UNIVERSITY OF JOENSUU
DEPARTMENT OF COMPUTER SCIENCE
Report Series A

VinEd - A System for Program Manipulation Through User-Definable Simultaneous Views

Jorma Sajaniemi, Kari Ikonen

Report A-1998-4

ACM D.2.3, D.2.6, K.6.3
ISSN 0789-7316
ISBN 951-708-632-6

VinEd - A System for Program Manipulation Through User-Definable Simultaneous Views

Jorma Sajaniemi, Kari Ikonen

University of Joensuu
Department of Computer Science
P.O. Box 111
FIN-80101 Joensuu
Finland

E-mail: Jorma.Sajaniemi@Joensuu.FI, kikonen@cs.Joensuu.FI

Abstract:

Programmers' mental representations of programs do not obey the order, form and immediate content of program text. Multiple visible representations, or views, of a program help programmers to construct mental representations that they need in, e.g., maintenance tasks. But current programming tools and environments provide only a very limited support for such views.

We have implemented VinEd, a language independent editor framework that supports an unlimited number of views of the program text. Programmers can use predefined views and create their own views by defining filtering functions. Programs can be edited through any view providing a reverse operation that constructs the original representation from this view. Filtering and reverse filtering can be any operations that accept textual input and produce textual output. Further, VinEd extends the notion of a view to any activity that can be based on the original program, e.g., program compilation, or version control functions. Thus VinEd can be extended to form a complete programming environment.

In this paper, we consider the role of program views in the programming tasks and look at the way views are supported by current systems. We then describe the salient properties of VinEd from the user's point of view, and present examples of views that help in program comprehension as well as of views that take care of software engineering activities. Finally, we discuss the most important implementation issues of VinEd, and analyze its performance.

Keywords: program views, program editing, programming environment, program comprehension

ACM CR categories: D.2.3, D.2.6, K.6.3

Contents

1. Introduction

Programmers' tasks are manifold including program creation, comprehension, debugging, modification etc. The corresponding cognitive processes are not well known but it is clear that programmers utilize many kinds of information about programs in these tasks [1, 5, 7, 11, 18, 22]. Computer programs are large and complex objects and include a huge number of details. Even when working with miniature programs people are unable to keep all important details in human working memory but must use external devices such as text editors to make notes for further actions [2].

With real size programs the limits of human working memory form a limiting barrier to programmers and they must continuously refresh and rebuild parts of their mental representations needed in their tasks. This rebuilding starts from the notes they have previously made and from the physical representations, usually the program text, provided by the programming environment. The ability to abstract high-level information from concrete program code has been found to be essential to the performance of programmers working with program modification tasks [6].

But all parts of mental representations do not obey the order, form and immediate content of program text [5, 22]. For example, the overall structure of a program forms one essential part of program representations but it cannot be easily seen from the program code itself as the important lines that drive the representation are dispersed around the program. Similarly, the way a certain variable is used in the program can be found only by searching for all the occurrences of the variable. The normal program text representation is all that a compiler needs but humans need multiple representations, or views, of the same program to support the construction of different aspects of mental representations. For example, Meyers and Reiss [9] have shown that programmers' productivity can be improved by offering them multiple views of the same program.

The exact characteristics of mental representations used in various programming tasks are currently not known. As a consequence, the construction of an ideal set of views that would fit all programmers and provide all representations required in all programming tasks is beyond the current state of art. It may even be impossible to define representations that every individual can call sensible, and views probably need to be defined according to the environment and background of the programmer [17]. This problem can be solved by letting programmers define their own views and add them to the programming environment. As programmers are programming experts they are able, and supposedly also keen, to create supporting programs that construct views that they feel to be helpful for their tasks.

Current programming environments provide some views but there are no provisions for programmers to add new views. We have designed and implemented a program editor framework, VinEd, that is capable of showing multiple editable views of the same program, and that allows programmers to define new views. VinEd extends the notion of a view to any representation that can be obtained from the original program. As a consequence, it is possible to create views that do not allow editing. A non-editable view can be as simple as the list of all function headings, or a result of a complicated operation like compilation. Thus VinEd can be easily extended from a mere editor to a complete programming environment.

VinEd consists of an infrastructure that supports change propagation between views. This is accomplished by the co-operation of a single main process and multiple view processes. The main process is responsible for opening and saving files, and creating view processes to handle views of the files. Each view process displays a view, i.e., some representation of the file, and gives the user the possibility to edit the original file through this representation. Views are created (and recreated when the user makes changes in other views) by filters that can be any programs or commands that accept textual input. For example, pretty printers like `cb` and `indent`, general tools like `diff` and `lpr`, and specialized view generators like VOCE [12] can be used as view filters. Moreover, users can easily use their own programs for view filters.

In this paper, we will first shortly describe current commercial and academic systems that support multiple views of the static aspects of programs. Then we will describe VinEd from the user's point of view, and the main features of its implementation. Finally, we present a performance analysis of the system. VinEd is available for download at `ftp://cs.joensuu.fi/pub/Software/VinEd`.

