Reification, Polymorphism and Reuse:
Three Principles for Designing Visual Interfaces

Michel Beaudouin-Lafon and Wendy E. Mackay

University of Aarhus, Department of Computer Science
Aabogade 34
DK-8200 Aarhus N - Denmark
+45 89 42 56 44 / +45 89 42 56 22

{mbl, mackay}@daimi.au.dk

ABSTRACT

This paper presents three design principles to support the development of large-scale applications and take advantage of recent research in new interaction techniques: *Reification* turns concepts into first class objects, *polymorphism* permits commands to be applied to objects of different types, and *reuse* makes both user input and system output accessible for later use. We show that the power of these principles lies in their combination. Reification creates new objects that can be acted upon by a small set of polymorphic commands, creating more opportunities for reuse. The result is a simpler yet more powerful interface.

To validate these principles, we describe their application in the redesign of a complex interface for editing and simulating Coloured Petri Nets. The cpn2000 interface integrates floating palettes, toolglasses and marking menus in a consistent manner with a new metaphor for managing the workspace. It challenges traditional ideas about user interfaces, getting rid of pull-down menus, scrollbars, and even selection, while providing the same or greater functionality. Preliminary tests with users show that they find the new system both easier to use and more efficient.

Keywords

Design principles, reification, polymorphism, reuse, direct manipulation, instrumental interaction, interaction model.

1. INTRODUCTION

Today's visual interfaces suffer from an overabundance of functionality: each successive version is marketed based on the number of new functions, with little regard to the corresponding increase in the cost of use. Simple things keep getting harder, as users spend more and more time deciding among an increasing variety of rarely or never-used options. Some users are at a breaking point and are less and less able to cope with new software releases [21]. Others have begun to

actively reject software upgrades and cling to older versions of products such as Microsoft Word (survey of Microsoft users, Business Week, 5 July, 1999).

New interaction techniques, such as toolglasses [4] and marking menus [17], have been proposed to reduce this trade-off between power and ease-of-use. Yet such interaction techniques tend to be developed in isolation, as the focus of a particular research project. While this is a critical first step, it is also important to understand how these techniques scale when combined with other techniques and are placed in the context of complex real-world applications. We also need to develop new interaction models that explain how these and other techniques can increase the functionality available to users without creating a corresponding increase in the cost of use.

This paper describes how three design principles, reification, polymorphism and reuse, have provided a framework for redesigning a complex tool for editing and simulating Coloured Petri Nets. Developed in the late 1980's, the Design/CPN tool used a then state-of-the-art WIMP (windows, icons, menus, pointing) user interface. The new tool, cpn2000, is the result of a participatory design process, in which users and designers have collaborated to recreate a tool that supports "Petri-Nets-In-Use". The goal is to provide Coloured Petri Nets developers with greater functionality through an interface that is more intuitive, efficient and pleasant to use; one that allows them to think in terms of Petri nets and not the mechanics of the interface.

We begin by describing the principles of reification, polymorphism and reuse and then describe the interface to cpn2000. We explain how these principles have influenced the design of the user interface and discuss how combining them helps address the trade-off between power and ease-of-use. We conclude with directions for future research.

2. DESIGN PRINCIPLES

Graphical user interfaces can be broadly defined as consisting of graphical objects and commands. Graphical objects are represented on the screen and commands can be applied to create, edit and delete them. Visualization techniques describe how to represent these objects while interaction techniques describe how to apply commands to them. Over time, users develop individual patterns of use that depend upon the available objects and commands, the particular application domain and the current context of use. The perceived "ease-of-use" of an interface depends upon many factors, including the effectiveness of the visual representation, the completeness of the command set and the support for efficient patterns of use.

We have developed three principles that address the issues surrounding objects, commands and patterns of use:

- *Reification* extends the notion of what constitutes an object;
- *Polymorphism* extends the power of commands with respect to these objects; and
- *Reuse* provides a way of capturing and reusing patterns of use.

2.1 Reification

Reification is the process by which concepts are turned into objects. For example, in a graphical editing tool, the concept of a circle is represented as an image of a circle in a tool palette. Reification creates new objects that can be manipulated by the user, thus increasing the set of objects of interest.

Instrumental Interaction [1] extends the principles of Direct Manipulation [26] by reifying commands into *interaction instruments*. An interaction instrument is a mediator between the user and objects of interest: the user acts on the instrument, which in turn acts on the objects. This reflects the fact that, in the physical world, our interaction with everyday objects is mediated by tools and instruments such as pens, hammers or handles. The menu items, tool buttons, manipulation handles and scrollbars seen in today's user interfaces are examples of interaction instruments. A scrollbar, for example, is both a visible object on the screen that can be manipulated by the user and also a command the user manipulates to scroll the document.

Turning commands into objects provides potentially infinite regression. Since instruments are objects, they can be operated upon by (meta)-instruments, which are themselves objects, etc. In real life, we see limited chains of regression, as we move our focus from pencils, to pencil sharpeners that sharpen pencils to screwdrivers that fix pencil sharpeners. In some user interfaces, menus and toolbar buttons can be reconfigured to tailor the interface: they become instrument objects that can be manipulated by meta-instruments.

Another example of reification is the notion of *style*: In a text editor such as Microsoft Word, a style is a collection of attributes describing the look of a text in a paragraph, e.g., the font and margins. The user can create and edit styles and apply them to paragraphs. Styles thus become objects of interest for the user.

