Revision Towers

Christopher M. B. Taylor and Malcolm Munro
Visualisation Research Group,
Research Institute in Software Evolution.
Department of Computer Science,
University of Durham,
Durham, DH1 3LE, UK.
{C.M.B.Taylor, Malcolm.Munro}@durham.ac.uk

Keywords: Software Visualisation, Version Control, Configuration Management, Software Evolution, Open Source

Abstract

The use and development of open source software has increased significantly within the last decade. With it has come an increased, and necessary, use of version control tools to provide project management. A typical repository contains a mine of information that is not always obvious, and not easy to comprehend in its plain form. A visualisation has been created from this information to display how the repository has evolved. The visualisation allows the user to see where the active areas of the project are, how often, and how, changes are made, and how work is shared out across the project. Colour, layout and animation are all important features of the visualisation. In addition, issues of the importance and use of animation and consistency are raised. A prototype tool has also been developed to show how the visualisation works in practice.

1. Introduction

The development and use of open-source software has increased significantly in the last few years [1]. Online repositories such as SourceForge [2] actively encourage and support the concurrent development of open-source projects, with a version control system to aid the development of the project.

An important part of these repositories is that they encourage anyone to participate in the development of the project. Many of these participants will never have met, and the sole source of communication will be mailing lists, or failing that, examination of the source code itself. The vast majority of the participants also do not work on the project full-time [1]. This means that it is unlikely that the participant will have a good understanding of the current state of the software project, particularly with a large number of developers working simultaneously. It is not uncommon for such projects to have weekly releases. Consequently, being unable to work on the project for even a short length of time may make it difficult for the participant to come to terms with the recent progress. The documentation, and particularly a change log, may aid this process, but the presence of this information can not be guaranteed.

Alternatively, it may be the case that a skilled software engineer wishes to join the project. As stated, the documentation often falls behind the current state of the software, and so the engineer must rely on the source code to comprehend the system. However, with possibly hundreds or thousands of files of source code to consider, and little idea of the importance of each, it may be that the engineer decides not to make that initial investment. The result is that the skills that they could have contributed to the project would be lost. In addition, it may not be apparent from the documentation who has responsibility for the various parts of the project, or who has the best understanding of the area. In this case, knowledge possessed by the more experienced developers working on the project may never be transferred.

A final picture is that of a manager of the project. It may be that they wish to understand the areas of the source code undergoing most development, and those that have not been recently worked on. Alternatively, they may wish to examine the reliability of code developed by a particular author, perhaps in order to encourage them to work on a more or less critical part of the project. Finally, they may wish to see the parts of the project that become affected when new functionality is implemented.

These scenarios are realistic, and may be applied to small-medium sized software projects (about 50 - 500K LOC). Typically, these projects could be anything from serious development tools and environments through to multi-player networked games. There is clearly no single solution to completely solve all of these problems. However, much relevant information can be gleaned solely from the version control repository that is an integral part of such open-source projects, as demonstrated by [3],[4], allowing conclusions to be drawn about the state and direction of the project.

Almost all version control systems will contain some means of viewing the current state of the repository. This is normally done using a text-based system, for example by CS-RCS [5]. Some newer systems (such as VRCE [6]) will also allow the viewing of the structure of a file within that repository, allowing information about a particular revision to be viewed quickly. However, what is less well supported is the viewing information relating to the entire repository. Due to the size of a typical repository, and number of relations that exist within this data, the task is one well suited to visualisation. Knight [7] states that software visualisation "makes use of various forms of imagery to provide insight and understanding and to reduce complexity of the existing software system under consideration." Therefore, by showing the information within the repository using visualisation techniques, it is hoped to provide an increased understanding of specific parts of the project, with the aim of finding some partial solutions to the problems encountered when comprehending open source software.

