International Conference on Software Maintenance (ICSM '99), Aug 30-Sep 3, Oxford, England, 1999.

VISUALIZING SOFTWARE RELEASE HISTORIES: THE USE OF COLOR AND THIRD DIMENSION

Harald Gall, Mehdi Jazayeri,
Distributed Systems Group,
Technical University of Vienna,
Argentinierstr. 8/184-1, A-1040 Wien, Austria
{gall, jazayeri}@infosys.tuwien.ac.at

Claudio Riva
Nokia Research Center,
Software Technology Laboratory,
P.O. Box 407, FIN-00045 Helsinki, Finland
claudio.riva@nokia.com

Abstract

The data regarding the components of a software system consists of a large amount of information such as version history, number of lines, defect density, and complexity measures. The ability to quickly grasp a comprehensive view of the evolution and dependencies of such information is the key to making informed decisions about future developments of the system. Managers usually make such decision based only on expert judgement.

For help in making such decisions, we can turn to the evolution history of large software systems, which contain a wealth of hidden information. Traditionally, this information is passed on through anecdotes without any supporting analytical data. This paper reports on our attempts to make such information more concrete through information visualization techniques. We present a three-dimensional visual representation for examining a system's software release history. The structure of the system is displayed by 2-D or 3-D graphs. The historical information is displayed by using time as the third dimension. Colors are used for displaying module properties and their historical changes in the system.

A supporting software tool enables not only visualization but also navigation in the 3-D space to change the viewpoint, to browse system information, to find interesting patterns and to discover previously unknown relationships among system components.

Keywords: software release history, information visualization, software evolution, third dimension, color, system architecture, software maintenance.

1 INTRODUCTION

Understanding a large software system requires the use of abstract representations to cope with the amount of details present at the source code level. Decomposition into modules and software measurement are technologies that can simplify this task. Software measures such as code size, complexity measures and defect density are a concise and informative representation of the properties of the source code. The evaluation and comparison of them can lead to the discovery of anomalous behaviors or unknown dependencies. Such information is valuable for software engineers, analysts and managers.

In evolving through releases, developers modify system structure and properties. Tracking their historical evolution allows the programmers to document the events that influenced the system evolution and to identify the reasons for structural problems [11].

The collection of the data regarding a software system (structural information and measures) and its evolution produces a large amount of data. A challenging issue in their examination is how to extract the useful information hidden within the data.

Information visualization [13] [27] is a recent and emerging technology which attempts to adopt graphics techniques for visualizing abstract entities that have no concrete shapes. Information visualization has proven to be an effective solution for understanding and uncovering information embedded in large data sets.

This paper reports on our attempts to develop new three-dimensional visualization techniques and to apply them to study the evolution of an industrial software system as represented by a database of 20 versions of the system released over a two-year period. Previous works identified some key metrics to measure and initially used simple two-dimensional graphs to plot them [11]. Previous work also concentrated on discovering patterns in the evolution of the software [12]. Our work investigates the use of information visualization and in particular of two graphical technologies: color and third dimension. Such techniques have rarely been applied to the field of software understanding. Related techniques have used primarily two dimensions and have focused on history visualization through animation [5].

Our 3-D representation provides the simultaneous visualization of *system structure, historical evolution* and *software properties*. The system structure is displayed by 2-D or 3-D graphs. The third dimension is used to represent time. Colors are used to represent a particular property (e.g. version number, code size, etc.).

The contributions of this paper are:

- the use of information visualization to study the release history of a software system.
- the combination of 3-dimensional views and color to the study of abstract software structures.
- presentation of a particular industrial system's visualization to show the kinds of analyses and observations that are possible from such release histories.

Our technique is supported by a tool written in Java, using VRML to render and navigate 3-D spaces. The tool is published on the web so that users can check the ideas reported in this paper. The link is: http://www.infosys.tuwien.ac.at/visualization.

The paper is structured as follows. In Section 2 we discuss the related work. Section 3 presents the issues in studying software release histories. Section 4 describes our industrial case study for which the visualization techniques were developed. Section 5 presents our visualization technique and some of the results of its application to the case study. Section 6 presents the benefits and challenges of the approach and Section 7 concludes the paper by drawing some conclusions and giving some ideas for future work.

This work was supported by the European Commission within the ESPRIT Framework IV project ARES (Architectural Reasoning for Embedded Systems). The technique was developed in conjunction with an industrial case study as part of the project ARES. The present publication only allows us to show black and white graphs. A copy of this paper, with color graphics may be found at: www.infosys.tuwien.ac.at/~riva/Docs/bibl.

2 RELATED WORK

Computer graphics has developed sophisticated techniques for visualizing real-world objects on the computer screen. Information visualization adopts such technologies to examine and comprehend masses of information contained in databases, documents and other data sources. Information visualization can represent data in a compact way and can uncover useful and interesting patterns in underlying data. A general overview of these technologies can be found in Keim's tutorials [15] [16] [17]. Some relevant approaches in the area of software visualization are mentioned here.

Most software visualization [30] work has concentrated on code-level visualization or on performance visualization. This is not the focus of our work. Our focus is on structural and architectural levels of large systems. In this area, Eick et al. have developed a set of tools for visualizing several classes of data [6] [7] [5]. In particular, SeeSys [1] is a visualization system for displaying the statistics associated with code that is divided hierarchically into subsystems, directories and files. Animation is used to show the historical changes of the system.

The graphical technique adopted in SeeSys is suggested by the work of Johnson and Shneiderman on visualizing hierarchical data using Treemaps [28] [14]. The approach uses the screen-filling method for displaying the attribute values of hierarchy components. The color of the region can provide an additional attribute.

Software visualization using three dimensional display is a current research area. Koike developed a 3-D framework to visualize the execution of two parallel/concurrent computer systems [18]. Koike et al. also proposed a 3-D framework for both version control and module management [20]. Ware et al. investigate how to visualize object oriented software in three dimensions [34]. An evaluation of the 3-D approach has been conducted by Ware et al. [34] and by Chu et al. [4]. Feijs et al. [9] have proposed a 3-D visualization for the analysis of software architectures.

3-D displays are also investigated for algorithm animation by Stasko et al. [29] and by Brown et al. [2]. 3-D graphics for displaying hierarchical structures have been developed by Card, Mackinlay and Robertson [25]. Their work provides ConeTree which is a technology for displaying the hierarchical objects in 3-D space. Improvements of ConeTree are described by Carrière and Kazman [3] and by Koike [19].

