Facilitating post-WIMP Interaction Programming using the Hierarchical State Machine Toolkit

Renaud Blanch

LRI & INRIA Futurs[⋆], Université Paris-Sud,
F-91405 Orsay Cedex, France
blanch@lri.fr

Abstract. Developing interactive programs is difficult because of the poor support for programming interactions in programming languages and the poor support for creative graphic design in traditional toolkits. This paper presents the Hierarchical State Machine Toolkit (HsmTk), a toolkit targeting the development of rich interactions. HsmTk has been designed to accommodate the needs of its users, namely programmers as well as interaction and graphic designers. It features a new control structure that makes interactions first class objects by extending C++ with hierarchical state machines. It also features the use of Scalable Vector Graphics (SVG) as the graphic language, enabling graphical designers to specify high-quality interfaces. Together, these features enable a tight coupling between graphic and interaction design by designers and software development by programmers. The paper provides examples illustrating the development process that results from using HsmTk.

Keywords: Advanced interaction techniques, hierarchical state machines, HsmTk, instrumental interaction, post-WIMP interaction, software architecture

1 Introduction

Programming interactive applications is known to be difficult and costly [1]. 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 behaviours. Most user interface toolkits are based on *widgets*, such as menus or buttons, which can be assembled to create a complete user interface.

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

[⋆] projet InSitu – Pôle Commun de Recherche en Informatique du plateau de saclay – CNRS, École Polytechnique, INRIA, Université Paris-Sud.

manipulation [2] that led to the first graphical interfaces. Indeed, the color attribute is manipulated *in*directly, through a dialog box [3].

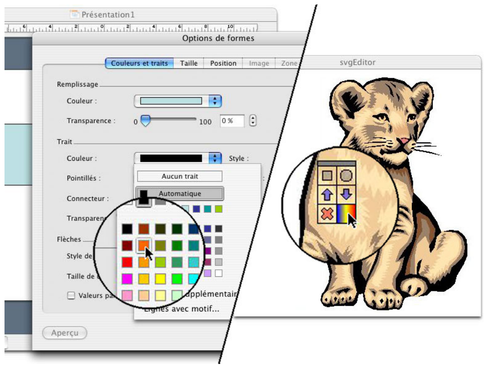

Fig. 1. Two interactions to set a color. Using a property box (*left*), or a transparent tool (*right*)

On the right of Fig. 1, the color is selected with a single click through a bimanual transparent tool or *toolglass* [4]: 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. Many post-WIMP techniques cannot be implemented as widgets and are therefore not available to developers using widget-based toolkits. The only alternative is to use program these techniques from scratch using low-level programming. The limitations of the WIMP approach and widget-based toolkits are illustrated by the amount of recent works conducted in the field of post-WIMP interaction toolkits [5–8]. Our goal, similar to some of this related work, is to make programming of hand-crafted interactions dedicated to a very specific need as easy as programming stereotypes interactions.

To achieve this goal, the Hierarchical State Machine Toolkit (HsmTk) provides the developer with multiple levels of abstraction. For example, the mouse can be seen as a device on its own, or as a two dimensional position and a separate set of buttons. If needed, the position can even be used as two unidimensional independent values. A collection of lightweight objects reduces the complexity of the development and helps create the proper levels of abstraction for the application at hand. Nothing is hidden, so developers can control what they are programming at any desired level, even the finest one. The toolkit was designed by applying a user-centered design approach where the user is the programmer using the toolkit. With traditional widget-based toolkits, the only alternative when no widget suits the application needs is to make do with the available widgets — at the expense of usability — or to develop the interaction from scratch — a very costly alternative. With HsmTk, the developer can choose the desired level of abstraction to start with, from the widget down to the low level physical device.

More precisely, HsmTk provides two main features:

– A system abstraction to access input and output devices as well as other system features required to program interactive systems, such as threads or timers.
– A programming abstraction designed to make interactions as first class objects, thus encouraging their factoring and reuse. To achieve this goal, we have extended the C++ programming language[1] with a new control structure for describing hierarchical state machines (HSM).