Many graphical editors also reify a collection of objects into the notion of a *group*. Since a group is itself an object, it can be added to a group, giving way to arbitrarily large structures such as trees and DAGs. These structuring mechanisms can be found in a wide variety of interfaces.

2.2 Polymorphism

Polymorphism is the property that enables a single command to be applicable to objects of different types. Polymorphism allows us to maintain a small number of commands, even as reification increases the number of object types. This property is essential if we want to keep the interface simple while increasing its power.

Most interfaces include the polymorphic commands *cut*, *copy* and *delete*, which can be applied to a wide variety of object types, such as text, graphics, files or spreadsheet cells. *Undo* and *redo* can also be considered polymorphic to the extent that they can be applied to different commands.

Applying a command to a group of objects involves polymorphism at two levels. First, any command that can be applied to an object can also be applied to a group of objects of the same type by applying it to each object in the group. Second, any command can be applied to a heterogeneous group of objects, i.e. objects of different types, as long as the command has meaning for each of the individual object types.

2.3 Reuse

Reuse can involve previous input, previous output or both. Input reuse makes previously-provided user input available for reuse in the current context. For example, the *redo* command lets users repeat complex input strings without having to retype them. Output reuse makes the results of previous user commands available for reuse. For example, *duplicate* and *copy-paste* let users avoid re-creating complex objects they have just created.

Polymorphism facilitates input reuse because a sequence of actions can be applied in a wider range of contexts if it involves polymorphic commands. Prototyping environments such as Self and its Morphic user interface framework [22], which are based on cloning and delegation, support and even encourage a high level of input reuse.

Reification facilitates output reuse by creating more first-class objects in the interface which are then available for reuse. Thus, for example a Microsoft Word user can create a new style object by reifying the style of an existing paragraph or by duplicating an existing style object, modifying the copy and reapplying it. A more elaborate form of reuse obtains when new styles are created through inheritance from an existing style, which allows changes made in the reused object to be propagated to the edited copies.

Macros, such as those found in Microsoft Excel, illustrate the power of combining these three design principles. The user begins by telling the system to "watch" as a sequence of commands is performed. Reification enables the user to capture the particular pattern of use as a sequence of commands that can be applied as a single new command to a new set of objects. A more advanced form of reification turns each component command into an object that can itself be edited, thus changing the pattern of use to accommodate different contexts.

The next section briefly describes the cpn2000 interface, which provides a testbed for exploring these three principles.

3. THE CPN2000 INTERFACE

The current cpn2000 interface was created over a period of ten months by a group of ten people. We followed a highly participatory design process beginning with observation of users of an existing system, Design/CPN, in various work settings. We developed scenarios to capture and articulate their work practices and engaged in a variety of video brainstorming and video prototyping exercises to develop the new interface. These activities involved a multidisciplinary group of user interface researchers, programmers and Coloured Petri Nets developers. The first version of cpn2000 was presented at the CPN International Workshop in October 1999. We also took advantage of the CPN Workshop and an earlier retreat for the University of Aarhus CPN group to conduct more formal studies using CPN developers who were not involved in the development of the new tool.

The following sections introduce the basic concepts and vocabulary of Coloured Petri Nets (CPN), the basic interaction techniques we selected and the overall design of the interface.
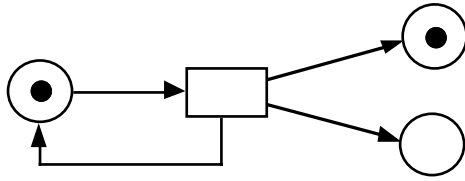
Fig. 1: A simple Petri net with three places, one transition and four arcs. Two places have a token.
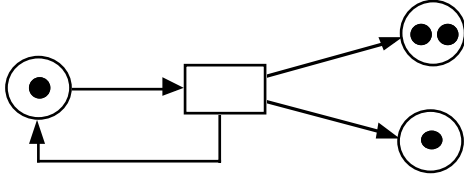

Fig. 2: The Petri net from Fig. 1 after the transition has been fired.

3.1 Coloured Petri Nets

Both Design/CPN and its successor, cpn2000, address the application domain of editing and simulating Coloured Petri Nets [14]. Petri nets are a graphical formalism with a strong underlying mathematical model that extends the power of simple finite state automata. Petri nets are particularly suited for the modeling and analysis of parallel systems such as communication protocols and resource allocation systems.

The graphical representation of Petri nets (Fig. 1) is a bipartite graph where the nodes are called *places* (depicted as circles or ellipses) and *transitions* (depicted as rectangles). Edges of the graph are called *arcs* and can only connect places to transitions and transitions to places. Each place typically represents a possible state or resource of the system. Places hold tokens, which represent the fact that the system is in a given state or the number of resources that can be allocated. The rules for simulating the net are very simple: a transition is *enabled* if all the places connected to it by an input arc have a token. *Firing* an enabled transition consists of removing a token from each input place and adding a token to each output place of the transition (Fig. 2). Mathematically, a Petri net can be represented by a matrix and simulation of the net is equivalent to a set of linear algebra operations. Properties of the net can be proven, such as the fact that the net has a bounded number of tokens or that there are no deadlocks.