2. Similar work

At least two visualisations have been developed which take this approach specifically. VRCS [8], shown in figure 1, shows clearly the relationships that exist within a small repository. A graph drawn in 3D space is used to show the state of the repository - heavy lines show the evolution of an individual file, and light lines indicate how these files affect particular releases. Time is shown using the Z-axis. The visualisation is very clear on a small scale - it is easy to see that file A has gone through four revisions, with v.3 used for the second release of the system. However, Young when evaluating Zebedee [9] - a visualisation using 3D space to lay out a call graph - commented that the visualisation ran into problems as the size and scale of the graph increased, often with the 3D nature of the graph contributing to the apparent complexity. Therefore, the benefits of VRCS when used with much larger projects are unclear.

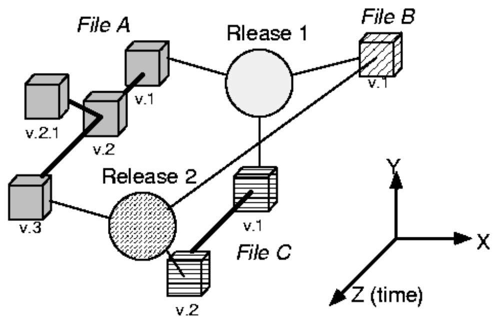

Figure 1 - VRCS [8]

The second, more promising, alternative to solving the problems set is 3DSoftVis [10], shown in figure 2. Although initially more difficult to understand than VRCS, a similar amount of information is displayed whilst handling repositories of a much larger scale. This visualisation is a result of zooming in to a subsystem of a much larger software project. Each column represents a file within that subsystem. Each row represents the state of that file at a given Release Sequence Number (RSN). Each RSN represents a release of the software. The colour of a square indicates the RSN at which it was last changed.

It can be seen from figure 2 that modules A and B were created at release five of the system, and have since undergone four changes. Module C has been modified at every release, in contrast to module D that has remained unchanged. Modules E and F were introduced for release 4, and Modules G to L were no longer incorporated at this point.

This visualisation also has a number of problems however. Firstly, although capable of handling a large number of modules - the visualisation has the potential for displaying several hundred - the number of releases that can be shown is limited, as a separate colour is required for each. Individual revisions that occur between releases are also considered as a single entity. Whether this abstraction is useful depends very much on the frequency of releases of the project. Projects that have releases every day will quickly exceed the number of colours that can be differentiated clearly. A project that has files updated frequently, but has a formal release on an infrequent basis, will abstract away many of the changes that a developer would be interested in.

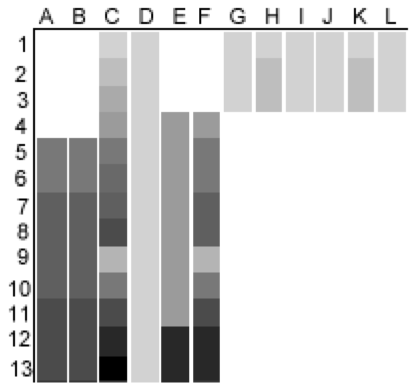

Figure 2 - 3DSoftVis [10]

Although 3DSoftVis covers some of the management aspects of the set problem, neither is ideally suited to the user. In the case of VRCS, although the detail exists it is presented in a restrictive and, on a realistic scale, confusing manner. In the case of 3DSoftVis, much of the interesting detail is abstracted away. It would be difficult to provide this information using the same view however, and so an alternative method is presented. This method provides most of the detail of the structure of the repository provided by VRCS at the cost of a less extensive view provided by 3DSoftVis. Furthermore, this alternative view allows more information from the log files to be displayed; dates can be considered in addition to the system releases, and there is support for providing more details of how the files have been modified, and by whom.

3. Revision Towers

Revision Towers, as with VRCS and 3DSoftVis, uses data obtained from typical version control log files. In this case, the visualisation is based around the information provided by using the 'log' options available within the version control systems RCS and CVS. This information can be provided quickly and easily, without extensive processing of the entire project. The log provides details for each file within the project, containing information such as the user who checked in the file, the date, and the version number. Also included is the number of lines changed since the last version, and a comment field which should be filled by the author when checking in.

A tower, shown in figure 3, represents two log files that are viewed side by side. (Towers are displayed in full colour, but have been converted to greyscale for printing purposes.) The central section represents software releases, as recorded within the log file, with earliest releases at the base of the tower, and the latest, as yet unreleased, at the very top. Each side section represents the history of a file, and how the individual versions of a file map to the releases.