2. Views in Current Systems

Current programming environments for object oriented languages provide object browsers that show the class hierarchy tree with information about class attributes and operations in each node of the tree. For example, the Borland Delphi object browser enables a programmer to visually explore the hierarchical relationships of objects, classes, and units (i.e., collections of classes). Moreover, it allows programmers to see a list of all global symbols with their types and locations of occurrences (in the form of file name, line number pairs). The actual occurrence can be found by double clicking the location information. Thus, Borland Delphi has two views (in addition to the standard program text view): object/class/unit hierarchy view, and global symbols list view. A program cannot be edited through either of these views. The programmer can select what properties (like attributes and operations) are shown but he or she can not create new views.

PECAN [14] is a family of language dependent program development systems that have a fixed set of editable and non-editable views. Each PECAN system can handle programs in some particular language and is generated from the descriptions of the syntax and semantics of the language.

PECAN represents programs internally as abstract syntax trees. The whole program can be seen through two views: the textual view operated by a syntax-directed editor, and the Nassi-Shneiderman view showing a graphical flow chart. There are also partial views that show some representation of some parts of the program: the symbol table view displaying symbols with their properties and scopes, the data type view showing similarly properties of data types, the expression view displaying the syntax tree of an expression, and the flow view showing a flow graph. All these partial views are bound to the syntax-directed editor textual view so that the partial views show the symbol, data type, expression etc. that is the focus of the current editing operation. Conversely, by clicking some element in any of the partial views the user can make the corresponding node in the syntax-directed editor the current location for editing.

MultiView [8] is similar to PECAN: it uses an internal representation consisting of an abstract syntax tree, and it provides a fixed set of views: the textual view, the flowchart view, and the abstract syntax view. Like PECAN, MultiView provides execution based views, also. These are meaningful when the program is under execution but are not relevant during program editing. PECAN and MultiView can not be extended by user-definable new views.

VOPE [15] is an editing tool that lets users alternate between the normal whole program view and a variable-oriented view. The variable-oriented view of a single variable consists of the declaration of the variable together with all its uses throughout the program. The programmer can choose whether the view will contain all variables or just a single variable. In both cases the view can be edited. VOPE works together with a standard editor, i.e. `vi`, so that the user need not quit editing session but VOPE loads the new view automatically to replace the old view in the editor. VOPE is language dependent and works on Pascal only. A similar tool for C has later been implemented [12]. MLR [19] is a multiple view system that monitors the coupling of variables and produces a flashing icon in a view to warn of potentially error-prone situations.

Shneiderman & al. [17] describe a hierarchical browser consisting of two windows: the program text and a non-editable listing of function headers. These windows scroll synchronously, i.e., when a user scrolls either window the system scrolls the other so that the same function is visible.

The Programmer's Torch [3] is a projected software tool designed to give answers to questions about data flow and to present the structure of a program in a clear way. There is no way to edit the results and the tool is language dependent, i.e., it can be used with BASIC programs only. The system demonstrates a divergent, dynamic, approach to views: the user does not automatically get the whole view but the appropriate pieces of information must be requested through explicit questions. Similarly, TELLYOU [19] is a documentation browser that is supposed to be able to answer questions concerning, e.g., delocalized plans which are pieces of code that are conceptually related but physically separately located in the program text. Green & al. [4] extend this idea of question-originated views by considering the possibility to browse object oriented programs by querying a knowledge base of attributes and relationships of objects and code fragments. Such a query-based approach minimizes the information shown in a view and may be practical if views become massive. On the other hand, the result must be explicitly requested and is transient by its nature.

GOOSE [13] applies view-based approach to the earlier phases of software engineering process. The system supports three views for system design. All views are editable and consist of graphical descriptions. The functional view takes the form of a hierarchical data flow diagram, the behavioral view consists of transition diagrams, and the structural view is a class diagram. The three views contain some common information but each view has some information that is not contained in others. There is no single complete view and hence all views are needed to describe a system. Changes made in one view do not automatically propagate to other views and they must be made manually in all views. A consistency checker is available to find inconsistencies between the views. A user cannot add new views or customize existing views in any essential way. This approach is well-known in various software engineering methodologies.

Welsh & Han [21] see software development as a document-based process and they are interested in the connections between program texts, design specifications, and requirements. They also consider various textual and graphical representations of programs but leave open the user interface aspects of such views.

All the systems described above lack extensibility: users are not allowed (or at least not supposed) to add new views, and the systems support a single programming language. Moreover, with the exception of VOPE and PECAN, views are either not editable at all or they do not appropriately propagate changes from any other view but from the normal, whole program textual view.

3. Views in VinEd

VinEd was designed to be a language independent program editor that is able to support an unlimited number of editable and non-editable views. Language independence means that VinEd cannot suppose that programs that are edited obey any syntactical rules. Thus VinEd cannot use a syntax tree, or any other syntax or semantics-driven representation internally. As a consequence, the internal representation of files is just the same as the external representation, i.e., plain text.

In VinEd, a view can be any text obtained by applying some filter program to the original file. As the output of a filter is text (and not a graphical representation), programming new filters is relatively easy. If a reverse filtering program is also given, files can be edited (and not just looked at) in the view. For example, the list of all function names in a program facilitates comprehension of the program by showing structured information and the underlying design [17]. A Pascal programmer can easily make a short program (using a single line UNIX shell script consisting of a call of the `grep` command with the result piped to a `sed` command) that makes such a list for his or her own programs (though the solution in the general case is somewhat more tedious). Thus he or she can see the list in a view window with the program text in another window. Even the reverse filtering is simple in this case (i.e., a single `sed` command) and hence the programmer can change the name of a function in the function name list, and the system will automatically apply the reverse filtering and thus change all occurrences of that name in the program.

