the system is evaluated according to the complexity of the task model associated to it, we show that small changes in the system model can significantly reduce the complexity of the task model.

3.1 The Initial System

The system available initially consists of a light bulb controlled by a timer. The timer is triggered by a button, and its period is 30 seconds. If the button is depressed while the light is on, the timer is reset and starts over counting down for 30 seconds.
Note that this presentation of the system is informal but that this information is absolutely necessary to start building a reasonably detailed model of the task. This information is the minimum that has to be provided to the task analyst in order to build a meaningful task model. If it is not provided, the task model designer will have to make assumptions about the behaviour of the system or else he/she will only be able to discuss about general goals, at a level that will not enhance further our ability to improve the system.

3.2 The System Model

The proposed system is straightforward to describe in terms of Petri nets, as shown in Figure 4. However, let's take some time in describing the Figure, in order to demonstrate various features of the ICO formalism.

Petri nets allow us to describe in a formal and concise way the dynamic behaviour of systems. The state of a system is described by tokens (represented by black dots) distributed in places (graphically represented by ellipses). The tokens' distribution can evolve according to paths made up of transitions (graphically represented by rectangles) and arcs.

```plaintext
Class InitialSystem { // (excerpt)

Services
   PressSwitch

Methods
   AutoswichOff
   ObCS // Definition of the places’ type
      LightOff : <Boolean>;
      LightOn : <Boolean>;
   // Definition of the behaviour of the class

   PressSwitch
      LightOff
      LightOn
         PressSwitch

   AutoSwitchOff
      [30]

   PressSwitch

Figure 4: The model of the initial system
```

The timer controlled light offers only one affordance, a button allowing the user to turn it on. The model of this device also highlights other primitives of our modelling approach: In the initial state of the system the light is off which is modelled by a token (a black dot) in the place LightOff. From that initial state the transition PressSwitch can be triggered by the user, removing the token from the input place LightOff and setting it in the output place LightOn.

The transition PressSwitch depicts a user service, meaning that the user has a way to trigger the occurrence of the transition through the associated input device. Graphically such transitions (called user transitions) are depicted with input and output unconnected broken arrows, which may eventually bear input or output parameters stating the data provided by the user or returned by the system. The AutoSwitchOff transition is an internal transition, not related to any user action, and performed on the system's behalf as soon as it is enabled. The arc between place LightOn and transition AutoSwitchOff is labelled with the timing inscription [30]. This means that a token staying in place LightOn is not available for the firing of transition AutoSwitchOff until it has stayed 30 time units (here, seconds) in the place. The effect of the second transition labelled PressSwitch, resets the time for the token in place LightOn by removing it and putting it back (this is graphically represented by a double arrow). This represents the resetting of the timer each time the switch is pressed. Note that two different transitions are labelled PressSwitch, which means that the button affords being pressed in two different states of the system.
3.3 The Task Description

To describe the task, we start by giving the high-level goal it aims to perform, and then detail the steps of the task in terms of actions put at the user’s disposal by the system.

In this case, the user’s goal is to maintain the light on without interruption for a period of at least 3 minutes. We want to minimise the number of actions performed by the user and thus the solution consisting in pushing and releasing the light switch at very short intervals is not considered relevant. In order for the user not to press the light switch more than 7 times the system must be upgraded as it is impossible for the user to perform the desired task without some information about the elapsing of time. In the following sections we will present two possible upgrades of the system, both of them supporting the requested task.

3.4 Upgrade 1: Adding a Stop Watch

![Figure 5: The first upgrade of the system](image)

The first possible upgrade is to provide the user with a stop-watch. Thus the system is actually made up of two unconnected subsystems, the stop-watch and the light bulb. In order to use this system, the user needs to use both his/her hands to push the light and stop-watch switches and vision in order to watch the elapsed time on the stop-watch. Both the user part and system part are represented in Figure 5.

3.4.1 The System Model

The model of the upgraded system is presented in Figure 6. Now the system is made up of two components, the TimerControlledLight and the Watch. As those two components are not related, the system model is composed of two unconnected Petri nets. The Petri net corresponding to the TimerControlledLight is thus exactly the same as in Figure 4, and another Petri net is added for describing the functioning of the Watch. The Watch offers two affordances: one button allowing the user to toggle it on and off, and the watch face itself, which allows the current time to be read, but only when the Watch is on.

