Concrete Browsing Of A Graphical Toolkit Library

Denys Duchier
Department of Computer Science
University of Ottawa

January 31, 1994

Abstract

The GUIDAR project aims to support the complete activity of Graphical User Interface Development And Reuse. We propose to organize the system as a collaborative architecture of independent automated agents that actively participate in the design and development process and promote reuse.

This paper introduces *Concrete Browsing* as an improved method of consulting a graphical library, and *Spreading Computation* as novel paradigm for search and retrieval.

A concrete browser allows the user to browse and interact with *prototypical instances* of graphical components, thus grounding the user's understanding in direct and concrete experience. We have implemented a prototype concrete browser for the GARNET toolkit.

We also present a combinator-based language with spreading computation semantics for expressing search through graphs of nodes and links such as an object-oriented graphical library. It serves as the mediating layer between the browser, and eventually all agents, and the library. Our prototype includes a graphical interface for editing queries in this language.

1 Introduction

Anyone who has ever attempted to design a user interface knows what a daunting task that can be. Large amounts of complex code are required to achieve the simplest interactions and configurations, and a tedious process of iterated experimentation is necessary to arrive at a design which is both aesthetically and ergonomically satisfactory.

Toolkits help alleviate the programming burden, but the granularity they afford is still quite small and typically not commensurate with the designer's conceptualization of the desired interface. The designer is stuck with a fixed set of primitive interface components and must start every new design from scratch.

Direct manipulation is a seductive technique to facilitate certain aspects of toolkit-based interface development. Using primarily the mouse, the designer is allowed to interactively assemble and modify the very components of the interface under construction. The chief advantage of this approach is the concreteness in which it grounds the development process. Yet, since there is only so much that can be conveyed with a pointing device, such systems necessarily have limited expressiveness. Moreover, the very concreteness of the process does not lend itself well to the elaboration of general, or abstract, designs.

We propose to overcome these limitations by encouraging the reuse of elaborate designs through improved browsing technology and case-based reasoning assistance. This research program is part of the motivation for the GUIDAR system, currently under development at the University of Ottawa, and whose purpose is to support the activity of Graphical User Interface Development And Reuse.

In order to promote reuse, it is necessary to locate, understand, evaluate, and adapt existing software components. In the first phase of the project, we tackled the problem of browsing libraries of graphical components. These libraries tend to be large and densely connected – primarily through the taxonomic and meronomic hierarchies – and require sophisticated ways of navigating and searching through them. We addressed this issue with a completely general, domain-independent, framework for searching and browsing densely connected graphs: it consists of an elegant language for expressing search programs using a small set of primitive combinators and of a novel execution paradigm, which we called *Spreading Computation*, that extends and generalizes the now classical AI technique of marker passing, or spreading activation.

To enhance browsing, we must not only make it easier to navigate and search the library, but also facilitate the understanding and evaluation of what is eventually retrieved. To this end, we introduced the notion of *Concrete Browsing* whose fundamental tenet is that the user should be allowed to browse the 'real' thing. Thus, when a query identifies a number of relevant components in the library, instances of these components are placed in the view where the user can look at them and interact with them. Furthermore, the user can select elements of a view to serve as the 'root set'

for subsequent queries, for example to find similar components modulo some similarity metric or to find other designs that use them as components. The latter possibility is presented as an example in this paper.

The GUIDAR project is investigating means of facilitating software development and promoting reuse through the application of knowledge-based, case-based, and machine learning techniques. However, rather than take a traditional approach and implement a CASE system with a fixed set of built-in passive capabilities – passive in the sense that it is the user's responsibility to exercise them – we envision an extensible collaborative architecture of independent automated assistants that actively participate in the design and development process, and promote the reuse of complex past designs.

