Visual Assessment Techniques for Component-Based Framework Evolution

Lucian Voinea
Technische Universiteit Eindhoven
L.Voinea@tue.nl

Alexandru Telea
Technische Universiteit Eindhoven
A.C.Telea@win.tue.nl

Abstract

Many component models have been proposed to address the challenge of reducing software development time and costs. Such models often offer similar functionality. We study how developers can assess the relative strengths and weaknesses of such models, based on information about the component development process, and the component framework performance. We use a visual tool to gather such information and support our approach. We validate our approach on ROBOCOP, a complex component framework used in the industry.

1 Introduction

Component based software engineering is regarded as a promising approach towards reducing the software development time and costs. While it has proven to be successful in many application domains such as office and distributed internet-based applications, the component-based approach toward development is still to be validated in the area of dependable systems, which have special requirements on the quality attributes. Möller et al. [4] have elaborated a set of such requirements, and classified a number of existing component models according to their conformance to the set. However, as the number of component models increases, a new challenge arises: how to discriminate among models that satisfy the same set of requirements such that the best suited one is selected as development base for a given system. Using the evaluation methodology proposed in [4], one can easily reach the conclusion that for example the Koala [6], and PECOS [9] component models offer similar benefits from the point of view of testability, resource utilization, and availability of a computational model. When these are the only important nonfunctional requirements for the component model, the selection of the best suited model can be further refined on two directions. First, one may try to establish which model fits better with the software development process that will be further used during the project's lifecycle. Second, one may try to select that component model which increases the granularity of the requirement conformance assessment, in order to spot better performance margins.

In this paper we propose a novel approach to implement these two specialized directions of assessment. We segregate component models based on history recordings about component structure evolution, on the one hand, and framework performance, on the other hand. We implement the above directions by using CVSscan [7] [8], a tool for visual assessment of source code evolution.

The remainder of this paper is as follows. In Section 2, we describe the component information data modeling and the visual mappings that CVSscan uses to encode source code evolution. In Sections 3,4, and 5, we present three mechanisms for distinguishing among component models based on information about component structure and framework state evolution, as follows: assessment of the component development process (Section 3); assessment of change propagation from framework to components (Section 4); and assessment of framework performance (Section 5).We illustrate each of the above techniques with examples from the ROBOCOP component framework. Finally, in Section 6 we conclude with a number of recommendations to maximize the efficiency of the techniques we propose, and we give a short view on our future direction of research.

2 Data modeling and visual mappings

CVSscan [8] is a visual tool developed to support program and process understanding in large software projects. It uses the information stored in CVS source code repositories to build graphical representations of the evolution of a given software system (or subsystem) along hundreds of versions. CVSscan is based on three main mechanisms, as follows. First, it uses a dense, pixel-filling display where every few pixels convey information, via their position and color, about a source code line. Second, CVSscan offers a rich set of navigation mechanisms, which allows users to follow the evolution in time of a given aspect of the source code, ranging from a single line to a whole file. Third, CVSscan includes a classical text editor whose navigation (scrolling) is correlated with the pixel-filling display. For a detailed overview of the implementation and functionality of the above techniques, as well as the functionality of CVSscan, we refer to [7] and [8].

CVSscan reveals information about both the structure of a program and the process it undergoes from creation until a given moment. To understand this, we next briefly present the program data model and visual mappings in use by CVSscan.