3 THE SOFTWARE RELEASE HISTORY

A software system evolves through successive releases. Release after release the system is modified adding new functionality, removing others or changing the existing ones. The *system release* is a mechanism for the controlled implementation of changes. Changes become an integral part of the system when the new release is delivered and no other changes are allowed after the delivery. Successive changes are integrated in future releases. Therefore, the system at a generic release may consist of software components (such as source files, documentation, configuration files) which have been added or changed in different moments of the life-cycle. The system release can identify uniquely a well-defined implementation (source code, documentation, configuration files) of the system. The Software Release History is a method to track the historical changes of system components. The next Section 3.1 describes the abstract model that the method uses. Section 3.2 describes the method itself.

3.1 System abstraction

Understanding a large software system by examining its source code is an arduous task. The amount of details present in the source code would overwhelm the examiner. Abstract representations of the source code simplify human comprehension. The Software Release History uses an abstract representation that is generated in two steps: system decomposition and measuring software attributes.

System decomposition

The software system is decomposed in two elements: *modules* and *relationships* between modules. They represent the abstract structure of the system.

A module is the basic software component of the decomposition. It has an internal structure that is invisible to the other modules and an external interface which exports its functionality. It can eventually be decomposed into further modules or be directly implemented in software. The main concern of this abstraction is reducing the complexity and moving to a more abstract level than code. According to [21] and [32] when dealing with large systems, system evolution is driven more by changes in functionality than by low-level tinkering with code. Changes are reflected by added, removed or modified modules. Therefore, system-level evaluation should be based on modules rather than on code. This is also consistent with the infrastructure trends of telecommunication companies described in [33]. The trend is to use a modular structuring which encapsulates a feature in a module instead of distributing its functionality over the system. This modular structure makes it easier to add, remove and manage the functionality of the system. The case study considered in this paper has a layered architecture. The system is organized hierarchically with each layer providing service to the layer above. Each layer hides its implementation from the higher layers.

Relationships between modules are related to the method used to decompose the system. The structure obtained by the decomposition depends on the method used: "part of" decomposition produces a hierarchical structure, "uses" decomposition produces more complicated graphs. The case study has a layered architecture, so modules can import only blocks from lower layers. This decomposition produces a hierarchical structure.

Measuring software attributes

The decomposition step extracts the structural information from the source code. Other useful information can be extracted. The theory of software measurement describes the technologies for mapping software code to a set of attributes. The objective is to create a concise and abstract representation of a piece of source code in terms of a set of attributes. The attributes can embody some characteristics of quality or software property that are under investigation. For example, we can calculate the size in terms of lines of code, complexity using complexity measures, age in terms of version numbers, error proneness in terms of defect density and other measures [10]. A set of attributes can be measured for each module. These attributes are a representation of the code that is associated with the module; they are its properties. In this way, each element of the structure owns a set of properties whose values are the ones of attributes measured on its associated source code.

This process of abstraction is exemplified in Fig. 1. The abstract representation constitutes three modules interconnected according to a decomposition model. Each module has three properties: the name, the number of lines of code and the version of the module.

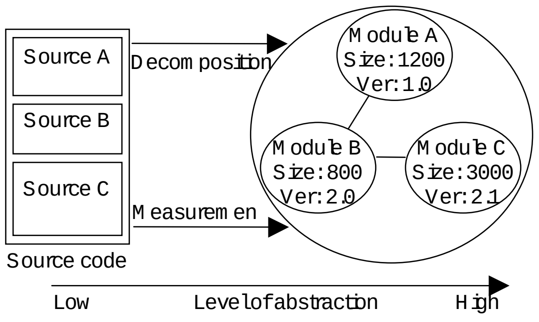

Fig. 1 The abstraction process

3.2 The Software Release History

The innovative part of our approach, called Software Release History, is the use of the history of the system to evaluate it. The historical evolution of a software system is determined by release deliveries: each new release incorporates changes to the code of the system. Software Release History tracks this evolution.

According to the abstract model just developed, system evolution is detected by modifications in two elements: structure and software attributes. The system structure is modified when modules are added to or removed from the system or the relationship among modules is changed. The software attributes can change when the underlying code is modified. For example, if a new module is added to implement a new functionality, the new structure is obtained from the old one plus the new module. As another example, if the code of a module is re-written, the module's version number increases. Therefore, the abstract representation modifies its information as the system evolves. For each system release an abstract representation can be extracted from the source code.

The Software Release History captures the evolution of the abstract representation. It is composed of three entities:

- Time: this coordinate is expressed in release sequence number, RSN. The advantages are that the system is defined precisely at times of releases and that the release intervals correspond to well-defined units in the system life-history.
- Structure: the system is decomposed into modules and relationships according to the abstract model.
- Attributes: a set of attributes, such as version number, size, complexity, defect density are measured on the source code associated with system modules.

The time line is discrete and a value of it identifies a unique release of the system. For each element of the time line the abstract representation (structure and values of attributes) is extracted from the code. The information is stored in a database. The database is called the *Software Release Database*.

4 THE CASE STUDY

We have developed the Software Release History approach for the analysis of large evolving systems. The technique was developed in conjunction with a representative case study. The case study examined is a product family of Telecommunication Switching Systems (TSS). The evaluation of the present work only concerns the software of the system. The system is mainly developed using the C programming language.

The system was under continuous development and several re-designs of the software and hardware have been done. The first delivery of the TSS in 1980 had 100,000 LOC (lines of code). In 1990 a typical TSS product consists of 3 MLOC (million lines of code). Today the size is about 13 MLOC. These figures already show some of the issues of such huge systems.

The next sections describe how the available data of the case study are mapped to the three entities of the software release history.

4.1 Time

The system releases are progressively numbered with increasing values. This numeration is called release sequence number, RSN. The case study contains twenty different releases which represent releases over 21 months. Eight of these releases are major releases (releases 1 through 6 and releases 19 and 20) and twelve are minor releases (releases 7 through 18). The time intervals between major releases (1-3 months) are normally larger than between minor releases (15-30 days).

4.2 The structure

The software architecture is organized as a *layered system*. Layered systems are organized hierarchically with each layer providing service to the layer above it. The structure is a tree hierarchy with four levels. The top level is the *system level*. It is based on the *subsystem level* (second level), *module level* (third level) and *program level* (fourth level).

Each level consists of one or more elements. Each element of a certain level is connected to one element of the higher level. The elements in each level are named corresponding to the names of the levels: *subsystems*, *modules* and *programs*[1].

