

# Combining SVG and models of interaction to build graphically rich user experiences

## A white paper on using IntuiKit to design User Interfaces (UI)

*Keywords:* SVG, user interface design, visual design, GUI toolkit, software architecture, model-driven architecture, finite state machine

Stéphane Chatty, Alexandre Lemort, Stéphane Sire



Prologue 1. La Pyrénéenne  
31672  
Labege Cedex  
France  
contact@intuilab.com

---

## Executive Summary

---

The gap between Web design and pure User Interface (UI) design is narrowing from day to day, as shown by recent announcements around the concept of rich internet application. SVG is at the heart of this evolution because it offers a high-end 2D graphical model with drawing primitives familiar to graphical designers.

IntuiLab designs highly interactive UIs that often feature multimodal capacities: gesture recognition, speech recognition, etc. We have started to use SVG for the graphical modality at the core of our UI programming environment: IntuiKit. Our portfolio includes UIs with high-end graphics for Air-Traffic control centers, in-car systems and for e-government services. Using SVG has led us to improve our usability driven software engineering life cycle and to combine SVG with imperative code written in any programming language and other proprietary markup languages.

SVG fits well in an agile software development process. At the early steps in a project the visual structure of a UI can be expressed as a logical structure. This structure is seen as a set of graphical components by graphical designers and as a set of software components by programmers. It can serve as a contract between themselves so that they can work in parallel: engineers start programming while graphical designers start drawing. Each time a mockup is required for an intermediate user test, the graphical designer sends the new designs as a set of SVG files. We have been able with this method to integrate the final look and feel of several UIs very close to the end of the project, with very low integration costs. Another advantage of this method is that graphical designs are reproduced with high fidelity in the final UI, which is not always the case when the programmers have to translate drawings into code instructions. Of course, this requires a rendering engine that conforms to the SVG requirements. For that purpose, we use the TkZinc open source graphical toolkit.

We have fostered this agile software development process based on a separation of graphics from behavior of an application by introducing models of control structures such as finite state machines (FSM). FSMs control what parts of the SVG trees are displayed at a given time. One advantage is that we do not need complex scripting language code to control the visibility of SVG nodes, as is often the case when programming interactive components with SVG and Javascript in Web browser applications. FSMs provide a simple interface between the code of the application and the SVG that represent the current states of the components.

This white paper gives an overview of the UI design process based on IntuiKit, from the most early design phases with participatory design sessions, to the development and integration of the different parts of the application for testing purpose.

---

## Table of Contents

<b>Combining SVG and models of interaction to build graphically rich user experiences.....</b>	<b>1</b>
1. Introduction.....	1
2. UI development process.....	2
2.1 Participatory design sessions.....	3
2.2 From graphical components to software components.....	5
3. Developing SVG applications with IntuiKit.....	6
3.1 The IntuiKit application object model.....	7
3.2 The switch module.....	8
3.3 The FSM module.....	8
3.4 Building control structures with switch and FSM.....	9
3.5 A component example.....	9
3.6 Making generic components.....	10
4. Application to a real project.....	10
5. Related works.....	14
6. Conclusion.....	14
Acknowledgements.....	14
Bibliography.....	15

# Combining SVG and models of interaction to build graphically rich user experiences

## 1. Introduction

The gap between Web design and pure User Interface (UI) design is narrowing from day to day. This is shown by recent announcements around the concept of rich internet application or dashboard widgets in Mac OSX Tiger. Graphic designers in UI development teams are one of the keys of this evolution. With the growing understanding that this work can improve users' performance and acceptance of new products, it is now sought in the design of specialised user interfaces and not only on the Web, from aircraft cockpits to plant supervision systems. For instance, figure 1 illustrates a graphic designer's work for air traffic control.



**Figure 1: Visual techniques when used by graphic designers can enrich the message conveyed, Jean-Luc Vinot**

SVG has a great potential in desktop applications because it combines high quality visualisation and user interactivity. Its wide palette of visual techniques makes it overwhelming superior to raster based GUI. With the coming of SVG 1.2, it will be extended with UI-controls and videos. But until now, the use of the full power of SVG is quite difficult. There is an increasing number of export facilities available in professional vector graphics editors such as Adobe's Illustrator, but they are far away from using all the features of SVG. For example, DOM manipulation, animations, interactivity and scripting are not possible. *This makes the export facility of such editors useful for small web applications, but nearly unuseable to produce real applications at reasonable costs.*

Wider acceptance of Human-Computer Interaction processes and graphic design requires solutions that preserve the ability to redesign graphics, and avoid duplication of effort. The direction we chose in the design of our UI software development suite, IntuiKit, is to support software engineering processes that give graphic designers a more central role, while preserving the ability of programmers to structure their code appropriately. The vision is that designers become software producers. Their artwork is a new type of software component that can be managed in its own way then merged with other components, just like software components obtained from different source files have to be merged by compilers and link editors in traditional software engineering.

This article describes a method that aims at making the design and development of applications with high-end 2D graphics more accessible to industrial companies. We then present IntuiKit and discuss implementation details. Finally, we illustrate the use of IntuiKit on a real application and present related work.

## 2. UI development process

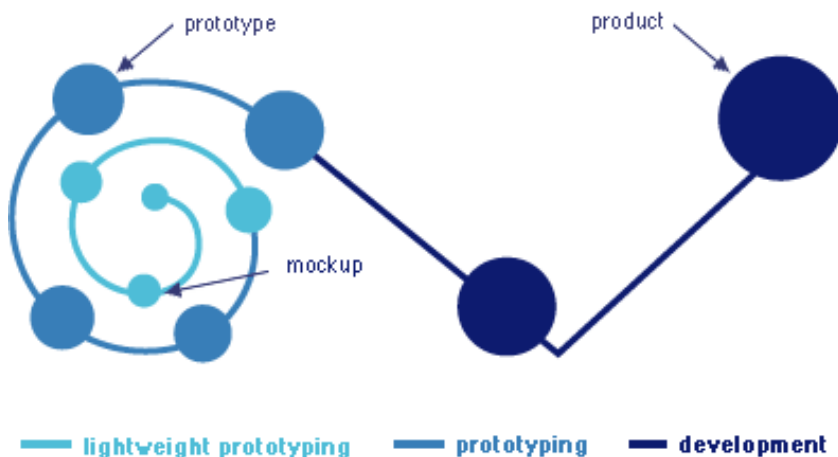
Initially, User interfaces (UI) were developed with traditional software engineering cycle such as V-cycle or waterfall model. While this is quite suitable when the specifications are fully defined, these approaches showed limitations in the case of UI, particularly for complex systems. It is difficult at the same time to define the requirements accurately and to formalize them before design begins. This introduces major risks concerning the acceptance by users.