In this paper, we introduce the idea of *Concrete Browsing*, which extends to library perusal the concreteness pioneered by direct manipulation systems. We also introduce *Spreading Computation*, an original paradigm of execution that generalizes marker passing and allows us to define an elegant combinator-based language for expressing search, evaluation, and retrieval strategies in object-oriented graphical libraries, and, more generally, in densely connected knowledge bases. At present, this language serves to support browsing and the programming of new search strategies by the user. We plan to use it as the common mediating layer between the library and all intelligent agents that we shall develop and plug into the GUIDAR architecture.

2 Concrete Browsing

The primary advantage of direct manipulation systems is the concreteness which they afford during the development process. We propose to harness this same advantage for the purpose of searching and browsing a library of graphical interface components.

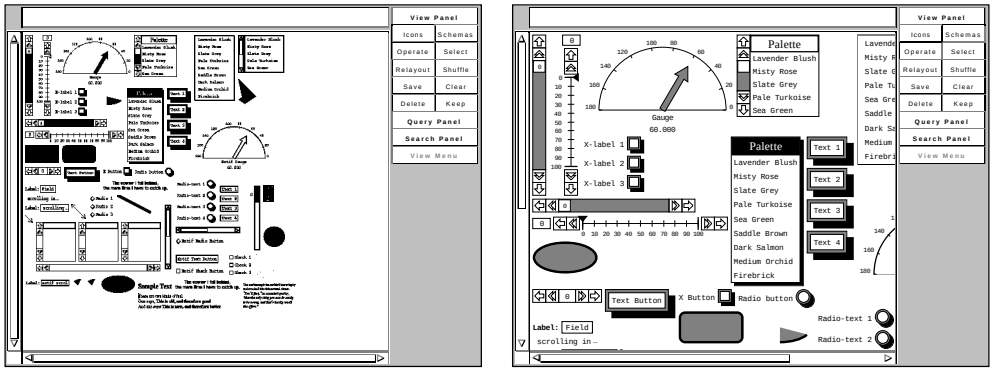
A concrete browser presents the user with a view, i.e. a subset, of the library. This view contains some number of graphical interface components. The user can directly see what they look like, and is able to interact with them (i.e. try them out) to understand what they do.

Naturally, a library will normally contain *generic designs* which can be instantiated in many different ways to obtain actual interface components. Since a generic design does not lend itself directly to concrete browsing, we must instead illustrate it using a small number of representative instances. For this reason, we introduce the notion of *prototypical examples*.

For each design in the library, a small number of well-chosen instances (typically, just one) must be supplied and become this design's set of prototypical examples. These are what the user gets to see displayed in a view. Each prototypical example provides a specific example of parameterization, instantiation and use and serves as a visual and interactive illustration of its design.

There is a secondary practical reason for introducing a distinguished set of prototypical examples: some interface components require a special protocol to insert them in a view.[1] Prototypical examples will be equipped with whatever additional handling protocol is necessary.

We selected the GARNET toolkit from CMU [4] to serve both as the application library and as the implementation vehicle. The GUIDAR concrete browser allows the user to search, inspect and interact with the contents of the GARNET library. In addition, it supports both actual and iconic presentations of the prototypical examples included in the current view. The figure below shows the two alternatives side by side for a view consisting of a large number of prototypical examples.



iconic view **actual view**

A concrete browser must address two issues: the search for information and its effective presentation. The GUIDAR system attends to the first one by supplying a general search language for expressing queries. It also makes available a graphical editor for visual programming in that language. The issue of presentation is addressed by a fast yet effective layout algorithm. The two figures above show that it is quite successful at arranging a large number of objects in a compact configuration.

3 Search Language

The GUIDAR search language is completely general and only makes the assumption that the application library can be described by a graph of nodes and links, where the nodes can be annotated. The execution paradigm can be summarized as follows: first, a subset of the nodes in the graph are somehow selected to serve as starting points. Then, search proceeds by propagation from this set of distinguished roots, evaluates

[1] For example, in order to place a component that involves a window $\mathcal{W}$ in the view, it is necessary to originally create $\mathcal{W}$ not as a top-level window, but as a subwindow of the view.

the nodes which are being visited and annotates them. Finally, interesting slices can be extracted from the graph by filtering on the basis of these annotations.