A number of higher-level Petri net formalisms have been developed to model complex systems. Most of these formalisms are equivalent in power to a simple Petri net, but are much more concise. One such extension is Coloured Petri Nets [14]. In this model, the tokens belong to a *color set* equivalent to a data type in a conventional programming language. Arcs are labeled with pattern-matching expressions that describe which tokens are used when a transition is fired. Typically, colors allow a conventional Petri net to be "folded" onto itself, making models much smaller. In addition Coloured Petri Nets can be hierarchical. A transition can be described by a subnet, equivalent to macro-substitution in a textual language. Hierarchical nets make it possible to structure a complex net into smaller units that can be developed and tested separately.

Over the past decade, the CPN group at the University of Aarhus has been developing an editor and simulator for Coloured Petri Nets, called Design/CPN (Fig. 3). This tool is freely available to the CPN community and is currently in use by over 600
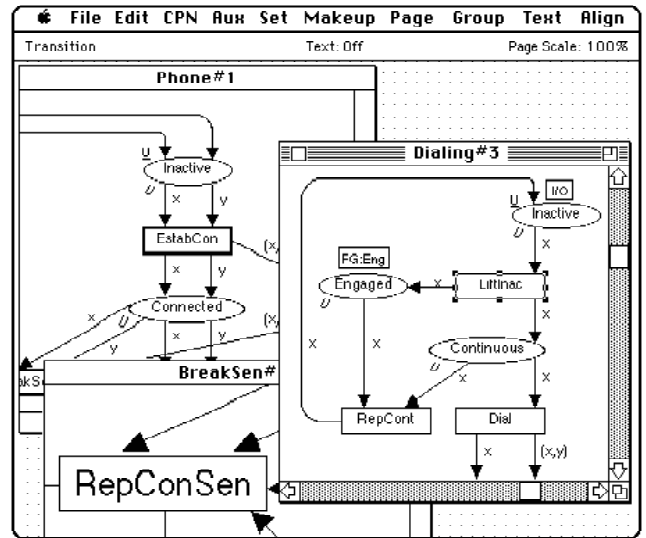

Fig. 3: Design/CPN, the current tool used by CPN designers.

organizations both in industry and academia. Design/CPN users have created models with as many as 100 modules and have run simulations lasting several days. The tool has been used far beyond the expectations of the designers and has reached its limits in terms of usability and complexity of implementation. The goal of cpn2000 is to reimplement the basic functionality of Design/CPN while improving the user interface and adding new editing and simulation capabilities. The project is a joint effort of the CPN, HCI and Beta groups at the University of Aarhus and is funded by the Danish Center for IT Research, Hewlett-Packard and Microsoft.

3.2 Interaction techniques

We began with two key decisions that have influenced many aspects of the design. First, we decided to explicitly support two-handed input, with a mouse for the dominant hand and a trackball for the non-dominant hand. The keyboard is used only to input text and to navigate within and across text objects. The design of the bi-manual interaction follows Guiard's Kinematic Chain theory [10] in which the non-dominant hand manipulates the context (container objects such as windows and toolglasses) while the dominant hand manipulates objects within that context. The exception is direct interaction for zooming and resizing, which, according to Casalta et al. [6], should give both hands symmetrical roles.

Second, we decided to incorporate a combination of new interaction techniques, rather than using a standard WIMP interface. Our goal is to provide cpn2000 users with easier yet more powerful tools and support more effective patterns of use. Users should be able to spend more time on developing Petri nets and less time on the mechanics of the interface.

The current version of cpn2000 incorporates four primary interaction techniques: direct interaction, marking menus [17], floating palettes, and toolglasses [4].

Direct interaction involves pointing directly at objects and either clicking on or dragging them. A direct bi-manual interaction, used for resizing and zooming, involves depressing a trackball button with the non-dominant hand and dragging the mouse with the dominant hand, as if stretching a piece of rubber.

Marking menus are radial, contextual menus that appear when clicking the right button of the mouse. Marking menus offer faster selection than traditional linear menus for two reasons. First, it is easier for the human hand to move the cursor in a given direction than to reach a target at a given distance. Second, the menu does not appear when the selection gesture is executed quickly, which supports a smooth transition between novice and expert use. Kurtenbach and Buxton [17] have shown that selection times can be more than three times faster than with traditional menus. Hierarchical marking menus involve more complex gestures but are still much more efficient than their linear counterparts.

Floating palettes contain tools represented by buttons. Clicking a tool with the mouse activates this tool, i.e. the user conceptually holds the tool in his or her hand. Clicking on an object with the tool in hand applies the tool to that object. In many current interfaces, after a tool is used (especially a creation tool), the system automatically activates a "select" tool. This supports a frequent pattern of use in which the user wants to move or resize an object immediately after it has been created but causes problems when the user wants to create additional objects of the same type. cpn2000 avoids this automatic changing of the current tool by getting rid of the notion of selection (see below) while ensuring that the user can always move an object, even when a tool is active, with a long click (200ms) of the mouse. This mimics the situation in which one continues holding a physical pen while moving an object out of the way in order to write.

Toolglasses, like floating palettes, contain a set of tools represented by buttons. Unlike floating palettes, they are semi-transparent and are moved with the non-dominant hand. A tool is applied to an object with a *click-through* action: The tool is positioned over the object of interest and the user clicks through the tool onto the object. The toolglass disappears when the tool requires a drag interaction, e.g., when creating an arc. This prevents the toolglass from getting in the way and makes it easier to pan the document with the non-dominant hand when the target position is not visible. This is a case where the two hands operate simultaneously but independently.