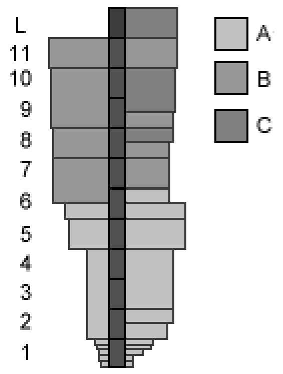

Figure 3 - A revision tower

The towers are then displayed in a grid formation to fill the available display area, ordered according to the date of file creation. The visualisation can be seen as one with similarities to 3DSoftVis, but where modules are paired off and separated. The purpose of this is to allow one-to-one comparisons to be made. The main intention of this is to compare a header file (.h) and an implementation file (.c) against each other, and so is particularly appropriate for a language using this structure such as C, or C++. In particular, use is made of the fact that header files and implementation files are usually named identically. This allows pairings to be generated automatically from the log file, without requiring further parsing of the actual files within the repository. This reduces the processing requirements significantly, allowing visualisations to be generated very rapidly. A header file will always be shown on the left side of the tower, and an implementation file on the right, to emphasise the differences between the two types of file.

Each tower is initially normalised to be the same height, and each release (central segment) given an equal proportion of this height. This provides an immediately accessible view that can be understood by a novice user. However, the visualisation supports the resizing of the central segments according to release date. Whereas ten file updates within a single release may appear intensive with the default view, if this release was shown to have taken ten times longer than the average release, a more accurate picture is obtained. A similar feature allows segments to represent a timeline, with segments representing days, weeks or months depending on the frequency of development. This is particularly useful when the 'symbolic name' feature within the log file is not used, as an automatic and accurate visualisation can still be created.

Each side segment of the tower represents a specific version of the file. The side segments have variable widths, and by default are used to show the change in file size. Although an RCS log file does not include the number of lines of code within a file, it does specify the number of lines that have been inserted, and the number deleted. As this information can be inaccurate depending on how the changes were made, the visualisation does not treat inserted lines and deleted lines separately, but rather relies on an aggregate of the two to show the overall difference.

The height of the side segment may be set in one of two ways. The heights may be allocated within the space of that release. For example, a file undergoing two revisions within one release would mean each side segment had a height that was half of the height allocated to the central release segment. This view is useful in providing a clear picture of how often each file changes per release, or within the project as a whole. Alternatively, the side segment heights can be allocated proportionally to the time of check-in relative to the release dates, so a version checked in very early after the previous release, and well before the next one would have a short height. This is a similar process to the resizing of the central segment. Implementation against header file comparisons should use this allocation method to gain an accurate picture, or use the timeline with the first method to display the same information.

Colour is used to show a further entity from the log. Possibly the most useful within a CVS log is the author that checked in the file. In this case, each side segment is filled with a colour that maps to a legend representing authors that has been extracted from the log files. The user of the visualisation can change these colours in order to highlight particular authors in a colour that is more easily recognised. Importantly, these colours are generated consistently, by allocating colours to authors in order of the time that they were first involved with the project. Therefore, the author who submitted the first file will be allocated the first colour, and this will remain throughout the lifetime of the project.

Finally, a timeline is also displayed. Although the log file does not contain specific release dates, the approximate date of a release can be derived from data contained within the individual entries, and it is this that is displayed. It is then possible to map individual towers onto the timeline, highlighting the evolution of the files within a release.

3.1. Animation

In Revision Towers, the user has partial control over the layout of the visualisation. Some changes that are made, such as changing the height of the central segment, could be disorientating. However, using animation to move smoothly between the different types of display allows context to be maintained.

Animation is also used to display the development of the repository. Using the timeline together with the main view allows a view of the repository at any point in time. This means the user is presented with a picture that is built up slowly, rather than being presented with an entire system instantly. This also means that files that are introduced to the system do not need to be shown from the beginning, but can be faded in at the suitable position. The position of these towers is determined at the start of the animation however, so that, for example, towers do not move to allocate space for a new tower. As unnecessary movement is eliminated, there is no danger of the user being distracted by animation unrelated to the data. The intention is to reduce the complexity of the animation, and so increase understanding. Chi et al [11] used a similar technique to show the evolution of web sites.