The process described above can be regarded as a generalization of marker passing, or spreading activation, where, instead of a simple marker, it is a program continuation that is being propagated. We characterize this mechanism as *spreading computation* and we have devised an efficient engine for it based on the concept of a *token passing graph* [3]. We have also developed a preliminary implementation in Concurrent ML [6] using *threads* and synchronous communication through *channels*.

Our approach not only subsumes traditionally keyword oriented retrieval systems, but also allows queries to take full advantage of the structure of the library and explore relationships by link traversal. The language is sufficiently powerful to express computationally challenging queries such as approximate structure matching.

By using the search language as the mediating layer on top of the graph description of the application library, we make it possible for users and automated agents alike to express equally sophisticated queries. Also, our approach, which abstracts away from the specifics of the library and provides a uniform interface for navigation, browsing, and search, should make independent automated agents easier to develop and plug into a cooperative architecture.

The complete language is an extension of LISP and will not be discussed here. We shall only describe the restricted version currently supported by the graphical editor. In order to more easily understand the language and its execution model it may be helpful to conjure up an analogy with UNIX processes and pipes.

In our language, a program fragment is like a process that reads an input stream of nodes (objects in the library) and writes as output another stream of nodes.[2] Complex programs are constructed by connecting the output port of one program fragment to the input port of another, i.e. by *piping* the former into the latter. A program fragment repeatedly reads a node from its input port, does some computation, and, as a result, may or may not write something to its output port. If it writes anything to its output port, it is either the node itself or the set of neighbours that can be reached from that node by traversing a particular link.

Nodes can be annotated. That is, they can bear named marks and named counters. For example, a search program might decorate certain nodes with the mark named selected to indicate that they should be included in the solution. Another program might use a counter to keep track of the degree of relevancy or matching for each component in the library.

The language currently contains the following primitive constructs:

[2] The whole truth is that these streams carry tokens rather than nodes, where a token is a pair ⟨node,data⟩ and data carries information such as partial results along the thread. To simplify the exposition, we do not make this distinction here.

(**mark** *name*)
Place the named mark as an annotation on the current node, then output the node.

(**incr** *name*)
Similarly, increment the named counter which appears as an annotation on the current node.

(**pipe** $e_1 \ldots e_n$)
This is the primary means of constructing complex programs. The output of each fragment e_i is piped into the input of the next fragment e_{i+1}.

(**branch** $e_1 \ldots e_n$)
This construct duplicates the incoming stream of nodes and sends a copy through each program fragment e_i. These fragments represent independent threads of computation, all writing to the same output stream.

(**to** *link*)
Spread this thread of computation to all neighbouring nodes that can be reached by traversing the named link.

(**when** *test*) (**unless** *test*)
Continue this thread of computation iff the node satisfies (resp. fails) the test.

(**repeat** *e*)
Each incoming node is written to the output port and also sent through the program fragment *e*. The output from *e* is fed back into the *repeat* loop's input port. The *repeat* construct is particularly useful for computing the transitive closure of a relation. For example:

(**repeat** (**to** :is-a))

will apply the remainder of this thread to the node and all of its ancestors.

(**until** *test e*)
If an incoming node satisfies the test, then write it to the output port, otherwise send it through the program fragment *e* and feed the resulting stream back into the *until* construct. For example:

(**until** prototypical? (**to** :is-a))

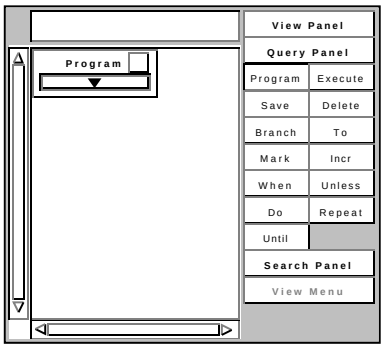
Reads an incoming node and initiates a new thread searching up the instance hierarchy for a prototypical example. If one is found, it is written to the output port and the thread terminates.

