Programming Rich Interactions using the Hierarchical State Machine Toolkit

Renaud Blanch,* Michel Beaudouin-Lafon†
LRI & INRIA Futurs
Université Paris-Sud
F-91405 Orsay cedex, France
{blanch|mbl}@lri.fr

ABSTRACT

Structured graphics models such as Scalable Vector Graphics (SVG) enable designers to create visually rich graphics for user interfaces. Unfortunately current programming tools make it difficult to implement advanced interaction techniques for these interfaces. This paper presents the Hierarchical State Machine Toolkit (HsmTk), a toolkit targeting the development of rich interactions. The key aspect of the toolkit is to consider interactions as first-class objects and to specify them with hierarchical state machines. This approach makes the resulting behaviors self-contained, easy to reuse and easy to modify. Interactions can be attached to graphical elements without knowing their detailed structure, supporting the parallel refinement of the graphics and the interaction.

Categories and Subject Descriptors

H.5.2 [**Information Interfaces and Presentation (e.g., HCI)**]: User Interfaces—*graphical user interfaces, interaction styles, prototyping*; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*user interfaces*

General Terms

Design, human factors

Keywords

Advanced interaction techniques, hierarchical state machines, post-WIMP interaction, scalable vector graphics, software architecture, structured graphics

1. INTRODUCTION

Programming interactive applications is known to be difficult and costly [18]. User interface toolkits are the most effective solution proposed so far to reduce the cost and difficulty of developing such applications. They usually provide high-level components that hide the underlying complexity of interactive behaviors. Most user interface toolkits are based on *widgets*, such as menus or buttons, which can be assembled to create a complete user interface.

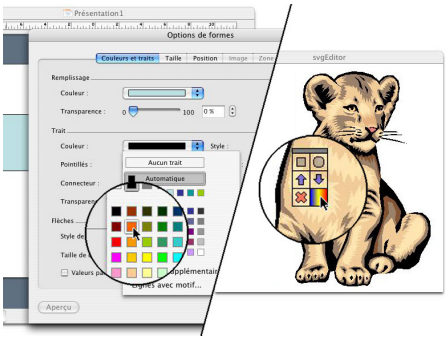

Figure 1: Two interactions to set a color.
Using a property box (*left*), or a transparent tool (*right*)

Widget-based toolkits have proved effective, as they are used for developing most graphical user interfaces. However they result in a WIMP (Windows, Menus, Icons, Pointing) interaction style that may not be well-suited to the task at hand. Figure 1 shows two interactions to perform a color change. On the left, the color is selected with a modal dialog box and requires three to five clicks. This type of interaction is common in current applications, because it is the path of least effort for the programmer when using a traditional widget-based toolkit. Yet this interaction violates the principles of direct manipulation [20] that led to the first graphical interfaces. Indeed, the color attribute is manipulated *in*directly, through a dialog box [2].

On the right of Figure 1, the color is selected with a single click through a bimanual transparent tool or *toolglass* [7]: a trackball used by the non-dominant hand moves the toolglass over the desired area while the dominant hand selects both the action (changing the color) and the object of the action in a single click. Such a bimanual transparent tool is an example of a so-called post-WIMP interaction technique. Many such techniques have emerged over the past decade and have been shown to be very efficient for a variety of tasks.

A drawback of the widget approach, is that the widget set has been designed in at a given time, and does not take advantage of new graphical possibilities that have emerged since this time. As an example, transparency was not anticipated and most toolkits can

*now at renaud.blanch@enst.fr.
†in|situ| project (http://insitu.lri.fr),
Pôle Commun de Recherche en Informatique du plateau de Saclay
CNRS, École Polytechnique, INRIA, Université Paris-Sud

AVI '06, May 23–26, 2006, Venezia, Italy.

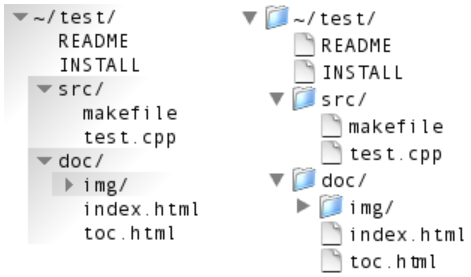

Figure 2: Two SVG representations of a file system

not handle this attribute. Non-rectangle widgets are often not possible either. Thus many post-WIMP techniques cannot be implemented as widgets and are therefore not available to developers using widget-based toolkits. The only alternative so far is to program these techniques from scratch using low-level programming.

The advanced graphical features needed by Post-WIMP interaction techniques are now available using structured graphics models such as Scalable Vector Graphics[1] (SVG), which can be rendered efficiently by taking advantage of modern GPUs [10]. Another advantage of SVG is that they can be both easily produced by graphic designers, and easily manipulated by programs. For the designers, authoring tools supporting SVG or equivalent output formats are now common. For programs, the SVG specification does include an API based on the Document Object Model (DOM), which is standard and well documented.