The Human-Computer Interaction (HCI) research community has advocated the User-centered and iterative design methods to overcome these difficulties. Interactive software is developed through a series of development cycles continually-refining the prototype/application. In each cycle, the design is elaborated with users, refined and tested, and the results of testing of each cycle determine the focus of the next cycle. The methods of user-centered design advocated in HCI literature are compliant with agile software development which value individuals and interactions over processes and tool among other principles (Agile 2001). However they are not widely practised in industrial companies due to a number of weakness:

- user-centered design is highly dependent on the willingness of users to engage in the process and their availability,
- the development of the models and prototypes during the design phase is lengthy and costly,
- there is often a break, both technical and human, between the phase upstream from design and the phase downstream from development, which occasionally leads to the solutions being called into question,
- some design choices are difficult, if not impossible, to implement in the technical, financial or scheduling constraints of the project.

Our UI development process tackles this issues by combining a spiral design cycle and a V development cycle. Figure 2 illustrates our approach divided into three phases:

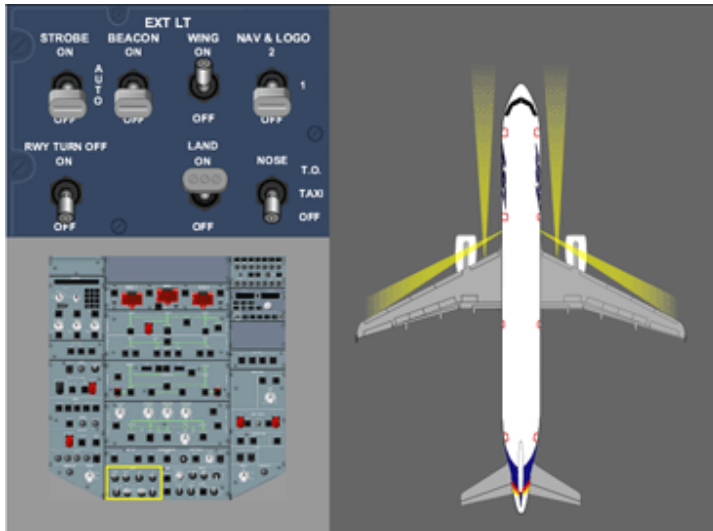
- lightweight prototyping: in this phase, we encourage users to express their requirements. Clarifications and enhancement of specifications, scenarii of use and a first graphical design can be obtained,
- prototyping: at this stage, the realism of mockups is increased and prototypes are developed to be able to experiment and evaluate the system under conditions close to those of the end product,
- development: when the specifications are validated, a classical industrial cycle is started.



In this section we describe our development process with the example of an application originally designed by Kriss Rockwell (US Airways) for training pilots of Airbus A321 (Figure 3). The goal of this development is to build an

interactive panel for learning how to control the lighting system of an Airbus A321 displayed in a separate window.

Applying our method the development starts by a participatory design session that should ideally bring together a graphical designer, one or more representatives of the end-users, a human-factor specialist, a programmer and eventually the project manager.



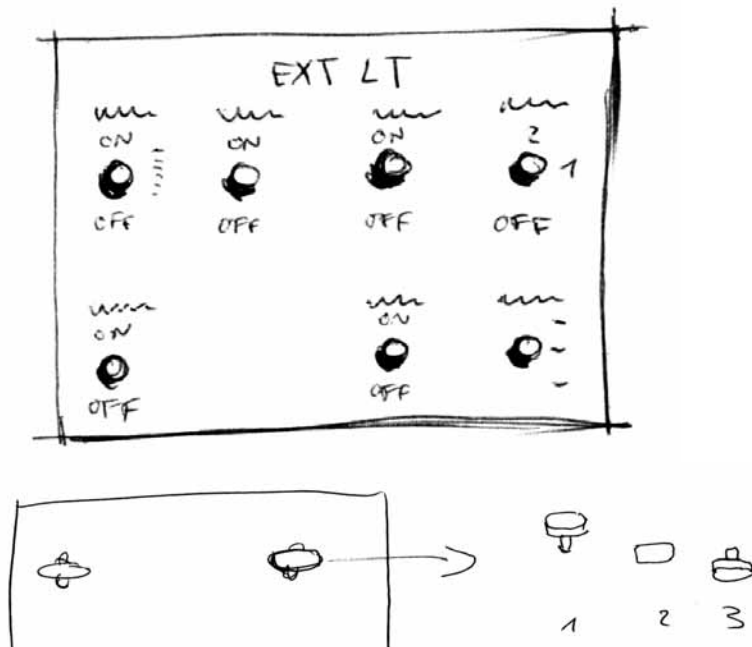
**Figure 3: The Airbus A321 training application reproduced in the scenario, originally by Kriss Rockwell**

### 2.1 Participatory design sessions

Participatory design has emerged as a method for understand what users need. Unlike other methods it is based on designing with the user rather than designing for the user (Muller and Khun 2003). In that way it goes against the misconception that people do not know what they want or cannot tell you what they want.

Participatory design assumes that users should play an active role in the creative process and contribute to the decision making process. It also encourages the participation of a wide variety of people, such as graphics designers, software engineers, sales persons, etc. The concept of participatory design is becoming standard practice in the computing industry. It involves different techniques such as role-playing activities and/or paper prototyping.

