HAL
open science

# Supporting multidisciplinary software composition for interactive applications

Stéphane Chatty

## To cite this version:

Stéphane Chatty. Supporting multidisciplinary software composition for interactive applications. SC 2008, 7th International Symposium on Software Composition, Mar 2008, Budapest, Hungary. pp 173-189, 10.1007/978-3-540-78789-1_14 . hal-01022139

**HAL Id: hal-01022139**

**https://enac.hal.science/hal-01022139v1**

Submitted on 23 Jul 2014

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Published in the Proceedings of the 7th International Syposium on Software Composition, Lecture Notes in Computer Science. Copyright (c) 2008 by Springer-Verlag. Available online at http://www.springer.de/comp/lncs/index.html

# Supporting multidisciplinary software composition for interactive applications

Stéphane Chatty

ENAC
Laboratoire d'Informatique Interactive
31055 Toulouse, France
chatty@enac.fr

IntuiLab
Les Triades A, rue Galilée
31672 Labège, France
chatty@enac.fr

**Abstract.** Producing interactive applications is a multidisciplinary software composition activity. This, and the nature of user interface code, puts particular requirements on component composition frameworks. We describe a component model that relies on a hierarchical tree of heterogeneous elements communicating through events and data flows. This model allows to assemble, reuse and apply late binding techniques to components as diverse as data management, algorithms, interaction widgets, graphical objects, or speech recognition rules at all levels of granularity. We describe implementations of the model and example uses. Finally, we outline research directions for making the model more complete and compatible with mainstream software models.

## 1 Introduction

Graphic designers and usability experts are increasingly involved in the design of applications, especially when the user interface goes beyond traditional widgets. Until recently, they did it by producing specifications that programmers tried to follow. This process was not optimal: work was duplicated, mistakes or technical constraints altered the original design, and it forced a sequential workflow between actors. It also impeded the redesign of applications. If a medical imaging company acquired a solution for analysing images, wanted to merge it with their image capture solution, and had consistency problems between the two user interfaces, they had to reprogram major parts of the software.

An emerging alternative process is the multidisciplinary production of software [1]. Graphic designers produce the visual parts, interaction designers produce interactive behaviours, and programmers only produce the functional core (data management and algorithms) and the overall application structure. This reduces the global amount of work, eliminates programmer-induced mistakes as well as incompatibilities, and allows for concurrent engineering: all actors can work in parallel and assemble their work just before delivering.

In this article we propose a component model to support this new process. The main contributions are:

- an analysis of how this process and the nature of interactive software call for a software composition model, applicable to all types of user interfaces and to the functional core at all granularities of code;

– the description of a hierarchical component model using events and data flows for communications among components, aimed at addressing the corresponding requirements.

Contrasting with most models that describe graphical interactive components, our model is aimed at describing all parts of an application, including non-visual interaction as well as the parts that do not belong to the user interface. We examplify the use of this model through several development scenarios, involving various degrees of interaction. Finally, we outline some research directions.

## 2 Motivation: assembling interactive software

Interactive software is hard to develop [2]. This is in part because the user interface per se, which accounts for half of the size of interactive applications, obeys different principles than the other half. It has external control, deals with state rather than computation, and heavily uses references because its objects have multiple interdependencies. With imperative or functional languages, its object behaviours tend to be split across multiple functions. The architecture patterns used for interactive components even give them concurrent semantics [3].

But most of all, the way interactive software is designed and produced poses a software composition problem that must be addressed. If software composition is about assembling components that have not be planned and designed together, then building an interactive application is a continuous software composition activity: from the beginning, unplanned reorganisation is the rule rather than the exception. Moreover, it deals with elements produced by actors with varied backgrounds and methods at any level of granularity and any degree of locality.

### 2.1 Varied stakeholders

Interactive software requires skills from usability experts, domain experts, users, programmers, interaction designers and graphical designers. Their contribution can be very concrete in the actual production of software, whether during the development or later during application 'revamping' or customisation:

– domain experts define sequences of user tasks (game levels, for instance);
– graphical designers define all the graphics and the geometrical layout;
– natural language grammar designers, sound designers or other specialists may also produce parts of the application;
– usability experts or interaction designers may define the fine behaviour of interactive components: should buttons highlight when one enters them from the side or only when pressing directly?
– users or support staff may redefine the layout or alternate input configurations to suit their special needs.

All of these are the owners of concerns that should legitimately form individual components. All have their own abstractions and tools to manipulate them, but in the end their productions must be assembled and run together.