Programs are the smallest logical unit of this structure. They represent the algorithms. The algorithms of a *program* are implemented in source files. So one or more source files are associated with a *program* element. The tree hierarchy limits the visibility of the algorithms contained in the program level.

4.3 Attributes

The case study is based on version numbers as a representative of software attributes. The numeration is based on the RSN.

[1] To avoid confusion the names of the structure elements of the case study are written in italic: *subsystem*, *modules* and *program*. When they are written in normal characters they refer to the usual meaning of those words.

The version number of this *system* element is the RSN of the release to which it belongs.

Subsystems and *modules* do not have any numeration because they are abstract entities. Their version numbers are the same as the *system* element to which they belong.

The version number of each *program* element is the RSN of the release where it had the latest change. For example, *program* A changes its implementation at releases 1, 2 and 5, so its version numbers are the sequence < 1 2 2 2 5 5 >. In releases 2, 3 and 4 the implementation is the same so the version number is 2 because the last change happened in release 2. The version number can assume also a null value. This value is used to indicate that the *program* element is not present in that particular system release.

We have adopted the same system structure for all the releases. Such common structure is the most generic one, so that the structures of each release can be fitted to the common one. In this way the same structure is used for all the releases and it eases comparisons.

The advantage of this approach is that all the elements are numbered with the same notation that captures the essential information of the version number: when the changes happen and what is the implementation of the *program* element. In this way it is possible to make comparisons because all the entities are measured in the same scale. Moreover the use of a null version number allows us storing also structural information about *programs*.

4.4 The Database

The data regarding the software release history are extracted directly from the source code. During compile time preprocessors extract and store the information in a database. For each release stored, the database contains entries for elements at the *system*, *subsystem*, *module* and *program* level. Two valuable information are present: relations between various elements of the system (e.g. *module* c consists of *programs* 1, 2, 3) and the version numbers of the programs (e.g. *program* 5 has version number 2.3).

The database considered in the present work is populated with 20 releases of the software product. Each release contains 8 *subsystems*, 47 to 50 *modules* and about 1500 to 2300 *programs*. The version numbers of the programs have been converted to the number described in Section 4.3 by specific programs.

5 VISUALIZING THE SOFTWARE RELEASE HISTORY

5.1 Overview of approach

The Software Release History is constituted of three entities: time, structure of the system, measures of attributes. These entities are visualized using one three-dimensional diagram (3-D diagram). The coordinates are called *x*, *y*, and *z*. In a 3-D diagram the entities are displayed in this way:

- Time: the coordinate *z* stores the time information. This time coordinate is expressed in release sequence number (RSN).
- Structure: for each RSN the system has an associated structure. The structure is displayed using 2-D or 3-D graphs and it is spatially positioned along the coordinate *z* at the value of its own RSN.
- Attributes: each diagram can display one attribute at a time. Each structure is identified by the RSN and has its own set of attribute values. These values are shown using colors. The values are mapped to a color scale and so each color of the diagram represents a value. Fig. 7 provides an example of color scale used for the case study. Each value of RSN is mapped to a color. For example, color black is associated with 0 which means that a *program* element is not present in the system structure; red color is associated with 1 which means that the *program* element has version number 1.

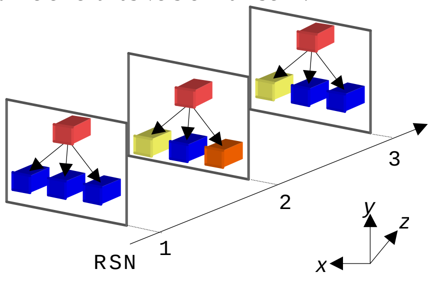

Figure 2 The 3-D diagram.

Figure 2 shows a graphical representation of this approach. For each system release (1, 2 and 3) a graph shows the structure of the system. The graphs can be of two types, 2-D or 3-D. In the figure, a tree structure is used as an example. Each colored block is associated with a module of the abstract decomposition. The colors are used to visualize a software attribute. Each color is mapped to an attribute value through a color scale. This visual approach provides an immediate representation of the data. Using the color scale it is possible to relate the colors to the values, and so immediate comparisons can be done. (Please see web site for color graphics.)

5.2 Visualizing one system release

The hierarchical structure of the software in the case study requires the implementation of specific solutions for hierarchical structures. 2-D and 3-D tree graphs may be used to visualize hierarchical structures. 3-D graphs are implemented using Cone Tree [25]. 2-D graphs use the same technologies of Cone Tree in two dimensions. Fig. 3 shows the whole structure of one system release of the case study. The large volume of data regarding one release is visualized in one view. The three dimensional layout allows the packing of more data onto the screen without overloading it. The 3-D layout also allows navigating virtually within the system structure. System architects or engineers who need to navigate the system structure can take advantage of this 3-D layout. Fig. 4 shows the details of several *subsystems*. By navigating through the space, the user can focus on interesting details, choosing the best view and can have a virtual perception of the visualized structures. Other features are:

- Visual retrieval of data: to extract simple data (like name or version number) the user can click with a mouse on the element and retrieve its attributes.
- Global and local view: the whole architecture of one system release is visualized in one view at any level of abstraction. In this way the viewer has all the data at his/her disposal. When something interesting or anomalous is found, the user can zoom in on the problem without losing the context.
- Visual aid: when understanding abstract information our minds create visual representations to simplify the process. Visualization can provide it for the viewer and relieves his/her mind. In this way imagination and creativity are free to address new ideas. This is one of the advantages claimed for information visualization.

- Ease of use: 3-D displays and navigation are easily understood because they take advantage of the human's innate perception of space. The training time for a new user can be remarkably short [4].

5.3 Visualizing large volumes of data

3-D and 2-D graphs, even if displayed in a 3-D space, have the limitations that they become incomprehensible when visualizing large sets of data. Specific solutions have to be adopted for large volumes of data. The percentage bar is a graphical object that offers a compact representation of a group of elements. It visualizes an attribute of the modules in terms of percentages. It is composed of a set of colored blocks. Each block has two properties: relative size and color. The relative size is proportional to the percentage of modules that have the same value of the attribute. The color depends on the value of the attribute through the color scale. The graphical object is shown in Fig. 6. It is helpful for visualizing the attributes of hundreds of modules. The percentages and their associated values are grasped quickly. For quantitative comparisons size is the most effective perceptual data encoding variable [8].