In the case of the Airbus A321 scenario, the goal of a participatory design session is to elaborate some sketches of the different parts of the UI and to define their expected behavior by simulating interactive sequences representative of typical tasks with paper mockups or storyboard drawings. Some examples are shown on Figure 4.



**Figure 4: Paper mockups obtained during the participatory design session**

The sketches and the manipulation of the paper mockup lead to define what we call a design contract. It consists in a list of the named graphical components that constitute the UI and in a description of the states attainable by each component. For example, to have both Taxi and Takeoff lights illuminated, the nose wheel light control must be in T.O. position (set to T.O.).

The following association list between sticks and their corresponding states is the design contract for the A321 application:

- Strobe:
  - ◆ OFF (strobe lights are off)
  - ◆ AUTO (strobe lights are automatically switched on when the shock absorber is not compressed)
  - ◆ ON (strobe lights flash white)
- Beacon: Operation of the two flashing red lights, one on top and one on bottom of the fuselage
  - ◆ OFF (beacon lights are off)
  - ◆ ON (beacon lights flash red)
- Wing: Operation of two single beam lights on each side of the fuselage, to illuminate wing leading edge and engine air intake to detect ice accretion
- Nav and Logo
  - ◆ OFF (lights are off)
  - ◆ 1 (logo lights are on when the main gear struts are compressed or the flaps are extended at 15° or more)
  - ◆ 2 (circuit for second set of navigation lights is activated)
- Runway turn off: Operation of runway turnoff light installed on the nose gear strut.
  - ◆ OFF
  - ◆ ON
- Land: Operation of landing lights
- Nose wheel
  - ◆ OFF (all lights are off)
  - ◆ TAXI (only taxi lights is illuminated)
  - ◆ T.O. (Both taxi and takeoff lights are on)

At the end the design contract gives a name for each objects such as noseWheel as well as a template name for object states such as noseWheel\_off, noseWheel\_taxi, noseWhell\_to which are not detailed here

## 2.2 From graphical components to software components

Programmers and graphics designers can work independently, once the initial contract has been established. The contract API defines a SVG structure for each component that enables interoperability between look and feel while leaving it flexible enough such that graphics designers may create visually stunning and impressive applications.

### 2.2.1 Design of graphical components

A component such as the nose wheel light control is composed of three layers to represent its three possible states (OFF, TAXI and T.O.). Figure 5 shows the Adobe Illustrator palette that represents the structure used as a contract between the graphics designer and the programmer for the full lighting control panel.

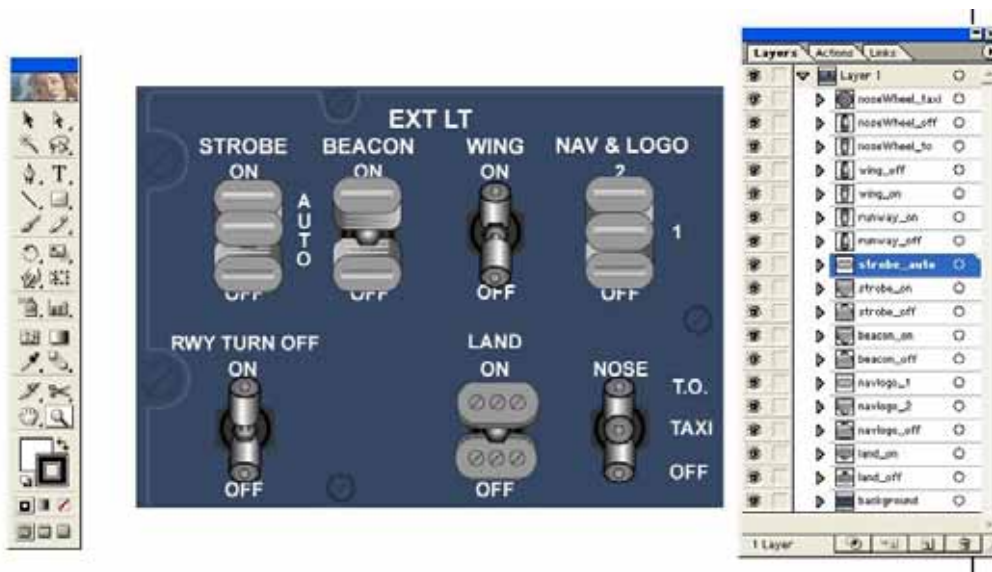


Figure 5: Structure of the SVG file for the lighting control panel

One of the most important advantages of this approach is the possibility to create several skins for the UI. There are two ways to accomplish this goal: through CSS files and through new SVG files. Although the easiest way to customize the representation of the UI is through the use of CSS, SVG files may be replaced and swapped with no effect to the application functionalities, provided that they respect the contract between the graphics designer and the programmer.

### 2.2.2 Design of behavior of components

Describing behavior is a different facet than graphics. The literature provides various models of discrete or continuous behaviour in user interfaces, such as UML State charts, Finite state machines, Petri nets, data flows, constraints, etc. To implement example the Airbus demonstration, we have chosen finite state machines (FSM). Although the FSM model has well-known limits such as state-explosion issue, it is rich enough to define simple discrete behaviours. The overall system behavior can be defined by two types of FSM:

- a two-state behavior to model lights and two-steps controls
- a three-state behavior to model three-steps controls

Figure 6 shows the FSM of a cyclic three-steps control. This FSM has three states (P0, P1 and P2) and three transitions

labelled with abstract events (turnP0P1, turnP1P2 and turnP2P0).