### 2.2 Planning issues

The above actors do not follow the traditional schedules of software development. As already mentioned, their tasks are best achieved in parallel. More, the design is iterative: because user needs are difficult to elicit, iterative processes based on prototyping and evaluation have been devised. The iterations may continue and important design decisions may be delayed while the rest of the software is being developed: from the beginning, the application is in an unplanned customisation phase. This creates difficulties in large projects such as air traffic control systems: one both has to delay interface design issues and choose an architecture very early, which often leads to architecture conflicts later in the project. Only a component model suiting all types of user interfaces would alleviate this problem.

Interactive software also has the classical issues of application redesign: graphics instead of text dialogue, post-WIMP[1], interfaces instead of widgets, or fusion of several applications under a unified interface. The adaptation to varying platforms (screen size, input devices) also requires a redesign or even a dynamic adaptation of the visual layout, the dialogue sequence, or even the interaction style (a button does not work the same with a mouse or a touch screen). This will culminate with the ubiquitous computing paradigm, when applications will discover their execution context at run time. In summary, software composition occurs at any time from initial development to run-time.

### 2.3 Component granularity

The components that are assembled or interchanged can have very different size and complexity. At the largest scale is the integration of two applications into one: an email application and a Web browser, for instance. At a smaller scale, a given widget must sometimes be replaced: changing a classic rectangular menu in favour of a pie-shaped one that works better on tabletop user interfaces, for instance. In some cases the replacement is at an even smaller scale: switching from a mouse interface to a touch screen interface just requires to change a part of the internals of some interactors[2]. For interaction designers this is best seen as a change of sub-interactor sized components from their own library of components: in buttons for instance, replacing the behaviour that reacts only to clicks initiated on the graphical object ('mouse-oriented' behaviour) with one that allows to start beside then enter ('touch-screen-oriented' behaviour).

Furthermore, these various granularities cannot be handled independently: one often has to replace a component by another of a very different size. Consider a desktop metaphor in which applications are shown as pages in a book. When turning the pages, you see an image on each page; but as soon as a page is flat, the image is replaced with the actual running application. As another example, when adding animation to user interfaces so that visual changes are not too sudden, one has to replace assignments of numerical values by whole animation

[1] interfaces that do not rely on the Windows-Icons-Mouse-Pointing paradigm

[2] components that deal with a given interaction: a widget, a drag and drop sequence, a speech-enabled dialogue box

sequences: instead of jumping to its position, the temperature dial of a cooler will make a continuous movement. And the target itself can be a constant, or can be obtained by activating a speech grammar rule, or through a 'wizard' application that helps the user make the choice.

### 2.4 Crosscutting concerns

Not all changes are as local as the ones above. When 're-skinning' a user interface, they only change the graphics but all the visual components in the application are modified. Many facets of user interfaces are "crosscutting concerns" in the sense of aspect-oriented programming: graphical style, geometrical layout, animation, drag and drop management, localisation, time constants, etc. Spaghetti of code is still the norm today in most reasonable-size interactive applications because of this. For instance, building drag and drop behaviour as a component rather than a black box or a series of code fragments is still a challenge.

## 3 Requirements on component models

The situation described above generates requirements on the component model used for organising applications. Some are not new and have been adressed individually in the past. But the new multidisciplinary processes and post-WIMP interaction exacerbate them and make it important to address them collectively.

*Unified framework.* Because of the planning issues described above, it is desirable to have a unique model for all components in an application, so as not to limit how and when components can be interchanged and connected. This applies to components in the user interface as well as the functional core: for instance, an animated object such as a scrollbar index can be connected to the mouse, to a clock, or to the file-loading component at different times. Post-WIMP interface designers rely heavily on this, whereas most user interface component models apply only to graphical interactive components, and to their sub-components as long as they are themselves graphical interactive components. This only covers very few composition scenarios, and forces programmers to use several software composition models in the same application.

*Heterogeneity support.* An interactive application is a heterogeneous system by itself. Not only do developers come from different backgrounds, not only do they manipulate very different entities, but their preferred computation models are also very different. Some interaction styles are best defined through state machines; others are easier with data flows. Graphics are often seen as pure declarative objects. Some dialogue sequences are purely linear (levels in a game, steps in a wizard) but concurrency is always present, if only to provide animated feedback. Gesture recognition and similar algorithms, like computations, are well described with functional programming, while input handling calls for reactive programming [5]. Within the proposed model, it must be possible to build components as different as graphical objects, computations, interactive behaviours or speech grammar rules, by taking any of these points of views.

