Verification of Interactive Systems, Springer Verlag, 1997.

Formal Transducers: Models of Devices and Building Bricks for the Design of Highly Interactive Systems

Johnny Accot [1], Stéphane Chatty [1], Sébastien Maury [1], Philippe Palanque [1,2]

[1] CENA
7 av. Edouard Belin
31055 Toulouse cedex, France
{accot,chatty,maury}@cena.dgac.fr

[2] LIS - FROGIS
Université Toulouse I
31042 Toulouse cedex, France
palanque@cict.fr

Abstract. Producing formal descriptions of low level interaction is necessary to completely capture the behaviour of user interfaces and avoid unexpected behaviour of higher level software layers. We propose a structured approach to formalising low level interaction and scaling up to higher layers, based on the composition of transducers. Every transducer encapsulates the behaviour of a device or software component, consumes and produces events. We describe transducers using a formalism based on Petri nets, and show how this transducer-based model can be used to describe simple but realistic applications and analyse unexpected defects in their design. We also identify properties that are meaningful to the application designer and users, and show how they can be formally checked on a transducer-based model of the application.

1. Introduction

The problem of applying formalisms to user interface construction has been an open issue for several years now. The goals are clear to all: interactive software is complex, and formalisms would help mastering that complexity by providing HCI designers and programmers with a common and precise language, and by allowing formal verifications of the behaviour of their software. However, no formal approach has been fully successful yet, and some even wonder whether a unique formalism will ever permit a full description of an interactive system [32] which leads to a wide variety of partial approaches. Some focus on the early stages of the design process such as requirements elicitation and analysis or early specification [15, 28, 25, 26] other on the elicitation of domain related properties [14]. Others describe the dialogue component of applications, making the reasonable assumption that low level interaction components have been taken care of by some error-free industry developer such as all the work done on the formal design of WIMP interfaces relying on the set of predefined interaction objects [5, 34, 35].

In any case, we believe that low level interaction cannot be ignored by formalisms and left to programmers and their craftsmanship. If a unified formalism ever exists, it will have to describe actions to their finest level of detail. And if several formalisms have to be used, one will have to be devoted to sequences of low-level events and the way they propagate to higher layers of software. We see several reasons to that opinion. First, as mentioned in [1], we observe that direct manipulation styles such as those

used in the Macintosh Finder do not rely on widespread reusable widgets. Actually, they use supposedly simple programming techniques, which are undocumented and misunderstood by many programmers and even by some HCI software researchers. Furthermore, most new direct manipulation styles that are currently being devised are not obtained by combining existing styles: they involve new handling schemes for low level events such as button presses and moves, time-outs, or key presses. We thus consider that languages and methods are necessary to design and reason about those new interaction styles.

Our second reason is more disturbing to software engineering researchers. Many programmers have been confronted to evidence that minor and unspecified behaviours of low level software or hardware components can dramatically affect the behaviour of their software system as perceived by users. For instance, we all think we know what a keyboard is and what is behaviour is: every key can be pressed then released. Then how comes that on some workstations, the key "3" is ignored when the keys "1" and "2" are held down[1]? What happens if those keys are mapped to actions that are supposed to be performed simultaneously, such as playing notes on a synthesiser, firing multiple guns in a video game, or showing different representations of data on an air traffic control workstation? We argue that one cannot predict the behaviour of an interactive application without a precise specification of all its hardware and software components.

Finally, we take the point of view that formalisms are the most useful when integrated with production tools such as UIMSs, toolkits or user interface description languages. Therefore, it is important that formalisms manipulate the same notions as toolkits. As of today and as far as highly interactive interfaces are concerned, those notions are limited to graphical objects, events, and event handling schemes such as callbacks or interactors.

Those reasons have led us to work on formal models for low level interaction, with the goal of producing a UI toolkit that manipulates formal notions. This brought to us a number of problems that can be generalised to other approaches: how can a formal description be structured in readable and reusable modules? what are the meaningful properties to be checked over a model?

In this paper, we outline a formal model that allows a modular description of applications while taking low-level interaction into account, down to the behaviour of models. Formal transducers encapsulate the basic behaviours involved in an interface: devices (mouse, keyboard, etc.), behaviours of graphical objects (click, drag, etc.), and behaviours of the functional part of the application, when applicable. Transducers consume and produce events. They can be combined in a client-server fashion. Each transducer exports a formal description of its behaviour; in this paper, we use a dialect of Petri nets for those descriptions: timed Petri nets. Such a formal declaration of behaviour allows consistency checks when connecting two transducers. It also makes it possible to observe how the actions of the user are successively translated, and to detect wrong assumptions that are made when using lower level components.

A lot of work has been devoted to devices and can be mainly classified in the following categories:

[1] This precise behaviour can be easily observed on SUN Sparc 10 workstations

- classifying devices [10]
- understanding of devices [21]
- building of transducers for managing low level events produced by physical devices [1, 11]
- assessing usability of devices according to task [8] and more generally tasks and to users' cognitive capabilities [22]
- evaluating the performance of input devices [2], [19].

This paper belongs the third category as its aims are:

- to propose a formalism allowing designers to describe the behaviour of physical devices,
- to propose a formalism allowing designers to describe the behaviour of high level modules (called transducers) extending the set of events offered by the device,
- to propose a model and tools for building application from transducers,
- to define a set of properties that characterise transducers.

This is significantly different from the work of [11] as it fits in a more generic framework based on object oriented concepts, properties of transducers and has been directly implemented. However, due to space reasons the implementation is not presented in this paper.

The paper is organised as follows. The next section gives a outline of formal transducers, that are exemplified in section 3. We then analyse a simple but realistic application, which highlights how bad assumptions on low level transducers may lead to unexpected software failures. We also show how transducers may exhibit undesirable properties for certain purposes. We identify some of those properties, and show how they can be checked.

