Generating Direct Manipulation Program Editors

Michael Read & Chris Marlin
Department of Computer Science,
The Flinders University of South Australia,
G.P.O. Box 2100, Adelaide, S.A. 5001,
{read,marlin}@cs.flinders.edu.au

Abstract

Language specific editors are cognisant of the syntax and semantics of the programming language they manipulate. Despite the various potential advantages of language specific editors, they have not been widely accepted by software developers for serious software development. On the other hand, direct manipulation editors, which are also cognisant of the entities they manipulate, have proven to be successful in other domains such as drawing and VLSI design tools. Thus, it is worthwhile investigating the incorporation of direct manipulation mechanisms into program editors. This paper presents a technique for specifying direct manipulation editing of programs which is amenable to the generation of language specific editors incorporating direct manipulation from a specification of the desired editing mechanisms.

Keywords: Language specific editors, direct manipulation, program editing, generation of language specific editors, state machines.

1. Introduction

There have been many different approaches to addressing the problems of software productivity. One of these has been the development of *language specific editors*; these are editors which are specifically intended for the creation and modification of programs. There have been a large number of language specific editors developed over the last twenty five years [6]; particularly well known examples include the Cornell Program Synthesizer [16], Mentor [3], Magpie [2] and ALOE [8].

Such editors have the potential to offer significant advantages over text editors by providing help with language syntax, avoiding syntactic and semantic errors before attempting compilation, providing semantically based searching, and so on. Despite these potential advantages, language specific editors have not been widely used by software developers for serious software development. Although many of the reasons for the lack of widespread use of language specific editors are no doubt non-technical, it is worthwhile exploring whether there are possible editing mechanisms which may be sufficiently attractive and easy to use that some of the other barriers to using these tools will be overcome.

One editing paradigm which is used successfully in a number of areas is *direct manipulation*. This approach to editing is characterised by being rapid, incremental and reversible, and providing continual feedback[15]. This technique can be found in various editing tools, such as many drawing packages (e.g., MacDraw and Idraw) in which graphical representations of objects are manipulated by pointing, dragging, stretching and gesturing, and the results are immediately visible. Given the success of direct manipulation as an editing mechanism in other contexts, it appears to be worth exploring in the context of program editing. In particular, it may contribute to making language specific editors more attractive to potential users.

Over the history of language specific editors, there has been a clear trend from manually constructed editors to the development of appropriate technology to generate editors and environments from a description of the language to be supported by the environment. Examples include the development of the Synthesizer Generator[14] from the Cornell Program Synthesizer[16], and the progression from Mentor[3] to CENTAUR[1]. This trend has occurred for very good reasons, specifically:

- removing a lot of the tedious work required to obtain an editor for a programming language, a significant part of which overlaps with what has already been carried out to implement editors for other languages, and
- the desire to customise the appearance of the editor, say to conform to coding standards used with a particular organisation.

If direct manipulation mechanisms were to be introduced into language specific editors, we would not wish to lose the

0-8186-7525-X/96 $05.00

advantages of being able to generate an editor from a specification of the programming language and some aspects of its appearance. Thus, there is a need to explore how program editors incorporating direct manipulation editing mechanisms can be generated.

This paper describes a technique for specifying direct manipulation editing mechanisms for program editors. This technique is amenable to the automatic generation of the direct manipulation program editors from a specification.

The next section of the paper will give an overview of language specific editing, covering both textually based editing mechanisms and some sample direct manipulation mechanisms. Section 3 will discuss the notion of unparsing a structured representation of a program into a depiction of it which can be edited; this discussion will cover both the specification of textual unparsing and the requirements for a technique which would accommodate the specification of unparsing to a representation which can be edited using direct manipulation. Sections 4, 5 and 6 will present our technique for specifying direct manipulation editing of programs. Finally, the future work planned to achieve the goal of automatically generating these editors will be presented.