Although the individual frames build up to the finished picture, the animation presents information that may not otherwise be obvious by looking at the final picture alone. For example, figure 4 contains five frames from an animation of the evolution of part of the Allegro [12] graphics library over five releases. What is immediately obvious is how the project management has changed during this time. Initially, all check-ins were executed by A. The second frame shows that almost all check-ins were done by B. However, over time, new project members have taken over, to a greater or lesser extent. This is not immediately obvious from viewing the final picture in comparison to the individual frames.

An important part of any animation such as this is that the user has full control over the playback. Animations that can only be played in one direction mean that the animation will often have to be viewed many times in order to understand how a particular state was achieved. However, by allowing the animation to be played in both directions, once an interesting state is displayed, it is trivial to immediately rewind the animation a few frames to determine how this state came about. The animations can also be displayed at a number of speeds. This is important as it allows familiar parts of the visualisation to be skipped, whilst concentrating on the unfamiliar parts at a slower speed.

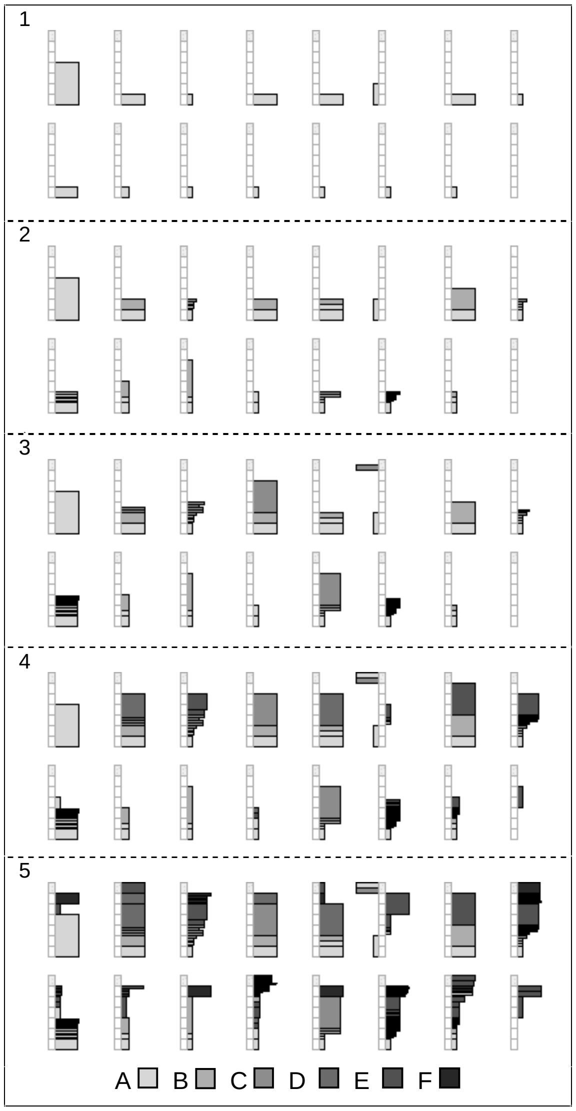

Figure 4 - Revision Towers animation

3.2. Layout

Within software visualisation, it is rare to consider the layout issues that arise with an evolving data set. The problem is not one of the layout of the current data, as many suitable algorithms exist. Rather, the issue is attempting to anticipate how the data might change at a future point in time. For example, a repository may gain several new files over a month. Visualisations created at the start of the month, and at the end of the month, should look similar in order to provide some context for the user of the visualisation, particularly if they use the visualisation on a regular basis. If the same object is placed at the top left of the screen when the visualisation is repeatedly created at different points in time, the user will assume that any similar object appearing in that position at a future date would refer to that same object. However, with many layout algorithms that will not be enforced. In particular, spring-based algorithms, which are suitable for many graph layout problems, respond particularly badly.