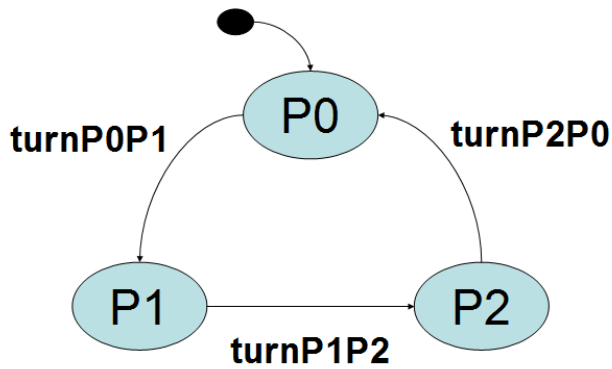


Figure 6: Finite State Machine of a cyclic three-steps control

### 2.2.3 Integration of graphical components and behaviors

The integration of graphical components and behaviors is the last step in a cycle of our development process. It consists first in pairing graphical objects with states of finite state machines (FSMs). Figure 7 shows the association of the FSM states with the corresponding graphical representations of the nose wheel light control.



Figure 7: The FSM-graphics pairs for the nose wheel stick

The programmer must have a mean to describe the association of events on graphical objects, such as mouse click, with the abstract events of the FSM. For example, to change the nose wheel light control state from TAXI (P1 state) to T.O (P2 state), we have to left-click on the graphical object named noseWheel\_taxi. Thus, the abstract event turnP1P2 must be set to "left-click on noseWheel\_taxi"

## 3. Developing SVG applications with IntuiKit

SVG allows a separation of the UI presentation from its control logic. This fits well in an iterative design process involving team work because graphical designers and developers can start working in parallel, as soon as they have agreed on a logical structure for retrieving the graphical components. However this can only be achieved if one does not clutter the SVG files with tens of line of scripting language code.

## Combining SVG and models of interaction to build graphically rich user experiences

We support the iterative UI design process with a custom software suite, IntuiKit, that makes use of SVG for the UI presentation. However, IntuiKit is not dependant on the graphical modality. It has been designed as a rendering engine for running model-based software components. This makes it closer in many respects to a language interpreter, or eventually to a browser, than to a GUI toolkit.

A UI is the result of the instantiation and combination of several models. SVG is one of these models. The models are bound together into software components. The definition of a component is itself driven by a model that we call the IntuiKit core model. Every other model is designed as an extension of the core model.

The set of models for a given UI depends on the interaction style and on the the preferred modelling approach: for instance, models of graphical objects and behaviours for direct manipulation; speech and grammar rules for speech interfaces. The software components making a given UI are declared and managed independently, their different constitutive models can be authored with specialized editing tools, such as vectorial drawing applications for SVG, by hand with a basic text editor, or they can be automatically generated.

The IntuiKit runtime engine loads the components directly from XML files. It is also possible to declare and to instantiate components directly with native code written in an imperative language, presently Perl or C++. In that case, the component code can be mixed together with other application code. It is also possible to mix native code and XML components in a same application. The IntuiKit application object model binds components together.

### 3.1 The IntuiKit application object model

As with a Web application based on XHTML or pure SVG, an IntuiKit application is based on a hierarchical structure which is a tree of elements. Elements are the basic blocks of all models: graphical objects, windows, control structures, etc. For instance, SVG drawing elements, such as rectangles and paths, are also IntuiKit elements. Elements define their own properties, which are key-value pairs.

The interpretation of an application consists in a series of traversals of the application tree in response to end-user actions or to internal changes in the system state. Each element in the tree has an internal state that determines the side effects of the traversal. This state is either unrendered, rendered or suspended. An unrendered element has never been traversed, it needs to be rendered when first traversed. A rendered element has already been rendered as its name suggests; any change to it since its last traversal must be converted to adequate side effects during the next traversal. Finally, the rendering of a suspended element is suspended, which implies that it does not provoke side effects when traversed, even though it must be able to be quickly rendered again if its state changes to rendered.

The rendering process and the side effects depend on the nature of the element. The rendered and suspended states of IntuiKit version of SVG elements have been mapped to their visibility attribute: rendered state maps to a visible value while suspended state maps to a hidden value. Changes to the values of the element properties result in changes of the corresponding attributes of the graphical presentation of the element when it is visible.

Some elements of an application tree can be escaped from the normal life cycle. They are treated as pure models, which means they will never be rendered directly, much like SVG elements between <defs> tags. They can be duplicated into different parts of the tree, which is similar to creating a new instance in a prototype-based language. So in a way, the IntuiKit application model is a generalization of the SVG document object model with custom elements and an extension mechanism for creating new elements.

Some builtin control structures described below control the current state of elements. The control logic in IntuiKit relies on an event processing model in which any element can send and receive events. There is a specialized Binding element which is used to bind event specifications with callbacks written in native code.

The application tree is loaded from one or several XML files, such as SVG files. It can also be created through an API in native code, or with a mixture of XML files and imperative programming language files. The values of the properties

defined by each element can be set from Cascading Style Sheets (CSS) files, or directly from within the XML files or the native code files. A referencing mechanism based on XPath expressions can be used to store references to elements inside properties in XML files. It is equivalent to the pointer based referencing system in imperative programming languages.

The separation of presentation from application logic comes from the provision of presentation as graphical components in SVG files, and from the provision of control structures as dedicated elements belonging to the switch or the FSM modules.

### 3.2 The switch module

The switch module of IntuiKit shares common points with the switch module in the XForms recommendation. Like it, this element contains one or more case elements, any one of which can be rendered at a given time while all the others are suspended. Put it in another way, it can also be seen as an extension of the SVG switch element that renders only the first of its direct child elements with a special attribute that matches given environment variables.

In IntuiKit, the switch element has a builtin 'current\_branch' property whose value determines the identifier of its case children that must be rendered at the exclusion of the others. Changing the value of this property and traversing the switch sub-tree results in changing the case that is rendered.

The following example is a switch-based component that displays the Airbus lights of the plane whose images are pointed to with the XPath expression "plane.svg#strobe" when the value of its 'current\_branch' property is "on". It displays nothing when its value is "off".

```
<switch id="switch" xmlns="http://www.intuilab.com/2005/intuikit">
  <case id="on">
    <use xlink:href="file://plane.svg#strobe"/>
  </case>
  <case id="off"/>
</switch>
```

#### Example 1: Switch XML example

The switch module takes advantage of a method often used by graphical designers who use Adobe Illustrator to build behaviours in Web pages: they store the states of their objects into different layers and manually simulate the transitions by turning the visibility flag of the layers on and off. To achieve that with IntuiKit, each layer must be given a unique id and be loaded inside a case branch of a switch element.

