Building an editor from existing components: an exercise in software re-use

J.W.C. Koorn[1]

Programming Research Group, University of Amsterdam
P.O. box 41882, 1009 DB Amsterdam, The Netherlands
email: koorn@fwi.uva.nl

H.C.N. Bakker[1]

Department of Software Technology , Centre for Mathematics and Computer Science
P.O. box 4079, 1009 AB Amsterdam, The Netherlands
email: huub@cwi.nl

Abstract

We show that a hybrid text/structure editor can be built by combining an existing text editor with an existing structure editor. We describe how we have built such an editor using a client-server architecture for the communication between each of the three components involved (text editor, structure editor and user-interface). The main advantage of this technique is the re-use and integration of existing software components.

Key Words & Phrases: Code re-use, text editor, structure editor, hybrid editor, generic syntax-directed editor, distributed editor, existing components, uniform syntax-directed user-interface, interactive programming environment.

1991 CR Categories: D.2.2 [Software engineering]: Tools and Techniques - Structured programming, User interfaces D.2.3 [Software engineering]: Coding - Program editors D.2.6 [Software engineering]: Programming Environments - Interactive D.2.7 [Software engineering]: Distribution and Maintenance - Restructuring D.2.10 [Software engineering]: Design - Methodologies

1 Introduction

When developing a computer program one makes frequently use of *programming environments*, i.e., sets of cooperating tools that help the programmer to carry out his or her task. These generally consist of an editor, a parser, a type-checker and a code generator. We will concentrate here on the role of the editor, the tool used to construct a program, within the environment.

While the datastructure used by an editor is text, the other tools in the environment usually operate on an abstract syntax tree [ASU86], which is a datastructure derived from the text. An abstract syntax tree can only be derived if the text is syntactically correct, so we need tools external to the editor for parsing, type checking and tree construction. In this case the editor itself has no knowledge of the language in which the text is written. Editors of this type, such as Emacs [Sta81], are known as *text* editors.

Structure editors on the other hand, do have knowledge of the grammar of the language in question. The construction of a program with a structure editor consists of manipulating the tree datastructure, using *placeholders* and *templates*. Since it is very difficult for humans to read

[1]Partial support received under Esprit contract nr. 2177.

trees, these editors also use an external tool, known as a pretty-printer, to derive text from the tree. Note that the abstract syntax tree, which is necessary for other tools in the programming environment, can be used immediately.

As an example of constructing a program, consider a grammar with the production rules:

```
EXP ::= EXP + EXP
EXP ::= EXP * EXP
```

Here, `EXP` is a placeholder and `EXP + EXP` and `EXP * EXP` are the corresponding templates. A user constructs a sentence like `1 + 2 * 3` from an `EXP` placeholder as follows: first select `EXP`, then replace it by the template `EXP + EXP`, next, select the rightmost `EXP` and replace it by the template `EXP * EXP`, etc.

The main advantage of using a structure editor rather than a text editor is that a structure editor can only construct trees that correspond to the language's grammar. This means that it is *impossible* to make a syntactical error during the construction of a program. The main disadvantage is that entering simple program constructions, such as variable names and numbers, requires more actions from the user than simply typing them in. The "perfect" editor combines the advantage of structure editors (it is impossible to make syntax errors) with the advantage of text editors (it is easy to enter simple program constructions).

Hybrid editors combine text editing with structure editing, enabling the user to switch back and forth between typing text and manipulating trees. Ideally, this "mode switch" is implicit.

Building a hybrid editor with an implicit mode switch is no easy job, as it is necessary to create *two* cooperating editors. Since both text editors and structure editors are already available, it is attractive to try to re-use existing editors, so that only the interconnection has to be coded. The subject of this paper is therefore: how can we build a hybrid editor by combining an existing text editor and an existing structure editor?

1.1 Paper overview

In Section 2 we motivate our work, present a model for a hybrid editor and discuss work related to ours. In Section 3 we describe the general implementation model used, present some basic features of the text editor we used, describe the so-called "focus" concept, and list the actual implementation in pseudo code. The user-interface component is the subject of Section 4. In Section 5 we show how multiple instances of our editor can be used as the user-interface of a programming environment. Section 6 discusses the use of the editor in the ASF+SDF meta-environment, which is a programming environment generator. We close with Section 8, where we discuss our approach and make some final remarks.

2 Connecting two editors

First, we will now motivate our work and present an editing model for hybrid editing, and briefly discuss related work. The editing model is the basis for implementing a hybrid editor in which text editing and structure editing are smoothly integrated, i.e., there is an implicit mode switch. We will use GSE [Koo92] as structure editor.

Keeping track of textual changes and of cursor movements is an essential part of the model. Therefore, we are only able to use a text editor which offers these tracking capabilities. Epoch [Epo92], an extended version of Emacs, fulfills this requirement and we will use it as text editor.