The initial state of the system is described by the initial marking of each Petri net, identical to the one described in Figure 4 plus one single token in the place WatchOff. The watch affordances are modelled by transitions in the Petri net, labelled Press and CurrentTime. The transition CurrentTime provides a return value labelled \(<t>\), the current time when the watch is read.

Class FirstUpgrade{ // (excerpt)
Services
PressSwitch
Press
CurrentTime <;int>

Methods
AutoswichOff
ObCS  // Definition of the type of the places
LightOff : <Boolean>;
LightOn : <Boolean>;
WatchOff : <Boolean>;
WatchOn : <Boolean>;
// Definition of the behaviour of the class

1 Special thanks to Alan Dix for calling attention to this possible behaviour of the user
3.4.2 The Task Model for the Upgraded System

Given the system defined in the previous section, the task may be described informally as follows:

1. User presses the light button, and starts the stop-watch (in any order, and within a short time interval)
2. User measures the elapsing of 25 seconds, Actions 1 and 2 are then repeated 6 times.

The formal task model corresponding to the above natural language description is described by the high-level Petri net in Figure 7.

Figure 7 demonstrates the surprising complexity of the task associated with this seemingly simple goal. This complexity mainly stems from the fact that the user has to perform a polling loop to consult the stop-watch, and then evaluate the value he/she has just read to decide whether he/she has to press the LightSwitch button once again. This can be seen in the model by the cycle made up of place ReadyToReadTime, transition T3, place CurrentTime and transition T1.

From the initial state (modelled by one token in both NeedToStartWatch and NeedToStartLight places and seven tokens in CountDown place), the user can press both buttons in any order. After the firing of the corresponding transitions one token is removed form each input place and one token is set in ReadyLight and ReadyWatch. Then the internal transition StartCounting is triggered internally by the user meaning that from that state the user knows that the system is started and it is possible to see the current time on the watch (this is represented in the model by a token set in the place ReadyToReadTime).

Class TaskFirstUpgrade { // (excerpt)
Attributes // They are usually represented as place in the ObCS.
// However, for readability reasons they are separated here
Syst: <FirstUpgrade>; // This is a reference to the class of the system.
// It is used by the task model to request methods to the system model

Services // Usually a task does not offer any service. It only request services to the system model

Methods // no method

ObCS // Definition of the type of the places
NeedToStartLight : <Boolean>;
NeedToStartWatch : <Boolean>;
NeedToStopWatch : <Boolean>;
ReadyLight : <Boolean>;
ReadyWatch : <Boolean>;
CountDown : <int>; // the value of the integer is the number of tokens in the place
CurrentTime : <int>;
ReadyToReadTime : <int>;
// Definition of the behaviour of the class
3.5 Upgrade 2: Adding a Bell

Faced with the complexity in the user’s task presented in Figure 7, the designer might choose to improve the system, in order to reduce the user’s workload. In this case, a solution might be to move the complexity out of the task and into the system, by ringing a bell 5 seconds before the timer expires.

The user then only has to count down the number of chimings, which is significantly simpler than to compare several times the current time with the 25 seconds to be awaited before pressing the light switch.

3.5.1 The System Model

The system model is the same as the one of the initial system except that an internal transition called AutoBeep has been added and that the timing of 30 seconds has been split on two arcs 25s before the ring bells and 5 second before the light switches off.

Class SecondUpgrade{ // (excerpt)
  Services
    PressSwitch
  Methods
    AutoSwitchOff;
    AutoBeep;
  ObCS // Definition of the type of the places
    LightOff : <Boolean>;
    LightOn : <Boolean>;}
LightOnBeeping : <Boolean>;
// Definition of the behaviour of the class

![System Model Diagram](image)

Figure 9: The System Model

3.5.2 The Upgraded Task Model

With this updated system specification, a new and significantly simpler task model may be built. The count down process is modelled by seven tokens in the CountDown place. The complexity of the task model, according to our metrics, is significantly smaller, which proves that the design decision was a good one. Indeed the only cycle of the net is the one corresponding to the repetitive task of pressing the light switch several times. It can be computed that the number of occurrence of the transition HearBeep is exactly seven thus stating that the task will correspond to the minimal number of actions requested by the goal.