The benefits of our toolkit are introduced by first presenting a small scenario of development using HsmTk. Next, an example of advanced interaction technique, the continuous keyboard zooming, demonstrates the versatility and expressiveness of HsmTk. The next section describes the abstractions provided by HsmTk and the extension of the C++ programming language with the HSM control structure. Then we discuss the advantages of our approach for the development of interactive software and highlight the perspectives of our work. Finally, we compare our work with other research on interaction toolkits.

2 Development Scenario

In order to better understand the process involved in using HsmTk, let us observe Peter, a software programmer and Dave, a graphic designer, as they create the simplest interactive object: a button[2] (Fig. 2).

Defining the Button. Peter and Dave agree that a button is a widget that can be pushed and popped, and that triggers an action when it is released. The button has two visible states that Dave can begin to design with his favorite editor (Fig. 2 left, and Fig. 3 for the resulting SVG). A formal description of the button behaviour is given by Peter using a finite state machine (FSM) (Fig. 2 right).

However, in order to capture the subtleties of the behaviour, the FSM should be refined: when one pushes the mouse button inside the button, then leaves the button and reenters it, popping 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, and down again when it reenters the button. In order to refine the behaviour, Peter uses a hierarchical formalism *a la* statecharts [9] (Fig. 4).

Programming the Button Behaviour. To program the behaviour described on Fig. 4, Peter uses the control structure provided by HsmTk: the hierarchical state machine. He maps the chart in directly onto the HSM language (Fig. 5). He then implements the actions of the button using various HsmTk facilities: structured graphics manipulation of the button look using the SVG API, and event dispatching to notify an observer that the button has been clicked.

[1] C++ was chosen for performance reasons.
[2] HsmTk provides a default button implementation. We use this simple example to avoid the complexity of describing a new interaction and a new formalism simultaneously. A richer example is provided in the next section.

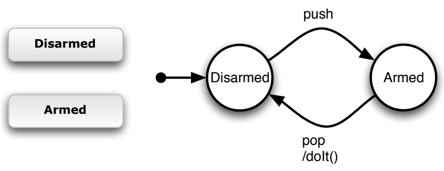

Fig. 2. The look (*left*) and the behaviour (*right*) of a button

```
<svg xmlns:hsm='hsm.insitu.lri.fr'>
   <defs>
      <!-- background gradient definitions, details omitted for readability -->
      <linearGradient id='armed' ... />
      <linearGradient id='disarmed' ... />
   </defs>

   <!-- button definition -->
   <g hsm:behaviour='Button'
      style='font-size:10; font-weight: bold'>

      <g><!-- armed version -->
         <rect width='80' height='20' rx='4' ry='4'
               style='fill:url(#armed); stroke:gray;' />
         <text x='19' y='13'>Armed</text>
      </g>

      <g><!-- disarmed version -->
         <rect width='80' height='20' rx='4' ry='4'
               style='fill:url(#disarmed); stroke:gray;'/>
         <text x='19' y='13'>Disarmed</text>
      </g>
   </g>
</svg>
```

Fig. 3. Button look enriched with behaviour binding

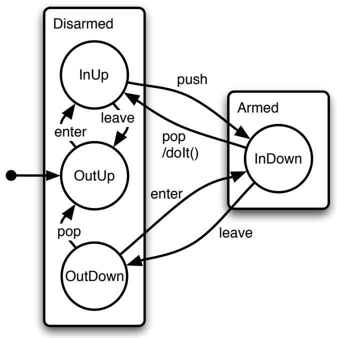

Fig. 4. A refined version of the button behaviour

The first lines of the declaration bind the button logic to the actual visual representation of the button. The compatibility between the implementation and the SVG document is insured by checking for the presence of the requested SVG fragments, specified as XPath expressions. This part of the code defines a *contract* expressing the requirements of the implementation. In this particular case, the SVG representation of a button must consist of a SVG element having at least two children (two SVG group elements) representing the two visual states of the button. This makes it easy to change the look of the button without changing the code — and thus without recompilation — and to reuse the same code across multiple applications. This feature is particularly useful to enable fast prototyping of interfaces and interaction techniques using coarse sketches while the graphics designer creates the final look of the application.
In order to glue together the behaviour with the visual representation, Peter has to annotate the SVG document produced by Dave (Fig. 3 shows the annotations in bold). This annotation system allows to specify a particular implementation for an object present in the SVG document. Here, a C++ object of class "Button" (corresponding to the HSM in Fig. 5) 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.