2. Program editing

Language specific editors assist the programmer by providing help with the syntax of the language and can also provide information based on the static semantics of the program. This is done by maintaining a structured representation of the program that is being edited. This approach contrasts with the alternative of sending the entire program to a compiler and awaiting the results. Thus, more meaningful structural and semantic information is available earlier in the development of the program, allowing an editor to take advantage of this information. A common underlying structural representation used by many language specific editors is an abstract syntax tree (AST); the rules for constructing an AST are defined in an abstract syntax, which is a description of the syntax of the language with the concrete syntax (i.e., keywords, etc.) removed. An advantage of using abstract syntax is that there are many proven techniques to extend the abstract syntax to maintain the semantic information needed during semantic analysis (e.g., [14]).

One example of how language specific editors can guide the programmer through the syntax of a language is through the use of templates. These templates contain all the keywords and punctuation characters required by the construct, along with placeholders for the other components. By selecting a placeholder, the programmer is presented with a choice of templates that are valid substitutions for the placeholder. For example, Figure 1 shows the *stmt_seq* placeholder selected in a language specific editor, with the possible templates listed to the left. When the selection is made all the associated keywords, text and placeholders that make up that construct are inserted, as shown in Figure 2, for the case of an **if** statement.

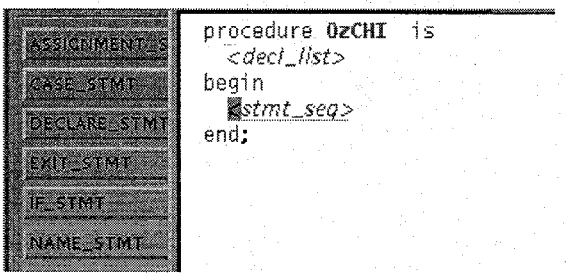

Figure 1. Templates in a language specific editor.

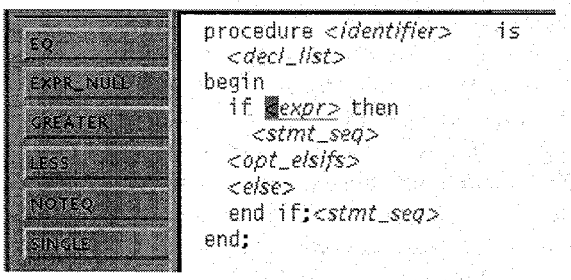

Figure 2. Templates in a language specific editor.

One disadvantage of this template style of editing is that it enforces a top-down approach to editing, as discussed further in [10]. To offset this shortcoming, many language specific editors adopt a hybrid approach, and allow both textual editing and structure based editing. By allowing textual editing, the possibility of errors is introduced and now the text must be parsed to discover the structural representation. There are many different approaches to parsing and error handling in language specific editors, none of which is clearly superior, and most of which tend to complicate the programmer's interaction with the editor. Many of these complications are introduced when programs, which are highly structured documents, are edited as simple text.

An interesting challenge is to find ways to interact conveniently with the structured representation, but maintain the flexibility that is gained through textual editing. Furthermore, there is no need to represent the program as text alone; perhaps augmenting or replacing the textual representation with graphical representations (which are often used to represent programs anyway, such as when flowcharts are employed) will enable some convenient and powerful editing mechanisms to be defined. One such pos-