### 3.3 The FSM module

The FSM module wraps native IntuiKit code for managing finite state machines. A FSM element defines one property for each input event that can trigger a transition. The value of this property must be set to a valid event specification at runtime. A FSM can also trigger events during transitions. Its builtin 'current\_state' property stores the name of its current state.

As an example, the following "automaton3p.xml" file defines the FSM produced by the developer for the nose stick component represented in Figure 5:

```
<intuikit xmlns="http://www.intuilab.com/2005/intuikit">
  <fsm id="fsm">
    <property name="turnp0p1"/>
    <property name="turnp1p2"/>
    <property name="turnp2p0"/>
    <transitions>
      <transition from="p0" to="p1" on="turnp0p1"/>
      <transition from="p1" to="p0" on="turnp1p0"/>
      <transition from="p2" to="p0" on="turnp2p0"/>
    </transitions>
  </fsm>
</intuikit>
```

```
</transitions>
</fsm>
</intuikit>
```

### Example 2: Automaton3p.xml file

When a FSM is rendered, it starts listening for input events and changes its state as they occur, triggering the corresponding transitions. When a FSM is suspended it stops listening to events. FSM, like any other element, can be nested inside switch elements. This is a powerful way to create hierarchical finite state machines.

### 3.4 Building control structures with switch and FSM

One of the main purpose of the FSM element is to be paired with a switch element for controlling the current branch that is rendered. This is conceptually equivalent to considering that they share a common state.

There are some examples of an approaching pattern in Web applications, where a hard coded Javascript control structure is used to move the focus between different elements. This is the case in the pizza-ordering application of IBM's Multimodal team, that accepts size and various toppings via check box or speech recognition (Richard 2003). However, in these type of examples the control structure has not been captured by a declarative model and requires a mix of XML and scripting language code.

IntuiKit introduces a means for declaring a property in a parent component that is shared with some properties of its direct children. In particular, a parent component can define a 'state' property that merges the 'current\_state' and 'current\_rule' properties of its child FSM and switch elements. The resulting object behaves as if the three properties were the same. As a side effect, each change of state of the FSM element will also change the state of the switch element. The pseudo-code below illustrates the corresponding XML syntax:

```
<component xmlns="http://www.intuilab.com/2005/intuikit">
  <property name="state" extends="f.current_state; s.current_branch">
    <fsm id="f">...</fsm>
    <switch id="s">...</switch>
  </component>
```

### Example 3: Merge of properties

### 3.5 A component example

The following code is the "nosestick3p.xml" file that declares one of the two sticks with 3 positions used in the Airbus demo application. It clearly shows the effective separation of graphical models from control models. That code defines one component containing a 3 states switch and a 3 states automaton (in an external file) that control the rendering of the stick.

The skin for the stick is defined in an external SVG file that contains the corresponding graphical components. The event specifications that triggers the transitions of the automaton are declared with an IntuiKit syntax. The event specifications use an XPath expression to point to the graphical objects that trigger the events.

```
<intuikit xmlns="http://www.intuilab.com/2005/intuikit">
  <component id="nosestick">
    <property name="state" value="p1" extends="fsm.current_state; switch.current_branch">

    <switch id="switch">
      <case id="p0" >
        <use xlink:href="lightPanel.svg#nose_bg"/>
        <use id="stick_p0" xlink:href="lightPanel.svg#nose_off"/>
      </case>
      <case id="p1" >
        <use xlink:href="lightPanel.svg#nose_bg"/>
        <use id="stick_p1" xlink:href="lightPanel.svg#nose_taxi"/>
      </case>
    </switch>
  </component>
```

## Combining SVG and models of interaction to build graphically rich user experiences

```
</case>
<case id="p2" >
  <use xlink:href="lightPanel.svg#nose_bg" />
  <use id="stick_p2" xlink:href="lightPanel.svg#nose_to" />
</case>
</switch>

<use id="fsm" xlink:href="automaton3p.xml">
  <property name="turnp0p1" source="#stick_p0" spec="ButtonPress-1" />
  <property name="turnp1p2" source="#stick_p1" spec="ButtonPress-1" />
  <property name="turnp2p0" source="#stick_p2" spec="ButtonPress-1" />
</use>
</component>
</intuikit>
```

**Example 4: XML description of a three-steps stick**

### 3.6 Making generic components

The models and the techniques described above support the creation of full applications. When a new model is developed as a custom native code extension and is not available in XML, then it can be instantiated in native code. However, it is also possible to create new XML parser modules. These parsers and the corresponding markup are isolated in their own namespaces. This boils down to create new "code-behind" elements that fits within the overall IntuiKit architecture.

In any case, it would be fastidious to have to declare all the interactive components of an application into individual XML files when they differ only by the name of the SVG files that contain their skins, or by the name of the identifiers of the skin components. This is the case for instance for all the sticks with 2 positions, or for all the sticks with 3 positions in the example used through this article. Their definition differs only by the identifiers of their graphical components.

For that purpose we have defined an 'eval(name)' function that can be used as a string value of any property, or as a value of the "xlink:href" attribute of the use element. That function returns the value of the property called 'name' of the component from within which it is called. The following example declares a generic component that displays a SVG file whose name is contained in its 'stick\_off' property. The component is also instantiated with a "file://panel.svg#noseWheel\_off" target:

```
<component id="display">
  <property name="stick_off" />
  <use xlink:href="value(stick_off)" />
</component>

<use xlink:href="#display">
  <property name="stick_off" value="file://panel.svg#noseWheel_off" />
</use>
```