2.1 Motivation

Our experience with the use of GSE forms the motivation of the work presented here. GSE is used as a generic building block in the ASF+SDF meta-environment [Kli91, Hen91], an interactive system that generates an interactive programming environment from a formal description. The older versions of GSE were all implemented as new, stand-alone, systems. Therefore, they contained functions to manipulate text and to display text in a window. This approach has three disadvantages: it is a lot of work to maintain the code, text editing functionality is far less compared to what is provided in existing editors like, for instance, Emacs and new users have to learn new editor commands. These observations led to the idea to eliminate all text editing and displaying functionality from GSE and replace it by an existing text editor.

2.2 A model for hybrid editing

The most important concept in GSE's editing model is the so called *focus* which designates a subtree. A focus is a pair [*text-part*, *tree*], where *text-part* is that part of the text that corresponds to the focus' *tree*. The *text-part* is implemented by maintaining two *positions* in the text, indicating the start and end of the text in the focus. The *tree* is a subtree of the abstract syntax tree of the whole program and therefore corresponds to some language construct. Normal text editing (cutting, pasting, inserting characters, etc.) is *only* allowed inside the focus. A *cursor* indicates the current character position inside the text. All text outside the focus is guaranteed to be syntactically correct. Moving the focus to another part of the text is done by invoking a *navigation* command, like "go to the next child", "go to the previous child" or "go to the parent". Note that if a navigation is invoked, the focus' *text-part* has to be computed. For this reason, GSE annotates nodes with positional information.

There is one navigation command which does not correspond to a tree traversal primitive, but to a breadth first search in the tree: the mousepoint command. It is invoked when the user points with a mouse device at some character and clicks. GSE interprets this as a request to move the focus to the smallest subtree S such that the text corresponding to S contains the character pointed at. Since the focus' *tree* is a subtree of the abstract syntax tree, the net effect of clicking on a part of a language construct is to move the focus to that construct and to move the cursor to the character pointed at.

There are a number of consequences of this editing model which have to be taken into account when we try to combine GSE with Epoch. First, since text editing is only allowed inside the focus, the cursor should always be positioned inside the focus. Note that when the cursor is moved by the user, this implies either parsing the focus' *text-part* or enlarging the focus such that it contains the cursor position. Second, again because text editing is only allowed inside the focus, deleting a piece of text that is outside, or partly outside, the focus also requires focus enlargement. However, deleting a piece of text that is inside the focus does not require a focus enlargement. Both situations for deleting text are depicted in Figure 1. Finally, parsing the

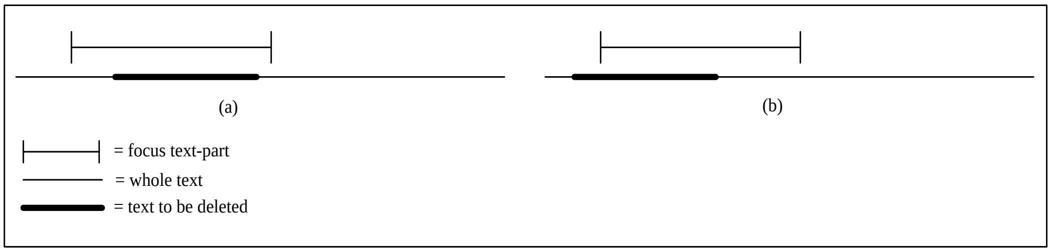

Figure 1: Deleting text may require focus enlargement

focus' *text-part* is, of course, only necessary when it has been changed since the last successful parse.

2.3 Text mode and structure mode

In the previous section, we saw that parsing the focus' *text-part* is only necessary when a textual change occurred. This property is used to define the *mode* of the editor. The editor is in *text* mode when the focus' *text-part* has been changed since the last successful parse, otherwise it is in *structure* mode.

One of our requirements is to build an editor in which the "switch" between these two modes is implicit. This can be achieved as follows:

- When the editor is in structure mode, any textual change causes a switch to text mode.
- When the editor is in text mode, it stays in text mode after any textual change, or an unsuccessful parse.
- When the editor is in text mode, a successful parse causes a switch to structure mode.

To summarize which action should be taken when the editor is in a certain mode, we present all actions for each mode in Table 1.

Command	Editor is in text mode	Editor is in structure mode
cursor motion	If the new cursor position is outside the focus, compute a larger focus containing that position.	The new cursor position should be inside the focus and the focus should be as small as possible $\Rightarrow$ compute a new focus accordingly.
text insertion	The focus size increases $\Rightarrow$ increase the focus end.	Text mode required $\Rightarrow$ switch to text mode and execute text insertion.
text deletion	If the text to be deleted is not inside the focus, compute a larger focus containing it. The focus size decreases $\Rightarrow$ decrease the focus end.	Text mode required $\Rightarrow$ switch to text mode and execute text deletion.
parsing	Try to switch to structure mode by parsing the focus text. If parsing succeeds, switch to structure mode.	The focus text has already been parsed $\Rightarrow$ no action required.
navigation	Structure mode required $\Rightarrow$ try to switch to structure mode by parsing the focus text. If parsing succeeds, switch to structure mode and execute navigation.	Navigation implies a focus move $\Rightarrow$ compute the new focus. If the cursor is not inside the new focus, move the cursor such that it is.