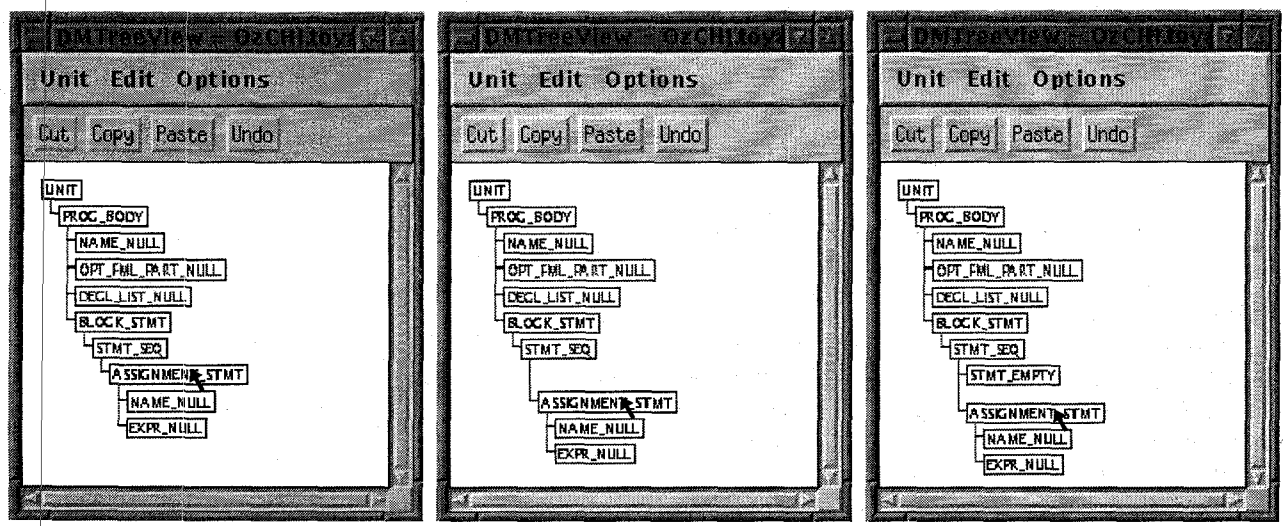

Figure 3. Inserting a new list item.

sibility is to extend program editors to include direct manipulation editing mechanisms. For example, SbyS[9] is an editor where some direct manipulation techniques are explored by the use of a *lego* metaphor, whereby the program is constructed using drag and drop techniques with language building blocks selected from a palette.

Some other possible direct manipulation mechanisms could include:

- a mechanism for moving code fragments around within a program in a way that provides continual feedback as to the current location of the fragment and which only allows the fragment to be inserted into valid locations, and

- allowing the conversion of a **while** statement to a **repeat-until** statement by grabbing the **while** statement and twisting it until it becomes the desired **repeat-until** statement.

A more concrete example of a possible direct manipulation mechanism is given in Figure 3. The example shows a program represented graphically as a tree with the cursor positioned over an assignment statement which is part of a list of statements. If the programmer wanted to insert another statement before the assignment statement, they would select the assignment statement and drag downwards as shown in Figure 3(b). When enough space is made before the assignment statement to accommodate the new statement, a placeholder would appear in the list, as illustrated in 3(c). An important aspect of this interaction is the continual feedback provided by the editor, so that the programmer is always aware of the result of the operation being performed.

3. Unparsing

Unparsing is the process of obtaining a displayed representation from an underlying structural representation. In many language specific editors, this is a matter of translating from an abstract syntax tree to the textual presentation with which we are all familiar. The way in which the underlying representation is unparsed is of crucial importance, since it is this unparsed, visual representation that the programmer will manipulate.

An example of a typical textual unparsing schema is given in Figure 4. This schema is taken from a specification of a version of TextView[13], which is a textual language specific editor within the MultiView programming environment[7]. The language in which the schema is written uses similar unparsing schemes to those employed by the Synthesizer Generator[14] and Gandalf[12]. The schema uses '/' to indicate a new line, and '<' and '>' to decrease and increase the indentation, respectively. If a string is enclosed in '$' marks, it is considered to be a keyword. Child nodes of the node to which the schema applies are indicated using the notation '#child_number'. Thus, lines 1 and 2 of Figure 4 indicate that the displayed representation of an **if** statement consists of the keyword "`if`" followed by whatever is the displayed representation of the first child of the if statement's node (presumably the condition within the statement), followed by the keyword "`then`"; a new line is then started and the level of indentation increased, before the second child (the part to be executed if the condition is true) is unparsed, and so on.