5.4 Visualizing the history

The third dimension allows us to visualize historical information together with the system structure. The *z* coordinate is expressed in RSN. For each value of RSN the associated graph is created by extracting the data from the database and displaying it at its own position. Fig. 5 shows 10 releases of the case study. Each release is visualized with a 2-D tree. The tree visualizes the structure of a *subsystem* of the case study. Fig. 6 shows the same data of Fig. 5 using percentage bars.

Two useful representations can be obtained by rotating and compacting the 3-D diagram. Fig. 8 and Fig. 9 show two examples. They visualize the attribute values of the *program* elements belonging to a *module* element of the case study. The former figure uses percentage bars, the latter visualizes separately each *program* element (each column is a *program*). Such 2-D representations are powerful tools for examining the historical information. They make such historical information accessible to the analysts who need to trace back the evolution of the system. The main features are:

- The whole history is visualized in a compact view so that the viewer can compare different releases.
- The main changes in the evolution are easy to detect because they are represented by "large" changes in color.
- The distribution of an attribute is visually perceived by region filling and colors.
- It is possible to focus on a single release and then to move the examination immediately on the next one without changing context.

Examining these two pictures, useful observations can arise about the evolution of the *module*. We first describe Fig. 8 and then Fig. 9

The graph of Fig. 8 visualizes the percentages of *programs* which have the same attribute value. This representation documents the events that influenced the whole *module*. At release 1 (first row) all *programs* have the same version number, that is, the version number 1 (red color). At release 2 (second row) 96% of *programs* have version 2 (pink color) and the rest (4%) have version number 1 (red color). This means a majority of *programs* (96%) have changed their implementation and therefore the *module* has been extensively modified. There are several reasons that can motivate such behavior. Motivations can be found by direct inspection of the code or of the *module*'s documentation. For example, the *module* could have been restructured, or, to add a new functionality, programmers had to modify many parts of it. Whatever the cause, such a view is a cause of concern and should trigger a closer examination.

Still in Fig. 8, at release 8 the *module* has its last major modification. In fact at release 8 (eighth row) the large dark green zone shows that many *programs* have changed their implementation. Then from release 8 until release 20 many of these *programs* maintain their version number: for each release from release 8 to 20 the green color zones are the biggest ones. Between release 8 and 20 a small fraction of *programs* change their version number: in Fig. 8 this is reported by the regions on the right colored with blue, purple and dark green. The *programs* change at releases 9, 10, 11, 12, 14, 15, 17, 19. It is clear that the programs have stabilized. An inspection of the module's documentation revealed that a considerable effort has been spent between release 7 and 8 to update a functionality of the module.

The representation of Fig. 9 visualizes separately the evolution of each *program* of the *module*. It provides the same information of Fig. 8 and allows studying the evolution of single elements.

To identify the small percentage of *programs* which do not change at release 2, Fig. 9 has to be used. It shows that on the second row (release 2) the first *program* from left and the sixth *program* from right have a red color, i.e. version number 1. In this way this representation allows identifying the outliers.

Looking at Fig. 9 it is possible to identify the small amount of *programs* which do not maintain their implementation between release 8 and release 20. In particular, there are five *programs* whose behavior is anomalous. They are the second, the third, the fifth, the seventh and the eighth *program* from the right. All these *programs* have a common behavior: they change their implementation at release 8, maintain it for several releases and then they change it at later times. The modification made at release 8 is successful for many *programs*. For the identified *programs* the modification is not so successful because after several releases they need other changes. Several reasons could give rise to this situation: the modification at release 8 has not been correctly implemented or the detected *programs* have particular problems and need to be restructured. Only a direct inspection of the source code or of the documentation can verify these hypotheses.

5.5 Observations on the case study

Fig. 10 shows all the *subsystems* of the case study, labeled with letters from A to H. For each *subsystem* its *modules* are visualized using the 2-D visualization with percentage bars. For each *subsystem* the *modules* are numbered from left to right. The *subsystem* A contains two rows of *modules*, the first row contains the *modules* from 1 to 8, the second line the *modules* from 9 to 16. Several observations can be made by examining the pictures. The purpose is to show how the qualitative observations can be extracted. Quantitative observations can be easily obtained by the database. The purpose of the technique is to provide the means for quickly examining the data.

Modules A-16, H-2, H-3 have been removed in the first system releases. In fact for each release from 3 to 20, the percentage bar is black. This means that all the *programs* of the *module* are absent.

In *subsystem* A the majority of *modules* is characterized by a low growth rate. High growth rates are localized in the first releases, after which *modules* reach a stable size. The sizes of the black regions are an indication of growth rate. High changing rates can be identified by a zone with high color changes, instead of low changing rate that are identified by plain color zones. The *modules* 1, 2, 5, 7, 9, 11, 13 and 14 are characterized by big plain color regions, therefore their changing rate is quite low. Release 5 is associated with yellow color. At release 5 *modules* 1, 2, 3, 4, 7, 9, 10, 11, 12 and 15 contain a large yellow zone. This means that at release 5 many *programs* have been modified. This may be due to a large modification of the whole *subsystem*. *Module* 5 is characterized by the fact that the majority of its *programs* are added at release 3 (orange zone) and then many of them do not change their version number any more (the big orange zone extends until the last release). This reveals that the programmers made appropriate choices when they added the *programs* because these *programs* do not require any modifications throughout 20 releases.

Subsystem D contains *modules* with the highest changing rate. Almost all *programs* of *modules* 1 and 2 change their version number in each release. This fact is shown by the horizontal colored lines which span almost all the *module* size. *Module* 3 has a high growth rate but the changing rate is modest. New programs are added and many of them are maintained for all the releases. The anomalous behavior of this *module* has also been detected in the work of Gall et al. using statistical analysis [11]. This *subsystem* is a candidate for reengineering.

In *subsystem* C *module* 1 has a stable size and it is mainly constituted of *programs* with version number 1 (big red zone). *Module* 2 increases in size at release 13 (dark purple color) when many *programs* are added. New *programs* are added with version number 13 and then many of them maintain it until release 20. In fact, many of these added *programs* do not change their version number. In all the *modules* we can identify a set of programs which never change their implementation (red vertical line on the left). This means that they are stable for all the releases.

Fig. 10 is also useful to detect relationships between *modules* of different *subsystems*. We can provide some examples.

Modules A-5, A-7, B-9, C-2, F-2 and G-3 are characterized by a considerable change that is maintained for many releases. This is perceived visually by the large monochromatic zones that start at a specific release.

Modules D-3, F-2, F-4 and F-6 have a high growth rate for all the releases. In fact the black region is constantly diminishing.