CVSscan is based on the data model used by the CVS version management system. Given some source file, CVS stores several *versions* of this file and allows, via a comparison technique similar to UNIX's `diff`, to detect which source code *lines* are modified, inserted, deleted, or stay unchanged between consecutive versions. Finally, for every file version, we extract (basic) syntactic information, such as function headers and bodies, comments, and include statements. CVSscan displays this information in various ways, controlled by several visual *mappings*. A visual mapping is, essentially, a way of assigning position, shape, and color attributes to (parts of) the data model described above. Since this model is centered on file versions, source code lines, and syntactic structures, our visual mappings will emphasize these concepts as well. CVSscan offers two such visual mappings: *file-based* and *line-based*. The file-based mapping essentially maps the time (version number) to the horizontal X axis and the line number in file to the vertical Y axis. Every code line gets then drawn as a pixel line, colored by various attributes. Every version appears thus as a vertical stripe. In Figure 1 (top), we use the file-based mapping to show 65 versions of a C source file colored by line status: green denotes constant, yellow modified, red modified by deletion, and light blue modified by insertion, lines. The line-based mapping maps also time to the horizontal axis. However, the Y coordinate of a code line is now computed globally, over all versions, rather than per version. In the line-based mapping, a given code line has the *same* Y coordinate in all versions it appears, and Y coordinates respect local line orderings in all file versions. Identity between code lines in different versions is evaluated with the `diff` tool. The exact algorithm for computing the line-based mapping is described in [8].

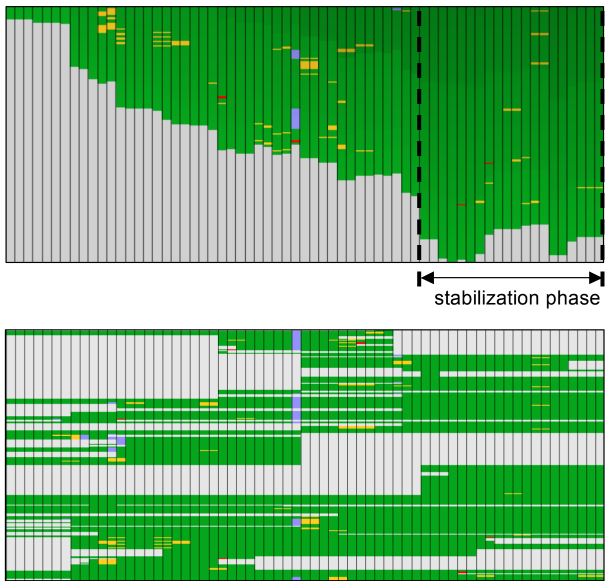

Figure 1: File-based (top) and line-based (bottom) mappings in CVSscan

The file-based mapping offers a simple, yet powerful overview of the complete evolution of a given file or, since in our concrete usage developers would code one component per file, of a given component. Individual code lines are not visible in Figure 1. However, this image manages to depict around 20000 lines of code spread over 65 versions. Colors indicate the type, frequency, and distribution of changes, and version sizes (on the Y axis) show the code evolution and eventual stabilization phase the project enters. In contrast, the line-based mapping allows identifying stable code fragments and quickly spotting significant inserted or deleted code as large gray areas.

2.1 Component System Assessment

Given the data model and visual mappings supported by CVSscan, as described so far, the question comes how one could use the above techniques to assess component-based software. Specifically, we are interested in assessing the component development process and the performance of component frameworks, for a given component model. To use the CVSscan, or similar approaches, such as its predecessor tool SeeSoft [2] for component-based software source code, the following must hold:

- this software can be described by a *linear* structure, i.e. an ordered sequence of *elements*. These elements can be source code lines, in the simplest case. However, any alternative data model can be used, e.g. syntactic blocks (component scopes, procedures, structures, or other namespaces).

- information about the structure *evolution* is available. In the simplest case of using a line-based structure, this implies being able to determine at which moment in time did a certain line appear in the source code, and respectively disappear from it. In other words, mechanisms must be available for extracting and comparing code structures.

Assessing a component development process is, in this respect, a conceptually simple task. Indeed, virtually all component models have a clear mapping between components and their source code. Additionally, many software projects use version control management (VCM) systems, similar to CVS, which hold history records about code evolution and offer mechanisms for code comparison.

Assessing the component framework performance using evolution visualization tools such as CVSscan is, however, a more difficult task. It depends on the ability of such tools to represent the framework state by a linear structure and map performance measurements and metrics on structure evolution patterns. It may be the case that each performance parameter requires its own line-based structure and visualization and has a specific set of evolution patterns.