It is clear that more powerful views are also needed. An example is the variable-oriented view of programs suggested by Sajaniemi & Niemeläinen [15]. In this view, occurrences of each variable are collected together to give a better understanding of the roles of variables in the program. Once a program that creates such a view is written, it is straightforward to use it in VinEd.

VinEd accepts also empty views that are produced because of the side-effects of the processing. For example, the check-in operation of a version control system may be defined as a "view" with no output, and hence no view window. There is no sense in checking the file in after each change, and VinEd allows the view to be updated on user request only. Program compilation is another view that is generated because of its side-effects. It has a textual representation, though, as users want to see the possible list of errors.

VinEd provides an infrastructure where users can hook their favorite views. Therefore, VinEd is shipped with some ready-made views for some programming languages but users are supposed to add their own views. To make the addition of new views simple, VinEd only supports non-synchronous textual views of static aspects of programs. As views are non-synchronous, i.e., scrolling one window does not affect the position of text in other views, filters need not tell the system how various parts of a view are connected to parts of the original file. The limitation to only use textual views is not an inherent restriction of VinEd, but an acknowledgment of the fact that filters for graphical views are perhaps too hard to write. Finally, views are (re)generated only when some other view is changed, and thus they cannot contain continuously changing information but are static representations of the form of the file at a given time.

Users can make editing operations not only in views but by using standard text editors like `vi` or `emacs`. VinEd treats standard editors somewhat different from other views as editors are long-term processes in contrast to filters that are called every time they are needed.

4. VinEd User Interface

To start editing a program text file, the user opens the file and asks for at least one editable view to be created. The user can use a built-in editor or a standard editor like `vi` or `emacs`. In Figure 1, the two large windows show two views of the same program: one with the built-in direct view editor (with the name "Direct (Proportional font)" in its title bar) and the other using `vi` editor in a VinEd editor window. Further, the view in the lower left corner shows the names of all functions in the program, and the view next to it shows all unsaved changes as produced by the UNIX `diff` command. Although the diff view contains a small fraction of the program only, it can propagate changes back to other views by recreating the whole program text with the `patch` command.

All open files are listed in the VinEd main window (located in the upper left corner in Figure 1) and each view that represents the same file shows, at the bottom of the window, the name of the file with the same background color. Thus the user can easily recognize views that belong together even when many files are edited simultaneously. When a file is selected, VinEd lists all currently open views of the file in the VinEd main window. In Figure 1, there is only one open file with three open views. Normally VinEd hides all views that do not belong to the currently active file but the user can nail views on the screen so that views of many files are available at the same time.

It is also possible that a view does not produce anything on the screen. For example, the print view creates a printer listing with no visible result on the screen. As a result, the view mechanism expands to contain general command interpreter functions. Thus, all software engineering tasks can be configured as views resulting in a uniform programming environment.