2. Formal Transducers

Most computer interfaces today are event-driven, and part of the job of designers is to manage the event flow produced by the user's interaction with physical devices. But managing all the possible combinations of event occurrence is hardly thinkable, especially considering the increasing complexity of systems (e.g. multimodal interaction). One possible solution to solve this issue is to supply designers with a formalism and an associated methodology, that would assist them in describing unambiguously the behaviour of the interface. However it is possible to model such complex behaviours using formal specifications, they unfortunately quickly get incredibly complex, and some research has still to be done on software engineering for formal models in order to supply designers with tools and methods to organise their work. Among the methodologies still to be explored, we believe that an object-oriented approach to formal specification of user interfaces is among the most promising ones. The difficulty in such a design approach is not only to define formally the behaviour of the objects but also to define the system modules and the rules to connect them. Even though this problem has been identified for a long time and description techniques are available such as Pre and Post conditions [29], statecharts [6] or using object-oriented Petri nets [30], work has still to be done to build a convenient software architecture (to favour reusability through encapsulation) and a methodology to help designers to organise their work.

As this research work is directed towards air traffic control (which is a critical domain regarding system security) we are very concerned with the validation of interactive systems. This validation mainly consists in the proof of some pertinent properties on the system. Splitting an application in a set of cooperating transducers makes easier the analysis of properties as they can be checked separately on each model.
Besides, it is important to characterise transducers according to the way they process events. We give hereafter three properties for transducers: *chatty*, *sly* and *regular*. In the next section we will show that a *chatty* transducer may introduce malfunctions in some applications and not in other ones, depending on the set of events used by the application. However, no such malfunction can occur with a *regular* one.
The Accepted Stream (AS_T) of a transducer $T = (I_T, O_T, PN)$ is defined by:

> Let L(PN; M) be the language defined by all the possible firing sequences of transitions of PN
> Let T_r be the set of transition of PN and T_e be the subset of transitions of T_r that feature an event place (grey circle in the Petri net)
> $AS_T = L(PN; M) \mid T_e$ (| means the mathematical restriction)

AS_T defines all the possible sequences of input events that can be processed by the transducer T.
The Produced Stream (PS_T) of a transducer $T = (I_T, O_T, PN)$ is defined by:

> Let L(PN; M) be the language defined by all the possible firing sequences of transitions of PN
> Let T_r be the set of transition of PN and T_p be the subset of transitions of T_r that feature an action part producing events (transition with the Post(event) actions)
> $PS_T = L(PN; M) \mid T_p$ (| means the mathematical restriction)

PS_T defines all the possible sequences of output events that can be produced by the transducer T.
A transducer T is said *chatty* (there is production of information during the filtering) if and only if:

> $\exists event \in I_T \Rightarrow PS_T(event) > AS_T(event)$
> This means that the transducer T is producing more events of the type of input event than it has received.

A transducer T is said *sly* (there is loss of information during the filtering) if and only if:

> $\exists event \in I_T \Rightarrow PS_T(event) < AS_T(event)$
> This means that the transducer T is producing less events of the type of input event than it has received. In that case, information is lost

A transducer T is said *regular* (there is no production and no loss of information during the filtering) if and only if:

> $\forall event \in I_T \Rightarrow PS_T(event) = AS_T(event)$
> This means that the transducer T is producing exactly as many events of any type it has received. However, it is possible for it to produce events of a different type as the transducer of Fig. 2 produces *repeat* events.

3. Formal Transducers and the Keyboard

This section aims at describing the actual behaviour of a physical device keyboard. The behaviour of this device is quite simple as it only consists in translating user's actions on the keys into low level events. In order to provide more interesting events the keyboard is coupled with a software transducer that interprets and enriches these low level events. The next section describes the formal model of the keyboard. Section 3.2 presents a formal specification of a transducer describing precisely how low level events are consumed and which higher level events are produced according to software designers' needs. The last two sections describes how these formal specifications can be used to prove properties on the models.

3.1 The Formal Model of the Device Keyboard

Fig. 1 describes the behaviour of a key using high level Petri nets [24]. When modelling with Petri nets, a system is described in terms of state variables (called *places*, depicted as ellipses) and by state-changing operators (called *transitions*, depicted as rectangles), connected by annotated *arcs*. The state of the system is given by the *marking* of the net, which is a distribution of *tokens* in the net's places. In coloured Petri nets, the tokens assume values from predefined types, or *colours*. State changes result from the *firing* of transitions, yielding a new distribution of tokens. Transition firing involves two steps: (1) tokens are removed from *input* places and their values *bound* to variables specified on the input arcs, and (2) new tokens are deposited in the *output* places with values determined by *emission rules* attached to output arcs. A transition is *enabled* to fire when (1) all of its input places contain tokens, and (2) the value of those tokens satisfy the (optional) Boolean constraints attached to the input arcs.

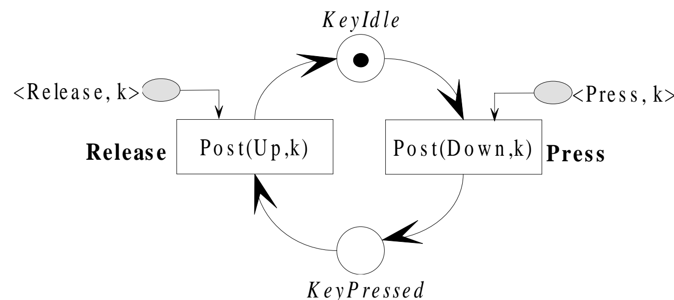


Fig. 1: Formal model of a key

The model of Fig. 1 is made up of two places modelling the two possible states of a key (pressed or released). The actual state of the key is fully stated by the token in the place *KeyIdle* stating that the key is not in use. From that state the user can press the key, that will trigger the transition Press thus removing the token from the place *KeyIdle* and setting it into the place *KeyPressed*. The connection between the transition and the user's action on the device is represented by the broken incoming arrow with the tiny grey circle.