Since floating palettes and toolglasses both contain tools, it is possible to turn a floating palette into a toolglass and vice versa, using the right button of the trackball. Clicking this button when a toolglass is active drops it, turning it into a floating palette. Clicking this same button on a floating palette picks it up, turning it into a toolglass.

None of the above interaction techniques requires the concept of selection. All are contextual, i.e. the object of interest is specified as part of the interaction. This greatly simplifies the application's conceptual model and, one hopes, the users' mental models. However, this also creates a problem. Traditional interfaces use multiple selection to apply a command to a set of objects. We solve the problem by reifying multiple selection into objects called groups (see below).

We considered several other interaction techniques including gesture input [25], zoomable interfaces [2] and dropable tools [3]. We selected the above set partly due to the participatory nature of our design process, which led us to select the techniques most appealing and natural for our particular set of users. However, the techniques we chose also cover each of the different possible syntaxes for specifying commands:

- *object-then-command:* point at the object of interest, then select the command from a contextual marking menu;
- *command-then-object:* select a command by clicking a tool in a floating palette, then apply the tool to one or more objects of interest;
- *command-and-object:* select the command and the object simultaneously by clicking through a toolglass or moving it directly.

Preliminary results from our user studies [13] make it clear that none of these techniques is always better or worse. Rather, each emphasizes a different, but common, pattern of use. Marking menus work well when applying multiple commands to a single object. Floating palettes work well when applying the same command to different objects. Toolglasses work well when the work is driven by the structure of the application objects, such as working around a cycle in a Petri net.

3.3 Workspace manager

Coloured Petri Nets frequently contain a large number of modules. In the existing Design/CPN tool, each module is presented in a separate window and users spend time switching among them. Early in the project, it became clear that we had to design our own window manager to improve this situation: the Workspace Manager.

The workspace occupies the whole screen (Fig. 4) and contains window-like objects called *folders*. Folders contain *pages*, each equivalent to a window in a traditional environment. Each page has a tab similar to those found in tabbed dialogs. Clicking the tab brings that page to the front of the folder. A page can be dragged to a different folder with either hand by dragging its tab. Dragging a page to the background creates a new folder for it. Dragging the last page out of a folder removes the folder from the screen. Folders reduce the number of windows on the screen and the time spent organizing them. Folders also help users organize their work by grouping related pages together and reducing the time spent looking for hidden windows.

Cpn2000 also supports multiple views, allowing several pages to contain a representation of the same data. For example, the upper-left page in Fig. 4 shows a module with simulation information, while the upper-right page shows the same module without simulation information but at a larger scale.

The left part of the workspace is called the *index* and contains a hierarchical list of objects that can be dragged into the workspace with either hand. Objects in the index include toolglasses, floating palettes and Petri net modules. Dragging an entry out of the index creates a view on its contents, i.e. a toolglass, a floating palette or a page holding a CPN module.

Pages and folders do not have scrollbars. If the contents of a page is larger that its size, it can be panned with the left button of the trackball, even while the dominant hand is using the mouse to, for example, move an object or invoke a command from a marking menu. Getting rid of scrollbars saves valuable space but makes it harder to tell which part of the document is currently visible. A future version will display relative position information on the borders of the page during the panning operation in a non-intrusive and space-saving way.