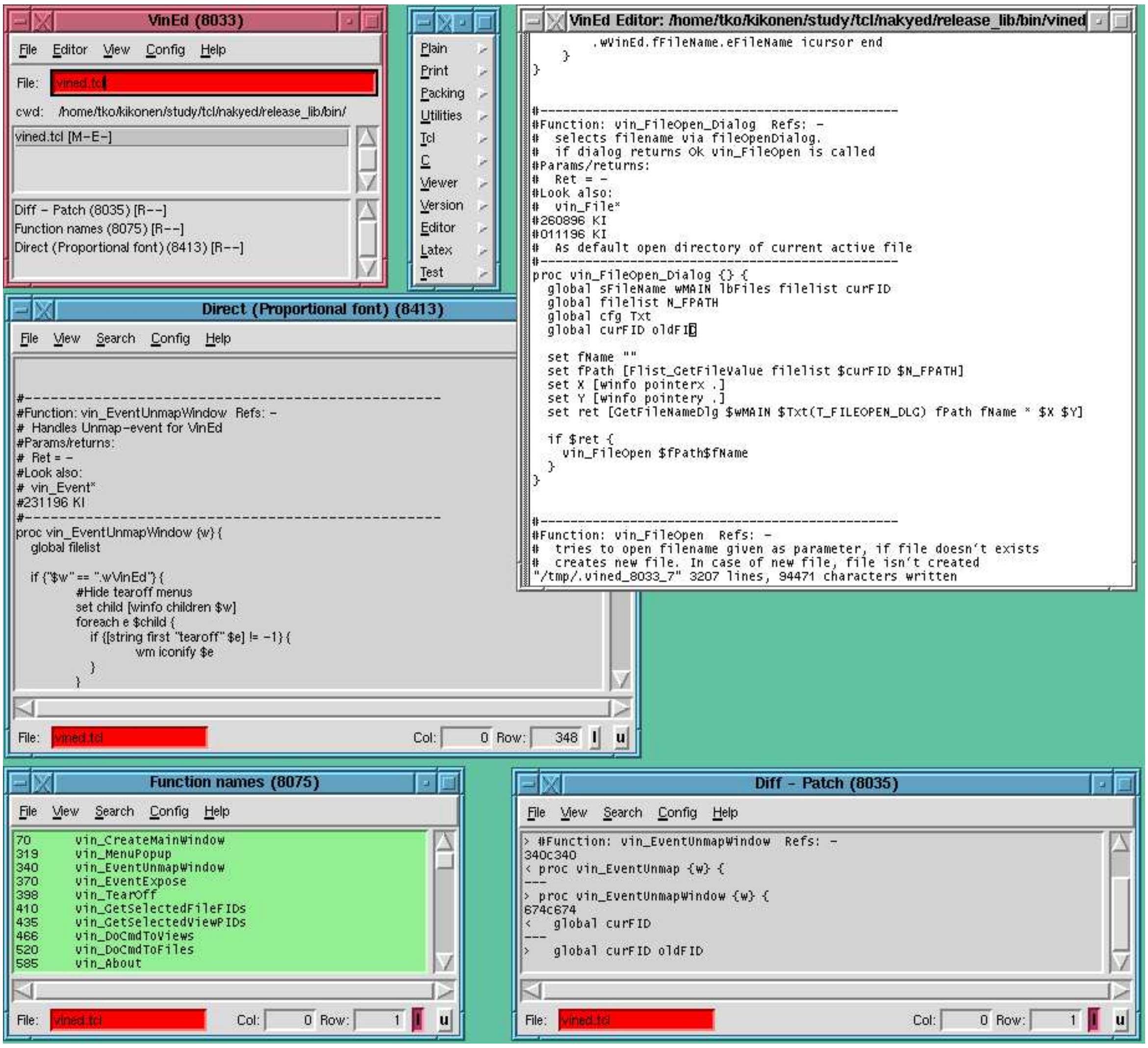

Figure 1: VinEd user interface with various views

VinEd does not automatically open views. Instead, the user must request each view through the view menu (on right hand side of the VinEd main window in Figure 1) that lists all available views. The view menu is divided into user-definable view categories, e.g., "Plain" for direct views with different fonts, "Print" for views that show nothing on the screen but print the file on a printer, etc.

Some views are updated by user request only. For example, there is no sense in producing a compilation view (i.e., the executable program) if the source program is not syntactically correct or if some part of a change is still to be done, nor is there any sense in producing a new printer listing each time the file is changed. A view can also be configured to update itself only when the original file is saved. Thus the user is relieved from explicit update requests but still updates happen only when the file is in a valid state.

5. View Definitions

For VinEd, a view is a textual entity formed on the basis of a text file. The text in the view may be empty, and the process of its creation may involve side-effects. Each view is associated with either one or two programs, so called filters. The first filter, in-filter, is compulsory. This filter gets the original file (or changed rows only) as its input and it must produce the view (or the changed part of the view) as its output. The second filter, out-filter, is used to recreate the file from the view when the text in the view has been altered. If the view has no out-filter then the view is not editable.

There are no restrictions concerning the programming languages used for creating filters as long as they produce executable filter programs. Thus traditional languages as well as scripting languages can be used for filters. Filters get the text to be processed as an input file and generate the result in standard output.

Filters are invoked by VinEd whenever the view has to be updated, or when changes in the view must be propagated back to the original file and further to other views. There is no need to describe the evoking mechanism in view definitions, as it is totally implemented in VinEd itself.

```
(a)   NORMAL REGISTER generic.tcl default_view.cfg "Direct view"
           {PARTIAL IMMEDIATE "cat <$IN"}
           {PARTIAL IMMEDIATE "cat <$IN"}

(b)   NORMAL REGISTER generic.tcl default_view.cfg "Diff - Patch"
           {FULL DELAYED "diff $OF $IN"}
           {FULL AT_USER_REQUEST "patch $OF -o $OUT <$IN"}}

(c)   NORMAL REGISTER generic.tcl default_view.cfg "Variable oriented C view"
           {FULL AT_USER_REQUEST "voce <$IN"}
           {FULL AT_USER_REQUEST "voce <$IN"}

(d)   NORMAL REGISTER generic.tcl gcc.cfg "gcc"
           {FULL AT_USER_REQUEST "gcc $U(1) $OF -o $U(2) |& cat"}
           {NONE NONE ""}

(e)   NO_WINDOW - - - "Check In"
           {FULL AT_USER_REQUEST {ci -u <<$V(1) $OF}}
           {NONE NONE ""}}
```

Figure 2: Examples of view definitions

A direct view, i.e., a copy of the text itself, can be defined as in Figure 2(a). The keyword `NORMAL` means that the view will be shown in its own window (as opposed to `NO_WINDOW` meaning that the view is used for side-effects only) and the keyword `REGISTER` means that this view will be registered in the view list of the VinEd main window. The view process that will be used to display the view on the screen is an instance of the program `generic.tcl` and it will use colors and window geometry described in the configuration file `default_view.cfg`. These two files are contained in the VinEd distribution but other view processes and configuration files may de defined, also. The name of the view, "Direct view", will be shown in the view menu and window titles.

Next the two filters of the view are defined. Normally views are (re)created by giving the whole file to the in-filter and replacing the old view by the output of the filter. In the case of large files this may take a long time resulting in time lags in view updates.