As all the keys (except the modifiers such as ALT, CTRL and SHIFT) are

independent, the behaviour of the keyboard is exactly the sum of the behaviours of all its keys. Thus the modelling of the behaviour of the keyboard can be done by adding one token for each key. However, it is important to differentiate the keys as rendering actions associated to keys might differ (this is the case for example in text editors). This is modelled using coloured tokens in the Petri net. In the following of the paper we shall represent all the keys of a keyboard using coloured tokens. In order to be exhaustive we should have represented as many coloured tokens as there are keys on the keyboard but for readability reasons only few tokens are displayed.

Each time a key is pressed, the device emits an event *down* along with the identification number of the key that has been pressed. Each time a key is released the device emits an event *up*. This is represented in the model of Fig. 1 by the action part of the transitions where the function Post is invoked with the corresponding parameters.

A transducer T = (I_T, O_T, PN) is defined by:

1. I_T be the set of input events received by the transducer
2. O_T be the set of input events produced by the transducer
3. PN the high level Petri net describing the behaviour of the transducer

The keyboard LL is a transducer between user's actions and low level events produced. It corresponds to a filtering function (F_0) and can be defined as follow:

Let Key be the set of keys of the keyboard.

1. $\forall k \in Key$, I_{LL} = {(Release, k), (Press, k)}
2. $\forall k \in Key$, O_{LL} = {(Up, k) , (Down, k)}
3. PN = the high level Petri net of Fig. 1

3.2 The Formal Model of a Transducer

The low level events produced by the keyboard are aimed at being used by software systems. However, most of the applications are interested in a set of richer events. For example the fact that the user is holding a key is usually relevant to the semantics of the application.

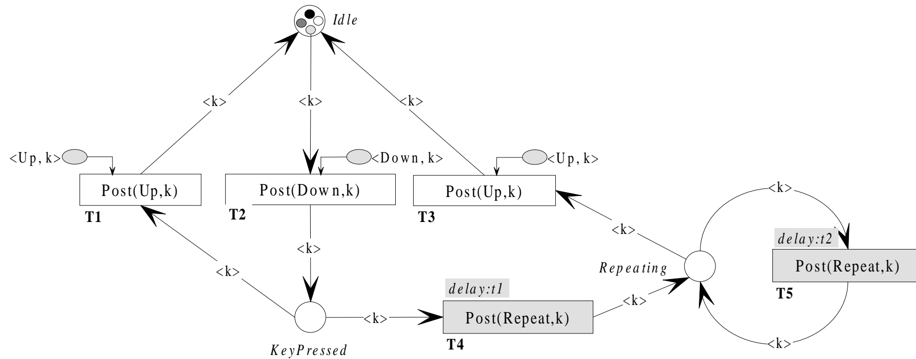


Fig. 2: Formal model of the high level events transducer

The Petri net in Fig. 2 describes such a transducer. In addition to the low level events,

a higher level event named *repeat* is produced by the transducer in order to represent the holding of a key by the user. The *repeat* event is related to temporal manipulation of keys. Two different temporal aspects can be taken into account:

- white transitions (as previously) that fire as soon as they are enabled (i.e. there is at least a token in each of their input places), they are called immediate transitions.
- greyed out transitions are associated with a delay. The semantic of these transitions is of Generalised Stochastic Petri Nets [3]. When a token is set in the place *KeyPressed* a timer associated to the token is started by the transition T4. If the token is still there after t1 seconds (the delay associated to transition T4) then T4 is fired. If within these t1 seconds an event *up* occurs then the transition T1 is fired and the timer associated to the token is destroyed[2].
- a first delay (t1) is used to differentiate between briefly pressing a key and holding it. The transducer waits during t1 seconds before a *repeat* event is emitted.
- a second delay (t2) corresponds to the delay between the production of two *repeat* events.

With respect to the keyboard model of Fig. 1 the transducer above models another state for a key represented by the place *Repeating*. The two delays t1 and t2 are associated to the timed transitions. If the user quickly releases the key (before t1 seconds) then the key returns to its initial state (Idle) before the first timed transition has been fired. The loop including the second timed transition represents the repetitive production of repeat events when the user holds down a key.

The filter HL is a transducer between low level events produced by the keyboard and higher level events. It corresponds to a filtering function (F_l) and can be defined as follow:

Let Key be the set of keys of the keyboard.

1. $\forall k \in Key$, $I_{HL} = O_{LL}$ = {(Up, k), (Down, k)}
2. $\forall k \in Key$, O_{HL} = {(Up, k) , (Down, k), (Repeat, k)}
3. PN = the high level Petri net of Fig. 2

3.3 Formal Analysis of Models

The use of a formal notation for describing transducers and input devices allows us to perform formal analysis on these models. Using analysis techniques it is possible to check generic properties of good allowing to assume the good functioning of the models but also to verify properties specific to the considered application.

This section concentrates on the generic properties such as liveness, reinitilisability, boundedness. Specific properties will be explicited and proved on the application described in section 4. As the transducers are supposed to be used repetitively it is important to be sure that they are reinitialisable, i.e. it is always possible to find a sequence of actions that put the transducer back to its initial state. As transducers are used very often it is also important to be sure that they are not over producing

[2] This semantics is quite different from classical timed Petri nets as there is no duration associated to transitions i.e. a token is not held by a transition but always remain in a place. However, if there is no immediate transition the semantic is the same as the one of T-Timed Petri nets [31]

information and that they do not feature dead branches i.e. part of the specification that can become unreachable. Using Petri nets this can be proved by checking conservative and repetitive components [27].