4 Query Editor

Query programs can be assembled interactively through a graphical editor. To illustrate the use of this tool, we are now going to write a program to select all prototypical examples of vertical scroll bars that can be found in the library.

The editor tool is invoked by selecting the Program option from the Query Panel. We are presented with a program frame (Figure 1) containing an empty pipe which we are going to fill by selecting constructs from the Query Panel. A small downward pointing triangular shape indicates the insertion point and can be set with the mouse.

Figure 1



In the GUIDAR library, each design is assigned a set of *facets*. At the moment, facets are simply keywords arranged in a loose conceptual hierarchy. The facet scroll-bar indicates a design that captures our intuitive concept of a scroll-bar, and the facet vertical is assigned to designs that have a vertical orientation. Therefore, our program need only search the set of prototypical examples, filter those that have both these facets, and mark them selected.

First, we are going to introduce a filter that lets through only those components that have facet scroll-bar. We select the When construct from the Query Panel which pops up a menu of tests (Figure 2). From this menu we choose the Facet option. We are then prompted for the name of the desired facet. In response we type scroll-bar (Figure 3).

Figure 2

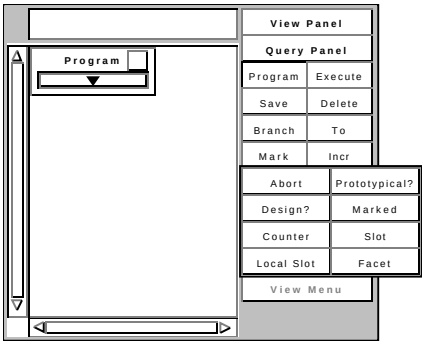
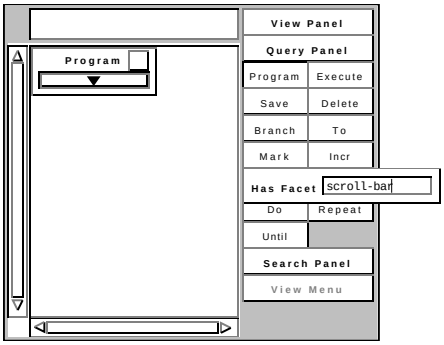


Figure 3



The system assembles the corresponding program fragment and places it at the insertion point (Figure 4). Following a similar interaction sequence, we introduce another filter that lets through only those components that have facet vertical (Figure 5).

Figure 4

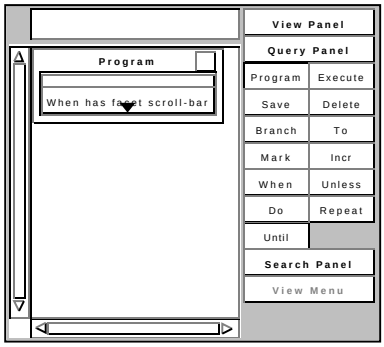
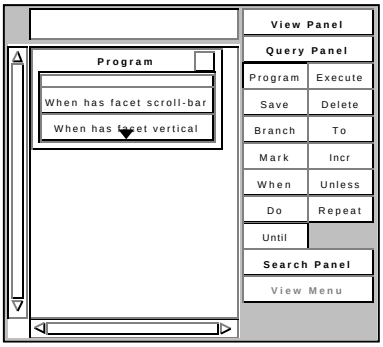


Figure 5



Finally, we mark as selected all the components that have made it successfully through both filters. We select the Mark option from the Query Panel and type in selected in response to the prompt (Figure 6).