To obtain better efficiency, VinEd supports also views that use filters on a line basis. The keyword `PARTIAL` means that the filter is capable of accepting any part of the file (consisting of consecutive lines) to produce the corresponding part of the result. Thus, when a user makes some changes in the file, the filter does not need the whole file to produce the changes in the view.

For an in-filter, the keyword `IMMEDIATE` means that the filter should be invoked whenever any out-filter (of other views) has processed changes made by the user. For out-filters, `IMMEDIATE` means that changes should be processed as quickly as possible, i.e., the out-filter should be evoked for each editing operation. In practice, there is a system-wide parameter (say, 0.1 seconds) giving the frequency of immediate out-filter invocations.

The filter programs in the example are simple: just pass the input file, denoted by `$IN`, as standard output. Note that for an in-filter, the special notation `$IN` means the contents of the file that must be shown in the view, whereas for an out-filter it means the current representation in the view. The use of the same notation is justified by the fact that in both cases `$IN` means input for the filter in question.

As another example consider the diff-patch view that shows all unsaved changes made to the original file, and is capable of patching changes made in the diff-listing back to other views. The view is defined in Figure 2 (b). Both the in-filter and the out-filter require the whole file (`FULL`) in order to operate. The in-filter is not invoked at each change but only at certain intervals (`DELAYED`) since the `diff` operation takes some time and it may be practical to wait some time to get more changes to be processed together. The out-filter is invoked by explicit user request only.

The filters operate on the original file (`$OF`) and the current (yet unsaved) version of the file (`$IN` for the in-filter; `$OUT` for the out-filter). The in-filter produces the diff-listing as its standard output. If the user makes changes in the view, the changed diff-listing in the view window is passed to the out-filter (as `$IN` of the out-filter).

The next example, Figure 2 (c), considers the variable-oriented view of a program [15]. Such a view can be produced for C programs by a program called VOCE [12]. VOCE uses an auxiliary file where it saves the original program text to be able to merge changes made in the variable-oriented view back to the normal view. It looks at the form of its input to deduce the direction of the required transformation, i.e., from normal view to variable-oriented view or vice versa. VOCE accepts syntactically correct programs only. Therefore, both filters are invoked at user request only.

As an example of an non-editable view consider the compilation of the program as defined in Figure 2 (d). The configuration file `gcc.cfg` instructs VinEd to show the result in the view window along the way it is produced. So the user can see the first error messages without waiting for the whole compilation to end. The user can even interrupt the compilation if there is no sense in finishing it.

The in-filter accepts two parameters (`$U(1)` and `$U(2)`). When the filter is invoked for the first time, VinEd asks the user to provide values for these parameters before running the filter program. Thus the user may specify additional compiler flags, and the name of the output file. The view has no out-filter as there is no way to edit the executable binary file and propagate these changes back to the C program.

Views can be used for other software engineering tasks as well. For example, the view in Figure 2 (e) implements RCS version control [20] check-in operation. This view has no window, so there is no need to describe it in detail as reflected by the three dashes. The command `ci` requires a user provided comment (e.g., "Changes described in Change Control Form 3054") as its input. The construction "`<<$V(1)`" is a special notation of VinEd meaning that the string provided by the user is given as standard input to the command.