Analysis of the transducer. The transducer of Fig. 2 is live (there are no dead branches in the specification), bounded (they consume as many resources as they produce, and vice-versa), and reinitialisable as there is always a sequence of actions that can lead the model back to the initial state. We detail hereafter the calculus of the these properties.
Conservative components are sets of places for which the number of tokens remains the same as the one of the initial state whatever sequence of transition is fired. In the model of the transducer there is only one conservative component which includes all the places of the model. Thus the sum of tokens in all the places of the nets remains constant and equal to the number of tokens in the initial state. As all the places of the Petri net belong to a conservative component then the net is bounded.
Repetitive components are set of transitions such that the firing of this state does not change the marking of the Petri net. In the generic transducer model there are six repetitive components namely: T5, T1+T2, T2+T3+T4. All the transitions belong to a repetitive component, and this is a necessary condition for the Petri net to be live [27].
The model of the keyboard presented in Fig. 1 features the same properties as the transducer, but as the model is very simple their proofs are not detailed here.

Verification of specific properties. Given a model, it is usually interesting to prove specific properties that depend only on the actual meaning of the model. Without referring to general properties on the class of system that is considered (such as predictability for example [14]) more precise ones are of interest for the designer.
Using Petri nets it is easy to model undetermined behaviours either by having conflicting transitions or multiple tokens. This happens when several transitions are available at the same time, but this is most of the time avoided by using events as triggers for a transition. However, the event might not be selective enough. In Fig. 2 for instance, if there is a token of type (k) in place *Repeating* and another token of the same type in place *KeyPressed*, and if the event (Up,k) occurs, then it is undetermined which of the transition T1 or T3 will be fired. In order to avoid this kind of undesirable behaviour, it is important to be sure that the token(s) representing a given key cannot be in several places of the net at the same time. This can easily be proved on the transducer as:

- all the places belong to the same conservative component (*Idle*+ *Keypressed* +*Repeating*)
- the value of this conservative component is equal to one

This means that there is no production of tokens by the model, and as at the initial state there is only one token for each key, then for a given event all the transitions of the net are mutually exclusive.

3.4 Conformance of Models

The models we have presented above function as a pipeline of events as it is represented on Fig. 3. Users are the sources of the pipeline as they produce the initial

events. The pipeline architecture of events introduces specific requirements for the behaviour of the components. Indeed, the consumption of events by a given transducer has to be compatible with its preceding transducer and its production has to be compatible with its following transducer.
This can be formally proven by the analysis of the language underlying to each model, i.e. the study of all the possible sequences of transitions as we have shown it in [1] for the mouse and a transducer handling Click, Drag, Drop and Double_Click events.

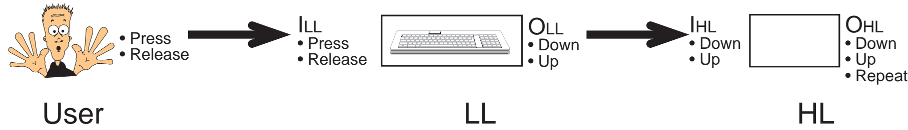
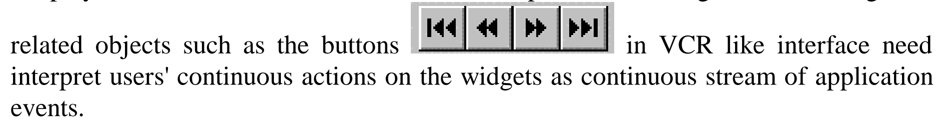


Fig. 3: The pipeline of events

Using regular expressions to represent the languages, the language accepted by the Petri net in Fig. 1 is: `(Press, Release)*` and the language generated is `(Down, Up)*`. According to the pipeline of Fig. 3, conformance has to be check between the generated language of this model and the accepted on of the high level event model of Fig. 2. The language accepted by this high level event model is `(Down, Up)*` and the one generated is `[(Down, UP)* | (Down, Repeat, Up)* | (Down, Repeat, Repeat*, Up) ]*`
These sequences of transitions can directly extracted from the marking graph of the Petri net which can be automatically calculated due to the boundedness of the models.
The temporal transitions have not been taken into account in the calculation of acceptance language because they are not related to incoming events. However, they are taken into account for the production of events as their internal actions consist in posting events. In all the cases the temporal value of these transitions are never taken into account as we only consider here what actions are available and not when they are available.
We can deduce from that analysis that the models are compatible i.e. the transducer will never wait for sequences of events that cannot be produced by the model of the keyboard device. However, as temporal aspects are not taken into account it is not sure on one hand that a model will not be waiting for events and on the other hand that the events produced will be immediately consumed.

4. Analysing defects in an application

This section aims at presenting an application in order to illustrate the effective use of formal transducers for understanding, analysing and then building reliable applications.
In order to demonstrate the importance of the transducer choice and transducer understanding, we will describe here a simple application which uses extensively the keyboard. This application allows users to use the keyboard as a piano. Beyond the toy aspect of this application, the problems highlighted by this example are widely

encountered as soon as a keyboard is used for other purposes than entering text. More precisely, our examples represents all applications where the keyboard is used as a continuous source of information. For example when arrow keys are used in order to move objects on the screen (such as in games), the information that the key is held by the player has to be considered as a stream of input events. Widgets for handling time related objects such as the buttons  in VCR like interface need interpret users' continuous actions on the widgets as continuous stream of application events.

Besides this specific use of the keyboard this section will demonstrate that formal specification of transducers are necessary in order to understand failures in the functioning of applications and to build new versions. Another aspect will be to raise relevant properties.

4.1 Informal Presentation of the Piano Application

This application simulates a musical keyboard with the computer keyboard. Each musical sound note is associated with a specific key on the computer keyboard.