One advantage to separately specifying unparsing, as opposed to hardwiring the unparsing into an editor, is that

```
1 if_stmt is "$if $ #1 $then$ /> #2 </
2             $else$ /> #3 </ $end if$";
3
4 equals is   "#1 ' = ' #2";
5 assign is   "#1 ' := ' #2";
6 plus is     "#1 ' + ' #2";
```

Figure 4. Textual unparsing used in TextView.

it makes it easier to customise the layout of programs, as the formatting information (held in the unparsing schema) is separate from the program; hence, it is simply a matter of applying a different unparsing schema to change the style/layout of the program. This would be very useful, for example, if there is a coding standard in place within an organisation, as the coding standard can be captured in the unparsing schema.

Another advantage of specifying the unparsing is that it allows for the generation of editors from descriptions. This is desirable, as it reduces the effort of reimplementing an entire editor for a different language, because the generator captures the knowledge about building editors. It also allows for user interface consistency between editors (e.g., if an organisation uses more than language), as they could all be derived from similar unparsing specifications, and would have similar appearance, interaction modes, and so on (since they are all generated by the same tool).

One disadvantage with the aforementioned systems and their unparsing schemes is that they only allow programs to be represented textually. The remainder of this paper describes an approach which allows the specification of unparsing in such a way that the program can be represented graphically, where a textual representation is considered to be a specific kind of graphical representation. Furthermore, the technique described here allows the specification of direct manipulation mechanisms to be applied to this graphical representation and is amenable to the generation of an editor from the specification.

4. Graphical unparsing

A technique for displaying a program with graphical aspects will be presented in this section. This technique involves building a hierarchy of *graphical objects* (GOBs). There are two kinds of graphical objects: containers and leaves. The leaves are graphical objects, such as lines and text, that have no children in the hierarchy. The container objects are graphical objects, such as a box, that have other graphical objects as their children. There is a special kind of container, a graphical object called a *subtreebox*; this object contains a reference to a node in the abstract syntax tree.

The unparsing is specified by attaching an unparsing function to a type of node in the abstract syntax tree. When an abstract syntax tree representing a program is unparsed, the relevant unparsing function returns a graphical object which is then associated with that node in the abstract syntax tree.

```
1  DISPLAY_LIST (LABEL_COL: COLOUR) is
2    subtreebox (COLOUR=>none)
3        [vbox_resize, vbox_allocation, press_subtree,
4         elide_subtree]
5      {
6      DISPLAY_SUBTREELABEL
7      box (COLOUR=>none)
8          [treebox_resize, treebox_allocation]
9        {
10       line (COLOUR=>green, STRETCHY=>true) []
11       box (COLOUR=>none)
12           [vbox_resize, vbox_allocation]
13         {
14         for CHILD in CHILDREN_OF (@THIS_NODE) loop
15           DISPLAY_CHILD (CHILD, [pulldown_listitem])
16         end loop
17         }
18       }
19     }
```

Figure 5. Textual unparsing for list nodes.

An example of an unparsing function is given in Figure 5, which specifies how to unparse a program so that it is depicted as a tree in a similar manner to the one shown in Figure 3. The function consists of a name, parameters and graphical object returned. In this example, it is a *subtreebox* that contains other GOBs as its children. The specification of a GOB contains three pieces: the name of the type of object required (e.g., line, box, etc.), a list of attributes for that object (including geometry and colour) and finally the name of any associated state machines for describing the interactions and the geometry behaviour. These state machines are described in more detail in the next section.

Unparsing functions can contain calls to other defined unparsing functions in place of a GOB specification; an example of this is shown on line 6 in Figure 5. The unparsing function can also contain an iterator, as shown on lines 14-16 of Figure 5, in this case iterating over all the children of the node in the abstract syntax tree. There is also a conditional expression to allow graphical objects to be optionally contained in the graphical object hierarchy.

Finally, some mechanism is needed to specify where the children of a node will be displayed, this is illustrated on line 15 of Figure 5 via the call to DISPLAY_CHILD, which specifies where the graphical object obtained from unparsing the given node will appear in the graphical object hierarchy. It also allows interaction state machines to be attached to these graphical objects; in this example, the interaction state machine called `pulldown_listitem` (which describes the behaviour when a list item is pulled downwards, as in Figure 3) is attached to all the children of the unparsed node.