Resizing a folder and zooming the contents of a page involves direct bi-manual interaction (as described above). Unlike traditional window management techniques, using two hands makes it possible to simultaneously resize and move a folder, or pan and zoom the contents of a page at the same time. Clicking the mouse on the page tab or on the folder pops up a contextual marking menu with additional commands, such as close, duplicate, collapse and expand.
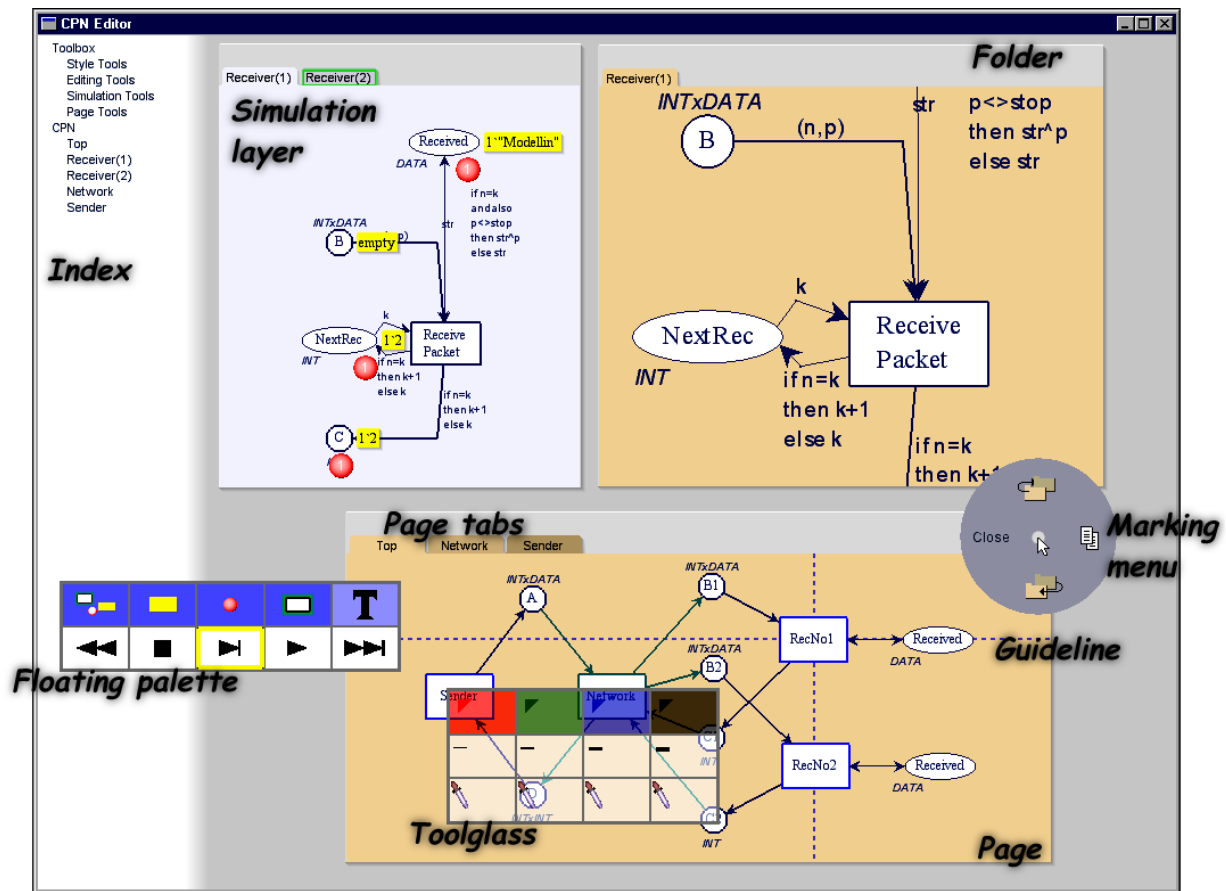
Fig. 4: The `cpn2000` interface. The index appears in the left column. The upper-left folder contains a page with the simulation layer active. The upper-right folder contains a view of the same page, at a different scale. The lower folder contains three pages: the top page shows a horizontal and a vertical magnetic guideline. The VCR-like controls to the left belong to the simulation floating palette. The toolglass in the center is positioned over objects on the page and is ready to apply any of the attributes shown. To the right, a marking menu has been popped up on a folder and is ready to accept a gesture to invoke one of the commands displayed.

3.4 Creating and Laying out Objects

Creation tools are accessible via any of the three interaction techniques. The user may select the appropriate object from the floating palette, move to the desired position and click, or use the non-dominant hand to move the toolglass to the desired position and click-through with the dominant hand, or move to the desired location and make the appropriate gesture from the marking menu.

Users of the current Design/CPN spend a great deal of time creating and maintaining the layout of their Petri net diagrams. The primary technique is a set of *align* commands, similar to those found in other drawing tools. The limitation is that they align the objects at the time the command is invoked, but do not remember that those objects have been aligned, unless the user groups them together into a new object. Aligning groups creates other problems, since the elements of the group cannot be edited further without ungrouping them, which is cumbersome and risks disturbing the alignment. Groups are also strictly hierarchical: an object cannot be a member of two independent groups. This makes it impossible to, for example, place an object both in a horizontal and vertical alignment group. We also observed that most current users use the same pattern to move an aligned object: They manually select all objects aligned to the object of interest and move them as a group. This dramatically slows down the interaction.

Many of the ideas from our early brainstorming sessions involved trying to make layout less cumbersome. We began by reifying the alignment command into alignment objects called *guidelines*. Graphic designers often use guidelines to define the structure of the layout and to position objects relative to this structure.

In the current version, we use horizontal and vertical guidelines. Guidelines are first-class objects that are created in the same way as the elements of the Petri net model, i.e. with tools found in a palette/toolglass or in a marking menu. Guidelines are displayed as dashed lines (Fig. 4) and are magnetic. Moving an object near a guideline causes the object to snap to the guideline. Objects can be removed from a guideline by clicking and dragging them away from the guideline. Moving the guideline moves all the objects that are snapped to it, thus maintaining the alignment. An object can be snapped to both a horizontal and a vertical guideline.

We have designed, but not yet incorporated, additional types of guidelines. For example, circular or elliptical guidelines would make it easier to layout the cycles commonly found in Petri nets. We also plan to support spreading or distributing objects

over an interval within a line segment, since this is a common layout that current users must implement by hand. Adding these new types of guidelines may create conflicts when an object is snapped to several guidelines. One solution is to assign weights to the guidelines and satisfy the alignment constraints of the guidelines with heaviest weight first. Such conflicts do not exist in the current system because only horizontal and vertical guidelines are available.

3.5 Editing Attributes

The tools to edit the graphical attributes of the CPN elements are grouped in a palette/toolglass that contains three rows: a row of color swatches, a row of lines with different thicknesses, and a row for user-defined styles (Fig. 4). The first two rows are fairly standard and are not described further here.