Figure 6

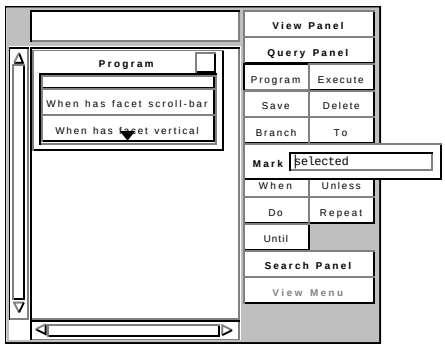
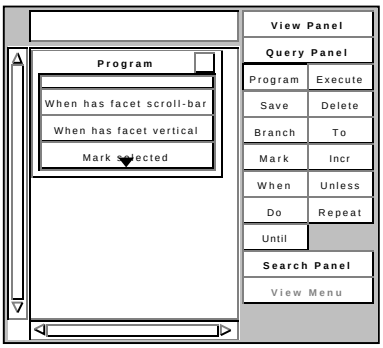


Figure 7



Before we can execute this query, we must specify the set of root nodes from which search will proceed and supply a selection criterion as well to extract the answer after the program terminates. After opening up the Search Panel (Figure 8), we click on the Library button to indicate that the root set should be the collection of prototypical examples recorded in the library.

Then we proceed to supply a selection criterion by selecting the When option, choosing the Marked entry from the pop-up menu of tests, and typing in selected in response to the prompt. Thus, the solution will consist of all prototypical examples bearing the selected mark.

Finally, we can initiate the query by selecting the Execute option from the Query Panel. The system compiles and runs the query program, then displays the result in the view. Three components were found (Figure 9).

Figure 8

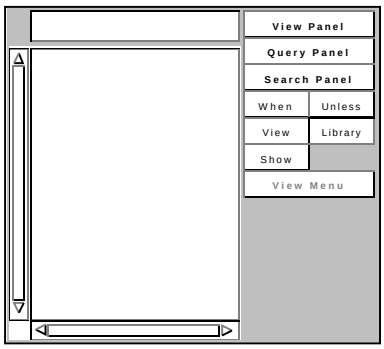
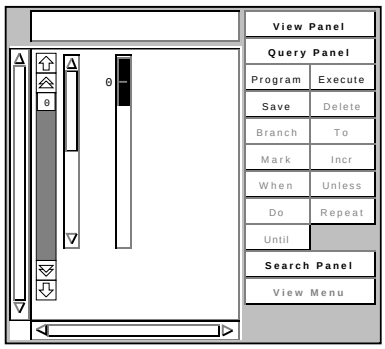


Figure 9



5 Searching Through The Library Structure

The GUIDAR search language allows us to formulate queries that take advantage of the very structure of the application library. In this section, we are going to demonstrate this ability by constructing a program to look for all designs that make use of any one of the three vertical scroll bars we have just found.

Before we get started, a brief introduction to the GARNET library structure is in order. The GARNET toolkit is implemented with an object-oriented language based on the prototype/instance model. As a consequence, the primary structure of the library corresponds to the specialization hierarchy as reflected by the is-a relation and its inverse is-a-inv. Naturally, complex interface components are constructed by aggregation thus yielding the usual *part-of* hierarchy embodied in the components and parent relations.

Let us now generalize the problem: given a collection of prototypical examples of components ($\mathcal{E}_i$), discover all prototypical examples of designs that include as a part some specialization of the design of any one of the $\mathcal{E}_i$'s.

Algorithm: For each $\mathcal{E}_i$, search up the is-a hierarchy for its design, i.e. an object which has been identified to GUIDAR as being a generic design. Find all specializations of this design using the transitive closure of the is-a-inv relation. For each specialization, climb up the parent hierarchy (aka. *part-of*) marking prototypical examples selected as they are being encountered.

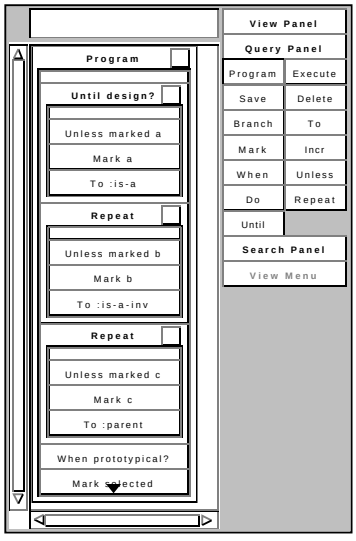
This algorithm can be expressed straightforwardly in our search language:

```
(pipe (until design? (to :is-a))
      (repeat (to :is-a-inv))
      (repeat (to :parent))
      (when prototypical?)
      (mark :selected))
```