*Multiple granularities.* Heterogeneity also exists in the granularity of components, as identified in the previous section. The model must therefore have a component concept that is the same at all scales, from basic instructions to whole applications, like does functional programming.

*Modularity.* Not all user interfaces are only graphical. Some have sound, speech recognition, or video capture. Some even change over time: an application can act as a voice server when you are away from the office and launch a graphical interface when you use your computer. It is important that the parts of the framework that manage these modalities are as modular as dynamic libraries are today, and that developers can choose to use them or not.

*Behaviour checking.* The model must provide support for checking component composition, because interactive software also has the issues of traditional software. It is particularly important to check the compatibility of component behaviours, and not only data types [6].

*Declarativeness.* Some stakeholders in the development use purely graphical tools. If they are to contribute efficiently, they must be allowed to modify applications without the help of programmers. Therefore the model must support a declarative style of composition, that is a style in which the existence of a given component at a given location fully determines its semantics.

*External control flows.* In user interfaces, what triggers an action is not a control flow from the main program but an external condition: user's action, clock signal, etc. Function calls do not properly support this because they require that the source of the control flow has information about the recipient and thus is developed after it. It usually is developped before: device drivers and interactive components predate applications. Function references and callbacks, or the use of late binding for that purpose are workarounds that sometimes induce programmers into mistakes [3]. Events are more useful than functions, especially with post-WIMP user interfaces.

*Concurrency.* Some interactive components require concurrent semantics, for instance when two users manipulate two menus on a tabletop interface. Concurrency also shows more subtly when two programmers subscribe two components to the same event without knowing about the other, and a third programmer combines their components and expects them to respect some sequencing properties. Not only does the model need to support concurrency, it also needs to provide ways of reasoning about it and expressing ordering constraints.

## 4 The I* component model

To address the above requirements we propose a hierarchical component model named I*, that combines features of computer graphics scene graphs, interactive

software models, and component models. Successive versions of it have been implemented in the IntuiKit model-oriented programming framework and used in an industrial context since 2003. Its major features are its tree of elements, its event-based communication model, and its modular execution model.

### 4.1 The element tree

In the I* model, an application is a tree of *elements*. An element is made of:

- a set of named *properties*, that store its state;
- a set of named children elements;
- an interface that exports the names of certain children, properties and *events*, and manages internal operations on children element and properties.

Some elements, called atomic, are built using the host language and their internals are not accessible within the model. All others, called *components*, are built by assembling elements, creating properties and defining interfaces. The I* model consists of the description of elements and operations on them, of a set of atomic elements that describe control, and of their execution semantics.

*Application structure.* The tree of elements not only represents the architecture of the application, but also the logical structure of its interface. It provides a reference framework for all actors of the development. For instance, a classical image editing program is made of a palette component, a menu bar component, a drawing area component, and a few pop-up dialogue box components. The palette contains buttons, and so on recursively. All interactors are components, and their children are components (smaller interactors) or atomic elements.

*Atomic elements.* In most user interface models, interactors are atomic. Here, atomic elements are smaller and more heterogeneous: computations, graphical objects, speech rules, state machines, event notifications, property assignments. The model allows a mapping from object-oriented classes to atomic elements, so as to facilitate integration with the host language. To build an atomic element one takes a class and turns an instance of it into an element, selecting some class members as exported properties and some methods as exported children.

Graphical objects and graphical context objects (brushes, gradients, etc) are atomic elements. For instance, a rectangle is an element, and thus forms a legitimate application; running it actually displays a rectangle on the screen. The code below shows how this is done with the IntuiKit Perl programming interface:

```
my $r = new GUI::Rectangle ( -x => 0, -y => 0, -width => 100, -height => 100);
$r->run;
```

The same principle applies to other interaction media; for instance, the IntuiKit environment also implements elements that represent 3D sounds and speech grammar rules. One of the classical techniques for describing the behaviour of interactors is the use of finite state machines. These, as well as dataflow connections for continuous behaviours and algorithms that recognise gestures from trajectories, are also implemented as atomic elements.

Finally, an interactive application also contains computation code and application domain objects. These too are elements, and are currently most often implemented as atomic elements by application programmers.

*Element aggregation.* Components are built by assembling other elements. Sometimes mere juxtaposition is enough, for instance when building a multimodal dialogue box by assembling a rectangular frame, two buttons (Yes and No), and a speech grammar rule that recognises "yes" and "no". More usually, elements need to be interconnected so as to exchange events or values, for instance when coupling a writing zone, a gesture recognition element, and a text element that shows what has been recognised; we will later see how event and data-flow propagation are described through specialised atomic elements.