5. State machines

Interactions are described with state machines, which are attached to graphical objects during unparsing. The idea of using state machines to describe the interactions is not new; it has been used to various extents in other systems (e.g., Garnet[11], Planit[5] and Artkit[4]). We have used state machines to describe three aspects of the system: the interaction with graphical objects, the geometric behaviour of the graphical objects and the recognition of sequences of events as new higher level events. These aspects are now discussed in turn.

5.1. Interaction state machines

The interaction state machines (ISMs) are used to define how graphical objects (and hence the program) will respond to user interface events, such as *key presses* and *button clicks*.

An example ISM is shown in Figure 6. When the ISM is attached to a GOB during unparsing, it is in the start state (represented by the double circled state). The state machine moves to another state in response to an event (represented in *italics* above an arc) and a transition function is called; in this case, the transition functions are called command procedures (COPs) and represented in UPPERCASE below the arc. The composition of these COPs is described in the next section.

The example in Figure 6 describes interaction with the list extension mechanism given in Figure 3. Initially, the ISM is waiting for a *Pull_Down* event with all other events being ignored. When the *Pull_Down* event is received, the ISM moves to the PULLING DOWN state, the command procedure called START DRAGGING LISTITEM is called and the ISM now waits for either a *Motion* event or a *Pull_Stop* event, with all other events being ignored.

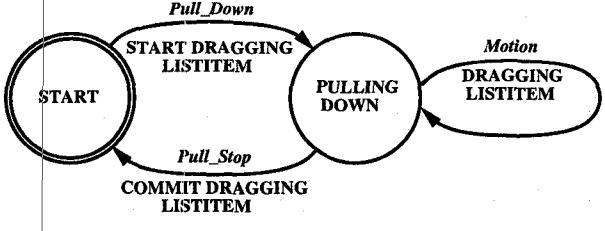

Figure 6. Pulldown Listitem interaction state machine.

5.2. Geometry state machines

A state machine model is also used to describe the geometric behaviour of the objects, more specifically, how a graphical object responds to resize requests and allocations.

Figure 7 shows the two geometry state machines (GSMs) needed to describe the behaviour of a graphical object called *HBox*, which is a box that horizontally tiles its children. The state machines themselves are very simple; in fact, most of the detail is hidden in the command procedures.

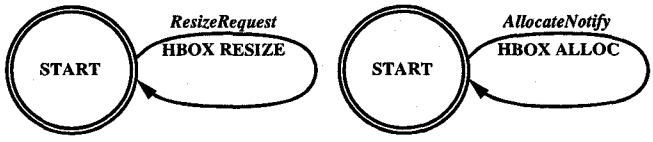

Figure 7. HBox geometry state machine.

One interesting aspect of the relationship between ISMs and GSMs is that state machines are being used to describe both interaction and geometric behaviour. It would be possible to combine the state machines for interaction and geometry into a single state machine to describe more complex interactions that may depend intimately on the geometric behaviour.

5.3. Monitor state machines

Monitor state machines (MSMs) are used to provide a richer set of event types that can be used in other state machines. To do this, they monitor all events in the system. Unlike the previous two kinds of state machines described, which are associated with graphical objects, monitor state machines receive all the events generated in the system. There are two classes of events in the system: primitive and generated. In the current prototype, primitive events come directly from underlying window system, which in our case is the X window system.

These monitor state machines are useful for generating events from sequences of other (generally more primitive) events. An example of this is given in Figure 8, which is a MSM that is used to generate some of the events used in specifying the example given in Figure 3. The MSM combines the mouse button press and motion to generate the appropriate Pull event in the direction of the mouse motion (namely, the events *PullLeft*, *PullRight*, *PullUp* and *PullDown*), and generates a *PullStop* event when the mouse button is released.