We next present three mechanisms for assessing a component model based on visualizations built with the CVSscan tool. We illustrate these mechanisms with real life examples. All these examples have been obtained from the ROBOCOP framework for component based software development [3].

2.2 Context of assessment

To understand the assessment results, we first sketch the context in which we used the CVSscan tool. ROBOCOP was developed on a period of four years by an international consortium of several industry and academic partners. Its focus is on providing a generic, flexible, and resource-efficient set of mechanisms and tools for implementing, composing, deploying, and monitoring component-based software applications with a focus on high volume embedded appliances such as mobile phones, set-top boxes, and embedded controllers. In this respect, the ROBOCOP component model is similar to Koala [6], Rubus [1] and PECOS [9]. ROBOCOP's core, the run-time environment (RRE), acts as a virtual machine providing application-specific component library management, component instantiation and destruction, and inter-component communication and control interfaces.

Component interfaces are described in the ROBOCOP Interface Description Language (RIDL). Just as in most component frameworks, this language can be translated, or compiled, to generate classical C skeleton code. Developers can next add component implementation code to this skeleton and compile it on a variety of platforms.

For both the RRE and the component libraries, several tens of versions exist, developed by the different consortium partners during its four year history. These versions emerged either due to changes in the framework and/or component interfaces or due to implementation changes when porting these to different hardware platforms.

Finally, we mention that CVSscan was built *and* used in assessing the ROBOCOP framework by different people than those who work on the component framework itself. We had thus the added difficulty of understanding a third-party large software system mainly through the perspective of our CVSscan visualization tool.

3 Assessment of component development

The choice of the component development process is an important issue when selecting the base component model on which a software system should be built. Investigating history recordings can give an indication about the effort required to have a minimal working component for testing purposes, the effort for modifying component interface and/or implementation, and the overhead related to maintaining framework compliance during development. Questions such as "how hard is to build a minimal component?" or "how much component code was changed when some framework interface got added?" can be directly targeted by CVSscan. Once we have been able to answer several such questions concerning the *past* evolution of our system, we attempt to extrapolate the results to the future. Most partners of the ROBOCOP endeavor have expressed their high interest in being able to answer, even if only qualitatively, the above questions.

Figure 2 depicts several CVSscan visualizations of a real-life ROBOCOP component evolution along 15 versions. We use a line-based layout, so the evolution of each source code line can be easily followed along the horizontal time axis. The upper part (*Content view*) shows three snapshots of the entire system evolution from the perspective of three different versions:

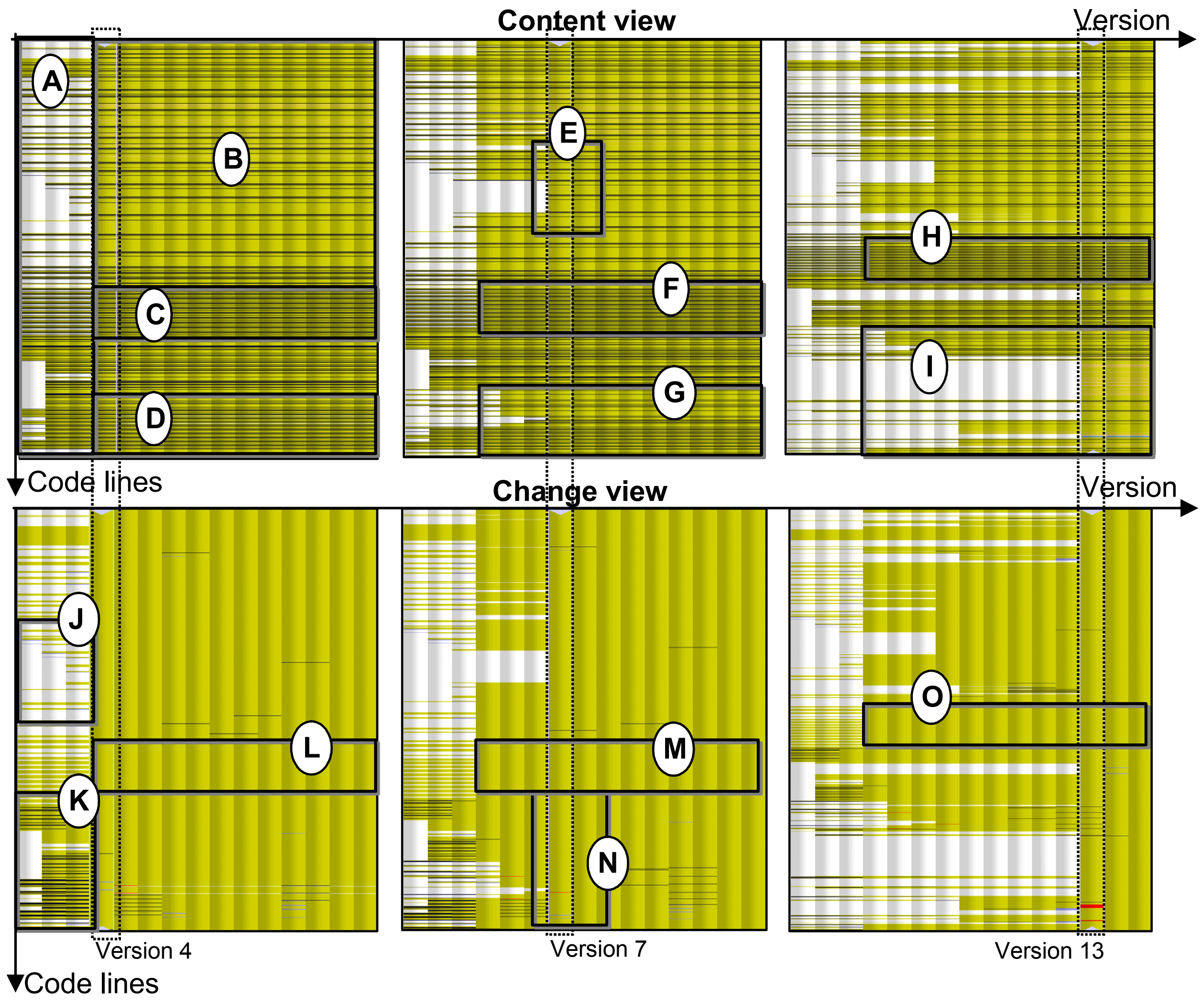

Figure 2: Visualization of ROBOCOP component development process

version 4, version 7 and version 13. Color encodes line content as follows: black (darker) indicates function headers, yellow (lighter) shows function bodies. The three snapshots in the lower part of Figure 2 (*Change view*) correspond to the ones in the *Content view*, but use a different color encoding: black (darker) indicates lines that do not change from one version to the next one, while yellow (lighter) encodes changing lines. In all snapshots, white shows gaps in the code, i.e. places where code was deleted in a previous version or will be inserted in a future version. For a detailed explanation of the line-based layout construction, see [8].

We can use Figure 2 to understand our component development process. In the beginning (A), the developer tries to build a stable component interface. He edits the RIDL component description file and then generates a C source code skeleton using the RIDL compiler. In the first three versions of the considered use case, the developer does not add implementation code to the generated C skeleton, but tries to refine the RIDL interface description. This is apparent in Figure 2 (J), as no code is inserted in the visualized file besides the C function headers automatically generated by the RIDL compiler, i.e. the thin dark lines inserted in versions 2 and 3. Additionally, we see that the code is automatically generated, since the function headers in region K are changed. Generated function headers have an automatically created textual reference to the line number in the RIDL description file that corresponds to that function. Inserting new interface specifications causes the textual references to the interface specifications following after them to change, since the specification location in file changes.