Table 1: Overview of all actions for each mode.

2.4 Related work

The idea to extend a structure editor with text editing functionality, where an existing text editor is used, is not new. For instance, Lang reports in [Lan86] that at some stage, the Mentor system[DGHKL84] had a link with Emacs. This allowed users to select a subtree in the Mentor editor, edit its corresponding text in Emacs, parse the changed text and replace the subtree by the parse result. However, the approach used did not incorporate automated focus enlargement.

The successor of the Mentor editor, called ctedit[BCD+88] currently uses Epoch instead of Emacs, but automated focus enlargement lacks here too.

Other hybrid editing systems sometimes have rich text editing functionality and customizable graphics, but use their own text editing and display facilities. For instance, Pan [BGV92] is extensible and customizable in the Emacs style and features completely unrestricted text editing. Another hybrid editing system with unrestricted text editing is PSG [BS92].

3 Implementation

Connecting an existing text editor with an existing structure editor, requires that adaptations are made to the source code of both editors. For GSE, this is not a problem, since we created its source code ourselves, but Epoch was created by others. However, Epoch is extensible because it contains a Lisp interpreter. This allows us to extend Epoch with code necessary for connecting it with GSE. Here, we first describe the general implementation model we use, after which we discuss two essential Epoch features called *zones* and *hooks*. Next, we take a closer look at focus enlargement and focus moves. At the end of this section, we present our actual implementation.

3.1 General implementation model

The basis for our implementation is a *client-server* architecture, as found in, e.g., the X-window system [SG86]. In this architecture, components can run on different machines while the system as a whole appears to the user to be running on a single machine. The actual machines involved communicate with each other through the network. The client-server architecture enforces a strict separation between components: they can only exchange information by using the interface layer to the network software. Therefore, the implementation as a whole tends to be better maintainable.

Each component involved is a *server* to the others, which are the *clients*. If a client needs data stored in a server, it builds a *command* that contains, amongst others, the destination. The command is handed over to the network software which dispatches it to the appropriate server. Next, the server interprets the command which leads to calling a function in the associated component. Finally, the result of the function call is sent back to the client which interprets the answer. The general architecture is depicted in Figure 2.

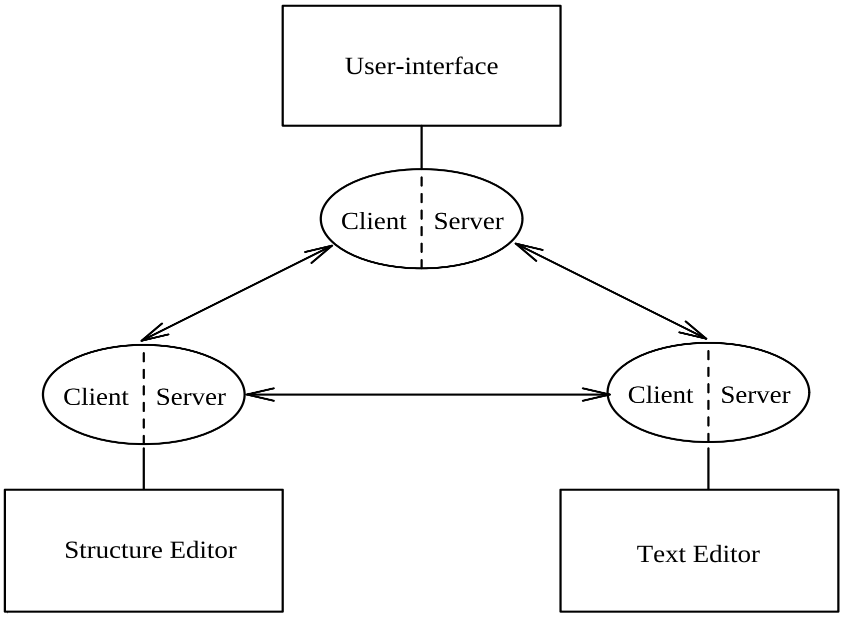

Figure 2: Client-server implementation model

3.2 Implementing the focus using Epoch

Two features of Epoch are essential for implementing the focus: *zones* and *hooks*.

Zero or more *zones* may be associated with an Epoch text. A zone in Epoch is a datastructure associated with a part of the text. It contains a *style*, a *start position*, an *end position*, and a *data* field[2]. A style describes how to display a zone, i.e., it contains font and color information. We will return to styles in Section 4. If the user inserts or deletes text, Epoch updates all data in its zones. For instance, if the cursor is positioned within a zone and a user inserts text in it, the zone's end position is updated. Since we need to keep track of all text in the focus by maintaining the focus start and end position, an Epoch zone is the natural choice for the datastructure to implement the focus. We use the data field of a zone to mark it as "added by GSE".

A *hook* in Epoch is a variable which may hold the name of a function. If a function name is stored in a hook, Epoch calls the function with that name with predefined arguments. Hooks are mainly used to keep track of what is happening inside Epoch. For example, there is a hook called `after-movement-function` to which a function with one argument may be attached. Whenever the cursor is moved, the attached function is called with the old cursor position as an argument. We use this hook to check whether or not the cursor is still positioned inside the focus. For this check we need, of course, the new cursor position as well, which is not an argument of the function called, but we can obtain it directly from Epoch by calling an internal function.