5.4. Event propagation

When an event is generated, the current model for event propagation checks if any of the MSMs are interested in the event and then the graphical object hierarchy is traversed to find the GOB deepest in the hierarchy whose geometry contains the co-ordinates of the event. The event is then propagated to this child first and is passed back up the graphical object hierarchy until a GOB's ISM uses the event to move

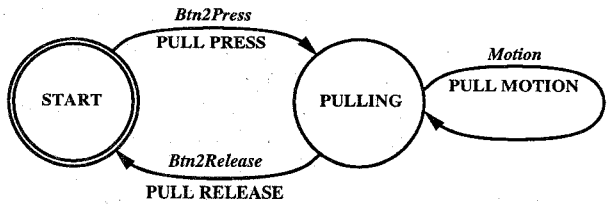

Figure 8. Pull Generator monitor state machine.

```
1 cop START_DRAGGING_LISTITEM is
2 begin
3   INITIALISE (SM_VARS);
4   SM_VARS.PRESS_Y := EV.Y; --store Y locn of btn press
5   SM_VARS.PLACEHOLDER_GOB
6       := rectangle (WIDTH=>5.0, COLOUR =>NONE) [];
7   ADD_CHILD_BEFORE (PARENT_OF(GOB),
8                     SM_VARS.PLACEHOLDER_GOB, GOB);
9   SM_VARS.EMPTY_LIST_SUBTREE
10      := CREATE_EMPTY_LISTITEM (GOB.SUBTREE);
11  SM_VARS.EMPTY_LIST_GOB
12        := UNPARSE (SM_VARS.EMPTY_LIST_SUBTREE);
13  SM_VARS.SPACING_GOB
14        := line (HEIGHT=>4.0, COLOUR=>NONE) [];
15  GRAB_FOCUS (GOB);
16 end;
```

Figure 9. An example command procedure.

to a new state. Once a GOB's ISM has used the event, it no longer propagates to other GOBs, but all ISMs attached to that GOB are able to use the event to move to a new state and trigger their COPs.

An exception to this event propagation model is that graphical objects are able to grab the focus. When the focus is grabbed by a graphical object, all events are sent to the state machines attached to that graphical object until it releases the focus.

6. Command procedures

Most of the detail for describing interactions is contained in the command procedures. An example of a command procedure is given in Figure 9, which is the command procedure triggered when the *PullLeft* event is received by the ISM given in Figure 6.

Each state machine can also have some variables associated with it. These variables are used in the command procedures to store values across state changes. These variables are referred to as SM_VARS in the command procedures. An example is given on line 3 of Figure 9, where these variables are given an initial value; this value would be defined when specifying the appropriate state machine. An Ada-like syntax is used to reference the variables, as shown on line 4, where the state variable called PRESS_Y is given the y-coordinate value of the mouse button press. The same Ada-like syntax is used to reference attributes of the graphical objects and the events. On line 5 and 6, a new graphical object is created using the same syntax as that used in unparsing (see Figure 5). This graphical object is then inserted in to the graphical object hierarchy on lines 7 and 8. Changes to the abstract syntax tree can also be specified in the command procedures. An example of this is given on lines 9 and 10, where a new piece of abstract syntax tree is created. There is an example of grabbing the focus, mentioned in Section 5.4, on line 15.

7. Future work

This paper has described an approach to the specification of graphical unparsing in the context of language specific editors which support direct manipulation editing mechanisms. Furthermore, the technique has been designed with generation of these editors in mind.

At this stage, the various components described in this paper have been prototyped in a sample editor (the one illustrated in Figure 3), although these components cannot yet be generated from a specification. Future work will address the construction of such generators.

In terms of the specification technique itself, the specification of the command procedures is currently in a preliminary (and relatively low level) form. Future work will refine the most appropriate set of primitives to allow common direct manipulation mechanisms to be described easily. One of many interesting issues in the development of the notation for the command procedures is how to conveniently specify the graphical feedback that is to be provided while the direct manipulation mechanisms are being performed.