Tools in the last row correspond to the reification of groups of graphical attributes into styles. Initially, each tool in this row is a *style picker*. Applying this tool to an object copies the object's color and thickness into the tool and transforms the tool into a *style dropper*. Applying a style dropper to an object assigns the tool's color and thickness to that object. Applying a style dropper to the background of the page empties it and turns it into a style picker. If this is done by mistake, the *undo* command restores its previous state. In practice, style pickers and style droppers make it very easy and efficient for users to define the styles they use most often and apply them to objects in the diagram.

3.6 Simulation tools

Once a CPN model has been created, the developer runs simulations to validate it. Simulations correspond to testing traditional programs by running them on different data sets. The simulation of a Petri net is visual: tokens are put into places and go from place to place according to the rules described in the text inscriptions. CPN developers debug Petri nets by running simulations step-by-step or in larger chunks, similar to debugging traditional programs.

Cpn2000 displays simulation information in a *simulation layer* that can be added to any page via any of the three interaction techniques. When the simulation layer is active (Fig. 4), the background color of the page changes, the number of tokens are displayed as small red disks, the value of the tokens are displayed as yellow text annotations, and enabled transitions are displayed with a green halo. Each of these types of feedback can be toggled by the user using the tools in the simulation palette or toolglass.

Cpn2000 uses a VCR metaphor to control the simulation. *Next frame* lets the user select which transition to fire. *Play* randomly fires enabled transitions until a deadlock is reached or the user hits the *stop* button. *Fast-forward* runs the simulation for a maximum number of steps set by the user. *Rewind* resets the net to its initial state. The *Next frame* command is polymorphic: If applied to an enabled transition, it fires that transition. If applied to a page, it fires a randomly-selected transition within the page. If applied to a folder or to the workspace, it fires a randomly-selected transition within the pages of the folder or the whole model, respectively.

Our user studies showed that users are either interested in the results of the simulation, and thus do not want to change the underlying diagram, or they are interested in editing the diagram and usually do not need the results of the simulation. Therefore, in our design, a diagram cannot be edited in a page while the simulation layer is active, which makes it easier to adjust the location of the simulation feedback. The user can always edit the underlying diagram in a different page with the simulation layer turned off (Fig. 4).

4. USING THE DESIGN PRINCIPLES

Many factors have influenced our design, including observations of users, brainstorming ideas, selection of interaction techniques, knowledge of other systems and, of course, the three design principles. The design principles played two key roles. First, they helped us find a way to combine the interaction techniques, preserving and even increasing ease-of-use. Second, they helped generate new ideas that solved problems or increased the power of the system. The design principles served to define a design space that helped us both evaluate and generate potential solutions, which in turn helped us manage the trade-off between power and simplicity.

4.1 Grouping

The concept of a group is very powerful, resulting from the combination of reification and polymorphism. Groups encourage reuse, since creating a group implies that the user plans to work with that set of objects at a later time.

Several aspects of the cpn2000 interface take advantage of the notion of group. Folders represent groups of pages, magnetic guidelines represent groups of objects with layout constraints and styles represent groups of objects that share graphical attributes.

Through polymorphism, commands applied to the group operate on its members, with semantics that depend upon the command. Thus, activating the simulation layer on a folder means activating the simulation layer in each page of the folder, while firing a transition in a folder means firing one transition picked from among the set of enabled transitions in the pages of the folder, rather than firing a transition in each page. Similarly, applying a style to a guideline applies it to each object snapped to the guideline.

Groups allow users to work at a higher level of abstraction, which increases the power of the interface. Yet using groups is no more complex than using individual objects, since the same interaction methods apply to both.

4.2 Decomposition

Commands in traditional user interfaces tend to be coarse grained. In order to keep the number of commands reasonably low, designers often use dialog boxes to specify the ever-increasing number of arguments for each command. For example, specifying an alignment or editing a style usually requires a separate dialog box to specify the characteristics of the alignment or style.

Our approach leads to a more fine-grained set of commands, as illustrated by our solutions for alignments and styles. Instead of an overall align command, the user creates a guideline, then adds objects to it. Different types of alignment correspond to different types of guidelines, e.g., horizontal or vertical. Similarly, different parts of objects, e.g., object centers or sides, can be snapped to the guideline.

This decomposition of the alignment command makes it possible to dynamically add and remove objects to the alignment, unlike traditional align commands. New types of alignment, such as alignment around a circle or distribution along a line segment, can be added easily, without increasing the complexity of the interface.

Instead of creating styles explicitly, with a set of dedicated commands, styles are extracted from existing objects with the style picker tool. Applying a style to an object involves the same interaction as changing an object's attribute. Reification results in an increase of power through the notion of style, while polymorphism keeps the interface simple. In addition, fine-grained commands encourage a wider variety of patterns of use and therefore creates more opportunities for reuse and the opportunity to better meet the specific needs of each user.

4.3 Layers

User interface modes are often criticized in the HCI literature, since they make interfaces harder to use and less direct. At the same time, modes appear necessary in complex systems, helping to structure the interface according to the different activities of users. For example, when working with Petri nets, users make a clear distinction between editing the net and simulating it. Yet, in the earlier Design/CPN tool, users complained about having to switch explicitly between the two modes.