The sound of a note (called the envelop) is here simplified and consists of two parts: the *sustain* and the *release* (

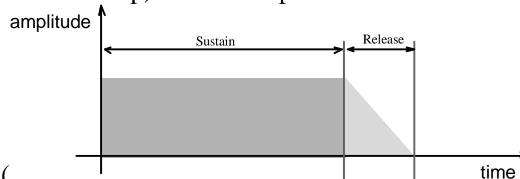


Fig.). Each time the user presses a key, a note is played. The sustain part is played until the key is released. When it is released the sound continues for few milliseconds, and this corresponds to the release part of the sound. Like using classical pianos the performer can play numerous notes simultaneously by pressing multiple keys. All the other musical aspects of the application are not considered here.

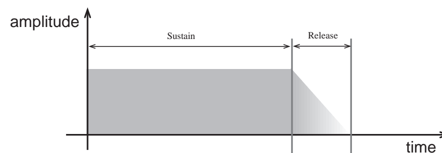


Fig. 4: The envelop of a sound note

4.2 Formal Model of the Application

As shown in Fig. 5 the keyboard is the source of the event stream for the application.

The formal specification of the application is split in two models: the sound generator of the notes (considered as the functional core FC) and the user interface (I) to this rendering engine. These two subsystems cooperate according to a client-server protocol defined itself in a formal way in terms of Petri nets [31]. The sound generator is only a server while the interface is only a client of this server.

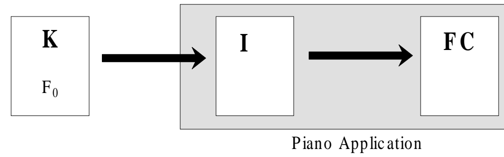


Fig. 5: The pipeline communications between components

The behaviour of the sound generator is described in Fig. 6. The Idle state corresponds to no sound (the associated key is released). To play a note, the T_2 transition has to be fired. The T_2 transition features an internal function called Play which starts the emission of the sustain part when the transition is fired. The sound hardware will maintain the emission of the frequency until an explicit stop. This stop corresponds to the firing of the transition T_1. At that time, the Stop internal function requires the sound hardware to stop the emission of the sustain sound, and to play a sound corresponding to the release part of the note.

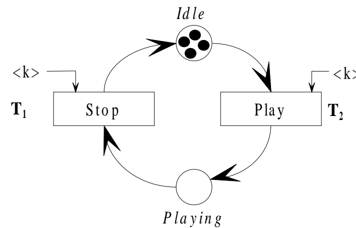


Fig. 6: The model of the note generator engine (FC)

It is important to notice that the model in Fig. 6 describes the behaviour of a note generator. The number of tokens in the place *Idle* correspond to the ability of the note generator to play several notes simultaneously (according to the implementation this could correspond to the number of different channel available). The exact number depends on the hardware characteristics.
Indeed, the sound hardware may still be playing a note "n" while the model of "n" is in the Idle state. This behaviour allows the performer to hit again the same note-key even if the previous release sound is not finished.

4.3 The formal model of the interface of the Piano Application

Fig. 7 presents a formal model of the piano application. This model highlights the communication with the sound engine as each transition of the model includes an action part describing a request to the sound generator. In Fig. 7 the place Idle features a set of coloured tokens modelling the fact that more than one note can be played simultaneously. As for the keyboard colours are used in order to differentiate the notes being played. The application is monitored by the user through events produced by the keyboard transducer. This is represented on the model by the input events of the transitions.

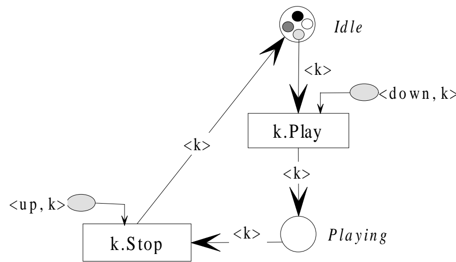


Fig. 7: The formal model of the interface part of the Piano application

This model has been directly implemented on the X Window system ™. The result of this implementation was quite different from what we expected at first as shown in

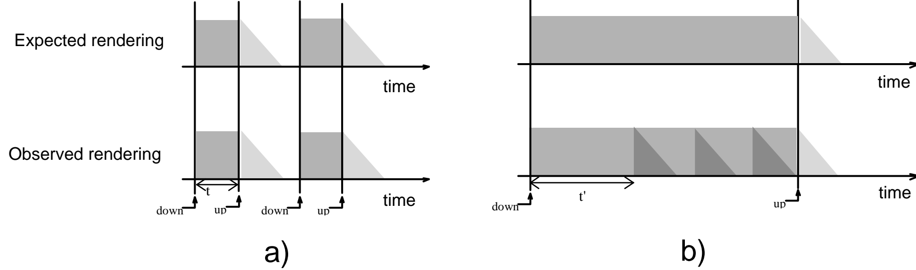


Fig. .