In *modules* A-1, A-2, B-3, B-4, B-5, B-7, B-8, B-10, C-1, C-3, E-1 and G-2 we can identify a common visual pattern. The pattern is made of a large red zone on the left and of multi-colored zones on the right. This pattern means that the *modules* are composed of a considerable number of *programs* at release 1 and of *programs* which often change their implementation, i.e. the *modules* contain a stable group of *programs* at release 1 and a variable group which are changing to implement the new functionality.

Such visual pattern can be extended to include *modules* A-5, A-7, A-8, A-13, A-14, B-9, B-10, F-7, G-1, G-5, G-6. This pattern is made of monochromatic colored bars on the left and of multi-colored zones on the right. The meaning behind this pattern is that these *modules* contain stable *programs* implemented in the first releases and variable *programs* which are often changing their implementation.

6 EVALUATION OF THE APPROACH

We have presented our attempt to use three-dimensional and color visualization of software release histories. For such a novel approach, it is difficult to give a definitive assessment of the technique. Certainly, the engineers and managers who viewed the visualizations were able to quickly grasp the main ideas and patterns in the software evolution. But there are also a number of questions regarding the generality of the approach and in how well we have implemented the approach. In the absence of quantitative assessments, in this section we provide some subjective evaluations. We first describe and summarize the main advantages of the visual approach to software history analysis and then discuss some of the difficulties and challenges in the approach.

6.1 Advantages

6.1.1 Visualization

The main advantage of the approach is that it provides simultaneous visualization of the three entities (structure, attribute and time) in one view.

The abstract structure of the system at a generic release can be visualized with both 3-D and 2-D graphs. The graphical notation adopted (cubes and spheres for modules and lines for relationships) is intuitive. The graphical representation adopted for the structures is close to the mental projections that humans' mind makes of abstract structures. For example, the hierarchical structure of the case study is visualized with trees that are the natural way we think of a hierarchical structure. The process of thinking of abstract information through mental images is alleviated because the process is carried out automatically by a graphical system of visualization.

Each module of the structure is associated with a part of the software system. From a piece of source code several measures can be calculated such as version number, size, complexity. These values are the properties of the module. The approach allows us to visualize with colors the attribute values of modules.

The third dimension is used as time coordinate to display the historical evolution of system structure and attributes of modules.

6.1.2 3-D visualization

The visual representation uses a three dimensional display. The main advantage of the third dimension is that it is possible to pack more information in one view. Other advantages of the third dimension are:

- Three coordinates allow us to visualize both system structure and historical evolution of the system. The viewer can perceive both structural and historical information looking at one view. The viewer can also select the best perspective when focusing only on one information.
- Graphical objects have a three dimensional layout. The visual effect is pleasing for human viewers. Rendering, shading and 3-D perspectives are the technologies that can simulate the reality to which our mind is accustomed. These graphical technologies can produce representations that are more natural to the human eye.
- The viewer can navigate virtually in the graphical representation. This allows choosing the best view of interest for the data instead of requesting the data from the database or changing the parameters of the graphs. New representations can be easily obtained just by rotating, zooming, projecting or moving the graph.

6.1.3 Coloring by measures

A module's attribute captures essential information about its associated source code. Visualizing how the attribute is distributed over the modules is useful for identifying anomalous behaviors or abnormal values of the attribute. The approach uses colors for visualizing such distribution. The basic idea is coloring the system elements by their attribute's values. This is achieved by mapping the range of values to a color scale. The advantages of this approach are summarized below:

- Attribute values are visualized together with system structure. In this way the values are in the same context of the elements to which they belong.
- Numerical values are mapped to colors, so that the process of comparing the data changes from being a cognitive task (i.e. numerical comparison) to being a perceptive task (i.e. visual comparison). Changes, commonalties, differences and patterns can be visually detected rather quickly.

6.1.4 Compact representation

In visualizing the attribute values of a large set of modules, the main problem is how to give an informative representation that would visualize large amount of details in a comprehensible manner. Our approach uses percentage bars. These graphical objects provide a visual representation of percentages instead of values. In this way it is possible to visualize a set of values of arbitrary size in a standard layout without being overwhelmed by the volume of data [1].

6.1.5 Visualization of History

Our approach allows us to visualize and compare multiple system releases. Two representations that are useful for studying the historical evolution are available. Representations that use percentage bars report the major modifications that influenced the history of a module and help discover anomalous behaviors. Representations that do not use percentage bars visualize the historical evolution of each module's components and expose the outliers precisely.

6.1.6 3-D navigation

Instead of accessing the database directly to look for the data and to extract them, it is possible to navigate through its visualization and to have access to the desired data immediately. We have developed a visualization system that supports both functionality. The viewer can navigate the 3-D space for examining the structure and can retrieve the data by pointing the mouse on an object and clicking on it. For example, Fig. 3 visualizes the whole structure of one system release that is contained in the database. The lowest level consists of almost 2300 elements. The viewer has a global view of all the database in just one picture. Then he/she can focus on a particular subsystem or can extract the values of the modules just by pointing.

6.2 Challenges

Applying information visualization techniques to the software engineering domain is a challenging task. We set out to evaluate the use of the third dimension and color in this visualization. Therefore, one way to evaluate the success of our approach is to ask whether tables or simple 2D charts wouldn't be able to present the same information, even more clearly.

3-D layouts have been investigated in the software engineering community but they are considered with suspicion by tool vendors. Our experience indeed supports such skepticism in the sense that 3-D layouts can be ineffective in displaying complex graphs. By keeping the complexity of the graphs low, however, 3-D layouts can be used to encode more information than 2-D layouts. Indeed, in the 3-D diagram we have developed, historical information and architectural views are put together. A 2-D layout would have difficulty encoding both these views. This motivated us to use the 3-D layout.

The use of colors is the second issue we investigated. As before, the question is whether a monochromatic color scale wouldn't provide the same results or if the cost of using color is worthwhile. In our experience, after a short period of familiarization with the types of visualizations, users find color plates to be easy to use and much more intuitive then textual tables.

One of the challenges in our scheme is the choice of the color scale. Our purpose was to maximize the number of distinct colors along the scale. We did not find a specific recipe in the literature to solve this problem [22] . Therefore, we decided to use the standard rainbow scale. The rainbow scale covers all the hues of the rainbow at different intensity. However, this scale didn't satisfy our purpose and we decided to manually customize the scale for our 21 color scale. This is a specific solution for our case study where the releases are limited to 20. The color scale could be a limitation for the adoption of this technique to other cases.