In order to address these issues, we reify the notion of mode into *layers*: activating a layer in a page gives the user access to the objects and commands relevant to a specific activity. For example, activating the simulation layer displays tokens and enabled transitions. Magnetic guidelines and text inscriptions are also accessible via separate layers. Users have the ability to define for themselves the complexity or simplicity of the visual interface, based on the number of activities they want to engage in simultaneously. Enabling several layers makes more objects directly accessible but may clutter the display while enabling a single layer allows the user to focus on a single activity. Since most activities are naturally separated through the context of the work, users are likely to display only the layers that are directly relevant to the problem at hand. Users thus control for themselves the trade-off between power for simplicity.

5. RELATED WORK

The work presented in this paper builds upon previous work in graphical user interfaces. The Xerox Star [15] and the Apple Lisa [23] have led to the desktop metaphor and the now-standard WIMP interfaces. Alternative models of the workspace have been proposed, including Rooms [12], the Information Visualizer [5], Translucent Patches [16] and Lifestreams [9]. Our design clearly draws from existing techniques. For example, the index is similar to the list of files and folders found in many file managers; Dragging pages into and out of folders resembles the manipulation of the palette tabs in Adobe Photoshop.

A number of systems have explored the reification of concepts into user interface objects, including ARK, the Alternate Reality Kit [27] and Self's Morphic user interface [22]. Dropable Tools [3] and the Raisamo and Räihä's Ruler [24] also fit into this design approach. Interaction models such as Direct Manipulation [26], Direct Combination [11] and Instrumental Interaction [1] have also strongly influenced this work.

Some of the interaction techniques described here have been applied to real-world applications. Marking menus and bi-manual zooming and resizing are used in the T3 prototype and in Alias|Wavefront's Studiopaint [18] and a variation of toolglasses is used in Maya [19]. Layers have been used as an architecture model [8] and in TicTacToon [7], a professional animation system.

6. SUMMARY AND CONCLUSIONS

This paper articulates three design principles, *reification*, *polymorphism* and *reuse*, and demonstrates how they, in conjunction with Instrumental Interaction [1], have affected the re-design of cpn2000, a tool for editing and simulating Coloured Petri Nets. One of our goals was to incorporate the latest research in interaction techniques. The design principles led us to the insight that we did not need the concept of selection, which in turn enabled us to combine and integrate four interaction techniques: direct interaction, marking menus, floating palettes, and toolglasses. These specific interaction techniques are of particular interest because they support each of the different patterns of use we observed during our studies of CPN developers at work. Marking menus emphasize the object of interest, floating palettes emphasize the command and toolglasses permit rapid switching between objects and commands as the focus of interest changes, such as working around a cycle in a Petri net.

Some users in our studies found that the new interaction techniques, marking menus and toolglasses, took a little bit of time to get used to. But everyone was able to successfully use all three tools after ten minutes of practice and all were more efficient than with the previous pull-down menus. Our approach accommodates individual preferences, allowing users to adapt the tools to their needs rather than having to adapt to the tool. Our preliminary experimental results show that different work contexts change the preferred patterns of use, even when the tasks are identical, which produces a corresponding difference in preference for tools.

Thinking about these design principles has also led us to re-examine a number of existing interaction features, extending or redefining them. The resulting interface has no menu bars, title bars or scrollbars, dramatically reducing visual complexity and allocating the extra space to users' objects of interest.

Reification is a property of Instrumental Interaction that allows the creation of first-class objects from a variety of concepts. When applied systematically, reification helps us to avoid some annoying aspects of existing interfaces, such as dialog boxes that seize control and block interaction with the rest of the interface until the dialog is complete. Polymorphism, which applies a single command to objects of different types, helps avoid the potential proliferation of commands that result from extensive use of reification. Explicitly considering reification and polymorphism together enables us to make specific design choices in the former that aid in the smooth execution of the latter.

Reuse allows us to capture patterns of use in the form of objects and commands and apply them in different settings. Simple reuse of reified objects and polymorphic commands directly improves the efficiency of the interface. More complex types of reuse provide subtle ways for users to capture and reflect upon their own work patterns, and subsequently tailor both the tool and their future work practices to meet the changing needs of the work and the work context. Tailoring choices need not involve choosing from long lists of difficult-to-interpret system options, but can be implicit records of work practices. As suggested in Mackay [20], reified patterns of use can be shared, enabling newcomers to quickly adopt the work style of the group and innovators to share their new strategies for accomplishing work.

Throughout our design process, these principles have helped us achieve our most basic goal, which is to provide users with

additional power and functionality, without simultaneously increasing the cost of use.

We have just begun the development of the next version of cpn2000, which will provide most of the functionality of the earlier Design/CPN tool. We will need to incorporate many more features, which will further test our claim that these principles make it easier to scale a prototype into a full-scale real-world application.

7. ACKNOWLEDGMENTS

8. REFERENCES

[1] Beaudouin-Lafon, M. Instrumental Interaction: An Interaction Model for Designing Post-WIMP User Interfaces. In *Proc. Human Factors in Computing Systems, CHI'2000*, ACM Press, 2000.

[2] Bederson, B. and Hollan, J. Pad++: A Zooming Graphical Interface for Exploring Alternate Interface Physics. In *Proc. ACM Symposium on User Interface Software and Technology*, UIST'94, ACM Press, 1994, p.17-26.

[3] Bederson, B., Hollan, J., Druin, A., Stewart, J., Rogers, D., Proft, D. Local Tools : an Alternative to Tool Palettes. In *Proc. ACM Symposium on User Interface Software and Technology, UIST'94*, ACM Press, 1994, p.169-170.