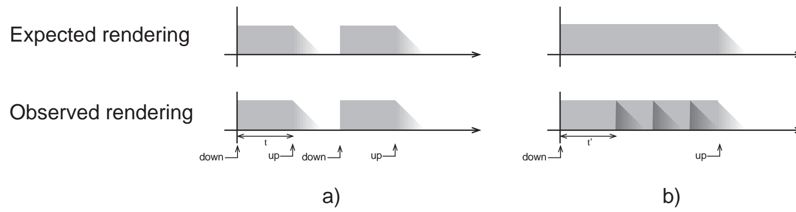


Fig. 8: Expected and effective rendering of the Piano application

The upper part of the

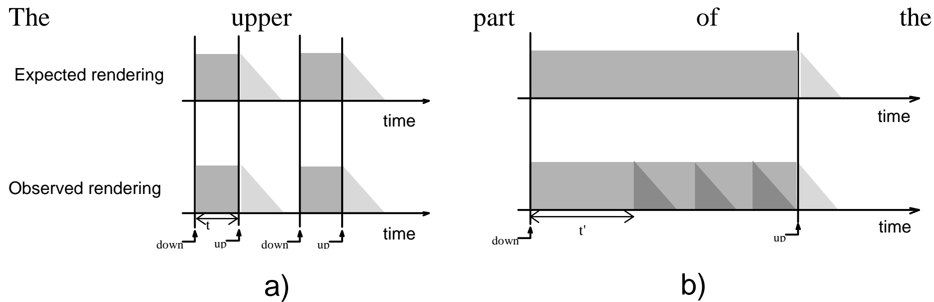


Fig. represents the expected continuous sound from he sound generator between the events emitted by the keyboard. The lower part of the figure describes the effective behaviour observed by the user. As we can see in the left hand part of the figure if the user releases the key after a short period of time t then the expected sound is exactly the one perceived by the user. However, if the key is held for a longer period of time t', then the sound produced is different. The sound is not stable but presents variations

both in amplitude and in pitch. This is represented on the right hand part of

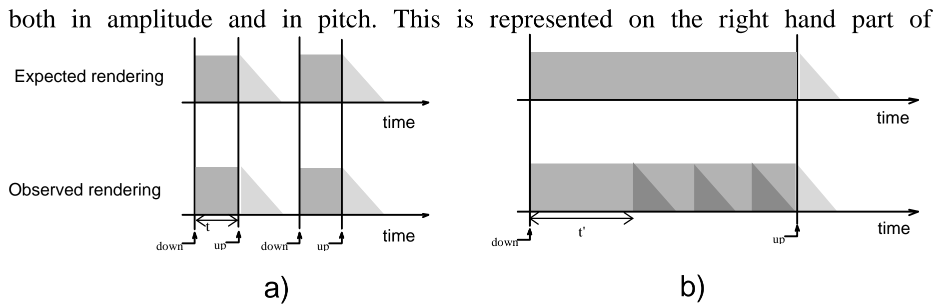


Fig. .

Introspection of the code of the application has not provided any information about the origin of this problem. So we decided to trace the stream of events. Fig. 9 presents the results of this investigation for the long down-up sequence that has led to the unexpected behaviour described earlier. We have analysed the events received by the application at run time. This has highlighted the existence of a X Window' transducer (named F_1') responsible having produced these events.

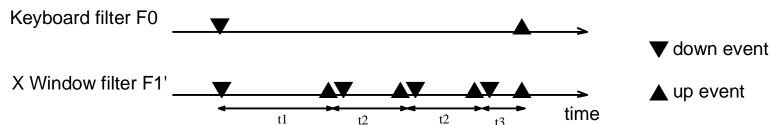


Fig. 9: The stream of events produced by the transducers

The first line of Fig. 9 represents the events produced by the keyboard. The total amount of time during which the key has been pressed is represented at the bottom of the figure and is equal to t1+2t2+t3. The second line represents the stream of events received by the application. We can see that several up-down sequences have been inserted between the two user's events.

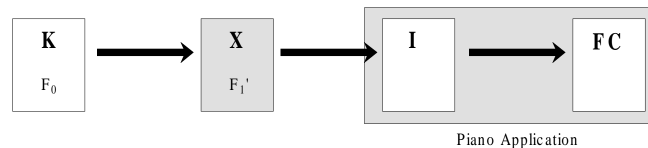


Fig. 10 : There was a ghost in the machine

This explains the observed rendering of the application: it made a wrong assumption on the sequences of events it would receive. This mistake is caused by a bad understanding of X Window, which had its own transducer to emulate repeat events (Fig. 10).

The model presented in Fig. 11 describes, using the same formalism as before the behaviour of this transducer. This model differs from the one presented in Fig. 2 only by the events produced by the transducer in the timed transitions. Indeed, instead of producing repeat events it produces the same events as the one received by the low level keyboard transducer i.e. up and down.

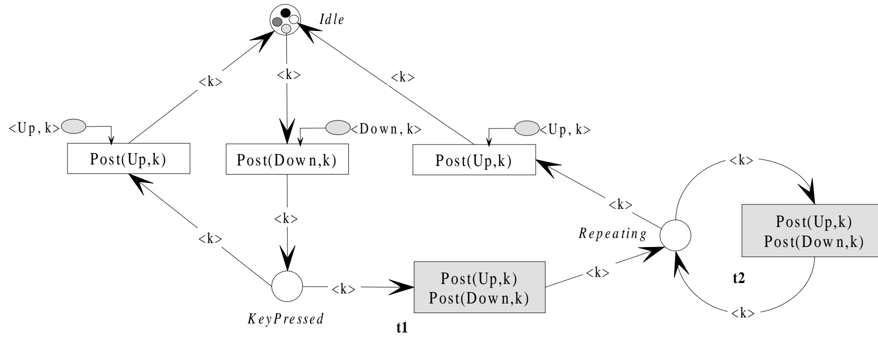


Fig. 11: The behaviour of the keyboard transducer F_1' of the X Window system

The filter XL is a transducer between low level events produced by the keyboard and a different sequence of the same low level events. It corresponds to a filtering function (F'_1) and can be defined as follow:

Let Key be the set of keys of the keyboard.
1. $\forall k \in Key$, $I_{xL} = O_{LL} = \{(Up, k), (Down, k)\}$
2. $\forall k \in Key$, $O_{xL} = \{(Up, k), (Down, k)\}$
3. PN = the high level Petri net of Fig. 11

This transducer is used by all the applications running over the X Window system but no specification of it was available. Its understanding by programmers that have to deal with it could only occur through empirical testing and experience.

4.4 The piano application adapted to the X Window Transducer

In order to use the piano application despite the X Window transducer, we have written a new transducer $F_1'^{-1}$ (Fig. 12). This transducer aims at hiding the X Window transducer by featuring the opposite effect on both the production and the consumption of events.