7 CONCLUSIONS AND FUTURE WORK

The evolution history of large software systems contains valuable information for software engineers and managers. We have developed a visual representation that can make this information concrete and applied it to the evolution of a TSS system.

Our work shows that information visualization technologies can be applied with relative ease to the analysis of software evolution and to uncover valuable information. We have only scratched the surface in this area. We believe that these kinds of analyses can be of significant help in software engineering management. A fruitful area of research is to develop new visualization techniques and supporting analysis.

The 3-D visualization allows the viewer to visually perceive the abstract information of a software system. Navigation makes it possible to change the viewpoint and to quickly extract the data. Such visualization has several potential applications. For example, it may be used as a user-interface for a software configuration system in which the user navigates the visual space in order to find, query, or open the desired module or release.

The 2-D color maps are a powerful instrument for examining the historical evolution of the software systems. Colors can effectively highlight the main events of the system evolution and can clearly reveal unstable areas of the system as regions of many color changes (because the same color means that the implementation does not change).

Visualization appears to be a good solution for extracting hidden information in large software databases. Software companies store thousands of documents about their applications and are unable to extract useful advice from them. Providing these databases with appropriate visual interfaces, software engineers can get more insights about their applications by bringing software "to life."

Below, we list some areas of possible future work:

- Extension of the database of the case study, in order to include more detailed information about the modules. For each module it could include its size in LOC, defect density and complexity measures. Then, by setting up the mapping of the color scale, the visual representation can be used also to visualize these attributes.
- Use of nonhierarchical structures. The case study has a peculiar hierarchical structure. For this reason, the present

work has adopted only tree graphs (2-D and 3-D). Future work could be directed to investigating other structural representations that could be used for non-hierarchical architectures.

- Verification of observations about the case study that were produced based on the visualizations. These observations could be verified by using additional information such as bug reports, enhancement requests, design changes, or by direct inspection of the source code.
- Automatic detection of change patterns. A related work [12] addressed the problem of identifying module dependencies by detecting common patterns in the Software Release History. Future work could investigate the possibility of automating the task of detecting patterns. In particular, two types of capability should be supported: visualization of known patterns and automated detection of patterns requested by users.

8 References

[1] M. J. Baker and S. G. Eick, Visualizing Software Systems, *AT&T Bell Laboratories*, 1994.

[2] M. H. Brown and A. M. Najork, Algorithm Animation Using 3D Interactive Graphics, *Proceedings of the 1993 ACM Symposium on User Interface Software and Technology*, Nov. 1993, pp. 93-100.

[3] J. Carrière and R. Kazman, Interacting with Huge Hierarchies: Beyond Cone Trees, *Proceedings of Information Visualization '95*, Atlanta, Georgia, Oct, 1995, pp. 74-81.

[4] H. Chu and H. Koike, How does 3D Visualization Work in Software Engineering ?: Empirical Study of a 3D Version/Module Visualization System, *International Conference Software Engineering 98 (ICSE 98)*, 1998.

[5] S. G. Eick, J. L. Steffen and E. E. Sumner Jr, Seesoft - A Tool For Visualizing Line Oriented Software Statistics, *IEEE Transactions on Software Engineering*, Vol. 18, No. 11, Nov 1992, pp. 957-968.

[6] S. G. Eick and D. E. Fyock, *Visualizing corporate data*, AT&T Technical Journal, January/February 1996, pp. 74-76.

[7] Thomas A. Ball and Stephen G. Eick. *Software visualization in the large*, IEEE Computer, April 1996, pp. 33-43.

[8] S. G. Eick. *Engineering perceptually effective visualizations for abstract data*, Scientific Visualization Overviews, Methodologies and Techniques. IEEE Computer Science Press, February 1997, pp. 191-210.

[9] L. Feijs and R. de Jong, 3D Visualization of Software Architectures, *Communications of the ACM*, Vol. 41, No. 12, December, 1998, pp. 73-78.

[10] N. E. Fenton and S. L. Pfleeger, *Software Metrics: A Rigorous & Pratical Approach*, International Thomson Computer Press, Second Edition, 1996

[11] H. Gall, M. Jazayeri, R. Klösch, and G. Trausmuth, Software evolution observations based on product release history, *International Conference on Software maintenance (ICSM '97)*, Bari, Italy, pp. 160-166, September 1997.

[12] H. Gall, K. Hajek, M. Jazayeri, Detection of Logical Coupling Based on Product Release History, *International Conference on Software maintenance (ICSM '98)*, Washington, DC, 1998.

[13] Gershon, Nahum, S. G. Eick, Visualization's new track: making sense of information, *IEEE Spectrum*, 32(11), Nov. 1995, pp. 38-56.

[14] B. Johnson, Visualizing Hierarchical and Categorical Data, Ph.D. Thesis, Department of Computer Science, University of Maryland, 1993.

[15] D. A. Keim, Databases and Visualization, Tutorial, *Proceedings of ACM SIGMOD International Conference. On Management of Data*, Montreal, Canada, 1996 URL: http://www.informatik.uni-muenchen.de/~keim.

[16] D. A. Keim, Visual Techniques for Exploring Databases, Invited Tutorial, *International Conference on Knowledge Discovery in Databases (KDD'97)*, Newport Beach, CA, 1997, URL: http://www.informatik.uni-muenchen.de/~keim.

[17] D. A. Keim, Visual Data Mining, Tutorial, *International Conference on Very Large Databases (VLDB'97)*, Athens, Greece, 1997 URL: http://www.informatik.uni-muenchen.de/~keim.

[18] Koike H., The role of another spatial dimension in software visualization, *ACM Transactions on Information Systems*, 11(3), July 1993, pp. 266-286.

[19] H. Koike, Hirotaka Yoshihara: Fractal Approaches for Visualizing Huge Hierarchies, *Proceedings of the 1993 IEEE Symposium on Visual Languages (VL'93)*, 1993, pp.55-60.

[20] Hideki Koike, Hui-Chu Chu: VRCS: Integrating Version Control and Module Management using Interactive Three-Dimensional Graphics, *Proceedings of 1997 IEEE Symposium on Visual Languages (VL'97)*, 1997, pp.170-175.

[21] Lehman M.M. and Belady L.A., *Program evolution*, Academic Press, London and New York, 1985.

[22] H. Levkowitz and G. T. Herman, Color Scales for Image Data, *IEEE Computer Graphics and Applications*, Vol. 12 1, 1992, pp. 72-80.