An important factor in generating language specific editors is the ease in which they can be specified, and an extension to the unparsing schemes that we would like to explore is to be able to specify them graphically. If this were done, the editor designer would be given better feedback as the editor is being designed; suitable direct manipulation mechanisms in the editor for the unparsing schemes may also be appropriate.

The direct manipulation mechanisms presented in this paper are meant to be indicative of the kind of mechanisms that could be provided, rather than definitive. This work will facilitate the rapid prototyping of these direct manipulation style mechanisms, and hence allow exploration of the suitability of these mechanisms for program editing. Ultimately, we trust that such mechanisms will contribute to the availability of better, and more widely accepted, language specific editors.

References

[1] P. Borras, D. Clement, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang and V. Pasqual, "CENTAUR: the system", *Proc. A.C.M. SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, A.C.M. SIGPLAN Notices, Vol. 24, No. 2 (February 1989) (Boston, Massachusetts, 1988), pp.14-24.

[2] N. M. Delisle, D. E. Mincosy and M. D. Schwartz, "Viewing a programming environment as a single tool", *Proc. A.C.M. SIGSOFT/SIGPLAN Software Eng. Symp. on Practical Software Development Environments*, A.C.M. SIGPLAN Notices, Vol. 19, No. 5 (May 1984), pp. 49-56.

[3] V. Donzeau-Gouge, G. Kahn, G. Lang, "Programming Environments Based on Structure Editors: The MENTOR Experience", *INRIA Research Report* No. 26, 1980.

[4] T. R. Henry, S. E. Hudson and G. L. Newell, "Integrating Gesture and Snapping into a User Interface Toolkit", *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, Snowbird, Utah, October 1990, pp.112-121.

[5] S. E. Hudson and R. King, "Implementing a User Interface as a System of Attributes", *Proceedings of the ACM/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, Palo Alto, California, December 1986, pp.143-149.

[6] C. D. Marlin, "Language Specific Editors for Block Structured Languages", *The Australian Computer Journal*, Vol. 18, No. 2 (May 1986), pp.46-54.

[7] C. D. Marlin, "A Distributed Implementation of a Multiple View Integrated Software Development Environment", *Proc. 5th. Conference on Knowledge-Based Software Assistant*, Syracruse, New York, September 1990, pp.388-402.

[8] R. Medina-Mora, R. J. Ellison, D. B. Garlin, G. E. Kaiser and D. S. Notkin, "ALOE Users' and Implementors' Guide (Interim Edition), Department of Computer Science, Carnegie-Mellon University, Pittsburg, Pennsylvania, 1983.

[9] S. Minör, "On Structure-Oriented Editing", Ph.D. Thesis, Department of Computer Science, Lund University, Lund, Sweden, 1990.

[10] S. Minör, "Interacting with structure-oriented editors", *Int. J. Man-Machine Studies*, 37, 1992, pp.399-418.

[11] B. A. Myers, D. A. Guise, R. B. Dannenberg, B. Vander Zanden, D. S. Kosbie, E. Pervin, A. Mickish and P. Marchal, "Garnet: Comprehensive Support for Graphical, Highly-Interactive User Interfaces", *IEEE Computer*, 25(8), August 1992, pp.61-73.

[12] D. Notkin, "The GANDALF Project", *The Journal of Systems and Software*, Vol. 5, No. 2 (May 1985), pp.91-106.

[13] M. C. Read, "TextView: a textual view for the MultiView environment", Honours Thesis, Department of Computer Science, The Flinders University of South Australia, Adelaide, South Australia, November 1993.

[14] T. W. Reps and T. Teitelbaum, "The Synthesizer Generator", *Conference of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pp. 42-48, 1984.

[15] B. Schneiderman, "Direct Manipulation: A step beyond programming languages", *IEEE Computer*, 16(8), August 1983, pp.57-69.

[16] T. Teitelbaum, T. Reps, "The Cornell Program Synthesizer : A Syntax-Directed Programming Environment", *Communications of the ACM*, Vol. 24, No. 9 (September 1981), pp.563-573.