In this paper, we present the Hierarchical State Machine Toolkit (HsmTk), which takes advantage of these characteristics to provide an advanced graphical model. The toolkit provides elaborate mechanisms to attach interactive behaviors to fragments of the interface without requiring the full knowledge of their structure, allowing designers and programmers to refine concurrently the graphics and the interaction. The toolkit also provides a programming abstraction designed to make interaction as first class objects, thus encouraging factoring and reuse. This abstraction is an extension of the C++ programming language with a control structure for describing hierarchical state machines (HSM). The HSM formalism is well tailored to design and implement interactive behaviors.

We first present those two main features of the Hierarchical State Machine Toolkit, illustrating them with toy samples. Next, some more realistic and useful examples demonstrate the versatility and expressiveness of HsmTk. Then we compare our work with other research on interaction toolkits. Finally, we highlight the perspectives of our work.

2. LINKING CODE TO GRAPHICS

The SVG format describes two-dimensional graphics in XML. SVG is a hierarchical 2D scene graph in which child nodes inherit graphical properties and geometrical transformations from their ancestor. The SVG documents shown in Figure 2 are views on a filesystem tree structure, and Figure 3 shows the XML document corresponding to the left view. Such documents can be easily produced by designers using authoring tools like Adobe Illustrator, or the free Inkscape[2] editor, which makes it possible to manipulate the SVG tree without requiring to edit manually the XML document.

[1]SVG is a language for describing two-dimensional graphics in XML. It is a W3C recommendation. http://www.w3.org/TR/SVG/

[2]http://www.inkscape.org/

```
01 <svg xmlns:hsm="http://hsmtk.insitu.fr/">
02    <defs> <!-- definitions -->
03       <!-- open/close arrows -->
04       <symbol id="closed">
05          <path d="M 9 3 l 5 5 l -5 5 z"
06             style="fill:gray" />
07       </symbol>
08       <symbol id="opened">
09          <path d="M 6 6 l 5 5 l 5 -5 z"
10             style="fill:gray" />
11       </symbol>
12
13       <!-- background gradient -->
14       <linearGradient id="bg" ... />
15    </defs>
16
17    <g> <!-- tree -->
18       <g hsm:behavior="tree" hsm-arg:isOpen="1">
19          <!-- label -->
20          <rect style="fill:url(#bg)"/>
21          <use xlink:href="#opened" />
22          <text x="22" y="12.5">~/test/</text>
23
24          <!-- content -->
25          <g transform="translate(16,16)">
26             <g hsm:behavior="node">
27                <text x="22" y="12.5">README</text>
28             </g>
29             <g hsm:behavior="node">
30                <text x="22" y="12.5">INSTALL</text>
31             </g>
32             ...
33          </g>
34       </g>
35    </g>
36 </svg>
```

Figure 3: XML (simplified) specifying the SVG document shown on the left of Figure 2

2.1 SVG annotations

To make the document interactive, objects implementing the behavior of particular SVG fragments can be attached to some elements of the XML tree. This is done by using custom XML attributes. Lines 18, 26, and 29 in Figure 3 show such annotations. The SVG group starting on line 18 correspond to the top level directory named `~/test`, and includes its content. The first annotation (`hsm:behavior="tree"`) states that a C++ object of class "tree" will be created when loading the SVG node from the document and will be attached to this node. The annotation system can also be used to pass values from the SVG representation back to the implementation, in the same way as C++ constructor arguments. The second annotation (`hsm-arg:isOpen="1"`) is such an argument specifying that the directory content should be visible.

From the C++ code perspective, a behavior is a specialization of the provided `hsm::Behavior` class (Figure 4). Upon loading of the SVG node, an instance of this class is created, and a pointer to the corresponding SVG element is passed to its constructor (`elem`, line 3). This pointer can be used to retrieve the arguments specified as annotations in the XML document (line 8, the `getAttribute` function retrieves the value of the `isOpen` argument specified as an attribute of the SVG element, Figure 3, line 18).

In order to dynamically instantiate the class specified by the `hsm:behavior` XML attribute, the C++ class must be registered inside a behavior factory. This registration is done at run time, providing that the programmer adds a "magic" variable to its class, and initialize it with the name chosen for the behavior (lines 12 and 15, Figure 4). This mechanism is dynamic, meaning that behaviors can be registered while the application is running by loading plugins

```
01 struct Tree : hsm::Behavior {
02    // constructor
03    Tree(svg::SVGElement *elem) :
04       hsm::Behavior(elem) {
05
06       // parameter retrieval
07       bool isOpen = false;
08       hsm::svg::getAttribute(elem, "isOpen", isOpen);
09    }
10
11    // behavior registration
12    static const hsm::Behavior::Factory::Reg< Tree >
                                          registration;
13 };
14
15 const hsm::Behavior::Factory::Register< Tree >
                              Tree::registration("tree");
```

Figure 4: Tree behavior implementation

```
18       <g hsm:behavior="tree" hsm-arg:isOpen="1">
19          <!-- label -->
20          <rect style="fill:url(#bg)"/>
21          <use xlink:href="#opened" />
21'         <image xlink:href="./folder.png" />
22          <text x="22" y="12.5">~/test/</text>
```

Figure 5: SVG document shown on the right of Figure 2

from the filesystem. Thus, if an application have to present some elements of its interface containing behaviors for which it is not aware of an implementation, it can search inside behavior repositories, and load the requested implementation if one is found. This is a powerful way to support extensibility and reuse, since behaviors can be shared between applications, as well as they can be updated without needing to recompile the whole application.