But in some cases the children elements are too fine grain to have a significant semantics as such, and must be combined tightly to produce a significant effect: the state of a state machine has a meaning only if it is also the state of a perceptible element. By extending the model to sub-interactor elements, we have lost the natural sharing of data between the two parts of an interactor. Using event communication would be a solution, but at the cost of a poorly justified memory overhead. To avoid it, a tight aggregation mechanism called property *merging* is proposed, and managed in the parent component's interface. The result is that a memory slot for one property only is used, and this property is accessible under different names from the children elements. Control propagation when the property changes occurs as a special case of data-flow.

For instance, here is how one would describe a button made of arbitrary graphics for each of its two states, with an element of type *Switch* that uses its *branch* property to choose which of its children is active at a given time:

```
$btn = new Component;
$sw = new Switch (-parent => $btn);
$on = load Element (-parent => $sw, -name => 'on', -file => 'on.svg');
$off = load Element (-parent => $sw, -name => 'off', -file => 'off.svg');
$fsm = load Element (-parent => $btn, -file => 'behaviour.xml');
$btn->merge (-names => [$fsm->state, $sw->branch]);
$b->run;
```

Because it allows to delay the association of graphics (or any other perceptive channel) and behaviour, merging is useful for managing heterogeneity in a group of developers. Once a convention has been established about the names of elements and their properties, a programmer can build a component in which he or she just names the children and specifies the merged properties, an interaction designer builds a state machine that describes how the user's input is managed, and a graphic designer builds a set of graphical objects; the final component is assembled in a compilation or linking phase, just prior to executing the program.

*Information hiding* The names given to children elements and properties are visible to all children of a component, as well as the events defined by children. From this internal symbol table is built an external one, by deciding what names

are visible; during that operation, renaming is also allowed. Note that merging is also a manipulation of symbol tables within the component's interface.

Name hiding is for classical software engineering purposes. Renaming is for interactive software architecture purposes. Interactive components are defined in terms of interaction concepts; names such as 'button', 'icon', 'click', 'press', or 'drag' are used. At some point, they need to be connected to the functional core that uses names such as 'file', 'application', 'launch', or 'ship', 'profile', 'match'. This connection implies two operations. First, one needs to match concepts; for instance, a ship is represented as an icon, a profile as a button, and the 'match' operation is associated to the 'press' event. Then, because the names are different, one needs to translate them. This is called functional core adaptation; in object-oriented frameworks, it is implemented with classes whose only role is to glue objects of incompatible types together. Here, renaming makes functional core adaptation a framework-level feature, optimised out at compile time.

Another peculiarity is that there are different publics to hide information from. Application programmers do not need to see the implementation details of a button, but designers who customise an application do need it; symmetrically they do not care about the external interface of the button. This is currently handled at the implementation level only: name hiding applies to all programming interfaces to the I* tree languages such as Perl, C++ or Java, and not to interfaces in languages for designers such as CSS.

### 4.2 Communication and control

The prevalent execution model in user interfaces, and particularly post-WIMP user interfaces, is the reactive model. This model usually coexists with the procedural model brought by the programming language. To satisfy the uniform framework requirement, I* solely relies on event communication and a variant: data-flow communication.

*Events.* Some elements in the tree are able to emit events when certain conditions are met. A clock emits events at regular intervals, a graphical object emits events when it is clicked on with the mouse, a finite state machine when it changes state, an animation when it ends, a button when its state machine changes state. Functional core elements can also be sources: a plane emits an event when it changes altitude or position, a file when it changes size, and so on. Event subscriptions are represented as *Bindings*, that is atomic elements that associate actions to conditions. A Binding is defined with:

- a reference to a *source*, that is a property or an element that may emit events;
- an *event specification*, that is a source-specific expression that describes what events are selected;
- a reference to an *action*, that is an element that is executed when a matching event is emitted.

For instance, this creates a rectangle whenever a multitouch surface is touched:

```
my $table = find Element (-uri => 'input:/intuiface');
my $b = new Binding (-source => $table->pointers,
                     -spec => 'add',
                     -action => "GUI::Rectangle␣(-x␣=>␣%X,␣-y␣=>␣%Y)");
```

A finite state machine is an atomic element made of a set of bindings that are only active when the machine is in a given state. Atomic actions named Notification allow component builders to emit their own events. Others named Assignment set the values of properties. Callback functions in the host language can be encapsulated as actions named NativeCode.