To avoid this problem, towers are laid out initially according to creation date, and then alphabetically, left to right and top to bottom. It is rare for files to be physically deleted from a repository, but rather for them to be marked as deleted. Therefore, providing space exists within the view, towers will not need to move from their original position, and so should be easily located within future visualisations. As hinted, the main problem arises when there is insufficient space to display the towers, and either the size of the towers must be reduced to allow more to be displayed, or the space must scroll to allow more to be shown. The ideal solution would depend very much on the type of repository being visualised - a very large repository might be better suited to scrolling, but a smaller one could work better with a reduced size. If the size of a tower is reduced, this has an impact on the layout of the towers, and recreates the original problem. The solution is to retain the layout used when the visualisation was last used by a particular user. Animation can then be used to morph the original layout with the new, reduced size, layout.

A common argument for using animation for a process such as this is that it will maintain context between the two views. Unless great care is taken over the two layouts this is unlikely to be the case, as there will be too many moving objects to be able to see the source and destination of all of them, and this would be necessary to completely maintain context. Any objects that had to cross to reach their new position would further confound the problem. Instead, in this case, the purpose of using animation rather than displaying the new layout directly is to offer the user of the visualisation some idea of the extent of the layout changes. If only a few objects change position, the user will be able to realise that the new layout is reasonably similar to the previous one, and therefore will be able to reuse the knowledge built up from examining the previous visualisations. However, if many objects change position, the user will realise that there has been a significant change to the layout, and possibly the repository, and so will also realise that previous assumptions that had been made about objects at specific locations are now likely to be unfounded.

Towers may also be rearranged to provide the user with an alternative view. Whilst the initial view provides consistency, it is not always the most appropriate layout.

Instead, it may be more useful to show the towers in order of the number of changes made, or grouped according to whether they were modified by a specific author. This can be based on either a specific release, or for the entire history. The location of the towers is then fixed by default during the playback of the animation.

4. Scenarios

Three scenarios have already been presented describing why an individual would have an interest in viewing the contents of a repository. In order for the visualisation to be considered useful, it should provide a solution to each of those scenarios. To demonstrate how the visualisation works, the tower shown in figure 3 will be used as an example.

This shows a typical tower from a relatively small project. The left side represents the header file, and the right side represents the implementation file. Three authors have worked on the file at some point, which is clear from the three colours used. More specifically, the first author (author A) appears to be no longer involved with these files, as no check-ins have been made by them after the first 6 releases. Author B has then taken over the bulk of the work, with some assistance from the author C.

Although this view provides a brief summary of the ownership of the file, closer examination allows more information to be derived and deduced. In particular, the role of author C is interesting. The author has only made changes to the implementation files with the header files unaffected. The file size after these changes (represented by the length of the segment) has also increased in at least two cases. (It may have increased in the top case also, but due to the limited screen resolution, this can not be confirmed from the picture alone. However, moving the mouse over the segment in question displays all details related to this segment, including the number of changes). Therefore, it is clear that no major functionality was introduced, as the header file remained constant during this time. As this is a C++ program, it is possible to assume that the header file would contain the class prototype, and therefore the prototype has not changed. So the likely changes would include bug fixes, adding comments, or rewriting the method implementations to improve them in some way.

Information such as this would be useful to both a programmer and a manager. A programmer would know that the class interface has changed only once in the last three releases. Depending on when the programmer was last involved in the project, they may wish to examine the source code more precisely in order to determine exactly what these changes were. However, rather than needing to examine changes for every file in the system, or even files related to their project area, they are immediately aware of which files have changed, and by how much. This enables them to concentrate on a smaller set of files.

Similarly, a manager may be able to draw some conclusions about the skills of authors B and C. If the pattern of these authors emerged elsewhere in the project also, where C only makes changes to the implementation files, it may be worth examining more closely the nature of changes being implemented. If it is apparent that the author is particularly proficient at optimising, for example, it may be worth encouraging this author to work in a more speed-critical part of the project. If, instead, it appears that C is implementing a lot of bug fixes, particularly as a result of coding by B, the manager may wish to encourage B to work on a less critical part of the project, or even leave the project altogether.