2.2 Structural contract

So far, the only part of the SVG known to the behavior is the element to which it is attached. The programmer can use the DOM API to walk through the SVG structure, and to manipulate it. However, doing so introduces dependencies between the behavior implementation and the SVG structure, and those dependencies are not explicit.

For example, switching from the tree representation on the left of Figure 2 to the one on the right introduces a new element: the folder icon. The corresponding change in the XML tree is shown in Figure 5. The icon consist of an SVG image element added between lines 21 and 22. If the behavior needs to manipulate the text of the label to reflect a name change of the folder, a simple approach consisting of taking the third child of the root element is not robust to such a structure change. Moreover, if such navigation in the tree structure is distributed inside various behavior methods, the dependencies between the code and the associated representation are left totally implicit. Such implicit dependencies lead to a code that is a nightmare to maintain.

HsmTk provides a way to explicitly express a structural contract between the SVG and its associated behavior. Such a contract is shown in Figure 6, lines 10 and 11. It makes use of the C++ typing system to search the childs of the SVG node `elem`. It first tries to find a text element (`label`, declared as `svg::SVGTextElement` on line 3), and then a group element (`content`, declared as `svg::SVGGElement` on line 4) in the remaining childs. The search skips unmatched child nodes, and throws an exception if an element is not found. Thus, the contract in Figure 6 expresses that to pretend to be a tree, a SVG node must have at least one text child,

```
01 struct Tree : hsm::Behavior {
02    // SVG parts
03    svg::SVGTextElement *label;
04    svg::SVGGElement    *content;
05
06    Tree(svg::SVGElement *elem) :
07       hsm::Behavior(elem) {
08
09       // structural contract
10       hsm::svg::Childs(elem).assert(label)
11                             .assert(content);
12       ...
13    }
14 };
```

Figure 6: Structural contract

the first of them being considered as the label of the tree. It must also have at least one group child after this label. This group is then considered as the content of the tree.

This contract is simple to express and understand. It is flexible enough to support some changes of the SVG structure. For example, adding an icon to the tree representation does not affect the structural contract of the tree behavior. The only limitation is that elements of a type already used by the contract should be appended at the end of the child list.

3. HIERARCHICAL STATE MACHINES

HsmTk provides an abstraction that makes it easy to keep interactive behaviors self-contained and reusable. It is well known that interactive software often leads to code that is difficult to maintain and reuse and that can look like a "*spaghetti*" of callbacks [17]. Due to the lack of appropriate control structures, imperative programming languages are not adapted to the implementation of interactions. They are tightly bounded to the computer execution model that, as noted by Wegner, "cannot accept external input while they compute; they shut out the external world" [21].

Formalisms adapted to the description and specification of interactions do exist. HsmTk extends the C++ programming language with a control structure, the hierarchical state machines (HSM), borrowed from such a formalism. We first present an example of behavior programmed using HSMs. Then the syntax and the semantic of the HSM control structure are presented.

3.1 Button Behavior Example

Figure 7 shows on the left the look of a button produced using a SVG authoring tool, and on the right the basic behavior of a button specified using a finite state machine (FSM). This behavior does only take into account the press and release events. In order to capture the subtleties of the complete behavior, the FSM should be refined: when one pushes the mouse button inside the button, then leaves the button and reenters it, releasing the mouse button should trigger the action. Meanwhile, the look of the button should have changed from up to down, to up again when the mouse is outside,

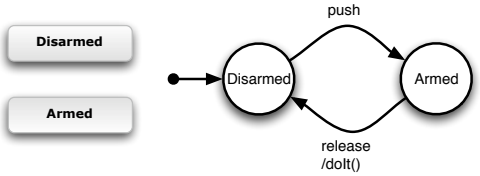

Figure 7: The look (*left*) and the behavior (*right*) of a button

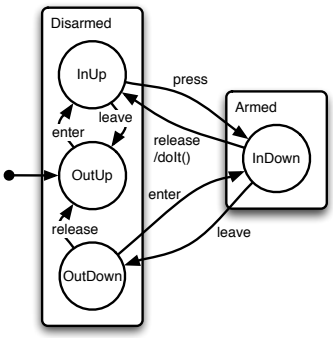

Figure 8: A refined version of the button behavior

and down again when it reenters the button. This behavior is captured by the HSM shown graphically in Figure 8. The hierarchy of states allows to keep at the first level the two states relative to the visual state of the button. Those two states can be refined to handle cursor enter/leave events as well as mouse button press/realease events.