However, we have not demonstrated that this task is actually supported by the system i.e. that the sequence of user actions embedded in this task model can be performed by the system model given in Figure 9. This is described in the next section.

Class TaskSecondUpgrade { // (excerpt)
Attributes // They are usually represented as place in the ObCS.
    Syst: <SecondUpgrade>; // This is a reference to the class of the system.
    Services // Usually a task does not offer any service. It only request services to the system model
    Methods
        HearBeep //the user has to hear the bell chiming
    ObCS // Definition of the type of the places
        Start/Elapsed : <Boolean>;
        Waiting : <Boolean>;
        CountDown : <int>; // the value of the integer is the number of tokens in the place
        // Definition of the behaviour of the class

![Uprgraded Task Model Diagram](image)

Figure 10: The upgraded task model

}
4 Consistency of Task and System Models

When both task and system models have been built their consistency must be ensured at the lexical, syntactic and semantic levels. The use of Petri nets for the design of those models is of great help in order to check this consistency. Indeed, the same mathematical verification that is used for proving the correctness of a model can also be used to prove their mutual consistency.

4.1 Lexical Consistency

This kind of consistency is related to the demand of actions from the task model towards the system model and vice versa. For example if at some point in the task model the user has to press a push-button it is mandatory to ensure that this button exists in the system (and the system model).

This kind of consistency can be automatically proven by checking the action part of the transitions for each model. This means that if there is a service request such as Light.Pressswitch in a model (see Figure 10) there exists at least one transition labelled PressSwitch in the model of the light, and that the signature of the service and of its invocation match. This resembles the link editing process in programming languages.

4.2 Syntactic Consistency

This kind of consistency is related to the sequencing of actions supplied by the models and their related demand. Indeed, as the lexical consistency corresponds to the existence of a requested action, the syntactic one is related to the actual availability of the action at the moment when it is requested. That is to say that the system can perform any sequence of actions requested by the task model.

The analysis is based on the calculation of the languages corresponding to the task and system models. Indeed, a Petri net can be considered as an acceptor for a language in the same way that a finite state automaton can be. The supply of a system model is the language defined by its user transitions (see Figure 4) and the demand of a task model is the language defined by the transitions where a service request is made of the system (such as transition T3 in Figure 7 which contains the invocation watch.currenttime()). Syntactic consistency thus consists of proving that the demand of the task model is included in the supply of the system model. This proof can be performed automatically by building the marking graph of the Petri nets and the application of this kind of analysis to a case study can be found in [27].

According to the case study, syntactic consistency between the task model in Figure 7 and the system model in Figure 6 ensures that the sequence of services requested by the task model towards the system model fits with the sequence of services offered by the system model. For instance a task model with the service invocation t:= watch.currentTime before the service watch.Press would present a syntactic inconsistency with respect to what is offered by the system model in Figure 6.

4.3 Semantic Consistency

To ensure semantic consistency, some meaning has to be ascribed to the places of the models, and it has to be checked that this meaning is preserved in the interaction between task and system models. The semantic of the places is usually apparent from the name of the places themselves (for example StopWatchOn, StopWatchOff in Figure 6 of NeedToStopWatch, NeedToStartWatch in Figure 7). In this case, for example, we have to check that each time the place NeedToStopWatch is marked in the task model, the place WatchOn is marked in the system model. The formulas to be checked have to be given by the designer (in terms of place invariant) but the checking itself can be carried out by automatic means.

Semantic consistency can be checked after the merging of task and system models. This merging is possible because the protocol for the communication between two Petri nets is itself defined in terms of Petri nets (as discussed in section 2.3). It is therefore possible to merge the nets pertaining to the task and system models and to perform analysis on the merger. Figure 11 presents a partial reconstruction of the global models made by the combination of the task model in Figure 9 and system model in Figure 10. The transitions related to the operation of the switch (both on the user’s and the system’s side) have been expanded and linked. It is important to consider firstly that this global model may be constructed by completely automated means, and secondly that it is not meant to be read by humans, but merely to be processed by automatic tools, in order to perform analysis and prove properties. The result of this analysis on the reconstructed net allows us to be sure that the models have been built in such a way that they can cooperate together without altering their inner behaviour. For example, using Petri net theory it can be automatically proved that the reconstructed net allows the place Countdown to be emptied and that there will be no token in place LightOff before the place Countdown is actually empty, stating that the light will be kept on.
5 Formal Verification of Models