In summary, Peter and Dave have created a complete button from scratch. They have not just skinned a preexisting widget, they have defined a behaviour using a HSM, defined a look using a standard authoring tool to produce a SVG document, and connected the behaviour to the view of the widget. The SVG document shown on Fig. 3 just needed a simple annotation to make the binding between the representation of the button and its implementation. All the code needed is given on Fig. 5.

3 Sample Application: Continous Keyboard Zooming

Before describing the HSM language in more detail, we present a more advanced example. We have prototyped an interaction technique that uses the keyboard as a non positional input device to perform a simple gesture recognition. This technique can be used to zoom in and out, e.g. in a drawing program.

Concept. Graphics software with zooming capabilities use various interaction techniques to perform magnification. Some of them rely on the mouse, e.g. dragging with a particular button or modifier, using the mouse wheel, or clicking or dragging using a particular tool. Others rely on the keyboard, e.g. one key to zoom in, one to zoom out. Interaction techniques using the mouse can be difficult to use on laptops that only have a track-pad to control the cursor, while those using the keyboard can be tiresome when zooming across large zooming factors because the magnification step is fixed.

We propose to use a row of keys as a unidimensional tracking device. The interaction to zoom the object lying under the cursor consists of sliding the finger on the key row from right to left to zoom out and from left to right to zoom in (Fig. 6). Since the keys of the laptop keyboard are thin and soft, this movement is continuous.

```
hsm Button {
   // structural contract
   svg elem {
      armedLook is elem/SVGGElement[1]
      disarmedLook is elem/SVGGElement[2]
   }

   // initialization
   init { elem->removeChild(armedLook); }

   // logic
   hsm Disarmed {
      hsm OutUp {
         - enter() > InUp
      }

      hsm {
         - leave() > OutUp
         - push()  > Armed::InDown
      }

      hsm OutDown {
         - enter() > Armed::InDown
         - pop()   > OutUp
      }
   }

   hsm Armed {
      // changing the look of the button
      enter { elem->replaceChild(disarmedLook, armedLook); }
      leave { elem->replaceChild(armedLook, disarmedLook); }

      hsm InDown {
         - leave() > Disarmed::OutDown

         // performing the action
         - pop() broadcast(DO_IT) > Disarmed::InUp
      }
   }
}
```

Fig. 5. Button behaviour specified by a Hsm

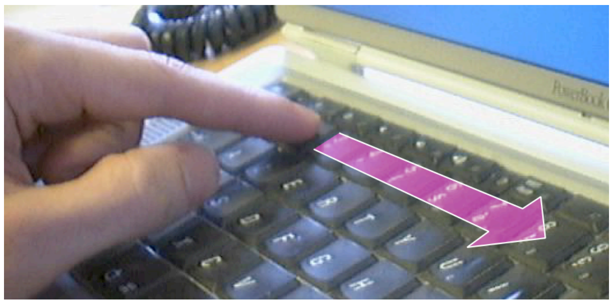

Fig. 6. Continuous keyboard zooming

Implementation. The continuous keyboard zooming can be implemented easily using HsmTk. The major issue with such an interaction is that the timing of actions has semantics. Pressing a single key does not perform any action whereas the successive press of two contiguous keys zooms in or out depending on the order of their activation. The HSM used to perform the keyboard zooming (Fig. 7) consists of two main states, a do-nothing `Idle` HSM and an active `Do` HSM. The latter is entered when a key is pressed. The requirement ensures that a zoomable object is located at the picking position[3]. If this requirement is not met, then the `KZoomer` HSM goes back to the idle state.

```
hsm KZoomer {
   hsm Idle {
      // waiting for a key press to enter the Do HSM
      - key [event.key != 0] > Do::Zoom(startKey = event.key)
   }

   hsm Do {
      var hsm::SVGLWindow &window;
      var hsm::Point &p;

      var hsm::Zoom *zoom = 0;

      // ensuring the presence of a zoomable object
      require ((zoom = window.pick< hsm::Zoom >(p)) != 0) else Idle

      hsm Zoom {
         var char startKey = 0;
         var char currentKey = 0;

         // performing the actual zoom
         - key [(currentKey = event.key) >= '1'] [currentKey <= '9'] {
            zoom->zoom(currentKey - startKey);
         } > Zoom(startKey = currentKey)

         // waiting at most 200 ms between successive keys stops zooming
         - 200 > Idle
      }
   }
}
```