The HsmTk code corresponding to the HSM shown in Figure 8 is given in Figure 9. States are recursively nested using braces as block delimiters, as does C++. The `Button` HSM thus defines two sub-HSMs: `Disarmed` (lines 2 to 32) and `Armed` (lines 34 to 41). The first one starts with some variable definitions (lines 3 to 6). These variables are used by the behavior to manipulate the SVG it is attached to. They are initialized using the structural contract featured by HsmTk (lines 10 and 11). Here, a button must have two SVG group childs each one representing the button in the armed or disarmed state. The initialization removes the armed version of the button from the displayed scene graph using the standard DOM API (line 12).

The synchronization between the internal state of the button and its visual representation is insured by updating the SVG document when the `Disarmed` state becomes active (`enter` action, line 16) or inactive (`leave` action, line 17). These actions are triggered when the `Button` active state switches from `Disarmed` to `Armed`. The remaining parts of the HSM do specify the other sub-HSMs, and the transitions governing the global state changes. For example, the `OutUp` state (lines 19 to 21), which is active when the cursor is outside the button and when the mouse button is not depressed, specifies a single transition (line 20). This transition is triggered when the cursor enters the button on screen (`enter()`). The transition specifies its target `InUp`, which becomes the active state. The transition given line 39 associates some C++ code to the transition. This code is executed when the transition is triggered, performing the button action when the mouse button is released.

3.2 Hierarchical State Machines

In this section, we present more formally the syntax of the HSM control structure and its semantics. Since we are extending C++, we use C++ definitions of type, identifier and statement.

3.2.1 States

A HSM has a name, can define variables and inputs, and can specify initialization, enter and leave actions. It can contain sub-HSMs, transitions and some other constructs described below. The syntax for defining a HSM is:

```
hsm Name { content }
```

```
01 hsm Button {
02    hsm Disarmed {
03       // variables
04       var svg::SVGElement *elem;
05       var svg::SVGGElement *armed = 0;
06       var svg::SVGGElement *disarmed = 0;
07
08       // initialization
09       init {
10          hsm::svg::Childs(elem).assert(armed)
11                                .assert(disarmed);
12          elem->removeChild(armed);
13       }
14
15       // visual aspect coherence
16       enter { elem->replaceChild(armed, disarmed); }
17       leave { elem->replaceChild(disarmed, armed); }
18
19       hsm OutUp {
20          - enter() > InUp
21       }
22
23       hsm InUp {
24          - leave() > OutUp
25          - press() > Armed::InDown
26       }
27
28       hsm OutDown {
29          - enter()   > Armed::InDown
30          - release() > OutUp
31       }
32    }
33
34    hsm Armed {
35       hsm InDown {
36          - leave() > Disarmed::OutDown
37
38          // action invocation
39          - release() { doIt(); } > Disarmed::InUp
40       }
41    }
42 }
```

Figure 9: Hierarchical state machine specifying the behavior of a button

Name is an identifier starting with a capital letter, and content is a succession of any number of the declarations below, and zero or more sub-HSMs.

Initial HSM

When several HSMs are defined inside a parent HSM, the first one is the *initial* HSM. When a HSM is entered (after a transition), it sets its own current state to its initial HSM, which is then entered, and so on recursively. However, to put the HSM in a state consistent with its inputs, it is possible to specify rules for choosing the initial state as follows:

```
[condition] : Name
```

Name should be one of the direct sub-HSMs. The condition can be any boolean expression involving variables or inputs that are in the scope of the HSM.

Variables

HSMs can declare variables of any type, initialized or not. Variables are accessible inside any action of the HSM. The scope of variables in the HSM hierarchy is unusual. First, nested HSMs cannot access their parent's variables. Second, if a variable is declared and not initialized, it is aliased with an implicitly declared variable in the parent HSM that has the same type and name. As a result, a parent HSM can access inner HSM variables when they are not initialized. This process recurs to the top, unless a parent HSM explicitly initializes the variable. The instantiation of the top HSM

will then require the specification of all the nested non-initialized variables. The syntax for declaring variables is one of the following (the first one is the declaration without initialization):

```
var type identifier;
var type identifier = value;
var type identifier(value, ...)
```

Inputs

An input is a special variable that can trigger transitions. It has the same scope properties as regular variables and is declared and defined as follows:

```
in identifier;
in identifier = value;
in identifier(value, ...)
```

3.2.2 Transitions

Transitions between HSMs are triggered by events it receives. They are driven by ordered rules. A typical rule has the following form:

```
- input.EVENT [condition] {
    code
} send(EVENT) > Target(var = value, ...)
```

EVENT denotes the type of event that can trigger the transition. Some predefined event types are provided to notify modification, creation and removal of components. To catch all event types one can use a wildcard (`*`). Omitting the event type is a shortcut for the most common event: modification. If the input is omitted, events coming from any sender and matching the event specification will trigger the transition.