5.1 Introduction to formal verification

The use of formal specification technique for the design of an interactive system provides designers with additional features such as the formal verification of the models. This can only be done if the formalism used for building the model is based on mathematical concepts. In that case it is possible to verify properties on the system model by performing proofs and not only by empirical testing of the system after its implementation. One advantage of proving properties on the model of the system is that the cost (in terms of resource consumption) of correcting errors is significantly reduced. Besides, analysis techniques not only provide abstract information about errors in the models (the model is error free or not) but also precise information on the location of the errors.

A problem that is relevant to the discussion is the executability of the models. Indeed as the proofs are made on the model and not on the system itself one can wonder what will happen at the implementation phase. If the models are executable then the probability to introduce flaws during the implementation is much smaller than if implementation is done manually by a developer. Errors may be introduced from different reasons such as misunderstanding of the model or mistranslation of the model into the programming language.

The ICO formalisms belongs to the set of executable specification and thus these problems can be avoided provided an environment for the execution of models is available. This environment, named PetShop, is under development and the architecture can be found in [2].

The aim of the formal analysis is to prove that there is no flaw in the models. Using the ICO formalism for describing the models, the analysis is done by using the mathematical tools provided by the Petri net theory. Using those tools, one can prove general properties about the model (such as absence of deadlock) or semantic domain related properties (i.e. the model cannot describe impossible behaviour such as the Light is on and off at the same time). In [26] we have described general properties that can be proven on a formal model of an interactive system while [29] has shown in a complete example how this kind of analysis can be performed.

5.2 Formal analysis of system model

We give hereafter three properties that have been proven on the system models: the absence of deadlock, the stability of resources and the reinitialisability.

The absence of deadlock is the possibility for the user to issue another system command whatever was the sequence of previous ones. This kind of property heavily relates to the reachability property that can be refined in:

- it is always possible to perform a sequence of actions that will make a given action available and
- there is no dead branch in the model i.e. there is no action that will never be available

The stability of resources is the fact that the system is not producing or consuming resources provided at the initialisation. This can be proved by the calculation of invariants in the models related to the number of actions available at a time. For example the presence of invariants in the models can prove that in some state precisely three action is available. The stability of resource is a necessary condition for the reinitialisability of a system. However, one might be interested in representing a system producing resources in an infinite way. For instance, if the number of use of an Automated Teller Machine (ATM) is counted we can find that this number is always growing and that
no limit for this number is known. However, most of the variables describing the internal state of a system (for example the number of cards in an ATM) are bounded and remains an invariant during the use of the system, which correspond to what we call the stability of resources.

The reinitialisability is the possibility for the user to reach the initial state of the system, or a given predefined state (this is very important when the system has to used several time for the same purpose e.g. an Automated Teller Machine). This property means that whatever use of the system has been done before the same set of actions will be available for the next use of the system. This property can only be proven if the previous properties absence of deadlock and stability of resources are satisfied.

Those properties can be automatically proven on the system models of the case study. If those properties were not proven the designer would have to modify the model of the system until they are verified. The verification of deadlock-freedom is done by the calculation of repetitive components in the Petri net (see [29] for more information) which are a set of cycles of transitions. The result of the analysis can state for example that a given transition is not in any cycle, meaning that it is a dead transition. Thus the analysis itself can help the designer to improve the design by precisely stating where are the flaws in the models.

Usually "well-designed" reactive systems feature the previous properties but this is usually not the case for transformational ones. Without going in detail in the distinctions between these two kinds of systems, the latter aim at processing data and of course it is not usually interesting for these systems to be reinitialisable. In that case, proving that a property does not hold might be the aim of the formal analysis.

How to express properties is a problem by itself which is strongly connected to the formal analysis. Providing designers with techniques for describing properties enables them to describe properties related to the application itself rather than generic properties as the ones presented above. A lot of work has been devoted to this kind of problem and can be divided into two categories: model checking techniques and theorem proving techniques. These techniques aims at checking whether or not a given property expressed in a given language (usually a declarative language such as Temporal Logics) is valid on a model. We will not detail the differences between these two techniques here as details can be found in [8].