**Example 5: Example of customization**

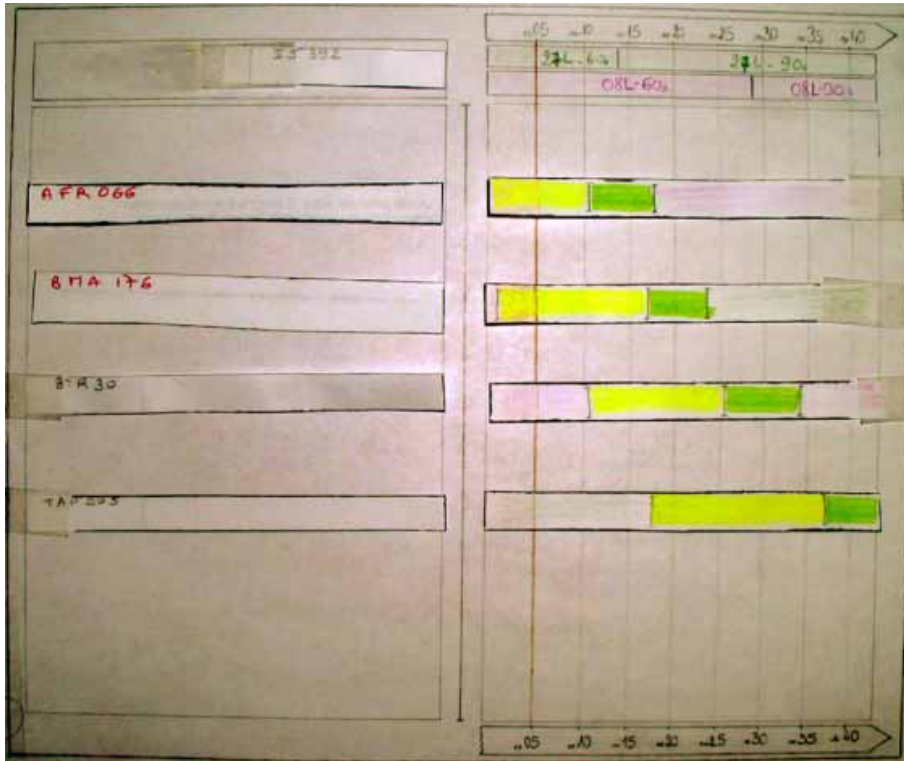
The full Airbus demo application can be programmed with only 6 short XML files (automaton2p.xml, automaton3p.xml, stick2p.xml, stick3.xml, light.xml, airbus.xml) and 2 SVG files (plane.svg, panel.svg) by using the parameterization mechanism. In that particular application there is no need for native code. Most of the files, such as the automaton description files, can be reused in other projects. They can also be generated with specialized graphical editing tools.

## 4. Application to a real project

We have developed prototypes and pre-operational products in the automotive, aerospace, manufacturing, telecommunications and defence domain with IntuiKit. This has given us a chance to test the concepts described here. Four graphic designers were involved in these projects, one at a time. Some had no previous experience of working with programmers. In all cases collaboration was remote, meetings being reserved for participatory design sessions. These experiences gave us evidence of the gains of a model-based UI design process in terms of schedules and effort.

## Combining SVG and models of interaction to build graphically rich user experiences

The development of a departure manager for airports is a typical use case. It has been designed and developed over a period of a few weeks in late 2003. A company had developed algorithms to optimize the sequence of departures and coordinate controllers who guide taxiing aircraft. Air traffic controllers are known to be very demanding professional users, and the company wanted to embed their algorithms in touchscreen workstations that would both provide excellent usability and seduce users and decision makers. The application was meant for pre-operational tests, and consequently had to offer full functionality. The company also had a very tight schedule because they wanted to exhibit the product at a professional convention so as to gain customers. This strict deadline obviously played a key role in the organisation of the project.



**Figure 8: The paper prototype that served as a reference for group work**

The project team was composed of a graphic designer and two programmers (among whom a lead interface designer and a domain expert). The project started with a discount participatory design session that produced a paper prototype for every interface to be built in the project. Figure 8 shows one of the prototypes.

This prototype served as the reference for all further developments on the interface, in several ways. First, the layout of the different parts of the interface was the result of collective work and served as the basis for future composition work by the designer. Second, all parts of the prototype were given a name: static parts (printer, column, timeline, etc) as well as a template name for dynamic parts (strips, plan). The names served as a basis for informal communication between participants, who never met again until the end of the project. Third, immediately after the session the lead designer decomposed the prototype into a tree of components, starting with the toplevel parts. This tree represented the architecture of the application, both in terms of software components and graphical components. She used the tree as a contract between the actors of the projects, especially between programmers and the graphic designer.

Programmers and the designer worked independently after this meeting. Contact has been limited to clarification questions or visual design proposals and feedback. The designer started working on the general impression he wanted to convey, and tested ambiances, colours, harmonies, textures, etc. Figure 9 gives a sample of his work.

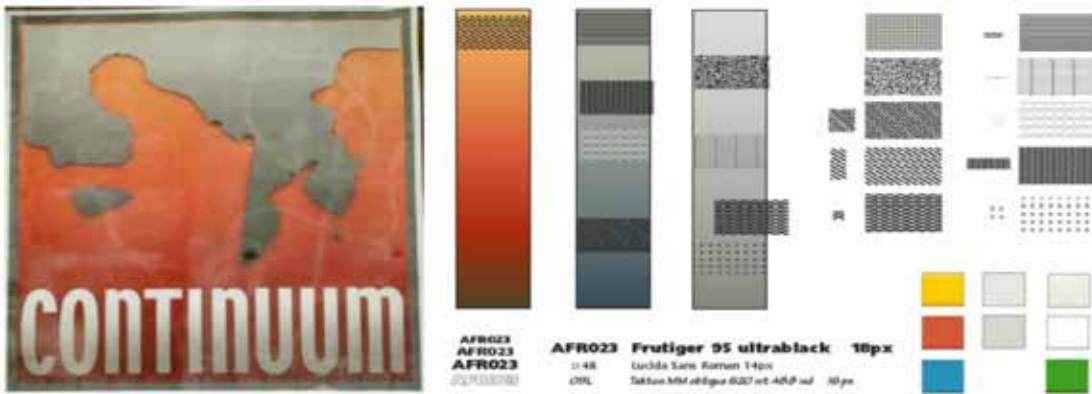


Figure 9: The designer's work on the global picture

Meanwhile programmers used the reference component tree to code the application. For each component they defined control structures, calculations and connections with a functional core resident on a network server. They also drafted basic graphics for testing purpose. They used a professional drawing tool to produce some of the graphics, others being coded with the native code Perl API. The result of their work is shown on Figure 10. It is of very limited visual quality, but sufficient to test usability as well as the network connection.