Since search proceeds by propagation, we must consider the issue of redundant threads and the possibility of loops. Redundancy is primarily the result of a node being visited by several computationally indistinguishable threads, each proceeding to repeat the same work already performed by its predecessors. In the program above, many independent threads are created by the high branching factor when going down the is-a hierarchy. Some of these threads may subsequently meet when going up the parent hierarchy, at which point they become similar and only one need survive.

A useful technique to avoid redundant search is to leave a trail of marks and only explore regions that have not been visited before[3] as evidenced by these marks. Our program includes three link traversal instructions and we shall guard them with three distinct marks, namely a, b, and c. The source form and graphical representation of this program are shown side by side below.

```
(pipe (until design?
        (unless (marked :a))
        (mark :a)
        (to :is-a))
      (repeat
        (unless (marked :b))
        (mark :b)
        (to :is-a-inv))
      (repeat
        (unless (marked :c))
        (mark :c)
        (to :parent))
      (when prototypical?)
      (mark :selected))
```



[3] More precisely, which have not been visited for the same purpose before.

Extending the interface. We could simply execute the program. Alternatively, by selecting the Save option from the Query Panel, the user may cause the newly defined program to become an integral part of the GUIDAR interface. The user is prompted for a name (Figure 10), and a new button with that name is added to the Query Panel (Figure 11).

Figure 10

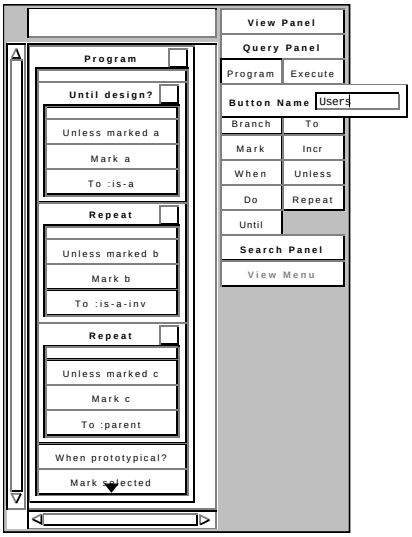
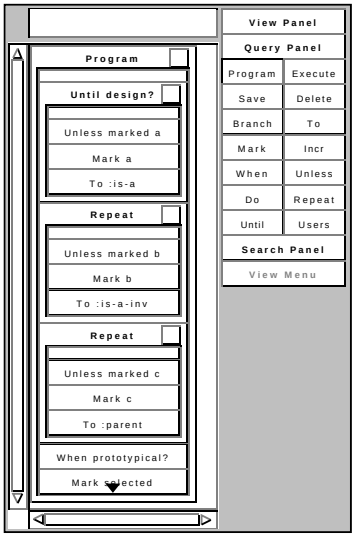


Figure 11



This button serves two functions: clicking it in browsing mode causes its program to be executed, whereas in programming mode, it simply inserts a representation of that program.

Before we execute the query, we must do one last thing: indicate to the system that the root set should be taken from the view rather than from the library. This is done by selecting the View option from the Search Panel. At last, we switch back to browsing mode (by toggling the Program button) and click on the newly created Users button (Figure 12). The query finds two scrolling menus and one directory browser (Figure 13).