Another hook is used to keep track of textual changes, since deleting text may lead to a new, larger, focus. Also, if the editor is in structure mode, we must record a mode switch when the text changes. We use the `before-change-function` for this purpose. The function associated with this hook is called before each textual change and is supplied with two arguments: the position of the change and the position of the end of the region affected. If the two positions are equal, the change is an insertion.

3.3 A closer look at focus enlargement and focus movement

Recall that the focus corresponds to a pair [*text-part*, *tree*]. Now consider the problem of computing a focus enlargement. First of all, any focus enlargement implies changing the focus' *tree* to its parent tree, or to any of its grandparents. Secondly, the new focus' end position depends on the old focus' end position, since there might be text inserted or deleted. Therefore, the new focus' end position can not be computed in GSE without asking Epoch for the old focus' end position. Furthermore, focus enlargement is sometimes necessary for deletions (cf. Figure 1). This implies that for some deletions Epoch needs to call a function in GSE, while for other deletions there is no such need. As a consequence, the response time of a deletion operation is not constant, which may be annoying for a user.

A different scheme is therefore adopted: for each focus surrounding the current one we add a separate zone to the Epoch text. This scheme is depicted in Figure 3. New focus end positions are now immediately available, since Epoch updates its zones during text editing. If a focus enlargement is needed, the zone corresponding to the focus is deleted. The new focus zone is now the smallest of the remaining zones. In this way, we do not need any communication with GSE during text editing, yielding constant response times for deletions. However, GSE looses track of the focus' *tree* when the focus is enlarged. Therefore, when a parse command is invoked, which is a function in GSE, GSE first asks Epoch how many zones are associated with its text. By comparing this number with the number of focus parent nodes in the tree, GSE is able to

[2] An Epoch zone contains even more fields, but those are not relevant here.

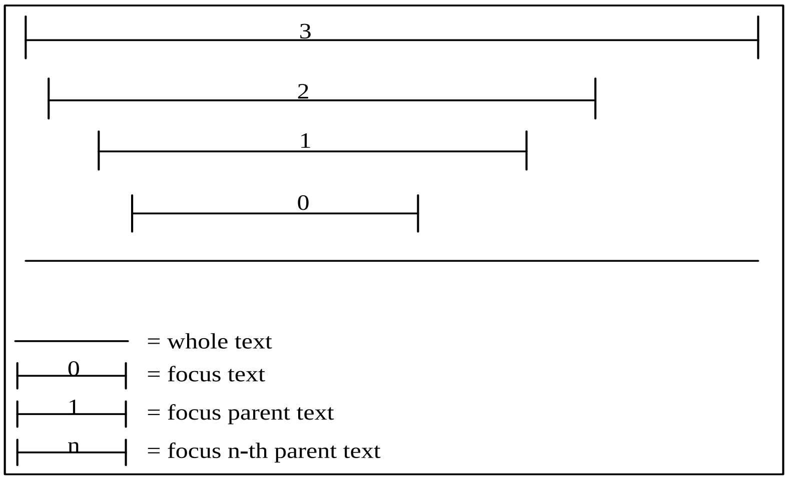

Figure 3: Focus zone, focus parent zone, etc. for a focus tree with three parents.

compute how many focus zones were deleted since the last parse. Consequently, GSE moves the focus' *tree* to the computed parent node before the focus' *text-part* is parsed.

When the editor is in structure mode and the user moves the cursor, we might have to change the focus as well, since the focus' tree must be the smallest subtree S such that the text corresponding to S contains the character at the cursor position. In the new scheme this requires deleting the zones not containing the cursor position, and possibly adding new ones. In this situation, we call the `GSE-add-focus-zones` function, which is supplied with the number of remaining zones and the new cursor position. The smallest subtree S is computed in GSE and the focus' tree is set to S. Furthermore, GSE counts the number of nodes from S to the root and compares it with the number of remaining zones. If these numbers are unequal, GSE instructs Epoch to add a zone for S, the parent of S, etc., until the number of zones in Epoch equals the number of nodes from S to the root. Moving the cursor in structure mode is depicted in Figure 4. Figure 4(a) shows the situation just before the move and Figure 4(b) shows the situation just after the move.