Fig. 7. Keyboard zooming

The `Do` HSM has a `Zoom` sub-HSM that performs the actual zooming operation when successive keys are pressed, using the distance between the two pressed keys as value for the zoom factor. The meaning of "successive" is encoded by a timer-activated transition. This transition puts the HSM back into the idle state after 200 ms if no key has been pressed during this delay. The timer is automatically reset when reentering the state after a transition has been activated by a key, so this delay is only counted after the last key press event, and the HSM loops in the active state while keys are pressed.

[3] The pick function returns a non null pointer on a zoomable object if such an object exists at the picking position. Otherwise, it returns null.

4 The Hierarchical State Machine Toolkit

HsmTk provides several abstractions and a control structure adapted to interaction programming. We present both aspects with a stronger emphasis on the latter since it is the main originality of HsmTk.

4.1 System Abstraction

Programming interactions can require fine tuning of the code, because details are important for the usability of interaction techniques. For example, navigating hierarchical menus is a lot easier if proper timeouts are built in so that the user can move diagonally from an item to its submenu. The lowest level of detail provided by HsmTk is a thin portable layer on top of the operating system to help make code portable across platforms[4]. The highest level is the component abstraction, which can be a widget, a device such as the mouse, a hierarchical state machine, or any object that can interact with the user or with other components.

Low Level Abstractions. The low level abstractions deal with services usually provided by the operating system. HsmTk provides threading and synchronization objects, interprocess communication facilities, and management functions. It provides a framework for plug-ins in order to support modularity and reuse. HsmTk also provides window management facilities based on the portable low-level graphic library OpenGL.

HsmTk also provides general mechanisms such as events for communicating between components, function objects to ease the manipulation of callbacks and active values that broadcast modifications when they are updated.

Device Abstractions. The only output device supported so far are the windows mapped on the screen. HsmTk uses SVG as a structured graphical model to describe the windows' content. Many editors, such as Adobe Illustrator, can export to SVG, making it easy to create high quality graphics. SVG supports gradients and transparency, arbitrary geometrical transformations, and more. Such features are not only useful to produce good looking interfaces, but they are necessary for many post-WIMP interactions such as transparent tools or zoomable user interfaces. HsmTk uses the svgl library[5] [10], based on OpenGL, to render SVG at interactive speeds.

The input device model provided by HsmTk is hierarchical. For example, a mouse can be seen as a whole device, or as a position and a set of buttons, or even as two distinct coordinates, and two distinct buttons. Depending on the platform, USB devices can be plugged in and out dynamically. HsmTk also supports Wacom INTUOS tablets, making it easy to develop bimanual as well as pen-based interaction techniques.

[4] HsmTk runs on MacOS X, Unix/Linux, and Microsoft Windows.

[5] svgl is freely available at `http://svgl.sf.net/`.

Interaction Support. HsmTk provides support for programming interactions. It provides an extensible set of basic interaction protocols, e.g. pan/zoom, enter/leave or push/pop, and default implementation for the most common ones. Those protocols can be used as bricks to build more complex interactive behaviours such as drag-and-drop.

HsmTk uses a plug-in architecture that encourages reusing behaviours across applications. Once a behaviour has been written, it can be compiled into a plug-in, implemented as a dynamic library, and added to the interaction repository. When a SVG file is loaded, HsmTk checks for SVG elements that have an associated behaviour (specified with the `hsm:behaviour` attribute in Fig. 3) and instantiates these behaviours. When a behaviour is unknown, the library looks it up in the plug-in repository. If it finds it, it loads it and instantiates the resulting component. As a result, applications do not need to redefine generic behaviours, however they can override them to suit their needs.

4.2 Hierarchical State Machines

From the developer's point of view, the main feature of HsmTk is the HSM language extension. Interactive software often leads to code that is difficult to maintain and reuse and that can look like a "*spaghetti*" of callbacks [11]. 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" [12].