A clause between square brackets specifies guards for the transition. The boolean condition must be true for the transition to occur. It should be calculable within the scope of the HSM. Multiple guards are or-ed together. Guards are optional.

The code is any set of C++ statements valid in the scope. It is executed when the transition is triggered. The execution occurs in the context of the current HSM, before leaving it. The broadcast section allows to generate events and dispatch them to other components. Thus HSMs can be used as event filters. Dispatching occurs right after executing the code and just before leaving the current HSM. Code and broadcast clauses are optional.

The last part of a transition defines the name of the target HSM. The name resolution rules are those of C++ namespaces. For example, one can specify a sub-HSM of a sibling HSM using scope qualification (e.g. `Armed::InDown` in Figure 9). The final part of the target clause allows to pass values to the target's variables, and can be omitted. If the target is the current state, the HSM is left and then reentered, with the associated code being executed. Conversely, if no target is specified, the HSM stays in the same state without leaving and reentering it.

Special Transitions

Two actions can trigger special transitions: the explicit invocation of a method and the firing of a timer. In the first case, the arguments passed by the method call are available within the code. In the second case, *ms* denotes the time in milliseconds after which the transition is fired, the timer being armed when the HSM is entered. In both cases, only the first part of the transition is special. The rest of the transition can consist of the same optional parts than any other transition, i.e. conditions, code, broadcast and target clauses:

```
- method(type var, ...) { code } > ...
- ms > ...
```

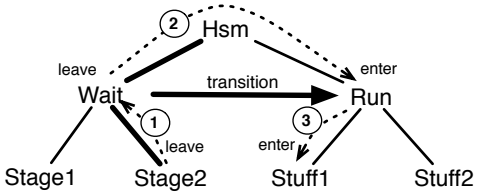

Figure 10: Target resolution
Lines denote HSM hierarchy, bold ones the active state, the bold arrow the transition, and dashed arrows the target resolution order.

3.2.3 Event Handling

An event triggering a transition is consumed by the transition. In some cases, it can be useful to propagate the event to the target HSM instead of consuming it. This can be done using a special arrow (**=>**) for the transition.

When a transition triggers, the target HSM is entered after a three-step process:

1. First, the HSM where the transition occurred is left. This process is recursive, starting by leaving the inner-most current HSM, and recursing upward until the source of the transition is reached.

2. Then, the target HSM is resolved. To perform this resolution, HSMs are left upward until a common ancestor of the source and target HSMs is found. Starting from this HSM, the branch leading to the target HSM is followed down and each HSM along this path is entered.

3. Finally, the target HSM is entered. The initial HSM of the target HSM is entered unless a *history* transition is specified. In this case, the last current sub-HSM is entered, and the rules specifying the initial HSM are ignored. This process is recursive, going downwards. To specify a history transition, a special arrow (**h>** or **=h>**) is used.

This process is illustrated in Figure 10. The three steps —leaving the current HSM, finding the target HSM, entering the target HSM— are labeled 1, 2, and 3.

Enter & Leave Actions

During this process, custom code can be executed by specifying enter and leave actions for each HSM (in Figure 9 such code is used to set the look of the button according to its current state):

```
enter { code }
leave { code }
```

Requirements

A last convenience structure, called requirements, is provided in order to handle exceptions. A requirement can take the following forms (the code section is optional):

```
require (condition) else { code } Target
! (condition) : { code } Target
```

The validity of the conditions is tested before entering the HSM. If the condition evaluates to false, the optional code is executed, and target resolution restarts with the supplied target as new goal. The target specification can include parameters as for a normal transition target.

```
01 hsm Translater {
02    [button] : Translating
03
04    hsm Idle {
05       - button > Translating
06    }
07
08    hsm Translating {
09       - button > Idle
10
11       hsm Op {
12          var hsm::SVGLWindow *w;
13          in point;
14
15          var hsm::Translate *op = 0;
16          var hsm::Point origin(2);
17
18          require
19             (op = w->pick< hsm::Translate >(point))
20              != 0
21          else
22             Nop
23
24          enter { origin = point; }
25
26          - point {
27             hsm::Point delta(2);
28             delta = point;
29             delta -= origin;
30             op->translate(delta);
31             origin += delta;
32          }
33       }
34
35       hsm Nop {}
36    }
37 }
```

Figure 11: HSM translating mouse action into dragging interaction

```
01 hsm::SVGLWindow *window = new hsm::SVGLWindow("test");
02 window->setDocument("test.svg");
03
04 hsm::Device *pointer = hsm::getDevice("Pointer");
05
06 Translater::Hsm translater(
07    window,
08    *pointer/"buttons"/hsm::Button::L,  // left button
09    *pointer/"position");            // mouse position
```

Figure 12: HSM initialization

4. SAMPLE APPLICATION

We present an example of HsmTk use illustrating how it can facilitate the incremental refinement and the reuse of existing interactions. Starting from a simple implementation of the dragging direct manipulation, we show how it can be easily enhanced to perform a constrained drag, and then how it can be multiplexed with another interaction using a control menu [19].