It is also possible to use the visualisation to get a good idea of how the project is evolving. For example, the state of the project just before the sixth release is worthy of investigation. Here, it is possible to see that whilst the size of the header file has increased, the size of the implementation file has decreased. This, at first glance, is an unusual situation. In this case, this was due to a section of code being cut from the file, which was then pasted into a new file. At the same time, some interface changes were made to the header file to take account of the section of code being moved. Therefore, the size of the header file increased, and the size of the implementation file decreased. This information may also be useful to a programmer returning to the project after time away, as it highlights the fact that code has been moved or deleted.

Further information that can be revealed from the tower is that the file is relatively significant. Initially there was a flurry of activity, which can be seen from the fact that there were several versions before the first release. Since that point, the files have been changed approximately once per release, and, in general, have been increasing in size at a constant rate. These frequent changes, and particularly those to the header file, indicate that it is often necessary to change the file in order to incorporate new functionality into the project. Therefore, it is probably worth spending time to understand the role of the file. The frequent changes to the header file would also be of interest to someone considering the use of the software within their own development. It may be that these changes regularly alter the interface, and so the developer may wish to find an alternative, more stable, solution to their requirements.

Finally, when multiple towers are displayed, animating the creation of these towers conveys further information.

Although a limited number of towers may be mapped simultaneously onto a timeline, this is not possible for all of them. Similarly, whilst towers may be structured in proportion to the date of release, this is not always appropriate. Therefore, viewing the animation allows the temporal details regarding the individual versions to be displayed. For example, the fact that only two changes occurred during releases two to four may be due to the fact that there was very little time between these. This information would again be useful to potential users of the software, in determining the level of activity within the project, and whether that activity is increasing or decreasing. Alternatively, the animation can highlight patterns within the repository that may not otherwise be seen, as figure 4 demonstrates. Here, it is clear to see how the project ownership has changed from author A in frame 1, to B in frame 2, and C in frame 3. Frames 4 and 5 show a more open ownership, with more authors responsible for check-ins. Developers preferring a more open project management would find this useful.

Clearly, it is not possible to draw concrete conclusions as to the behaviour of a large project by examining the small degree of data presented. However, this is not the intention of the visualisation, and it would be unwise to use it as such. It is the intention to reduce the amount of data that has to be considered before comprehending the latest version of a system, and this is done by highlighting potentially interesting areas within the repository, which should then be investigated more closely. It should also be considered that a single tower has been used to demonstrate the visualisation. In reality, it will normally be the case that several towers should be examined before drawing any real conclusions about the state of the project.

5. General Issues

Some outstanding issues remain with the visualisation however, which are influenced by the data set used. The programs RCS (and CVS), which provide the data used in the Revision Towers tool, are based around the line-based differencing tool ‘diff’. This means that changes to the file that only include whitespace, for example, blank lines, will affect the recorded number of lines changed within the log. Similarly, changes to comments will also be flagged as a change. Although necessary to record changes such as this within the repository so that they are not lost, a visualisation such as Revision Towers would benefit strongly from a version control tool that records the changes intelligently, rather than just line by line. The primary reason for this is that it would allow a much more accurate comparison between the header and implementation files. For example, whilst working with an implementation file, it may be that the associated header file is changed accidentally, for example, by adding a blank line at the end of the file. Checking in the files again would record this as a change, and so the visualisation would suggest that more work than was actually done took place.

A second issue regards the role of the ‘author’ within the log. For a small project with a few developers who have all been given full read/write access to the repository, the author as listed in the log is likely to be the author of the code that was added or modified. However, within a larger project, it is the developer with the responsibility for checking in work to that module that will be credited with the change within the log, sometimes with the actual author included as a comment within the log, or within a separate changes file. Therefore, when colour-coding the versions of a file according to the author, it is not clear without examining the management of a project in more detail whether this generally refers to the author making the change, or the project member with the responsibility for that module.