Formalisms adapted to the description and specification of interactions do exist. We propose to extend an imperative programming language with a control structure, the hierarchical state machine (HSM), borrowed from such a formalism. HSMs are close to Statecharts [9], the visual language chosen by UML to describe behaviours. HSMs are not equivalent to Statecharts however. For example, they take advantage of the textual form of the code to avoid nondeterminism: states and transitions have a natural order, unlike in a visual programming language. This order defines a precedence relationship that is used in case of potential ambiguity.

We now present the syntax of the HSM control structure and its semantics. Since we are extending C++, we use C++ definitions of type, identifier and statement.

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 }
```

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 (see the button variable on Fig. 5 for an example). 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, ...)
```

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 } broadcast(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 Fig. 5). 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 > ...
```

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 Fig. 8. The three steps — leaving the current HSM, finding the target HSM, entering the target HSM — are labeled 1, 2, and 3.

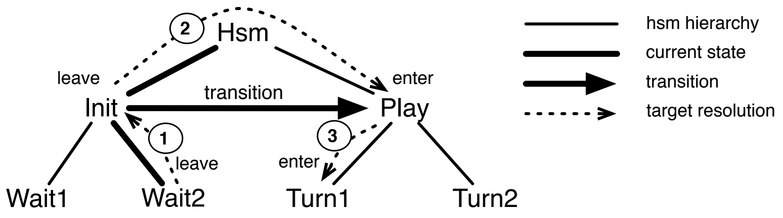

Fig. 8. Target resolution

Enter & Leave Actions. During this process, custom code can be executed by specifying enter and leave actions for each HSM (in Fig. 5 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 (see the `Zoom` HSM Fig. 7).

5 Discussion and Perspectives

We have used HsmTk to create sample applications. The largest project to date is Indigo, a generic post-WIMP distributed application. 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 behaviour 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 behaviour is then ready for reuse by other object servers.

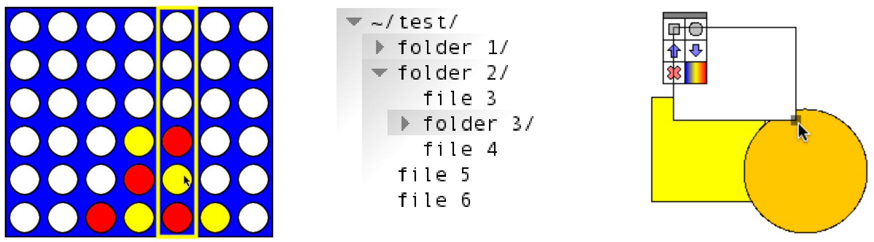

Fig. 9. Sample applications developed using HsmTk

We have developed a generic rendering and interaction server using HsmTk, as well as several client applications, some of which are shown in Fig. 9: a multi player game, a file system explorer and a simple chart editor. In the multi-player game (left), each player adds a token in a column by clicking on it. The coloured token then falls from the top of the screen to the lowest possible position in an animated motion. This game does not use any widget. Writing it required only a couple hours. The file system explorer (center) is based on a tree widget that performs the tree layout according to the opened folders, with a customized SVG representation. Folders can be opened and closed, and files can be moved around. The chart editor (right) uses bimanual interaction and a toolglass to perform simple chart editing.

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

Toolkits. A good example of a post-WIMP application is CPNTools [13], an environment designed to create, edit and simulate coloured Petri nets. The CPNTools visual editor uses advanced interaction techniques such as bimanual interaction, toolglasses [4] and marking menus [14]. At the time it was developed, no toolkit supported simultaneously multiple mice, transparency and zooming. So CPNTools was developed from scratch, highlighting the need for a toolkit that allows rich graphical output and extensible support for multiple input devices.

Some toolkits do provide support for specific interaction needs: the Jazz and Piccolo toolkits [8, 15] provide a well-designed Java framework to build ZUIs. ICON [16] supports advanced input techniques for Java/Swing applications by customizing the mapping between available physical devices and interaction techniques. However, the graphical model of these Java-based toolkits is rather poor and there is no support for directly t using he work of graphical designers.

Similar limitations apply to the Ubit toolkit [5]. 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 behaviour of objects is included directly in the brickget graph. Magglite [6] and IntuiKit [7] do provide a similar mixed-graph approach, where interaction specification and graphic specification are mixed together.