5.3 Formal analysis of task models

The properties to be proven on the task model are different from the ones on the system model. For example absence of deadlock is very important for the system as it is necessary for proving the reinitialisability. At the opposite the presence of deadlocks is very important within the task model as this property is necessary for proving the termination of the task which we usually use for proving that the goal of the user has been reached.

It can be easily proven that the task models presented in Figure 7 and in Figure 10 are not live and thus not reinitialisable. Numerous tools are available for doing this kind of analysis such as the ones presented in [14].

6 Performance Evaluation of Models

Performance analysis has been used for a long time in the formal design of systems. This kind of analysis is usually done by integrating temporal aspects in the models. Petri nets have also been used as a tool for supporting performance evaluation. For example in [88] timed Petri nets have been used for the performance analysis of network protocols.

In the area of Human Computer Interaction performance evaluation has started in the early eighties with the work presented in [8]. Building on this early work, the GOMS family has grown up and a summary of this work can be found in [23,22]. The work presented in this section is very close to the one of the GOMS family as it aims at evaluating interactive systems according to the efficiency of use of the system. However the main advantage of the work presented here is that task and system modelling are formally related. Besides, GOMS approaches lack of a representation of the system and therefore the quantitative results provided heavily depends on assumptions on the underlying system. This lack of system representation does not allow for taking into account the response time of the system that might completely jeopardise the evaluation of the system.

In this section we will present the principles of the performance evaluation of interactive systems modelled using the ICO formalism. Quantitative temporal aspects will be added both to the task model and the system model and thus we will have a precise and complete performance analysis of the couple (user, system). The performance evaluation will be used in order to compare the two versions of the system presented in the case study. It is important to notice that not only the task models will be compared (as it would have been done following a GOMS approach) but also the system models.

6.1 The user model

The model of the human processor has been proposed in the early eighties by Card, Moran and Newell in [8]. In this model the user is modelled as a set of three processors interacting together: the perceptual system, the motor system and the cognitive system.
The perceptual system is the boundary between the environment and the user. It is responsible for the inputs from the real world and is related to the body’s sensory system. The most studied sensory system is the visual one. The eye is responsible for receiving the information and the time needed to acquire the information is about 100 ms. The exact value depend on the intensity of the stimulus, on the individual and can vary from 50 ms to 200 ms.

The motor system is responsible for the physical movements of the users. As far as human computer interaction is concerned, the most interesting ones are the eye and the hand movement. There is no continuous behaviour but only a sequence of rapid elementary movements even though while the high level behaviour seems to be continuous such as in dragging an icon. The time needed to perform an elementary movement is about 70 ms. Actually it is within the range [30 .. 100] ms.

In simple interactions the cognitive system is basically serving as an interface between the perceptual system and the motor one. In more complex tasks it deals with comparison, data storage, learning, ... The average time for a cycle in the cognitive system is 70ms Actually it varies according to several parameters such as the cognitive workload, the number of items in the working memory, ... and can vary from [25...125] ms.

6.2 Quantitative Analysis

The use of a formal model for task modelling results in the same advantages as formal system modelling but, as the task model describes the sequences of actions the user will have to perform in order to reach a given goal, it is very important to compute some performance evaluation on the model. This is fully supported using stochastic or timed Petri nets as they have been used for a long time for performance evaluation of systems and a lot of theoretical work is available in this area.

The kind of analysis results that can be done on the task model is:

• number of actions the user has to perform in order to reach the goal,
• the number and the length of cycles in the models

if some actions have to be performed by the user under temporal constraints (such as entering a password within a given amount of time) it is possible to compute the frequency of those actions and to prove that the temporal constraints are consistent i.e. they do not contradict each other.

Another kind of quantitative analysis is related to the complexity of the tasks the user has to perform. This is done by automatically building the marking graph of the Petri net and computing complexity measures on it. For example the number of nodes (corresponding to the number of states in the task model) the number of actions (corresponding to the number of arcs with different labels) and the length of the path to come back to the initial state are associated with weights in order to have a quantity of complexity of the task model. This quantity is used within the design process to decide whether or not the task and system models meet the requirements.