4.1 Simple Drag

Figure 11 show the implementation of the simple drag direct manipulation using HSMs. The `Translater` HSM has two sub-HSMs: `Idle` and `Translating`. Its first line (line 2) specifies that the initial state depends of the state of the mouse button when the HSM is initialized. Both `Idle` and `Translate` HSMs start with the definition of a transition (lines 5 and 9). The transitions make these states alternatively active when the mouse button is depressed and released. Together with the specification of the initial state (which depends of the initial mouse button state), they create and maintain an invariant: the `Translater` HSM is in `Idle` state when the mouse button is released, and in `Translating` state when it is pressed.

```
01 hsm ConstrainedTranslater {
02    [shift] : Constrained
03
04    in point;
05    in p = point;
06
07    hsm Normal {
08       hsm Translater;
09       - shift > Constrained
10    }
11
12    hsm Constrained {
13       var hsm::Point constr(2);
14       in point = &constr;
15       var hsm::Point origin(2);
16
17       enter { constr = origin = p; }
18
19       local void constrain(hsm::Point &c,
20                            const hsm::Point &o,
21                            const hsm::Point &p) {
22          /* details omitted */
23       }
24
25       - p { constrain(constr, origin, p); }
26
27       hsm Translater;
28       - shift > Normal
29    }
30 }
```

Figure 13: Interaction refinement through reuse

The `Translating` initial sub-HSM, `Op`, actually performs the translation. It defines some variables and one input (lines 12 to 16). It then defines a requirement (lines 18 to 22) that should be met before entering in this state: the picking inside the window (`w`), at the cursor position (`point`) should return an object that implements the `hsm::Translate` interaction protocol[3]. If the requirement is not met, the active state becomes `Nop` (line 35) which is a "do nothing" state. The sub-HSMs `Op` and `Nop` allow to stay in the `Translating` state according to the mouse button state, while performing or not the translation action —the actual action being performed depending on the presence of a compatible object under the cursor.

If such a compatible object is found, `Op` becomes effectively active. Upon enter, line 24 stores the origin of the interaction. The transition defined lines 26 to 32 moves the object each time the mouse is moved. This transition has no target, so `Op` remains active when it is triggered. Only the transition of the parent state `Translating` defined line 9 will leave `Op` or `Nop` and the `Translating` when the mouse button will be released.

To use this HSM, one must instantiate it and give it explicitly values for its undefined variables and inputs (`w`, `button`, and `point`). Figure 12 shows the minimal code creating a window, loading a SVG document inside it, requesting an object representing the pointer, and initializing the `Translater` HSM we have just defined.

4.2 Constrained Drag

The preceding interaction technique is simple. However, it can be reused as a basis to create refined variations. Figure 13 shows a HSM implementing a constrained drag when the *shift*

[3]HsmTk provides a mechanism to declare interaction protocols as a set of methods implemented by an object. The `pick` method (line 19, Figure 11) performs a typed picking, returning only objects of the requested type —thus implementing the desired interaction protocol— present below the mouse cursor.

key is depressed, and a normal drag when it is not. The `ConstrainedTranslater` HSM consists of two sub-HSMs, `Normal` (line 7 to 10) and `Constrained` (line 12 to 29), that becomes alternatively active when the *shift* key is pressed or not. As seen previously with the `Translater` HSM, the combination of an initial state specification (line 2) and two transitions (lines 9 and 28) insure the coherence of the active state with the *shift* key state.

Both sub-HSMs include the previously defined `Translater` HSM without needing to redefine it (lines 8 and 28). The `Normal` HSM includes it just as it is without altering its behavior. The `Constrained` HSM adapts its input by constraining it using an auxiliary function `constrain`. Since input are identified using naming conventions, the HSM redefines the `point` input as an alias to the constrained position `constr` (line 14). This constrained position is computed using the real position `p` defined by the top level HSM as an alias to the original `point` position (line 5). The constrained position is updated each time the input position changes by the transition defined line 25. The included `Translater` HSM uses this altered position as input, resulting in a constrained movement when the *shift* key is depressed.

4.3 Multiplexed Pan & Zoom using a Control Menu

Control menus are an extension of pie menus where the selection of the action and its continuous control can be performed in a single gesture [19]. Figure 14 shows a control menu multiplexing a pan and a zoom interaction on a single interaction device. If the movement begins vertically, a pan interaction is started (Figure 14 left) whereas if the movement is initiated horizontally, a zoom interaction begins (Figure 14 right). As with marking menus, it is not required to show the menu since advanced users learn quickly its layout. Thus the menu is displayed only after a short timeout when the mouse has not been moved after the button has been pressed.