The Binding elements make behaviour declarative: one creates a control flow just by adding the appropriate Binding. It also helps to create state-dependent behaviours, by making bindings or state machines active or not depending on a state, without having to introduce hierarchical state machines or Statecharts: hierarchy is represented by the I* tree.

*Data-flow.* Data-flow is a special case of event communication. Properties are event sources, and atomic elements called *Connectors* are Bindings defined from two properties: they trigger an implicit action that copies the value of the first property into the second when it changes. For instance with the following code a rectangle follows the finger on a touchscreen.

```
my $t = find Element (-uri =>  'input:/touchscreen');
my $r = new GUI::Rectangle (-width => 10, -height => 10);
my $xc = new Connector (-in => $t->X, -out => $r->x);
my $yc = new Connector (-in => $t->Y, -out => $r->y);
```

Atomic elements named *Watchers* are used within elements to bind actions to changes of their own properties. This allows to build data-flow bricks such as those described in [4] or [7], and produces the control flows associated to merging.

This definition of data-flow does not only provide a declarative way of building behaviours. It also allows to define a consistent scheduling for event and data-flow propagation, so that mixing them leads to predictable results. Implementations of I* include a scheduling algorithm based on properties, comparable to those used in synchronous programming.

## 5 Implementing element semantics

We have built two implementations of the I* model named IntuiKit Perl and IntuiKit C++. We now describe what semantics they give to elements and how their architecture helps fulfill the initial requirements.

### 5.1 A model-based implementation

For each type of elements, an XML format has been defined. For instance, the SVG format is used for graphics. IntuiKit includes parsers for these formats, in addition to a programming interface for instantiating elements, cloning them, or

creating components. Developers can thus build the application tree by loading XML files, instantiating elements from code, or both.

Using XML files has allowed to use IntuiKit in a research project as the final execution engine in a model transformation chain. It also helps manage the heterogeneity of actors and the planning issues: graphic designers use their own tools to build graphics and export them as SVG. Programmers or interaction designers can build the rest of the application in code or XML. Then one can choose to load the XML files at run time, thus delaying integration to the last minute, or to generate code from them. Using XML also allows to migrate application parts from one IntuiKit implementation to another. The typical intended use for this is to carry out iterative prototyping with the Perl implementation, then export the graphics, behaviours, and structure of the application tree in XML and reuse them in the final C++ development.

The manipulation of part of the tree as data files introduces preliminary phases in the execution of applications: the loading or instantiation of elements, then their linking, prior to executing the tree. So as to make the programming interfaces for instantiating elements compatible with element creation in graphical editors, instantiation has been defined along the lines of prototype-oriented languages: elements can be copied from others, then modified.

### 5.2 Modules and rendering engines

Following the construction of the tree, IntuiKit takes charge of executing ('running') it. The associated semantics is that each element represents an instruction for a part of the execution environment named a module: graphical objects are rendered by a graphical engine, speech grammar rules are managed by a speech engine, bindings, actions and other behaviour-oriented elements are executed by the core module. This addresses the modularity requirement: each module is in charge of a set of element types.

Each module defines an XML namespace and implements the associated parser, provides a programming interface for instantiating the elements it defines, and includes a rendering engine for them. Leaving aside user-defined modules that contain user-assembled components such as WIMP interactors (buttons, menus, dialogue boxes) or dials for cockpits, most modules introduce atomic elements. The core module provides the central concepts of the model and a few types of control elements: bindings, connectors, state machines. Other modules are used only when required: a GUI module for graphical objects and basic WIMP objects such as windows, mouse and cursors; an input module for atypical input devices; an animation module for animation trajectories; a speech recognition module for grammar rules. Such modules are implemented by reusing an existing rendering engine, either as a library or a server, and encapsulating its primitives into the execution methods of atomic elements.

Using modules provides support for the management of crosscutting concerns while preserving declarativeness: to enrich a component with a new media, one just needs to add a child element from the corresponding module. All other complexity is hidden in the module internals. Furthermore, modules interact

nicely with the application architecture, creating a two-dimensional structure: one dimension is the set of modules, the other is the application tree that drives the rendering in all modules. In our view, this is the key for providing an clear architecture for multimodal applications.

We have encountered two types of rendering engines with that regard. Some, such as OpenGL, do not store the objects they render and need to be called periodically. In this case, the I* tree serves not only as the application structure but also as the basis for rendering: once the tree is run, the graphical module periodically traverses the tree, updates its rendering context or the engine's, and has graphical objects rendered by the engine as it encounters them. In other words, the restriction of the tree to containers and graphical elements has the semantics of a graphical scene graph. Other rendering engines do manage their own internal structure. In that case the tree is only traversed once to create this structure, and the engine is then notified of changes in the tree that concern it; the engine acts as a server, and one can interpret this as an extension of event communication to the rendering itself.