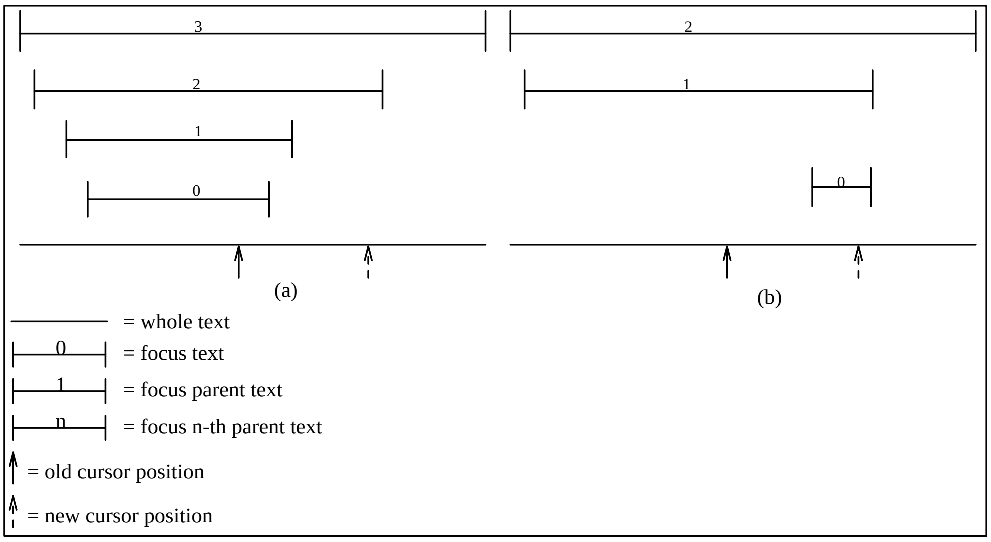

Figure 4: Focus zones (a) before and (b) after moving the cursor in structure mode.

3.4 Actual implementation

We present our implementation as a list of functions in pseudo code. Functions in GSE have names starting with "`GSE-`", likewise, functions in Epoch start with "`Epoch-`".

Epoch functions handling cursor moves and textual changes are:

```
Epoch-after-movement-function
   delete-zones-not-containing-new-cursor-position
   if in-structure-mode then
      call GSE-add-focus-zones (number-of-zones new-cursor-position)
   fi

Epoch-before-change-function
   if in-structure-mode then
      switch-to-text-mode
   fi
   if change-is-a-deletion then
      delete-zones-not-containing-entire-deletion
   fi
```

GSE functions handling parse and navigation commands[3] are now as follows:

```
GSE-parse
   if call Epoch-in-text-mode then
      N := call Epoch-get-number-of-zones
      Text := call Epoch-get-focus-text
      parse (N, Text)
      if parse-succeeded then
         call Epoch-switch-to-structure-mode
      fi
   fi

GSE-go-to-parent
   call Epoch-delete-smallest-zone

GSE-go-to-next-child
   if call Epoch-in-structure-mode then
      if next-child-exists then
         new-focus-position := compute-position-of-next-child
         call Epoch-move-focus-zone (new-focus-position)
      fi
   fi
```

4 The User-interface

There are two points in the editor under construction where the user-interface (UI) plays an essential role. First, the focus should be visible in the Epoch text. Second, the window in which the text is displayed must be "decorated" with a menu bar from which GSE commands, such as structure editing, parsing, and navigation commands, may be invoked. We discuss these aspects below.

[3] We omit `go-to-first-child` and `go-to-previous-child` since these are similar to `go-to-next-child`.

4.1 Displaying the focus

There are two reasons why the focus should be visible in the text. First, if the user invokes a parse command, parsing may either succeed or fail. Clearly some form of feedback is required here. Second, if the editor is in structure mode, all structural editing commands are relative to the focus' *tree*. In this case, a user wants to know what the corresponding text part is before he or she decides which structural editing command is to be invoked.

We have chosen to implement this using Epoch's zone styles (cf. Section 3.2). We use three different styles for the focus zone: the *ok-style* when the editor is in structure mode; the *error-style* when parsing failed; and the *text-style* when the editor is in text mode. Parent zones are displayed in the text-style. All three styles use the same font, but different colors for the text and the text background.

4.2 Decorating the text window

Structure editing commands, such as replacing a placeholder by a template, are menu driven. Consequently, we have to add a menubar to the window in which the text is displayed. Note that this window is created by Epoch. We are therefore confronted with a situation where we want Epoch's window to be a *subwindow* of a window that contains the menubar. However, this can be handled by supplying a so-called *parent-window* to Epoch's "create-window" function. For the implementation of windows that contain a menubar, we have chosen OSF/Motif™ [Fou90], a system for building graphical UIs based on the X window system.

4.3 Handling callbacks

It is standard practice in current windowing systems that each UI item — such as a button — has an associated *callback*. This is a function to be called when the UI item is activated by the user. In our case, we are unable to invoke such a function directly, because it might be a function in the text editor or in the structure editor component. These functions might require arguments only present in that component. Therefore, we need a more general mechanism to handle callbacks.

Recall from Section 3.1 that the UI is a separate component of our system. The creation of any window is thus the result of the interpretation of a command. All commands involving windows are invoked by GSE. The `create-window` command is supplied with a filename, the UI component creates an Epoch window displaying the file, and its surrounding window. This surrounding window has an *empty* menubar. The `create-window` command returns a *window identifier* to GSE. GSE stores this identifier in its datastructure as a *shadow window*. Next, GSE invokes commands to add entries to pulldown menus. One command per entry. Each of these commands return an *item identifier*, which is stored in GSE's shadow window. We call this a *shadow item*. The GSE function to call when there is a callback from an item, is now stored in the shadow item.