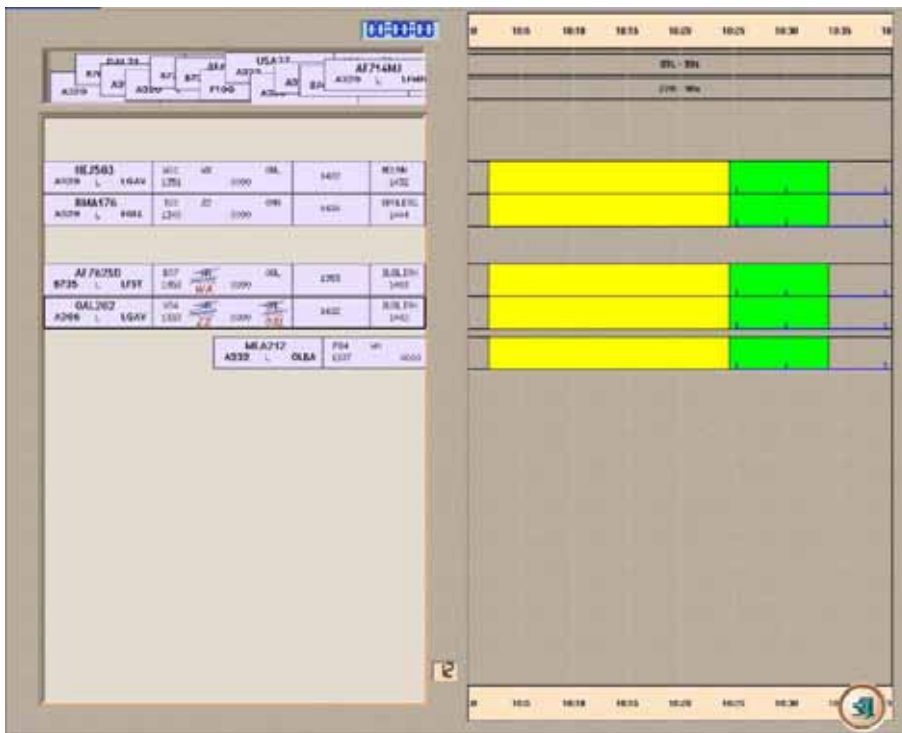


Figure 10: The application with programmer-made graphics

The designer had finished maturing the design and preparing his elements, such as fonts and patterns, three to four weeks before the deadline. He started then to create the structured graphical components. During that process he was able to test them in context himself with the version of IntuiKit installed on his computer: he just had to drop the SVG files in the appropriate folder to see them executed by the last version of the code received from the developers. He used to send back the updated SVG files to the programmers by email. Figure 11 shows two equivalent SVG designs from the application's point of view, though not exactly equivalent for the user.



Figure 11: Two skins for the strips

The final graphics were produced a few days before the deadline, after several cycles of test and redesign. The simplicity of graphics integration into the application as well as the ability to work in parallel allowed very late iterations while programmers were still busy testing and debugging the application. The final result is shown on Figure 12. As a final note, the product obtained a great success with potential users and customers at the exhibition, and the UI was perceived as a competitive advantage.

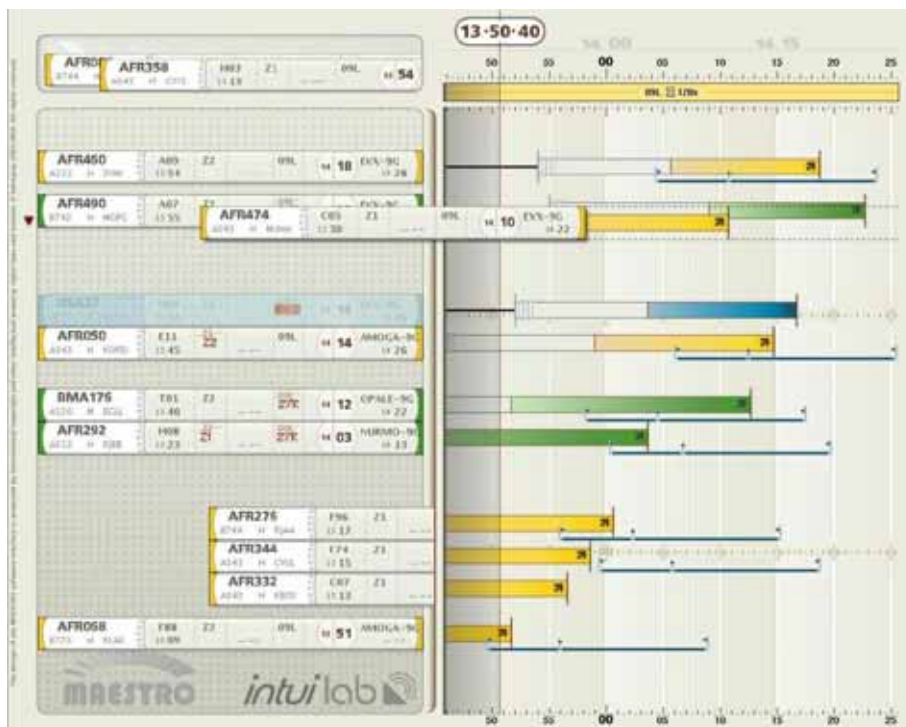


Figure 12: The final application after full integration

We have compared the departure manager project with an equivalent project conducted with a traditional approach: visual elements produced by a graphics designer are reproduced with a programming language by the programmers. We have observed three major improvements of our approach over the traditional one:

- a 20–30% reduction in programming effort, obtained by avoiding the re-coding of graphics;
- a 50–70% reduction in overall project duration, gained with the parallel production process, even if a pipe-line process is set up to incorporate visual elements as they are produced;
- a reduction in coordination costs that we did not measure but that we estimate at 50–70% in terms of number and duration of telephone calls.