As depicted in Figure 2, however, this can lead to misunderstandings, due to the current skeleton generation process, as follows. Every time the developer adds new specifications to the RIDL file, he needs to run it through the RIDL compiler in order to generate appropriate function headers, which cannot be created by hand. However, once developers start to manually fill in this generated skeleton, adding new interfaces can be a very cumbersome process. This is mainly caused by the RIDL compiler, which has no ability to merge the new skeleton information with the existing one, but generates the entire skeleton anew, discarding any hand-coded additions performed by the developer. To prevent this, users maintain copies of the old code, and every time new interfaces are added, the generated function headers are manually merged (i.e. by cut-and-paste) in the saved copy. In this way, however, textual inconsistencies are introduced in the existing function headers as they reference invalid locations in the RIDL specification file. This can be seen also from Figure 2, by comparing area (K) and (N). The introduction of new interfaces in version 2 and 3 (J) causes existing function headers to be updated (K). However, in version 7, the developer manually inserts automatically generated headers in the previous file version (E), which causes no update in the existing headers (N).

Figure 2 shows also the amount of effort required to have a minimal component running for testing purposes. Version 4 of the considered component was also the first functional one. We identify the main effort to achieve that as writing the code in the (B) area. From the *Change view*, we also see that the code required for a minimal component does not change in time except for some additional interface additions like the one highlighted in (E).

The evolution of the 'useful' component code can be noticed in the areas D, G and I, where most of the code inserted during component refinement (i.e. versions 4..15) goes away. Areas C, F, and H in the *Content view* and the corresponding regions L, M, and O in the *Change view* refer to empty function implementations, i.e. non implemented interfaces. These represent the code overhead required for compliance with the ROBOCOP framework and have no other useful purpose for the functionality of the component.

Summarizing the information in Figure 2, we conclude that developing ROBOCOP components requires a very careful code architecting. Subsequent interface changes are difficult to accommodate or lead to inconsistencies. Additionally, the effort required to have a minimal ROBOCOP component running may be relatively high, e.g. accounting for almost 50% of the developed code in the presented example. However, once we have this code, it does not change significantly during further refinement of the component. Eventually, ROBOCOP components might have to include pure 'overhead' functions (empty implementations) for compliance with the framework.

4 Assessment of change propagation

When component frameworks are not yet mature, it is often the case that new framework versions are not compatible with previous ones. In such cases, existing components need to be re-architected to various degrees in order to be supported by the new framework. The effort required for this step may be so high that migrating to a different, more mature, component framework or maintaining the old framework may be better alternatives. A detailed estimation of the transition cost at framework change is therefore of paramount importance. CVSscan can help make such estimations, based on history recordings for components that have been already re-architected to comply with new framework versions.

Figure 3 shows four CVSscan snapshots visualizing the evolution of the same component as the one discussed in Section 3, but including two additional versions that correspond to the transition from a first framework version to a second one. In the upper part of Figure 3, we use a file-based layout in conjunction with a version filter (see [8]) to depict the amount of code from one version that may be found in other versions. Color is used to encode change: yellow (light) areas are lines that did not change during development; black (dark) areas show line changes. By analyzing the upper left image in Figure 3, we infer that a lot of code had to be changed when passing from component version 16 to version 17. Additionally, only about 75% of the component implementation code from version 16 is found in version 17. Furthermore, the upper right image shows that new code had to be written for version 17 in addition to what was preserved from version 16. The amount of newly written code is almost 40% of what was preserved. Overall, about 50% of the component code in version 17 differs from the one in version 16: This signals a quite high effort to adapt components cope with changes in the ROBOCOP framework.