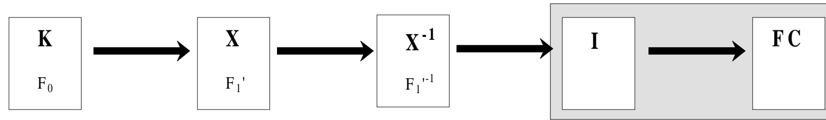


Fig. 12: The new transducer X^{-1} to adapt our application to X

The model of the new transducer (Fig. 13) is based on this characteristic and differentiates user's from synthetic events using a temporal transition. The model must be read as follows. After a down event has been received the token corresponding to the key that has been pressed is set in the place *KeyPressed* and a Down event is produced. Then only an up event can be received which removes the token from place *KeyPressed* and sets it into the place *Repeating*. This up event may be either a synthetic event or a real one. In the case of a synthetic one, a synthetic down event should be received right after and thus the transition T3 will be triggered and the token set back to the place *KeyPressed*. If after t seconds (this quantitative time is expressed aside the transition T4) no down event has be received, the application assumes that

the up event was a real one, hence transition T4 is fired requesting the sound generator to play the release sound and the token is set back to the *Idle* place.

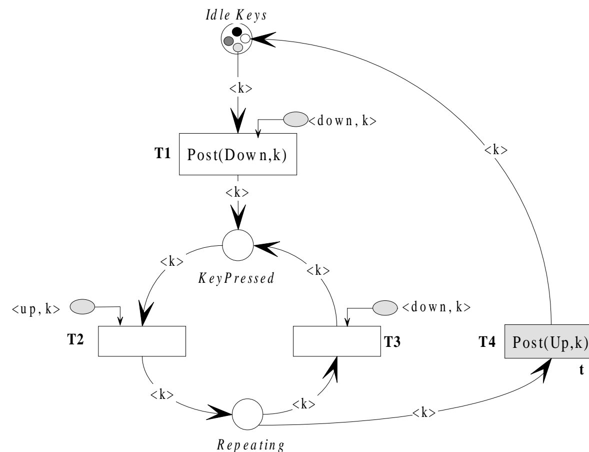


Fig. 13: The model of the $F_1^{'-1}$ transducer

However, usability testing of the new piano application has shown the critical aspect of the temporal parameter t.

The consequences in our Piano application is that it is waiting t seconds before playing the release of the sound (as shown in

Fig. 14.a). If the delay is more than few milliseconds then it becomes perceivable by the user.

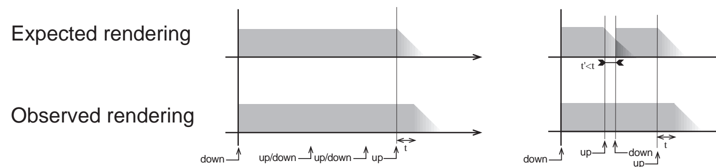


Fig. 14: Two problems with a long temporal parameter t

Indeed, as shown in

Fig. 14.b if this parameter is too long it is possible for the user to press and release the key in a shorter delay. This means that the application will interpret user's real actions as synthetic events. The observed rendering (described on the second line of the figure) is significantly different from the expected one. Instead of the note being played twice for a very short period of time, the note is played continuously. Besides, the same problem of the note being played after the event up has occurred, still applies.

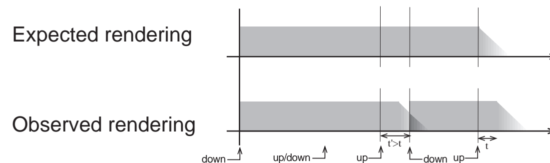


Fig. 15: A problem (c) with a short temporal parameter t

Another problem is related to the client-server architecture of the X Window system. This client-server architecture may dissociate the synthetic (Up, Down) events produced if the user holds the key pressed. If the delay between the two events is longer than t, then the system will interpret these events as user's events thus stopping the note and starting it again. The observed behaviour of such a misinterpretation is described in the second part of Fig. 15.

4.5 Discussion and design options

The problems identified in the previous section are related to the to the parameter t. Designers must take into account this parameter in order to improve the quality of the application. However, the first two options of
Fig. 14 recommend to have a short delay associated to t while the last option of Fig. 15 recommend to have a long delay associated to t
This means that the probability of having disturbance in the rendering of the piano application will never be null. The maximum disturbance for the rendering is produced by the problem a) thus we should have a very short t. However, using an heavily loaded network the problem c) will occur each time the user holds the key thus resulting in an unusable application.
These three problems are the result of the X Window transducer presented in Fig. 11 because there is a loss of information when the transducer is in the pipeline an can be characterised in a generic way by the properties presented in section 2.
The first information that is lost is the number of keys that have been pressed by the user. Indeed as the transducer produces the same Down and Up events when the user has pressed a key or is holding it, it is no more possible to know how many times the user has pressed a given key. This can be overcome using the information that the Down event is emitted immediately after the Up one, but due to the client-server architecture of X Window they may be received significantly separated (problem C).
It can be easily proved that the X Window transducer of Fig. 11 is not *regular* and more precisely is *chatty* as it produces Up and Down events when no such events are received (the user is only holding the key).

5. Conclusion

The building of reliable interactive systems featuring a direct manipulation interface requires a clear understanding of all the input devices used by the user to drive the application.
In this paper we have discussed this thesis and proposed a transducer-based model in order to cope with the problem of the design of such interactive systems. We have considered here keyboard-like input devices, but the approach can be directly generalised to graphical input devices such as mouse or track-ball from the work presented in [1]. We have characterised necessary properties for transducers that have been highlighted by the effective development of a piano application.
This work is part of a more ambitious project aiming at developing a toolkit for direct manipulation interfaces based on a formal description of all the basic bricks to be at the basis of the applications. Indeed, most of the applications currently developed for

the management of Air Traffic Control present a direct manipulation interface and there is a need for both reliability and efficiency.