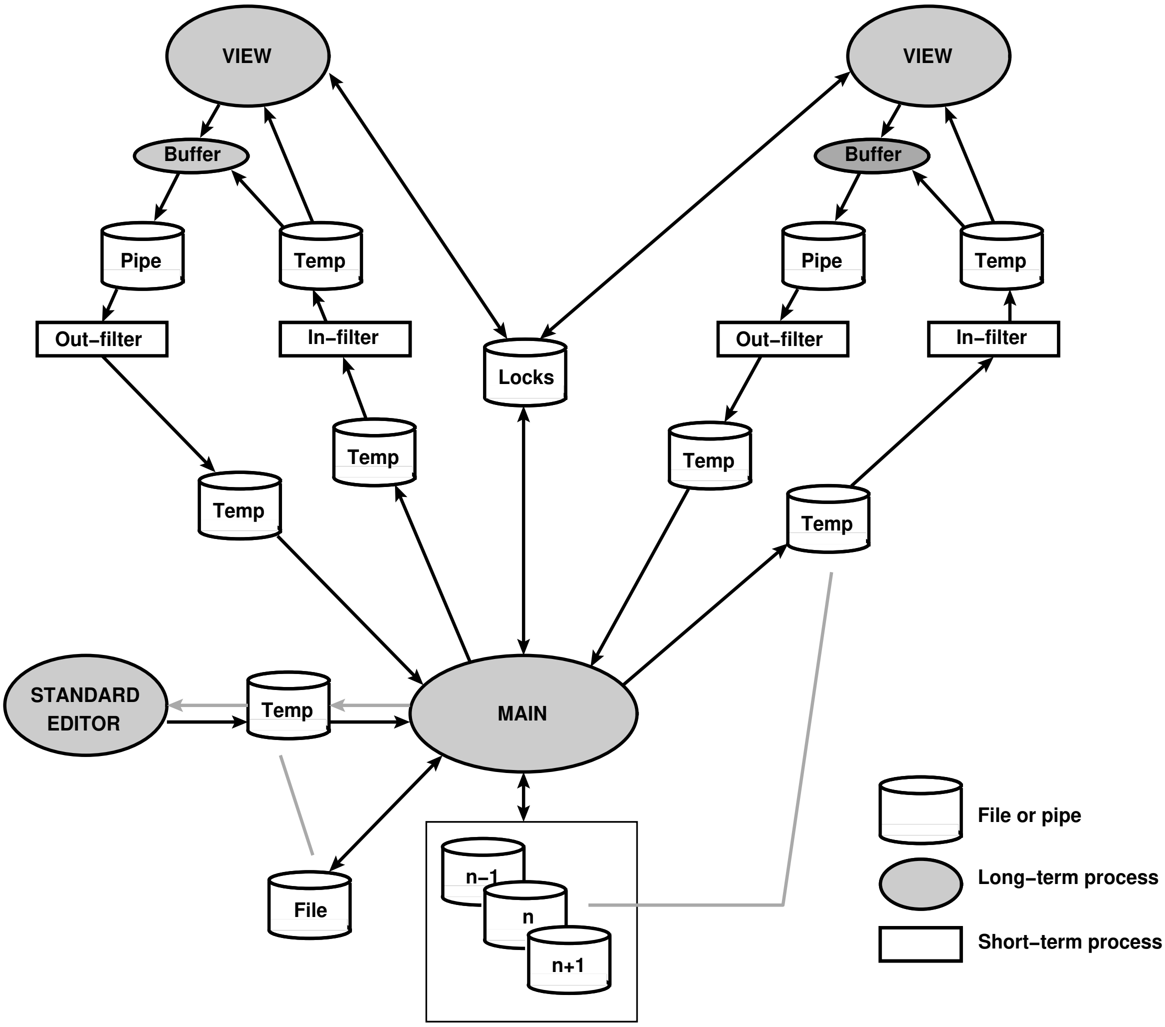

Figure 3: Overall structure of VinEd

6. Implementation

VinEd is implemented on UNIX using Tcl/Tk [10]. UNIX provides a good platform for handling simultaneous processes and interprocess communication.

The kernel of the system is the main process (`MAIN` in Figure 3) that is responsible for opening and saving files, starting up views, sending update information to views, and arranging all communication between views and standard editors. The main process also controls the VinEd main window and other view independent windows.

Each view is shown in its own view window by its own view process, which is usually an invocation of the `generic.tcl` program. All view windows work as separate processes responsible for updating and handling modifications of its own content, as well as sending update information to the main process. The communication and coordination between view processes and the main process is implemented by message channels and lock files.

View processes have no direct access to the original file being edited by the user. The main process controls a set of copies of the file representing different versions corresponding to different update levels of the file. Whenever a view process sends changes to the main process, `MAIN` creates a new version, assigns it a new update level number, and passes the new version to other views. Due to time lags views may at a given time show different versions of the file in their windows.

When the user asks the system to save the file, the main process replaces the original file by the version having the highest update level. Figure 3 shows the original file and some of the versions denoted by update level numbers `n-1`, `n`, and `n+1`. The system supports simultaneous editing of any number of files, but for simplicity only one file is shown in Figure 3.

Each view knows the update level of the version it is showing. Only views which are on the highest update level allow the user to make changes in their view windows. Thus users are prevented from changing a view that does not accurately reflect the current state of all editing operations.

If the in-filter of a view can handle partial data, the main process may pass a new version of the file to the view by giving a temporary file containing the changed lines only. More often, however, the main process gives directly the file name of the currently highest version. This is reflected in Figure 3 by the gray line unifying one of the temporary files with one of the versions. The version processed by an in-filter at a given time may not be the highest version as a new version may be created during the filtering process. The filtered view is passed to the view process to be displayed on the screen and to a buffer process that is used to accelerate the out-filtering process and to reduce the work of the view process so hat it can respond to user actions quickly.

The most important lock files are update locks and a focus lock. Every process, including the main process, has its own update lock which tells whether the process is currently receiving update information. The focus lock is owned by the view which is on the highest update level and has the window manager focus, i.e., is receiving user keystrokes. Thus all changes are always originated in the view possessing the focus lock.

Filters are invoked repeatedly when views have to be updated. Filter processes have a relatively short life every time they are invoked. Theoretically, each invocation of a filter is independent of other invocations, but filters can use their own files to carry information from one invocation to another. The data flow to and from the filters is implemented by pipes and temporary files. Pipes are used in cases where the lifetime of the producing process is so long that the consuming process might reach the end of the file before the file has been totally created.

Standard editors are treated in a different way because they do not respect VinEd locking protocols and they don't monitor changes in the file being edited. When a standard editor is started, the main process gives it either a temporary file or the original file for editing. Thereafter the main process propagates changes in this file to views but views cannot be edited.

The problem of standard editors can be solved by writing a new view process program that sits between a standard editor and the user, directs user actions to the editor, shows editor output on the screen, and communicates with the main process. When changes occur in other views, this view translates them to user actions and passes these to the editor. So the file in the editor is updated. Such a view could be implemented but this has not been done yet.

7. Performance

We measured the time it takes for changes made in one view to propagate to other views. The tests were run on a 200 MHz Pentium/MMX processor with 64 MB memory and running Red Hat Linux (kernel 2.0.32) with Tcl/Tk 8.0. The system-wide parameter giving the frequency of immediate out-filter invocations was set to 0.1 seconds. Moreover, the times were recorded by a stop watch implying a 0.1 seconds accuracy of the measurements. There is an understanding [16, pp. 284-297] that response times less than 2 seconds are acceptable in this kind of a situation and so there was no need to measure the view update times with better accuracy.