A callback is now handled as follows. Consider a callback from an item with identifier I located in a window with identifier W. The callback from the UI item is first sent to the UI component, which builds a command indicating that there was a callback from I in W. Then, the UI component sends this command to GSE using the network. GSE searches the shadow item with identifier I in its shadow window with identifier W and calls the associated function, supplying it with the appropriate arguments when necessary.

If we want to call a text editor function from a UI item, this function is invoked by first creating a command at the GSE level and then sending it to the text editor component. At first glance, it seems that this scheme implies that there is no need for communication between the UI and the text editor component. However, these components do need to communicate, because the Epoch window must be created as a subwindow of our editor. Furthermore, a user might resize or destroy the editor window, in which case the Epoch subwindow must be resized or destroyed as well.

5 Using the editor in a programming environment

As with the older versions of GSE, the new editor is used in programming environments generated with the ASF+SDF meta-environment. Generated environments are based on a *collection* of syntax-directed editors, and each editor may use a different language. Until now, we have discussed a situation in which there was only one combined editor, the question therefore is how to implement multiple instances.

5.1 Multiple editors

Epoch is able to handle several texts at the same time by displaying each text in a separate window. These Epoch windows are subwindows of OSF/Motif windows managed by the UI component. By replacing the structure editing component by a *GSE manager* (GM), we are able to implement multiple instances. The collection of syntax-directed editors, needed in a generated programming environment, is thus modeled by GM. We have to make some adaptations however, since in the case of multiple instances, the problem of how to identify each instance arises.

5.2 Identifying instances

The programming environment identifies each editor instance by means of a *filename*. For example, adding an instance to the environment is done as follows. After the user has supplied a filename, GM first creates a new GSE datastructure in which the filename is stored. Next, a new window is created of which the identifier and the identifiers of UI items are also stored in the new GSE datastructure.

If a structure editor needs information stored in its corresponding text editor, the structure editor sends a command to the text editor component of the system. This command is supplied with a filename by which the text editor component can identify the appropriate text editor.

Callbacks are now handled by GM instead of by GSE itself. Each callback comes with a window identifier W and an item identifier I. GM responds to a callback by searching the shadow window with window identifier W in all GSE instances. When this shadow window is found, the item with identifier I is searched, and the associated function is called.

6 Using the editor in a programming environment generator

So far, we have only discussed the windows of our new editor itself. Besides these type of windows, there are other types of windows, such as: dialog boxes, error windows, etc. Furthermore, in the ASF+SDF meta-environment (cf. Section 2.1), the notion of a *module-editor* exists. A module-editor is a composition of *two* editor instances which are displayed in one window. In an ASF+SDF module-editor, a user defines syntax rules in one of the editors, which can immediately be used in the second editor. This two level editing process is described in [Kli91]. Finally, in the ASF+SDF meta-environment there are components that use editor instances, or other window based objects, as part of their user-interface. Now we describe how we extended the implementation of the UI to allow the handling of an arbitrary number of additional components.

6.1 Window types

Windows, such as dialog boxes, error windows, etc., all have a *type*. Each window type has specific commands to build its window and has its own callback. For example, a dialog box with "OK" and "CANCEL" buttons is built using three commands: `create-dialog`, `add-button("OK")` and `add-button("CANCEL")`. Each command returns identifiers as before, such that a shadow dialog box can be built. Executing a callback from an item in a dialog, called a *dialog-callback*, now amounts to looking for a shadow dialog with the appropriate dialog identifier, searching the item in the shadow dialog and calling the associated function.

6.2 Combining editors in frames

Combined editors, such as ASF+SDF module-editors, are displayed in a window that contains subwindows, one for each editor involved. We call such a subwindow a *pane* and we call the surrounding window a *frame*. This window type is implemented in a similar manner as the editor window described earlier. That is, we send several commands from GM to the UI to build a combined editor. First, a frame is built that does not contain any panes. Second, we add a pane for each editor involved, which is an editor window with an empty menubar. Finally, the menubar of each pane is filled with menu entries.

Callbacks from items in a frame now contain three identifiers: the frame identifier (F), the pane identifier (P) and the item identifier (I). In this way, we are able to use the same scheme for callbacks as described in Section 5.2, except that we have to search for a frame with identifier F first. This approach allows the addition of UI items to a frame as well. A typical example of such an item is the `destroy window` button.

6.3 Windows created by other components

Apart from the windows created by GSE, other components of the system may create windows as well. Consider, for example, a dialog box that is created by another component. Any callback from this dialog box will be sent to GM, which searches in all GSE instances for a shadow dialog with the identifier derived from the callback. This dialog will not be found, because the shadow dialog is not a part of any GSE instance. To solve this problem, we add a level of indirection: the *name* of the component that created the window. Any callback is supplied with this name as an argument. The system now becomes responsible for handling all callbacks. It first strips off the name argument, and then calls the callback manager for that component, this time without the component name. Consider, for instance, a callback from a frame that was created by GSE. In the system, the callback arrives as `frame-callback("GSE" F P I)`. The system maps this call to a call in GM: `GSE-frame-callback(F P I)`. Callbacks from frames created by the debugger (DB) however, arrive as `frame-callback("DB" F P I)` and are mapped to `DB-frame-callback(F P I)`. In this way, any component in the system can build its own user-interface and is by itself responsible for handling any callbacks. When a new component is added to the system it can use the library of user-interface functions immediately due to the fixed mapping of callbacks by the system.