Figure 15 gives an implementation of such a control menu using HsmTk. Once again, the top level `Controler` HSM consists of two sub-HSMs, one waiting the beginning of the interaction (`Idle`) and the other performing the real work (`Do`). This HSM consists itself of three sub-HSMs. Its initial state, `Choose`, handles the visual feedback and the interaction selection. The transition on lines 27 to 30 is triggered by the expiration of a timeout. It displays the circular menu reflecting the possible actions after 500 ms if no interaction has been selected so far. This menu is destroyed when the state is leaved (line 32). Transitions on lines 34 to 39 do select the appropriate interaction according to the principal direction of the movement. The pan and zoom implementations (lines 42 and 43) are not detailed since they use the same techniques as previously seen.

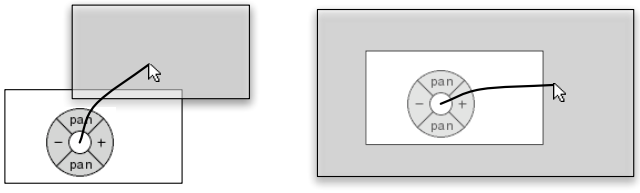

Figure 14: Pan (*left*) and zoom (*right*) multiplexed using a control menu

```
01 hsm Controler {
02    [button] : Do
03
04    hsm Idle {
05       - button > Do
06    }
07
08    hsm Do {
09       - button > Idle
10
11       hsm Choose {
12          var hsm::SVGLWindow *window;
13          var hsm::Point o(2);
14
15          var hsm::Translate *tl = 0;
16          var hsm::Zoom      *zm = 0;
17
18          var hsm::SVGLCursor *menu = 0;
19
20          enter {
21             o  = *point/hsm::Point::cast;
22             tl = window->pick< hsm::Translate >(o);
23             zm = window->pick< hsm::Zoom      >(o);
24             menu = 0;
25          }
26
27          - 500 {
28             menu = new hsm::SVGLFeedback(
29                       o, window, getMenuSVG(tl, zm));
30          }
31
32          leave { if(menu != 0) delete menu; }
33
34          - point [abs(o/Y - point/Y) > 10]
35          > Translate(translate = tl)
36
37          - point [abs(o/X - point/X) > 10]
38          > Zoom(zoom = zm)
39       }
40
41       hsm Translate { /* details omitted */ }
42       hsm Zoom      { /* details omitted */ }
43    }
44 }
```

Figure 15: HSM defining a control menu

5. DISCUSSION & PERSPECTIVES

HsmTk has been successfully used for a larger project: Indigo, a generic post-WIMP distributed application [8]. In this application, so-called conceptual objects are managed by client applications and sent to a rendering and interaction server (RIS) as XML streams. They are transformed into SVG fragments on the RIS side, using XSL rules specified by the application (and according to the capabilities of the display device). These SVG fragments are annotated to specify their interactive capabilities, and then presented to the user. When a particular behavior is unknown to the interaction server and no corresponding plug-in is found, the interaction server requests its implementation from the application, compiles it into a plug-in and loads it. The behavior is then ready for reuse by other object servers. We have developed a generic rendering and interaction server using HsmTk, as well as several client applications: a multi player game, a file system explorer and a simple chart editor.

Our use of the toolkit so far validates our approach: we have used it to implement a wide variety of techniques and have found that it makes simple things simple and complex things possible. Some limitations do however exist as well as room for improvement. For example, the SVG graphic model does not support visualization techniques using non linear transformations such as fisheye views. Another drawback is the lack of supporting tools. In particular, the consistency between SVG and HSM is checked only at run time,

leading to potential problems when only one of them is upgraded to a new version. Another tool is missing for the development of HSMs: at compilation time, C++ code is generated from the HSMs, and the errors reported by the compiler are relative to this generated code. Going back to the actual source of the problem in case of an error can be somewhat difficult.

6. RELATED WORK

Some toolkits do provide support for specific interaction needs: the Jazz and Piccolo toolkits [6, 5] provide a well-designed Java framework to build ZUIs. However, the graphical model of these Java-based toolkits is rather poor and there is no support for directly using the work of graphical designers. Similar limitations apply to the Ubit toolkit [16]. However, Ubit relies on the interesting brickget concept —a dynamic combination of fine-grained brick elements, e.g. box, text, image. This approach combines the advantages of scene graph and widget-based toolkits, but the behavior of objects is included directly in the brickget graph. Magglite [14] and IntuiKit [9] do provide a similar mixed-graph approach, where interaction specification and graphic specification are mixed together. The DoPIdom toolkit [4] relies on an annotation mechanism similar to our to add behavior to graphics.

Various types of finite automata have been used for programming interactions before the advent of GUIs, e.g., [12]. Statecharts [13] were designed for this purpose but their semantics is difficult to implement. Petri nets have clear semantics and can be used to perform some automatic verifications [1]. However, these formalisms can not easily be integrated into a programming language. They require a dedicated environment for programming and a runtime framework to support their execution. Dataflow or reactive languages have also been used to support interaction, together with state based approaches [15], or standalone as a visual programming language to configure the mapping between actual physical input devices and logical devices using ICON [11]. However, they usually require a visual representation, and they are inherently stateless.