## 6 Example applications

IntuiLab and their partners have used IntuiKit during five years for developing dozens of interactive applications as diverse as car dashboard and multimedia displays, air traffic control tools, geographical information systems on tabletops, multimodal information query systems or lotto kiosks. We describe here some example uses that demonstrate the robustness of the I* model.

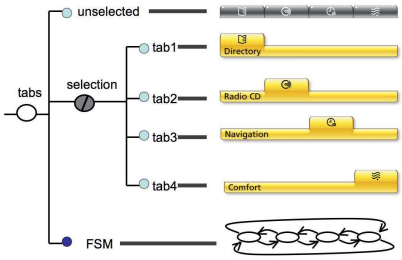


**Fig. 1.** An set of tabs for a car multimedia system

### 6.1 Skinning a visual component

Figure 1 shows the tree structure of a component that was built for a car multimedia system. It has a static background, four tabs that represent four parts of an application, a Switch element, and a finite state machine. The transitions of the state machine are bound to events from a set of keys located near the steering wheel, and its state is merged with that of the Switch. Depending on the SVG file used for the graphical elements, the result looks as in Figure 2a or Figure 2b.

**Fig. 2.** a. With one graphics file b. and another

### 6.2 Building a multimodal dialogue box

The following code shows how one builds a simple multimodal Yes/No dialogue box from atomic elements: a rectangular frame; two rectangles and bindings on them that emit Y (resp. N) events when they are pressed on; a speech grammar; two bindings on the recognition of words by the grammar. For concision the parent component does not appear here, nor the arguments that create the elements within this parent.

```
my $r = new GUI::Rectangle (-x => 0, -y => 0, -width => 200, -height => 100);
my $y = new GUI::Rectangle (-x => 20, -y => 30, -width => 60, -height => 40);
new Binding (-source => $y, -spec => 'ButtonPress', -action => "notify('Y')");
my $n = new GUI::Rectangle (-x => 120, -y => 30, -width => 60, -height => 40);
new Binding (-source => $n, -spec => 'ButtonPress', -action => "notify('N')");
my $g = new Speech::Grammar (-grammar => 'yes-no');
new Binding (-source => $g, -spec => [command => 'yes'], -action => "notify('Y')");
new Binding (-source => $g, -spec => [command => 'no'], -action => "notify('N')");
```

The same events are emitted by this dialogue box whether the mouse or voice is used. The speech grammar, since it is a child element of the dialogue box, is only active when the box is active; the same holds for the rectangles and the bindings of course.

### 6.3 Application design and development

Figure 3 illustrates the use of IntuiKit in a phase of the multidisciplinary process described in the introduction of this article. The illustrated air traffic control project involved "virtual paper": objects that felt like paper strips through a combination of visual effects, animation and gesture recognition. A first phase of iterative design yielded a paper prototype that outlined the structure and the behaviour of the application. Designers and programmers used this prototype to define an I* tree and give names to elements to be produced by designers. Then each started to program, draw or otherwise build their elements and give them the appropriate names. For test purposes, someone in the group quickly produced very crude graphics, gave them the agreed names and saved them in a

SVG file. This allowed programmers to test their work by loading these elements from the SVG file (left). When the final data management, behaviour, animation and graphical elements were ready, the programmers just had to put XML files delivered by designers at the right place, and test the application (right). This application later had several sets of graphics for different customers in Europe.

Measurements carried out on this case study (comparison with a project of similar size and complexity, by the same team, using a linear process) showed a reduction of project duration by about 50%, expenses by about 30%, and a dramatic decrease of coordination costs (estimated number of phone calls) [1].