[23] Parnas D. L., Clements P. C. and Weiss D. M., *The Modular Structure of Complex Systems*, IEEE Transactions on Software Engineering, March 1985, Vol. SE-11 No. 3, pp. 259-266, also published *in Proceeding of 7th International Conference on Software Engineering*, March 1984, pp. 408-417.

[24] Parnas D.L., Software Aging, *Proceeding of ICSE 16*, Sorento, Italy, May 1994, pp.279-287.

[25] G. G. Robertson, J. M. Mackinlay and S. K. Card, Cone Trees: Animated 3D Visualizations of Hierarchical Information, *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI '91)*, ACM Press, 1991, pp. 189-194.

[26] A. Sarma, Introduction to SDL-92, *Computer Networks and ISDN Systems*, Elservier Science Publishers, Vol. 28, No. 12, 1996, pp. 1602-1615.

[27] S. Card, J. MacKinlay, B. Shneiderman, Readings in Information Visualization : Using Vision to Think, Morgan Kaufman Publishers, 1999.

[28] B. Shneiderman, Tree Visualization with Treemaps: A 2D Space Filling Approach, *ACM Transactions on Graphics*, Vol. 11, No. 1, 1992, pp. 92-99.

[29] J. T. Stasko and J. F. Wehrli, Three-Dimensional Computation Visualization, *Proceedings of the 1993 Symposium on Visual Languages*, Aug. 1993, pp. 100-107.

[30] J. T. Stasko, J. B. Domingue, M. H. Brown and B. A. Price, *Software Visualization*, MIT Press, 1998.

[32] Turski W. M., Reference Model for Smooth Growth of Software Systems, *IEEE Transactions on Software Engineering*, Vol. 22, No. 8, Aug. 1996, pp. 599-600.

[33] F. J. van der Linden and Müller J. K., Creating Architecture with Building Blocks, *IEEE Software*, Nov. 1995, pp. 51-60

[34] C. Ware, D. Hui and G. Franck, Visualizing Object Oriented Software in Three Dimensions, *Conference Proceedings of CASCON' 93*, Toronto, Ontario, Canada, October, 1993, pp. 612-620.

[34] C. Ware and G. Franck, Viewing a Graph in a Virtual Reality Display is Three Times as Good as 2D Diagram, *Proceedings of the 10th IEEE Symposium on Visual Languages*, 1994, pp. 182-183.

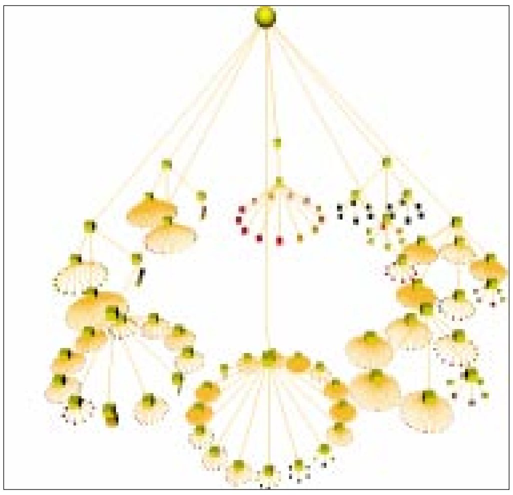
Fig. 3 3-D visualization of the structure of the case study.

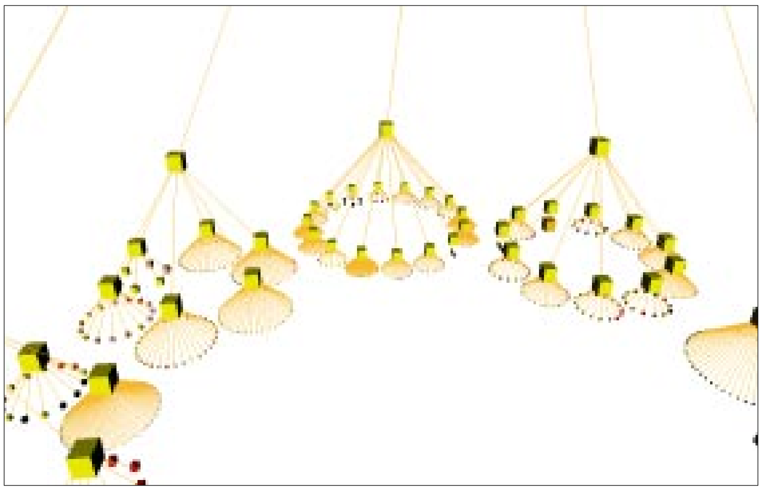
Fig. 4 Navigation: zooming on *subsystems*

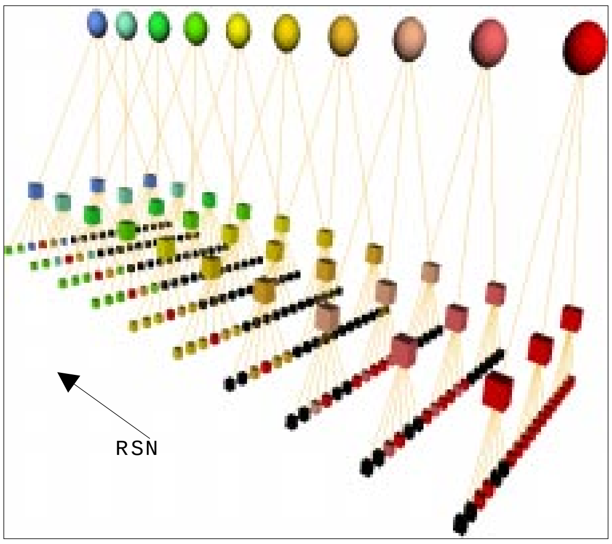

Fig. 5 Visualizing the history.