Figure 12

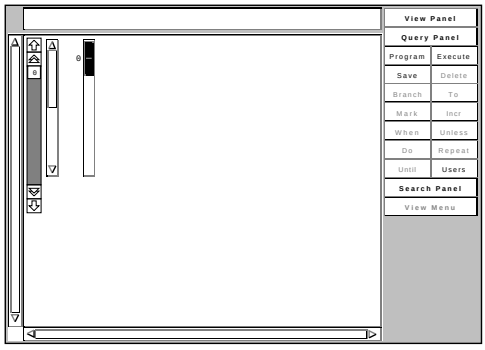
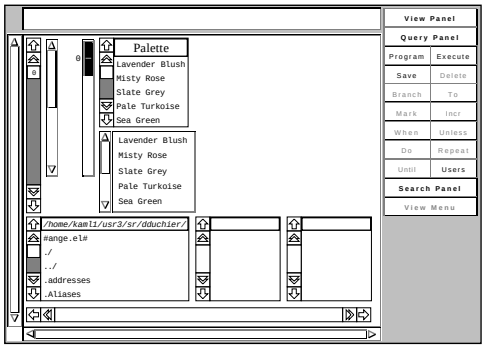


Figure 13



6 Conclusion

The GUIDAR system introduces the idea of concrete browsing to facilitate the development of graphical user interfaces and encourage the reuse of past designs. The user is able to display and interact with instances of graphical components available in the library, and thereby gains direct and concrete experience with them to better guide his decisions.

The browsing activity is supported by a powerful search language whose execution mechanism is based on the spreading computation paradigm. This language makes it possible for the user to express queries far beyond the limited capabilities of keyword oriented retrieval systems. In particular, the search process can take full advantage of the very structure of the application library by navigating through its network of links and relations while carrying out arbitrary computations to evaluate and annotate each component thus visited.

Query programs can be elaborated interactively through a perspicuous visual interface to a structure editing tool. Such programs can not only be executed, but may also be automatically integrated into the GUIDAR interface.

Future Work: An underlying theme of this project is that it should eventually provide the framework for a cooperative architecture of automated agents, all conspiring to facilitate and contribute to the design and development task. These agents will be able to search the library using the same programmatic interface available to the user. For example, we plan to implement an agent for *active browsing* as described in [1], who, by observing the user's browsing action, formulates and constantly refines a conjecture concerning the user's current interest, and, on the basis of this conjecture, consults the library to determine entries relevant to that interest, and non-intrusively offers suggestions for further browsing.

The development task can be further enhanced with such facilities as specification matching and similarity-based search by example. *Active browsing* itself can be formulated as an incremental and adaptive technique of automating relevance feedback

search. Our search language, drawing on its ability to annotate components with incrementally updated numeric values, is well-suited to the implementation of these strategies. More generally, we believe that it is a convenient means of expressing search strategies in densely connected knowledge bases.

The next immediate item on our agenda is to combine the GUIDAR browser with an interactive tool for constructing graphical user interfaces, such as GARNET's GILT and LAPIDARY tools, or the forthcoming MARQUISE [5]. However, the project now enters a more ambitious phase, namely the development of a module to support case-based design as well as the incremental refinement of specifications and requirements. We view this second phase as essential for the practical development of user interfaces and the promotion of software reuse, and as a necessary step towards fully supporting the idea of non-monotonic design [2].

References

[1] Chris Drummond. Automatic goal extraction from user actions to accelerate the browsing of software libraries. Master's thesis, University of Ottawa, Department of Electrical Engineering, 1992.

[2] Denys Duchier. Reuse and non-monotonic design of user interfaces. Technical Report TR-92-17, University of Ottawa, Department of Computer Science, Apr 1992.

[3] Denys Duchier. Implementing search strategies with token passing graphs. Technical report, University of Ottawa, Department of Computer Science, 1993.

[4] Brad A. Myers, Dario A. Giuse, Roger B. Dannenberg, Brad Vander Zanden, David S. Kosbie, Edward Pervin, Andrew Mickish, and Philippe Marchal. Garnet: Comprehensive support for graphical, highly-interactive user interfaces. *IEEE Computer*, 23(11):71–85, November 1990.

[5] Brad A. Myers, Richard G. McDaniel, and David S. Kosbie. Marquise: Creating complete user interfaces by demonstration. In *InterCHI'93*, 1993.

[6] John Hamilton Reppy. Concurrent programming with events – the concurrent ML manual. Technical report, AT&T Bell Laboratories, 1993.