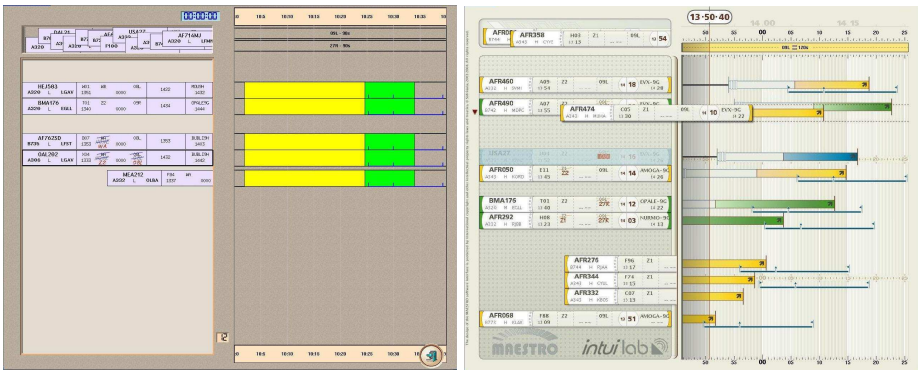


**Fig. 3.** ATC application before and after final integration

### 6.4 Transferring more tasks to designers

In the above example, graphic designers only produced graphics. However, some are willing to take more tasks from programmers, and particularly visual layout and its adaptation to size changes. We have designed artistic resizing [9], a technique where graphic designers provide examples of graphical objects at different sizes, and the system interpolates their appearance for any chosen size.

Implementing the artistic resizing algorithm with IntuiKit was a simple application of the I* model: we built a new atomic element that has properties `width` and `height`, implements the artistic resizing algorithm, is defined by passing it the examples as children elements, and then behaves as a single graphical element. This new element can then be placed in the tree wherever a resizable graphic element is desired, and its properties connected to the size of the available window. From then on, the graphical object adapts to the size of the window, respecting the designer's non-linear transformations.

### 6.5 Input management

One of the future challenges for interactive software is that when building an application, developers will not have a precise idea of what input devices will be available at run time. We have been able to build an IntuiKit module to address this problem, by slightly extending the semantics of the I* model.

Input devices are event sources and hence candidate tree elements, but they are out of control of developers. It makes sense to decide that the application tree is just an element of a larger I* tree that contains the computer devices. Therefore, we just had to create a new element set and element discovery functions to allow programmers to test and use input devices [8]. Using a technique used for communicating dynamic data creation from the functional core to the user interface, the hot-plugging of devices is reported as an event by the set of input devices, which is an automatically managed component that contains all input device elements. In that context, multimodal fusion, that is the combination of inputs from different sources, becomes a matter of creating elements that subscribe to different sources and implement one combination policy or the other: time windows, for instance.

## 7 Research directions

The I* model and its implementations have allowed us to turn innovation in user interfaces into a more industrial activity. But questions remain to be addressed, to give the model more solid foundations and to cover issues currently not addressed. First of all, the control structures described above are insufficient for building all of the functional core; this forces developers to build it as a set of atomic elements, and breaks the requirement for a uniform model. Similarly, one needs to devise a data-passing scheme that makes the implementation of data-flow elements as easy as functions in a functional framework, as well as a typing system for controlling bindings and connections. We may also need to propose a "service call" communication system on top of event communication, for the few cases where the caller is defined after the callee.

In another direction, defining a formal semantics for the I* tree and its communications would provide developers with an unmbiguous understanding of how their components behave, and help compare with more general frameworks. It could also serve as the basis for compiling components rather than just interpreting them: whereas during execution IntuiKit, even in Perl, compares favourably with all rich graphics frameworks, interpretation times are not satisfactory.

Finally, a strong similitude appears between elements and processes in reactive systems or other concurrent models, but the consequences of choosing a given semantics need to be explored. In particular, we must understand what level of control programmers and designers need over the sequencing of their actions, and how it fits in the available models of concurrency.

## 8 Related work

Many composition scenarios and requirements have been studied by user interface software specialists. The proposed solutions either have been high level guidelines or patterns focused on a given requirement: for instance MVC or PAC [10] for separing the interface from the functional core; the use of active

values (examplified recently by Cocoa's bindings) then data flows or one way constraints for describing user input, layout or animation [4, 11]; hierarchies of visual components as in Self [12]; the Java source/listener and Qt signal/slot patterns for event communication. Most such patterns implement a reactive composition model on top of an existing function-oriented language (using inheritance, for instance), thus not addressing the uniform framework requirement. None of these have explored the heterogeneity requirement.

Recent products support the new development processes. Flash allows graphical designers to build complete applications; programmers can extend these using a dedicated language or even a mainstream language. Other solutions for Web applications, such as SVG+Javascript or Microsoft Silverlight, take a similar hybrid approach. However, such solutions are very specific to graphics, and do not propose a unified framework for complex applications: Flash has limited encapsulation features and the others fall in the hybrid model category.

Solutions for programming user interfaces have been proposed for nearly every programming paradigm: object-oriented programming of course, but also reactive programming [13], functional programming [5], etc. Many of these approaches, with the notable exception of reactive programming and the Smalltalk language [14], consist in providing patterns that extend or alter the semantics of the original framework to support interactive components.