There are a number of potential solutions to this problem. The first is to parse the comments associated with a change intelligently, to try and determine the developer who wrote the code. It may be that a specific project has a standard way of generating comments, and so it may be possible to tailor the visualisation to work with a specific project. The second is to modify RCS (or CVS) to include the developer as a separate field within the check-in information, within the environment that it is to be used. This would be feasible if the visualisation was integrated into an online repository environment, such as SourceForge [2]. Both of these solutions would raise further issues within the visualisation, such as how multiple authors contributing to changes checked-in as a single version would be displayed.

6. Conclusions and Further Work

This paper has demonstrated that it is possible to use the data contained within a version control repository to create a useful and meaningful visualisation of the state of an open source project. A specific visualisation, Revision Towers, has been presented in order to demonstrate some of the relationships and conclusions that can be drawn from examining this data. A prototype tool has been developed to show how the visualisation would work in practice, and an example provided showing this working with data taken from the Allegro project.

The layout of the information within the log is an important part to the usefulness of the visualisation. By pairing related files together, provisional conclusions may be drawn rapidly about the state of the project, and how individual files are evolving. Size and colour are used to display further information about file size and ownership.

Animation is used by the visualisation in order to present this information. Although animation has often been suggested as a useful means of displaying time-related data, there are few implementations where this has actually been done. Some guidelines for the use of animation in this way have been presented, and an example of how the use of animation is beneficial over an equivalent static picture has been provided.

Finally, the importance of the consistency of a visualisation over time has been raised. The fact that a visualisation should be consistent is well known, but this concept has now been extended so that the visualisation should remain consistent when provided with an evolved data set. This is a little considered area within software visualisation, and a possible means of achieving some consistency has been presented.

There is much that could be done to improve Revision Towers. Currently the visualisation is stand-alone, and has no integration with a version control system. Providing this integration would be valuable, and enhance the usability of the visualisation. For example, clicking on a version could check that version out of the repository. Selecting a release could do the same for all files within the project. Selecting two versions of a file could call a file comparison program, and so on. Similarly, the visualisation could be integrated with an online repository CVS view as an alternative to the text-based log that is often provided.

References

[1] A.Mockus, R.Fielding, J.Herbsleb, “A Case Study of Open Source Software Development: The Apache Server”, *Proceedings of the 22nd International Conference on Software Engineering*, Limerick, Ireland, 2000. pp263-272

[2] SourceForge web site. http://www.sourceforge.net

[3] T. Ball, J.-M. Kim, A. A. Porter, H. P. Siy. “If your version control system could talk...” *Proceedings of the Workshop on Process Modeling and Empirical Studies of Software Evolution*, Boston, MA, 1997

[4] S.G.Eick, T.L.Graves, A.F.Karr, J.S.Marron, A.Mockus, “Does Code Decay? Assessing the Evidence from Change Management Data”. *IEEE Transactions on Software Engineering*, Vol.27, No.1, Jan. 2001, pp1-12

[5] CS-RCS website .http://www.componentsoftware.com

[6] RCE website. http://wwwipd.ira.uka.de/~RCE/

[7] C.Knight, M.Munro, “Comprehension with[in] Virtual Environment Visualisations”, *Proceedings of the IEEE 7th International Workshop on Program Comprehension*, Pittsburgh, PA, 1999. pp4-11

[8] H.Koike, H-C Chu, “VRCS: Integrating Version Control and Module Management using Interactive Three-Dimensional Graphics”. *Proceedings of 1997 IEEE Symposium on Visual Languages*, Capri, Italy, 1997. pp170-175

[9] P.Young, "Visualising Software in Cyberspace", PhD thesis, Dept. of Computer Science, University of Durham, 1999

[10] C.Riva, “Visualizing Software Release Histories: The Use of Color and Third Dimension”, Masters thesis, Information Systems Institute, Technical University of Vienna, 1998

[11] E. Chi, J. Mackinlay, P. Pirolli, R. Gossweiler, S. Card. "Visualizing the Evolution of Web Ecologies". In *Proceedings of ACM Conference on Human Factors in Computing Systems*, Los Angeles, CA, 1998. pp400-407

[12] Allegro website. http://alleg.sourceforge.net