Besides, it is possible to include in the Petri nets models the values of the human processor model described in [9] about the human performance while interacting with a computer. This is really important as it is related to the actual behaviour of the user interacting with the system. Computing performance evaluations on the tasks models labelled with these values allows us to have precise information about the cost for the user to achieve a task.

6.3 Analysis of the case study

Now we will analyse quantitatively the task models in order to evaluate the cost in term of cognitive and physical workload for the user. The analysis is performed on the two different task models in order to compare their relative efficiency. This is use in order to choose the more efficient system model among two upgrades.

6.3.1 The first task model :

In this section we present the marking graph of the task model for the first upgrade of the system. The actual marking graph is more complex than appears in Figure 12, since this figure has been graphically simplified by factoring the seven iterations needed to reach the state corresponding to the achievement of the goal of the user. This factoring is done by introducing the variable n, that decreases from seven (the initial number of tokens in the place \textit{CountDown}) to zero.

---

2 Timed Petri nets is a dialect of basic Petri nets where time is added on transitions, places or arcs of the Petri net, thus allowing to describe quantitative temporal behaviours in the models (see 28)
The following table relates transitions of the Petri net task model of Figure 7 to the time associated to user's action in the human-processor model.

<table>
<thead>
<tr>
<th>Transition</th>
<th>Processor</th>
<th>Type of action</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>Cognitive</td>
<td>Comparison of values</td>
</tr>
<tr>
<td>T2</td>
<td>Motor</td>
<td>Press button (Stop Watch)</td>
</tr>
<tr>
<td>T3</td>
<td>Motor</td>
<td>Press button (Light)</td>
</tr>
<tr>
<td>T4</td>
<td>Motor</td>
<td>Press button (Stop Watch)</td>
</tr>
<tr>
<td>T5</td>
<td>Cognitive</td>
<td>Comparison of values</td>
</tr>
<tr>
<td>T6</td>
<td>-</td>
<td>Nothing</td>
</tr>
<tr>
<td>T7</td>
<td>Perceptual</td>
<td>Look at the Stop Watch</td>
</tr>
</tbody>
</table>

According to the values associated to user's behaviour it is possible to compute the minimal time needed in order for the user to achieve the goal. The goal of the user can be expressed in terms of reachable markings in the task models. For example in the model presented in Figure 7, the goal of the user is to empty the place *Countdown*. According to the marking graph presented in Figure 12 it is straightforward to compute this minimal time.

Indeed, using the table above we associate to each transition (according to its type) the corresponding amount of time given by the human processor model (minimum, mean and max time to perform an action). The sequence of transition that have to be fired in order to perform one cycle is (see Figure 12): T3+T4 (the first two transitions in the sequence) T6+T7+T1+T2+T3+T4 (one cycle). Associating the minimum time of the processors to these transitions we obtain: T3 is motor (30ms) + T4 is motor (30 ms) and T6 nothing (0 ms)+ T7 is perceptual (50ms) + T1 is cognitive (25ms) + T2 is motor (30ms) + T3 is motor (30ms)+ T4 is motor (30ms) = 60 + 165

As the cycle must be performed 7 times in order to reach the goal, we obtain the following results:
Min Time (Goal) = 7*165 + 60 = 1205 ms
Mean Time(Goal) = 7*380 + 140= 2800 ms
Max Time(Goal) = 7*625 + 200 = 4575 ms

It must be noted that the task model (and therefore its marking graph) is non-deterministic, as the cycle T7,T5 (corresponding to the user looking at the watch before the time interval has elapsed) can be performed a number of times. The results of the performance evaluation is performed on an optimistic scenario where the user looks at the watch only once in each iteration.
6.3.2 The second task model

\[
\begin{align*}
T1 & \quad (7,1,0) \quad \rightarrow \quad (7,0,1) \quad \rightarrow \quad (6,1,0) \quad \rightarrow \quad (6,0,1) \quad \rightarrow \quad (5,1,0) \\
T1 & \quad (3,0,1) \quad \leftarrow \quad (3,1,0) \quad \leftarrow \quad (4,0,1) \quad \leftarrow \quad (4,1,0) \quad \leftarrow \quad (5,0,1) \\
T2 & \quad (2,1,0) \quad \rightarrow \quad (2,0,1) \quad \rightarrow \quad (1,1,0) \quad \rightarrow \quad (1,0,1) \quad \rightarrow \quad (0,1,0) \quad \rightarrow \quad (0,0,1) \\

\end{align*}
\]

Figure 13: The marking graph of the task model in Figure 10

The following table relates transitions of the Petri net task model of Figure 10 to the time associated to user’s action in the human-processor model.