With the advent of large heterogeneous systems [15], research on software architecture and software composition addresses requirements that are very similar to ours. The I* tree can be compared to the hierarchy of components in the Fractal framework [16]; component interfaces, including the experimental behaviour inspection features, and some aspects of internal control in I* components not described here can be compared to Fractal membranes. The main difference is probably that Fractal is service-oriented while I* is event-oriented. Aspect programming [17] also shares requirements with I*, particularly modularity (for handling cross-cutting concerns) and external control. One can interpret point-cuts and advices as the I* binding of actions to particular sources, with a particular event specification language. The main difference is that this event communication is the main control construction in I* whereas aspect programming uses it only for particular software engineering cases. I* can also be considered as an architecture description language, but one that would aim at describing the internal architecture of components as well, down to the level of instructions.

## 9 Conclusion

We have analysed in this article how the new multidisciplinary processes used for interactive software influence software architecture and composition. They create a need for a component model that unifies the heterogeneous concepts used by the various stakeholders, that combines with the more traditional requirements of user interface software. We have described the main features of the I* component model that addresses these issues. In particular, the ability to apply late binding techniques to heterogeneous components such as behaviours,

graphical objects, speech rules or computations allows to implement concurrent development processes. One of the main challenges now is to compare our model with more mainstream results in software engineering. Understanding the links between interactive software and other heterogeneous systems may prove fertile, as well as comparing I* with formal models for describing concurrency. In the long term, our objective is to reconcile user interface design with software engineering theories, practices and tools.

## Acknowledgements

This work was partly funded by the French government through the ITEA Emode project and by Agence Nationale de la Recherche through the Digitable and Istar projects. L. Bass, R. Kazman and S. Conversy provided useful advice on this article. The anonymous reviewers helped a lot to improve it.

## References

1. Chatty, S. et al: Revisiting visual interface programming: creating GUI tools for designers and programmers In: Proc. of the ACM UIST, Addison-Wesley (2004)
2. Myers, B.A.: Why are human-computer interfaces difficult to design and implement? Technical Report CMU-CS-93-183, Carnegie Mellon University (1993)
3. Chatty, S.: Programs = data + algorithms + architecture. In: Proc. of the 2007 conference on Engineering Interactive Systems. LNCS, Springer-Verlag (2008)
4. Chatty, S.: Defining the behaviour of animated interfaces. In: Proceedings of the IFIP WG 2.7 working conference, North-Holland (1992) 95–109
5. Elliott, C., Hudak, P.: Functional reactive animation. In: International Conference on Functional Programming. (1997)
6. Accot, J. et al.: Formal transducers: models of devices and building bricks for the design of highly interactive systems. In: Proc. of DSVIS'97, Springer-Verlag (1997)
7. Dragicevic, P., Fekete, J.D.: Support for input adaptability in the icon toolkit. In: Proceedings of ICMI'04, ACM Press (2004) 212–219
8. Chatty, S. et al.: Multiple input support in a model-based interaction framework. In: Proceedings of Tabletop 2007, IEEE computer society (2007)
9. Dragicevic, P. et al.: Artistic resizing: A technique for rich scale-sensitive vector graphics. In: Proceedings of the ACM UIST, Addison-Wesley (2005)
10. Coutaz, J.: PAC, an implementation model for dialog design. In: Proceedings of the Interact'87 Conference, North Holland (1987) 431–436
11. Myers, B.: Separating application code from toolkits: Eliminating the spaghetti of callbacks. In: Proceedings of the ACM UIST, Addison-Wesley (1991)
12. Smith, R.B. et al: The Self-4.0 User Interface. In: OOPSLA'95 conference proceedings, Addison-Wesley 47–60
13. Clement, D., Incerpi, J.: Programming the behavior of graphical objects using esterel. In: Proceedings of TAPSOFT'89, LNCS 352, Springer Verlag (1989)
14. Kay, A.C.: The early history of Smalltalk. ACM SIGPLAN (3) (1993) 69–75
15. Hardebolle, C. et al.: A generic execution framework for models of computation. In: Proceedings of MOMPES 2007, IEEE Computer Society (2007) 45–54
16. Bruneton, E. et al.: An open component model and its support in Java. In: Proceedings of CBSE 2004. LNCS 3054, Springer-Verlag (2004)
17. Kiczales, G.: Aspect-oriented programming. ACM Comp. Surveys **28**(4es) (1996)