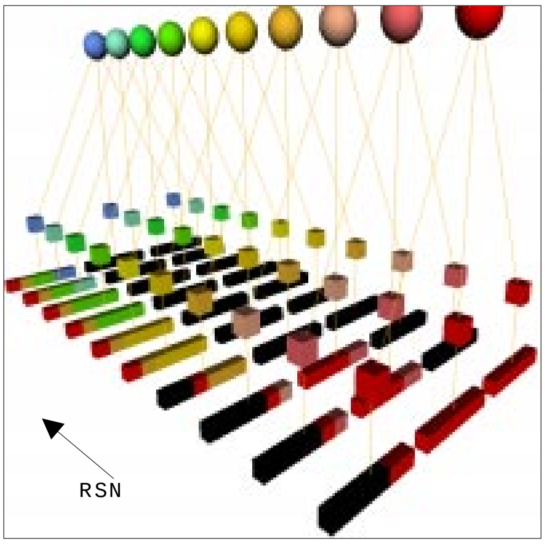

Fig. 6 Visualizing the history using percentage bars

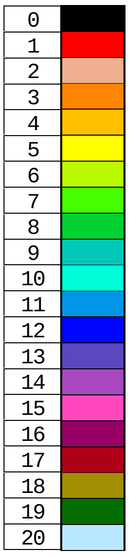

Fig. 7 Color Scale

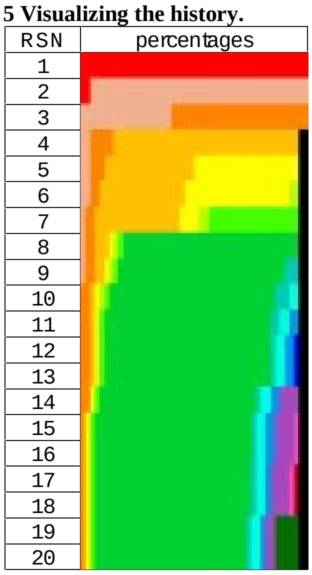

Fig. 8 History with percentages.

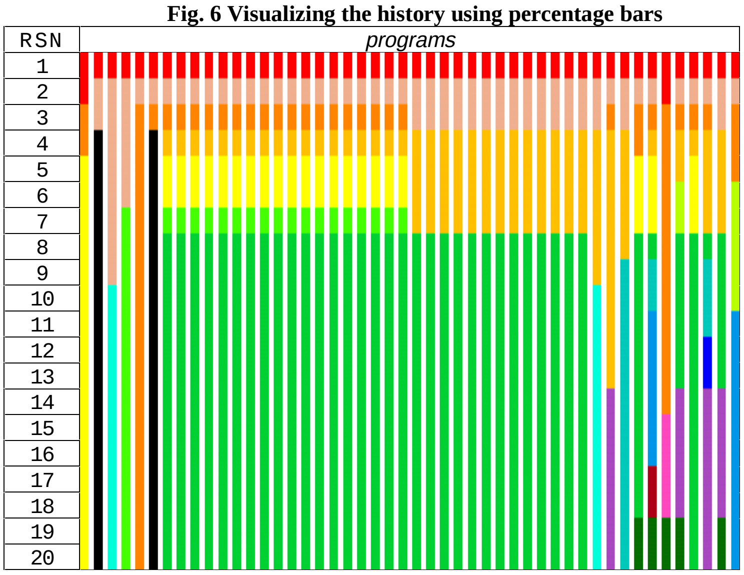

Fig. 9 History displaying *program* elements.

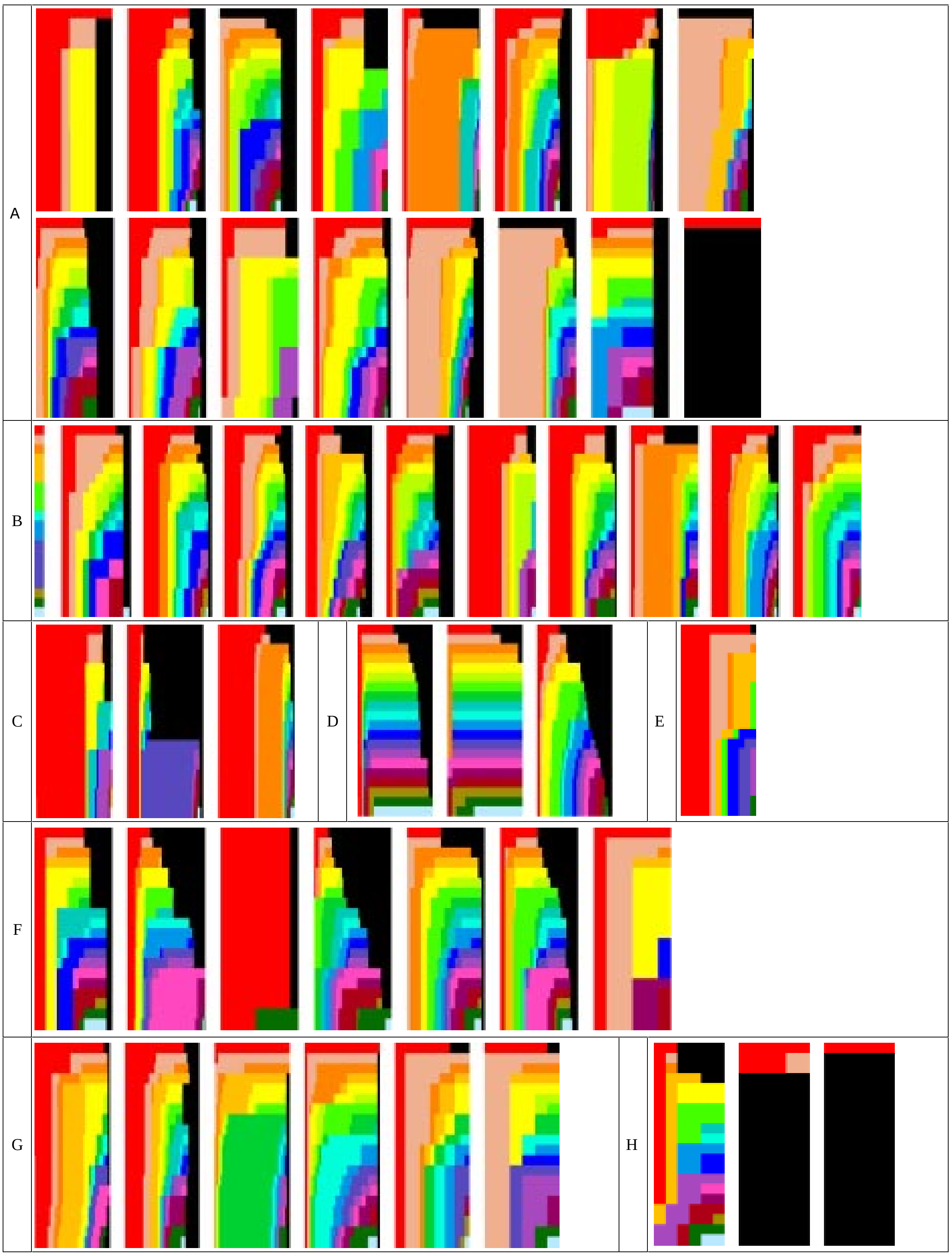

Fig. 10 2-D visualization of the case study (RSN as in Fig. 8).