[4] Bier, E., Stone, M., Pier, K., Buxton, W., De Rose, T. Toolglass and Magic Lenses : the See-Through Interface. In *Proc. ACM SIGGRAPH*, ACM Press, 1993, p.73-80.

[5] Card, S., Robertson, G., Mackinlay, J. The Information Visualizer, an Information Workspace. In *Proc. ACM Human Factors in Computing Systems*, CHI'91, ACM Press, 1991, p.181-187.

[6] Casalta, D., Guiard, Y. and Beaudouin-Lafon, M. Evaluating Two-Handed Input Techniques: Rectangle Editing and Navigation. *ACM Human Factors In Computing Systems*, CHI'99, Extended Abstracts, 1999, p. 236-237.

[7] Fekete, J-D. TicTacToon: A Paperless System for Professional 2D Animation. *Proc. ACM SIGGRAPH*, ACM Press, 1995, p 79-90, 1995.

[8] Fekete, J-D. & Beaudouin-Lafon, M. Using the Multi-layer Model for Building Interactive Graphical Applications. In *Proc. ACM Symposium on User Interface Software and Technology*, UIST'96, ACM Press, p. 109-118.

[9] Fertig, S., Freeman, E. and Gelernter, D. LifeStreams: An Alternative to the Desktop Metaphor. Video, *ACM Human Factors in Computing Systems Adjunct Proceedings*, p 410-411, 1996.

[10] Guiard, Y. Asymmetric division of labor in human skilled bimanual action: The kinematic chain as a model. *Journal of Motor Behavior*, 19:486-517, 1987.

[11] Holland, S. & Oppenheim, D. Direct Combination. In *Proc. ACM Human Factors in Computing Systems*, CHI'99, ACM Press, p.262-269, 1999.

[12] Henderson, D.A. & Card, S.K. Rooms: The Use of Multiple Workspaces to Reduce Space Contention in a Window-Based Graphical User Interface. *ACM Transactions on Graphics* 5(3):211-243, 1986.

[13] Janecek, P., Ratzer, A., and Mackay, W. Petri-Nets-In-Use. In *Proc. International Workshop on Coloured Petri Nets*, Aarhus, Denmark, 1999.

[14] Jensen, K. *Coloured Petri Nets: Basic Concepts (Vol. 1, 1992), Analysis Methods (Vol. 2, 1994), Practical Use (Vol. 3, 1997).* Monographs in Theoretical Computer Science. Springer-Verlag, 1992-97.

[15] Johnson, J., Roberts, T.L., Verplank, W., Smith, D.C., Irby, C., Beard,M. and Mackey, K. The Xerox "Star": A Retrospective. *IEEE Computer* 22(9):11-29, 1989.

[16] Kramer, A. Translucent Patches: Dissolving Windows. In *Proc. ACM Symposium on User Interface Software and Technology*, UIST'96, ACM Press, 1996, p. 121-130.

[17] Kurtenbach, G. & Buxton, W. User Learning and Performance with Marking Menus. In *Proc. ACM Human Factors in Computing Systems*, CHI'94, ACM Press, 1994, p.258-264.

[18] Kurtenbach, G., Fitzmaurice, G., Baudel, T., Buxton. W. The Design of a GUI Paradigm based on Tablets, Two-hands, and Transparency. In *Proc. ACM Human Factors in Computing Systems*, CHI'97, ACM Press, 1997, p.35-42.

[19] Kurtenbach, G., Fitzmaurice, G.W., Owen, R.N., Baudel, T. The Hotbox: efficient access to a large number of menu-items. In *Proc. ACM Human Factors in Computing Systems*, CHI'99, ACM Press, 1999, p.231-237.

[20] Mackay, W.E. *Users and Customizable Software: A Co-Adaptive Phenomenon.* Ph.D. Dissertation, Massachusetts Instititute of Technology, 1990.

[21] Mackay, W.E. Triggers and barriers to customizing software. In *Proc. ACM Human Factors in Computing Systems*, CHI'91, ACM Press, 1991, p. 153-160.

[22] Maloney, J.H. & Smith, R.B. Directness and Liveness in the Morphic User Intertface Construction Environment. In *Proc. ACM Symposium on User Interface Software and Technology*, UIST'95, ACM Press, 1995, p. 21-28.

[23] Perkins, R., Keller, D.S. and Ludolph, F. . Inventing the Lisa User Interface. *ACM Interactions* 4(1):40-53, 1997.

[24] Raisamo, R. & Räihä, K-J. A New Direct Manipulation Technique for Aligning Objects in Drawing Programs. In *Proc. ACM Symposium on User Interface Software and Technology*, UIST'96, ACM Press, 1996, p. 157-164.

[25] Rubine, D. Specifying Gestures by Example. In *Proc. ACM SIGGRAPH*, ACM Press, 1991, p 329-337.

[26] Shneiderman, B. Direct Manipulation : a Step Beyond Programming Languages. *IEEE Computer* 16(8):57-69, 1983.

[27] Smith, R.B. Experiences with the Alternate Reality Kit: An Example of the Tension between Literalism and Magic. In *Proc ACM Human Factors in Computing Systems*, CHI+GI'87, ACM Press, 1987, p 61-67.