We have also identified several limitations and possible improvements:

- freeing graphic designers from certain constraints has a major impact on performance: they create much more complex visual representations which can lead to graphical performance issues;
- some designers use blending techniques in their construction of visual effects that are not supported by SVG.

## 5. Related works

IntuiKit approach to developing SVG applications has evolved from our practical experience of working with graphical designers. It is also the result of our work on modelling languages for declarative UI programming. Some ideas presented in this article can be related with several different technologies which have been around during the last years of the Web evolution.

As we have already mentioned, the IntuiKit switch element is a generalization of the switch module in XForms (Dubinko 2003) and of the switch element in SVG. The IntuiKit core module is used to structure a UI into a set of reusable software components; it shares some ideas with the HTML Components W3C Member Submission (Wilson 1998) and with the XUL/XBL (XUL) way to create components through a binding. This work has recently been endorsed by the W3C with the ongoing sXBL recommendation (Ferraiolo 2005). In a near future the frontier should blur between the notion of document, that sustains the Web browser, and the notion of software component which is explicit in IntuiKit.

There have been some other attempts to introduce declarative control structures. The repeat module in XForms maps data to presentation; it is a kind of iterator, or a constraint declaration depending on the point of view. Laszlo has an explicit 'state' element, but it can be manipulated only programmatically with Javascript methods (Laszlo 2005). In this regards, Intuikit is closer to UML 2.0 with its hierarchical state machines.

Finally, IntuiKit is a prototype-based modelling language into which each component sub-tree can serve as a model for cloning new instances (Dony 2002). The mechanism to merge properties, such as the state property of FSM and switch elements, allows the dynamical creation of split objects (Bardou 1996). In such a split object, the parent component and its FSM elements represent the control logic view of the object, while the parent component and its switch elements represent the graphical view of the object (assuming switch elements contain only graphical elements).

## 6. Conclusion

Practical experience with IntuiKit shows that interactive UI design with SVG is worthwhile. The most obvious benefits of high-end 2D vectorial graphics is the visual quality of the result, when designed by professionals. But the efficiency gain of the separation of the graphic model from the interaction logic is also important. It gives more time to improve the usability of the design.

Now that we have successfully developed several design-intensive application with SVG files as application resources, our plan is to reuse or to invent more declarative models to increase the coverage of UI code that can be implemented with declarative languages. Our work includes constraint-based layout control, non-linear geometric transformations (fisheye, perspective wall, etc.), animations and speech grammars for multimodal applications.

In particular we are looking forward to the standardization of models such as finite state machine (or state charts) by the W3C and to the newest SVG 1.2. We also hope to convince people that the gap between UI design and system design processes can be reduced without losing creativity by working with a model-based approach to UI programming.

## Acknowledgements

Thanks to Jean-Luc Vinot (DSNA/SDER) who brought insightful ideas about graphical design and user interaction which are present in the background of this white paper. A special thanks goes to Patrick Lecoanet (DSNA/SDER), the main

author of TkZinc (<http://www.tkzinc.org>) an advanced open source graphical rendering engine used in IntuiKit. Pierre Dragicevic, Celine Schlienger and Stephane Vales contributed to the implementation of IntuiKit. Yves Rinato (Intactile Design) designed the departure manager, which is shown with the kind permission of Sofreavia. Thanks to those who made significant contributions but are not listed here.

## Bibliography

### [AGILE01]

*Manifesto for Agile Software Development*, <http://agilemanifesto.org/>

### [BARDOU96]

Bardou Daniel and Dony Christophe (1996). *Split objects: a disciplined use of delegation within objects*. ACM SIGPLAN Notices, Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, Volume 31 (10)

### [DONY02]

Dony Christophe, Malenfant Jacques and Cointe Pierre (2002). *Prototype-based languages: from a new taxonomy to constructive proposals and their validation*. ACM SIGPLAN Notices, conference proceedings on Object-oriented programming systems, languages, and applications, Volume 27 (10).

### [DUBINKO03]

Dubinko Micah, Klotz Jr. Leigh L., Merrick Roland and Raman T. V. (editors) (2003). *XForms 1.0, recommendation, W3C, 14 October 2003*, <http://www.w3.org/TR/xforms/>

### [FERRAILO05]

Ferraiolo Jon, Hickson Ian, Hyatt David (editors) (2005). *A SVG's XML Binding Language (sXBL)*, W3C Working Draft 05 April 2005, <http://www.w3.org/TR/sXBL/>

### [LAZLO]

Laszlo Systems, Inc. (2005). *Software Engineer's Guide to Developing Laszlo Applications*. Visited in June 2005 at <http://www.laszlo.com/lps-3.0/docs/guide/>

### [MIRANTI03]

Miranti Richard, Jaramillo David, Ativanichayaphong Soonthorn and White Marc (2003). *Developing Multimodal Applications using XHTML+Voice*, Voice XML Forum, Volume 3 (5), September/October 2003. Visited in June 2005 at [http://www.voicexmlreview.org/Sep2003/features/Sep2003\\_dev\\_mm\\_apps.html](http://www.voicexmlreview.org/Sep2003/features/Sep2003_dev_mm_apps.html)

### [MULLER93]

Muller Michael J., Kuhn Sarah (1993). *Participatory design*, Communications of the ACM, Volume 36 Issue 6, pp. 24–28, ACM Press, New York, NY, USA.

### [WILSON98]

Wilson Chris (editor) (1998). *HTML Components, Componentizing Web Applications*. W3C member submission, NOTE-HTMLComponents-19981023, <http://www.w3.org/TR/1998/NOTE-HTMLComponents-19981023>

### [XBL]

XUL Planet. *Introduction to XBL*. Visited in June 2005 at <http://www.xulplanet.com/tutorials/xultu/introxbl.html>