7 Assessment

How much effort was involved in changing GSE, implementing the network communication, and changing the user-interface? What was gained by this effort?

7.1 Quantification of code re-use

In this section we present how much code was involved when we made our new hybrid editor. We quantify the amount of code by counting lines of source code. While counting the number of source lines we included empty lines, comments and code used for testing and debugging. We sometimes list a total of the number of source lines used by a component. These totals should be interpreted with care since we add counts of sources that are written in different programming languages. That is, the old GSE is completely written in LeLisp [LeL91], Epoch is written partly in C [KR78] and partly in Emacs-Lisp [LLG90] [Epo92], the new UI is written in C and the network communication is also written in C. For this reason we list the number of lines written in each language where we use the abbreviations `LL` for LeLisp, `EL` for Emacs-Lisp and `C` for C. We included `*.h` files while counting C source lines, but we excluded `*.h` files used for hardware specific compilation.

Two major parts of GSE have been changed: the text editing component and the UI component. The third major part of GSE, used for parsing text and structure editing, is re-used. The sizes of the sources involved are listed in Table 2. We excluded the parser and the pretty-printer code, a total of 21348 lines, from the count of the re-used part. The numbers presented raise

Used for	old(LL)	new(total)	new(EL)	new(C)	new(LL)
Text functions	2178	2369	1478	-	891
Displaying text	3797	-	-	-	-
Building the UI	4971	6251	-	3158	3093
Network layer	-	4561	586	3924	51
Total (changes)	10946	13181	2064	7082	4035
Re-used	6331	6331	-	-	6331
TOTAL	17277	19512	2064	7082	10366

Table 2: Overview of sizes of the source code used by each hybrid editor version.

some questions. For instance, "Why is the new text editing component *larger* than the old one? (2369 new lines compared to 2178 old ones)". Indeed, at first glance this seems strange since we now use an existing text editor which is an *external* component. The answer is that in the new situation we have to interface with the network layer which implies some overhead. The overhead consists of building a command string, interpreting the command, building an answer string and interpreting the answer. To give an overview of how much code is used for the overhead and how much code is "really" used, see Table 3. This table shows that 1493 lines

Part	total	EL	C	LL
Interface to rest of GSE	325	-	-	325
Message building and interpreting	551	185	-	366
Reactions to messages	1493	1293	-	200
TOTAL	2369	1478	-	891

Table 3: Breakdown of the size of the new text editing functionality.

are "really" used. A similar effect can be found in the new UI. Here we wrote 6251 lines of new code and eliminated 4971 lines of code. Of the new 6251 lines, 5086 (6251 - 1165) are "really" used as shown in Table 4. Another question triggered by Table 2 is: "Why did you write 13181 lines of new source code in order to eliminate *less* code (10946 lines)?" We did this because we now have incorporated Epoch and this amounts to a gain in text editing functionality with a size of 208951 lines of code. For an overview of the amount of external source code used by each

Part	total	EL	C	LL
Interface for building objects	2253	-	-	2253
Message building and interpreting	1165	-	731	434
Reactions to messages	2833	-	2427	406
TOTAL	6251	-	3158	3093

Table 4: Breakdown of the size of the new UI.

version we refer to Table 5. The main advantage of the new situation is the percentage of the

Component	old(LL)	written	new(total)	new(EL)	new(C)	new(LL)	written
UI library	7663	by others	6348	-	6348	-	by us
Epoch	-	-	208951	64720	144231	-	by others

Table 5: Overview of sizes of the external source code used by each editor version.

code that has to be maintained in the future. This is reflected in Table 6. Also, we have created

Component	Old			New		
	code by us	code by others	%	code by us	code by others	%
GSE	17277	-	100	19512	-	100
UI library	-	7663	0	6348	-	100
Epoch	-	-	0	-	208951	0
TOTAL	17277	7663	69.3	25860	208951	11.0

Table 6: Overview of sizes of the all source code used by each editor version.

our new editor in such a way that different components can be run on different machines. The amount of source code involved in network communication is 4561 lines (cf. Table 2). This part is re-usable and may therefore be re-used by other parts of the ASF+SDF system in future.

8 Discussion and conclusions

We have shown that a hybrid editor can be built by connecting an existing text editor with an existing structure editor. The general advantages of this approach are:

- The text editing functionality of the combined editor is as rich as the functionality of the text editor.
- Users familiar with the text editor can use the combined editor immediately.
- Future improvements to the text editor are inherited by the combined editor.
- There is no need for maintaining the source code of the text editor by the programmers of the combined editor (and vice versa).

The advantages of connecting Epoch and GSE in particular are:

- The combined editor inherits the extensibility and customisiability from Epoch.
- Each component of the combined editor may run on a different machine.