Languages and Formalisms. Dataflow or reactive languages have been used to support interaction. For example, ICON [16] uses a dataflow visual programming language to configure the mapping between actual physical input devices and logical devices. However, they usually require a visual representation, and they are inherently stateless.

Various types of finite automata have been used for programming interactions before the advent of GUIs, e.g., [17]. Statecharts [9] 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 [18]. 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.

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 [19]: 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.

References

1. Myers, B.A., Rosson, M.B.: Survey on user interface programming. In: Proceedings of the SIGCHI conference on Human factors in computing systems, ACM Press (1992) 195–202
2. Shneiderman, B.: Direct manipulation: a step beyond programming languages. IEEE Computer **16** (1983) 57–69
3. Beaudouin-Lafon, M.: Instrumental interaction: an interaction model for designing post-WIMP user interfaces. In: Proceedings of the SIGCHI conference on Human factors in computing systems, ACM Press (2000) 446–453
4. Bier, E.A., Stone, M.C., Pier, K., Buxton, W., DeRose, T.D.: Toolglass and magic lenses: the see-through interface. In: Proceedings of the 20th annual conference on Computer graphics and interactive techniques, ACM Press (1993) 73–80
5. Éric Lecolinet: A molecular architecture for creating advanced GUIs. In: Proceedings of the 16th annual ACM symposium on User interface software and technology, ACM Press (2003) 135–144
6. Huot, S., Dumas, C., Dragicevic, P., Fekete, J.D., Hégron, G.: The MaggLite post-WIMP toolkit: draw it, connect it and run it. In: Proceedings of the 17th annual ACM symposium on User interface software and technology, ACM Press (2004) 257–266
7. Chatty, S., Sire, S., Vinot, J.L., Lecoanet, P., Lemort, A., Mertz, C.: Revisiting visual interface programming: creating GUI tools for designers and programmers. In: Proceedings of the 17th annual ACM symposium on User interface software and technology, ACM Press (2004) 267–276
8. Bederson, B.B., Meyer, J., Good, L.: Jazz: an extensible zoomable user interface graphics toolkit in java. In: Proceedings of the 13th annual ACM symposium on User interface software and technology, ACM Press (2000) 171–180

9. Harel, D.: Statecharts: a visual formalism for complex systems. Science of Computer Programming **8** (1987) 231–274
10. Convery, S., Fekete, J.D.: The svgl toolkit: enabling fast rendering of rich 2d graphics. Technical Report 02/1/INFO, École des Mines de Nantes (2002)
11. Myers, B.A.: Separating application code from toolkits: Eliminating the spaghetti of callbacks. In: Proceedings the 4th annual ACM Symposium on User interface software and technology, ACM Press (1991) 211–220
12. Wegner, P.: Why interaction is more powerful than algorithms. Communications of the ACM **40** (1997) 80–91
13. Beaudouin-Lafon, M., Lassen, H.M.: The architecture and implementation of CPN2000, a post-WIMP graphical application. In: Proceedings of the 13th annual ACM symposium on User interface software and technology, ACM Press (2000) 181–190
14. Kurtenbach, G.P.: The design and evaluation of marking menus. PhD thesis, University of Toronto (1993)
15. Bederson, B.B., Grosjean, J., Meyer, J.: Toolkit design for interactive structured graphics. IEEE Transactions on Software Engineering **30** (2004) 535–546
16. Dragicevic, P., Fekete, J.D.: Input device selection and interaction configuration with ICON. In: joint proceedings of HCI 2001 and IHM 2001. (2001) 543–558
17. Green, M.: A survey of three dialogue models. ACM Trans. Graph. **5** (1986) 244–275
18. Bastide, R., Palanque, P.: 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)
19. Beaudouin-Lafon, M., Mackay, W.E.: Reification, polymorphism and reuse: three principles for designing visual interfaces. In: AVI '00: Proceedings of the working conference on Advanced visual interfaces, ACM Press (2000) 102–109