<table>
<thead>
<tr>
<th>Transition</th>
<th>Processor</th>
<th>Type of action</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>Motor</td>
<td>Press button (Stop Watch)</td>
</tr>
<tr>
<td>T2</td>
<td>Perceptual</td>
<td>Hear the beep</td>
</tr>
</tbody>
</table>

According to the values associated to user’s behaviour it is possible to compute the minimal time needed in order for the user to achieve the goal. The goal of the user can be expressed in terms of marking in the task models. For example in the model presented in Figure 10 the goal of the user is to empty the place Countdown. According to the marking graph presented in Figure 13 it is straightforward to compute this minimal time which is: 7 firing of T2 and 8 firing of T1. As T1 is associated to the Motor system and T2 to the Perceptual one we obtain the following:

- Min Time (Goal) = (8*30) + (7*50) = 590 ms
- Mean Time (Goal) = (8*70) + (7*100) = 1260 ms
- Max Time (Goal) = (8*200) + (7*100) = 2300 ms

The result from the performance evaluation shows that even though the time spent by the user looking at the watch is not taken into account the upgraded system is twice efficient that the first system.

It is important to notice that the time presented above do not correspond to the time needed in order to reach the goal but the time spent by the user in order to perform the action needed in order to achieve the goal. Indeed, the transition T2 is associated to a timed transition in the model of the system and thus the user will be able to perform the action when the time associated to the transition is elapsed.

However, the real time needed in order to perform the task can be automatically computed using the value associated to the corresponding transition in the system model. The transition AutoBeep has to wait 25s before being fired thus this amount of time have to be added (each time the transition is fired) to the total time needed in order to perform the goal:

- Min Time (Goal) = (8*30) + (7*50) + (7*25000) = 590 + 175000 ms

It is important to notice that the time spent by the user in performing the task is much less than the time spent in wait for the system to beep. This is classical with interactive applications driven by the system instead of user driven ones.

The ratio corresponding to the time spent by the user in performing a task by time spent in waiting is a parameter that renders very accurately the interactiveness of the interactive system. The bigger it is, the more interactive the system.

In the case of the system described in Figure 9 the ratio is: \( R = \frac{590}{17500} = 0.00337 \)

7 Conclusion and Perspectives

This paper demonstrates the possibility and the benefits of using a common formalism to describe both the system and the tasks of users interacting with it. To this end, we use a formal approach, based on the use of Petri nets and structured according to object-oriented concepts. We also propose an iterative design process to support the use of this formalism in a sound way. This approach allows us to fully represent the formula \( \text{Task} + \text{User} + \text{Computer} = \text{System Performance} \) that can be found in \[\text{p. 406}\]. Indeed, the user’s behaviour is included in the task model which is made consistent with the computer model and the performance evaluation is performed using techniques available in timed Petri nets \[\text{p. 406}\].

This work is currently integrated in a software platform in order to make the design and the analysis of the models as automated as possible. This model-based environment \[\text{p. 405}\] integrates an Interactive Cooperative Object
editor, an Interactive Cooperative Object interpreter that generates C++ classes (see [2] for more information on the system) and will embed very soon some analysis tools already available in the Petri net community (an overview of the available tools can be found in [4]).

The ongoing work on this approach consists in applying the ICO formalism and the framework presented here to the Air Traffic Control area [31]. This aims at showing both the scalability of the approach to real size applications and the advantages of formal notations for the design of interactive safety critical applications. A characteristic of this kind of application is their direct manipulation user interface and current work is done on that topic by formally specifying the low level interaction [1].

8 Acknowledgements

The present work owes a lot to the working sessions held in the LIS laboratory in Toulouse, with the LIS team and part of the PII team in the CENA, Toulouse.

9 References