6. References

1. J. Accot, S. Chatty, P. Palanque. A Formal Description of Low Level Interaction and its Application to Multimodal Interactive Systems, In [18], pp. 92-104.
2. A. Albert. The effect of graphic input devices on performance in a cursor positioning task. In proceedings of the 26th Annual meeting of the Human Factor Society., 1982, pp. 54-58.
3. Ajmone-Marsan et al. Generalised Stochastic Petri nets, Wiley, 1996.
4. M. Beaudouin-Lafon, Y. Berteaud, S. Chatty. Creating direct manipulation interfaces with XTV. EX'90. European conference on the X Window System. London 1990.
5. A. Beck, C. Janssen, A. Weisbecker, J. Ziegler. Integrating object-oriented analysis and graphical user interface design. In Coutaz J. & Taylor R. (Eds) LNCS Springer Verlag 1995.
6. G. Booch & J. Rumbaugh Unified Method for Object-Oriented Development, Documentation Set Version 0.8, October 1995. Available by http at www.rational.com
7. W. Buxton. Lexical and Pragmatic Considerations of Input Structures. Computer Graphics, 17 (1), 31-37, 1983.
8. W. Buxton. There's more to interaction than meet the eye: Some issues in manual input. In Norman and Draper Eds. User Centered System Design: New Perspectives on Human-Computer Interaction, Hillsdale, NJ, Lawrence Erlbaum Publ. Pp. 319-337.
9. W. Buxton. A three state model of graphical input. In proceedings of the Interact'90 conference, p.449-456, North Holland 1990.
10. S. Card, J. Mackinlay, G. Robertson. The design space of input devices. In proceedings of ACM conference on Computer Human Interaction (CHI'90) pp.117-124.
11. L. Cardelli, R. Pike. Squeak: a Language for Communicating with Mice. In proceedings of the ACM conference on Graphics (SIGGRAPH'95), pp. 199-204.
12. S. Chatty. Defining the behaviour of animated interfaces. Engineering for Human Computer Interfaces conference 1992. p. 95-109. North-Holland.
13. S. Chatty. Extending a graphical toolkit for two-handed interaction. In ACM UIST'94, pages 195-204. ACM Press, 1994.
14. A. Dix. Formal Methods for Interactive Systems. Academic Press, 1991.
15. D. Duke, M. Harrison. Interaction and Tasks Requirements, in [17], pp. 54-76
16. Proceedings of the First Eurographics workshop on Design, Specification and Verification of Interactive Systems, F. Paternó Ed. Springer Verlag 1995.
17. Proceedings of the Second Eurographics workshop on Design, Specification and Verification of Interactive Systems, P. Palanque & R. Bastide Eds. Springer Verlag 1995.
18. Proceedings of the Third Eurographics workshop on Design, Specification and Verification of Interactive Systems,F. Bodard & J. Vanderdonckt Eds. Springer Verlag 1996.
19. B. Epps, H. Snyder, W. Mutol. Comparison of six cursor devices on a target acquisition task. In proceedings of the Society for Information Display, 1986, pp. 302-305.
20. O. Esteban, S. Chatty, P. Palanque. Whizz'Ed: a visual environment for building highly interactive interfaces. Proceedings of the Interact'95 conference, p. 121-126.
21. G. Faconti, A. Fornari, N. Zani. Visual representation of formal specifications: an application to hierarchical logical input devices. In [16] pp. 349-367.
22. G. Faconti, D. Duke. Device Models. In [18] pp. 73-91.
23. F. Feldbrudge. Petri net tool overview 1992. Advances in Petri nets 1993. In G. Rozenberg (Ed.), Lecture Notes in Computer Science n° 674, p. 169-209. Springer

Verlag 1993.
24. K. Jensen. Coloured Petri nets. Vol. 1 (Basic concepts) and Vol. 2 (Analysis methods and practical use) Springer Verlag, 1995.
25. C.W. Johnson. Literate Specifications. Software Engineering Journal, July 1996, pp. 225-237
26. C.W. Johnson, S. Jones. Human Computer Interaction and Requirements Engineering, SIGCHI Bulletin, Editorial For Special Edition On HCI and Requirements, vol. 29, n°1, pp.31-32, 1997
27. Lautenbach K. Linear algebraic techniques for Place/Transition Nets. Application and Theory of Petri nets, LNCS 254 & 255, Springer Verlag. 1986.
28. J. McCarthy, P. Wright, M. Harrison. A requirements Space for Group-Work Systems. In Proceedings of Interact'95, pp. 283-288, Chapman & Hall.
29. B. Meyer. Object-Oriented Software Construction. Prentice Hall 1988.
30. P. Palanque & R. Bastide. Petri net based design of user-driven interfaces using the interactive cooperative object formalism. In [16], p. 383-401.
31. P. Palanque & R. Bastide. Time modelling in Petri nets for the design of Interactive Systems. GIST workshop on Time in Interactive Systems. Glasgow, July 1995, and also SIGCHI bulletin vol 28 n°2, p. 43-46.
32. P. Palanque, F. Paterno, R. Bastide, M. Mezzanotte Towards an integrated proposal for interactive systems design based on TLIM and MICO. In [18] pp 162-187.
33. C. Ramstein et al. Touching and Hearing GUI's: Design issues for the PC-Access System. In ACM/ASSETS'96, ACM Press, pp. 2-10, 1996.
34. W. Van Biljon. Extending Petri nets for specifying man-machine dialogues. Int. J. Man-Machine Studies (1988) 28, p. 437-455.
35. A. Wasserman. Extending state transition diagrams for the specification of human-computer interaction. IEEE Transactions on Software Engineering, 11(8), August 1985.