We used C and Tcl/Tk programs of various lengths as files to be edited. The programs were 100, 500, 1,000, 3,000, and 10,000 lines long. In practice, hardly any modules are longer than 3,000 lines and we indeed had some troubles in finding 10,000 lines long modules. We were interested to see how soon after each editing operation the change was reflected in other windows. Fast propagation of changes is important for users not only for convenience but also for the reason that editing operations are not allowed in a window unless it reflects the highest update level of the file.

One of the views in each case was a direct view that was used for editing. In three cases we used one additional view that was either a second direct view, the diff-patch view described earlier, or a program analysis view which was VOCE [12] for C programs and a list of function names together with line numbers for Tcl/Tk programs. Figure 4 shows the times needed to propagate a change in the direct view to the other views. When the second view was the direct view or the diff-patch view, the response time was less than 0.6 seconds for programs shorter than 3,000 lines and less than 0.9 seconds for 10,000 lines long programs. The longest response times occur in the case of VOCE, but this is not a surprise as the analysis process of VOCE takes its time. For comparison, the gray line in Figure 4 shows the time required by VOCE to analyze the files when run directly from a command line.

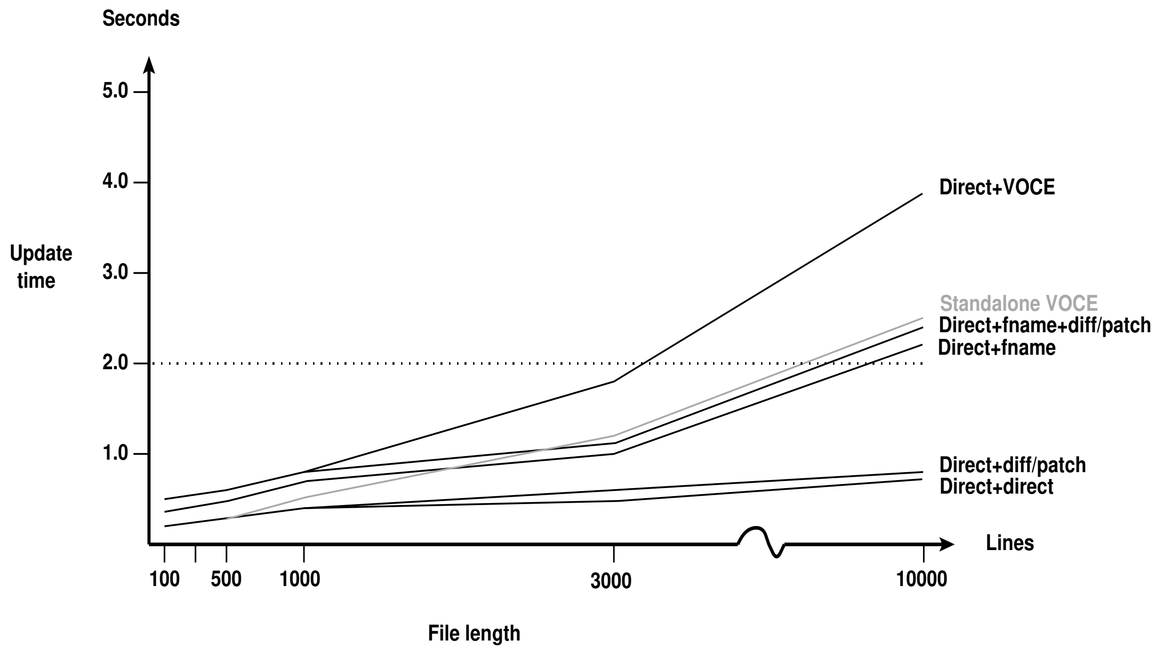

Figure 4: View update times

Figure 4 shows also the response time in the case of three simultaneous views: the direct view, the function names view, and the diff-patch view. The addition of the diff-patch view adds only little to the response times of the other two views. In general, as the VinEd overhead for version control and clerical tasks grows very little with respect to the number of simultaneous views, the use of several views is practical and does not slow down processing.

To see how VinEd works in extreme conditions, we used the combinations of two direct views, and one direct view with the diff-patch view to edit a file of 100,000 lines. In both cases the response time was around 2.4 seconds.

8. Conclusions

Programmers' mental representations of programs do not obey the order, form and immediate content of program text. VinEd provides a means to overcome this problem by representing any number of views of the same program text simultaneously. Users can define new views and attach them to VinEd easily. We have shown that VinEd extends the notion of a view from visible representations to executable actions. As a result, software engineering tasks can be considered to be views and be triggered by the same mechanism as normal view updates. This idea of considering executable actions as views makes VinEd applicable to other domains than programming as well.

In VinEd, all views having a screen representation are textual. This is not an inherent property of the mechanisms but merely a result of the standard view presentation programs delivered in the VinEd distribution. Users can define new programs to represent views on the screen, and they may well use some graphical notation. The way view presentation programs interact with the VinEd main process is, however, complicated, and few users are expected to be able to construct such programs. On the other hand, filter programs are much easier to write and the normal way to add new views is supposed to be to write or adopt new filters. VinEd is meant to be used for editing program texts and consequently it's users are supposed to be programmers who are able to write new filters for their own needs.