7. CONCLUSION

In this paper we have described the HsmTk toolkit and illustrated its versatility through several examples. The most important aspects of HsmTk are the support for a rich graphical model that is compatible with the authoring tools used by graphic designers and the explicit support for interactions as a language construct that programmers can easily adapt to. HsmTk promotes reification, polymorphism and reuse, three principles that have proved important to user interface design [3]: HSMs reify the concept of an interaction; the decoupling of graphics (SVG) and interaction (HSM) encourages a form of polymorphism where the same interaction can be used in different contexts; the use of SVG for graphics and plug-ins for interaction encourages reuse of both graphical and interaction components. While supporting tools would help make HsmTk more usable for real-world development, we believe it provides a sound base for a new generation of user interface toolkits.

8. REFERENCES

[1] R. Bastide and P. Palanque. A petri net based environment for the design of event driven interfaces. In *Proceedings of the 16th International Conference on Application and Theory of Petri Nets*, 1995.

[2] M. Beaudouin-Lafon. Instrumental interaction: an interaction model for designing post-WIMP user interfaces. In *Proc. CHI'00*, pages 446–453, 2000.

[3] M. Beaudouin-Lafon and W. E. Mackay. Reification, polymorphism and reuse: three principles for designing visual interfaces. In *Proc. AVI'00*, pages 102–109, 2000.

[4] O. Beaudoux. DoPIdom : Une approche de l'interaction et de la collaboration centrée sur les documents. In *Proc. IHM'06*. ACM International Conference Proceedings Series, 2006.

[5] B. B. Bederson, J. Grosjean, and J. Meyer. Toolkit design for interactive structured graphics. *IEEE Trans. Soft. Eng.*, 30(8):535–546, 2004.

[6] B. B. Bederson, J. Meyer, and L. Good. Jazz: an extensible zoomable user interface graphics toolkit in java. In *Proc. UIST'00*, pages 171–180, 2000.

[7] E. A. Bier, M. C. Stone, K. Pier, W. Buxton, and T. D. DeRose. Toolglass and magic lenses: the see-through interface. In *Proc. SIGGRAPH'93*, pages 73–80, 1993.

[8] R. Blanch, S. Conversy, T. Baudel, Y.-P. Zhao, Y. Jestin, and M. Beaudouin-Lafon. INDIGO : une architecture pour la conception d'applications graphiques interactives distribuées. In *Proc. IHM'05*, pages 139–146. ACM International Conference Proceedings Series, 2005.

[9] S. Chatty, S. Sire, J.-L. Vinot, P. Lecoanet, A. Lemort, and C. Mertz. Revisiting visual interface programming: creating GUI tools for designers and programmers. In *Proc. UIST'04*, pages 267–276, 2004.

[10] S. Conversy and J.-D. Fekete. The svgl toolkit: enabling fast rendering of rich 2d graphics. Technical Report 02/1/INFO, École des Mines de Nantes, 2002.

[11] P. Dragicevic and J.-D. Fekete. Input device selection and interaction configuration with ICON. In *Joint proc. HCI'01 & IHM'01*, pages 543–558, 2001.

[12] M. Green. A survey of three dialogue models. *ACM Trans. Graph.*, 5(3):244–275, 1986.

[13] D. Harel. Statecharts: a visual formalism for complex systems. *Sci. Comp. Prog.*, 8(3):231–274, 1987.

[14] S. Huot, C. Dumas, P. Dragicevic, J.-D. Fekete, and G. Hégron. The MaggLite post-WIMP toolkit: draw it, connect it and run it. In *Proc. UIST'04*, pages 257–266, 2004.

[15] R. J. K. Jacob, L. Deligiannidis, and S. Morrison. A software model and specification language for non-WIMP user interfaces. *ACM TOCHI*, 6(1):1–46, 1999.

[16] É. Lecolinet. A molecular architecture for creating advanced GUIs. In *Proc. UIST'03*, pages 135–144, 2003.

[17] B. A. Myers. Separating application code from toolkits: Eliminating the spaghetti of call-backs. In *Proc. UIST'91*, pages 211–220, 1991.

[18] B. A. Myers and M. B. Rosson. Survey on user interface programming. In *Proc. CHI'92*, pages 195–202, 1992.

[19] S. Pook, É. Lecolinet, G. Vaysseix, and E. Barillot. Control menus: excecution and control in a single interactor. In *Ext. abstracts CHI'00*, pages 263–264, 2000.

[20] B. Shneiderman. Direct manipulation: a step beyond programming languages. *IEEE Comp.*, 16(8):57–69, 1983.

[21] P. Wegner. Why interaction is more powerful than algorithms. *Com. ACM*, 40(5):80–91, 1997.