We are able to connect Epoch and GSE because Epoch offers some special functionality. These features are: extensibility, which we use to implement network communication; hooks, used to keep track of cursor moves and textual changes; zones, to display and implement the focus concept; and windows that can be subwindows, used to decorate a surrounding window with a menubar containing structure editing commands.

We have built our editor by extending Epoch and by removing all text editing and display functionality from GSE. In other words, we stripped the structure editor and extended the text editor. This leads to the question if a complementary approach would also work: can we strip an existing text editor and extend an existing structure editor. In our view, this will be much harder because in this situation we need to use the display functions of the structure editor. However, displaying in a structure editor is based on displaying text of which the underlying structure is known. This implies that these editors lack functions to display unstructured text, which is typical during typing. Therefore, using the display functions of the text editor seems the natural choice.

A related question is whether or not it would have been a better approach to extend Epoch with structure editing functions. However, this would require re-implementing GSE's structure editing functions as well as the parser and the pretty-printer. This amounts to re-implementing 6331 lines of GSE code and 21348 lines for the parser and pretty-printer (cf. Figure 2). If we compare the total of 27679 lines to the 13181 lines we now wrote, the approach chosen seems to be the best. We think the way we have built the combined editor is a school example of re-using existing code.

Acknowledgements

The authors would like to thank Paul Vriend for additional programming and Paul Klint for reading and commenting on earlier versions of this paper. Robert van Liere was our X, Motif and networking guru and Claus Bo Nielsen helped us implementing the Epoch client and server. Finally, we thank Philippe Kaplan and Chris Love who helped us to incorporate the Epoch window in a surrounding window containing a menubar.

References

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers. Principles, Techniques and Tools.* Addison-Wesley, 1986.

[BCD+88] P. Borras, D. Clément, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. Centaur: the system. In *Third Annual Symposium on Software Development Environments (SIGSOFT'88)*, Boston, 1988.

[BGV92] R.A. Ballance, S.L. Graham, and M.L. Van De Vanter. The Pan Language-Based Editing System. *ACM Transactions on Software Engineering and Methodology*, 1(1):95–127, 1992.

[BS92] R. Bahlke and G. Snelting. Design and structure of a semantics-based programming environment. *International Journal of Man-Machine Studies*, 37(4):467–502, October 1992.

[DGHKL84] V. Donzeau-Gouge, G. Huet, G. Kahn, and B. Lang. Programming environments based on structured editors: the MENTOR experience. In D.R. Barstow, H.E. Shrobe, and E. Sandewall, editors, *Interactive Programming Environments*, pages 128–140, New York, 1984. McGraw-Hill.

[Epo92] University of Illinois, Urbana-Champaign. *Epoch, GNU Emacs for the X Windowing System*, 1992. Release 4.0, based on GNU Emacs 18.58.

[Fou90] Open Software Foundation. *OSF/Motif Programmer's Guide, Revision 1.1.* Prentice Hall, Englewood Cliffs, New Jersey 07632, 1990.

[Hen91] P.R.H. Hendriks. *Implementation of Modular Algebraic Specifications.* PhD thesis, University of Amsterdam, 1991.

[Kli91] P. Klint. A meta-environment for generating programming environments. In J.A. Bergstra and L.M.G. Feijs, editors, *Proceedings of the METEOR workshop on Methods Based on Formal Specification*, volume 490 of *Lecture Notes in Computer Science*, pages 105–124. Springer-Verlag, 1991. Revised version to appear in: ACM Transactions on Software Engineering and Methodology.

[Koo92] J.W.C. Koorn. GSE: A generic text and structure editor. In J.L.G. Dietz, editor, *Conference Proceedings of Computing Science in the Netherlands, CSN'92*, pages 168–177. SION, 1992. Also appeared as Report P9202, University of Amsterdam.

[KR78] B.W. Kernighan and D.M. Ritchie. *The C Programming Language.* Prentice-Hall, 1978.

[Lan86] B. Lang. On the usefulness of syntax directed editors. In R. Conradi, T.M. Didriksen, and D. Wanvik, editors, *Advanced Programming Environments*, volume 244 of *Lecture Notes in Computer Science*, pages 47–51. Springer-Verlag, 1986.

[LeL91] INRIA, Rocquencourt. *LeLisp, Version 15.24, Reference Manual*, 1991.

[LLG90] D. Lewis, D. LaLiberte, and GNU Manual Group. *GNU Emacs Lisp Reference Manual.* Free Software Foundation, Inc., 675 Massachusetts Avenue, Cambridge, MA 02139 USA, 1.03 edition, December 1990. Describes Emacs Version 18.

[SG86] R.W. Scheifler and J. Gettys. The X Window System. *ACM Transactions on Graphics*, 5(3):79–109, 1986.

[Sta81] R.M. Stallman. Emacs, the extensible, customizable, self-documenting display editor. In *ACM SIGPLAN/SIGOA Symposium on Text Manipulation*, pages 147–160, Portland, Ore., 1981. The conference proceedings appeared as *SIGPLAN Notices*, vol. 16, no. 6, June 1981.