The view presentation programs delivered in the VinEd distribution do asynchronous scrolling, i.e., user actions in one view do not automatically scroll other windows to show the same part of the program. It is possible to implement a new view presentation program that would, after changes in other views, automatically scroll to some change in its own view. Note, however, that a single change in one view may result in many changes in other views and a view presentation program cannot in general know which of these changes is the most important for the user. Moreover, views let the main process know all changes but do not tell it what part of the program is shown in the view window, so VinEd cannot give any support to synchronous scrolling.

The performance analysis showed that VinEd can be run on current workstations with acceptable response times. Further, large screens capable of representing a large amount of information are becoming more popular. Thus the technology is ripe for simultaneous presentation of multiple views.

References

1. Davies S. P. (1993) The Structure and Content of Programming Knowledge: Disentangling Training and Language Effects in Theories of Skill Development. *Int. J. Human-Computer Interaction* **5**(4) 325-346.
2. Green T. R. G., Bellamy R. K. E., Parker U. M. (1987) Parsing and Gnisrap: A Model of Device Use. In: Olson G. M., Sheppard S., Soloway E. (eds.) *Empirical Studies of Programmers: Second Workshop*, Ablex Publishing Corporation, New Jersey, 132-146.
3. Green T. R. G., Cornah A. J. (1985) The Programmer's Torch. In: Shackel (ed.) *Human-Computer Interaction INTERACT'84*. Elsevier Science Publishers, Amsterdam, 397-402.
4. Green T. R. G., Gilmore D. J., Blumenthal B. B., Davies S., Winder R. (1992) Towards a Cognitive Browser for OOPS. *Int. J. Human-Computer Interaction* **4**(1) 1-34.
5. Green T. R. G., Navarro R. (1995) Programming Plans, Imagery, and Visual Programming. In: Nordby K., Helmersen P., Gilmore D. J., Arnesen S. A. (eds.) *Human-Computer Interaction Interact'95*. Chapman & Hall, London, 139-144.
6. Holt R. W., Boehm-Davis D. A., Schultz A. C. (1987) Mental Representations of Programs for Student and Professional Programmers. In: Olson G. M., Sheppard S., Soloway E. (eds.) *Empirical Studies of Programmers: Second Workshop*, Ablex Publishing

Corporation, New Jersey, 33-46.
7. Littman D. C., Pinto J., Letovsky S., Soloway E. (1986) Mental Models and Software Maintenance. In: Soloway E., Iyenger S. (eds) *Empirical Studies of Programmers*. Ablex Publishing Corporation, New Jersey, 80-98.
8. McCarthy M., Marlin C. (1994) Interprocess Communication Protocol Support in a Distributed Integrated Software Development Environment. *Australian Computer Science Communications* **16** (1) Part B 363-371.
9. Meyers S., Reiss S. P. (1992) An Empirical Study of Multiple-View Software Development. *Software Engineering Notes* **17** (5) 47-57.
10. Ousterhout J. K. (1994) *Tcl and the Tk Toolkit*. Addison-Wesley, Reading, MA.
11. Pennington N. (1987) Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs. *Cognitive Psychology* **19**, 295-341.
12. Porokka A. (1991) *VOCE - Variable Oriented C Editor*. (In Finnish) Special project, University of Joensuu, Department of Computer Science.
13. Reeves A., Marashi M., Budgen D. (1995) A Software Design Framework or how to Support Real Designers. *Software Engineering Journal* **10** (4) 141-155.
14. Reiss S. P. (1985) PECAN: Program Development Systems that Support Multiple Views. *IEEE Trans. on Software Engineering* **SE-11** (3) 276-285.
15. Sajaniemi J., Niemeläinen A. (1989) Program Editing Based on Variable Plans: A Cognitive Approach to Program Manipulation. In: Salvendy G., Smith M. J. (eds.) *Designing and Using Human-Computer Interfaces and Knowledge Based Systems*. Elsevier Science Publishers, Amsterdam, 66-73.
16. Shneiderman B. (1992) Designing the User Interface. Addison-Wesley, Reading, MA
17. Shneiderman B., Shafer P., Simon R., Weldon L. (1986) Display Strategies for Program Browsing. *IEEE Software* **3**(3) 7-15.
18. Soloway E., Ehrlich K. (1984) Empirical Studies of Programming Knowledge. *IEEE Trans. on Software Engineering* **SE-10** (5) 595-609.
19. Soloway E., Pinto J., Letovsky S., Littman D., Lampert R. (1988) Designing Documentation to Compensate for Delocalized Plans. *Comm. ACM* **31** (11) 1259-1267.
20. Tichy W. F. (1985) RCS - A System for Version Control. *Software - Practice & Experience* **15** (7) 637-654.
21. Welsh J., Han J. (1994) Software Documents: Concepts and Tools. *Software - Concepts and Tools* **15** (1) 12-25.
22. Wiedenbeck S., Fix V., Scholtz J. (1993) Characteristics of the Mental Representations of Novice and Expert Programmers: An Empirical study. *Int. J. Man-Machine Studies* **39**, 793-812.