The lower part of Figure 3 uses a line-based layout together with a version filter to show what interfaces have been removed and what was inserted during re-architecting. Color is used to encode line content: black (darker) = function headers; yellow (lighter) =

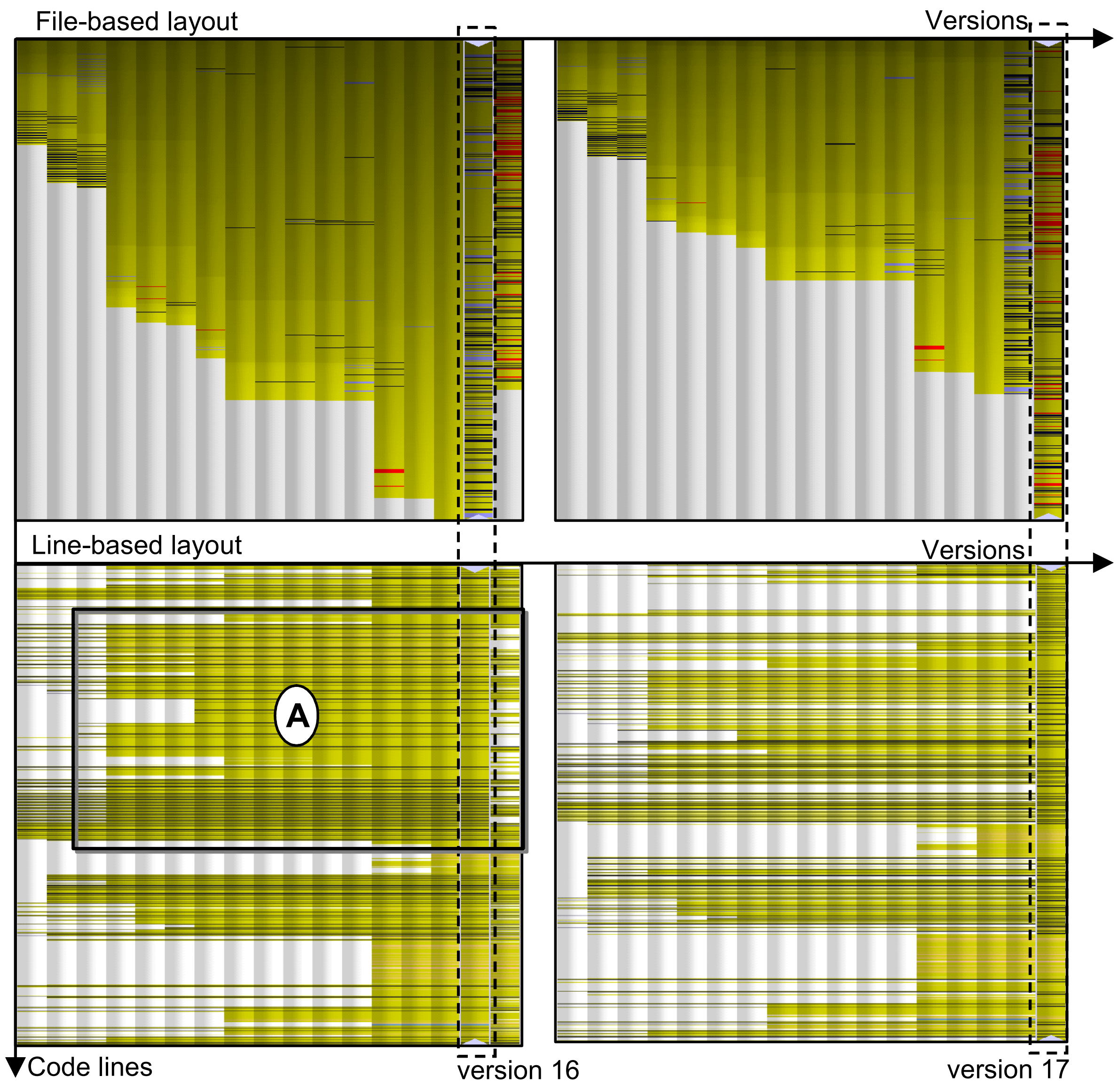

Figure 3: Migrating a component from ROBOCOP 1.0 to ROBOCOP 2.0

function bodies; white = deleted or inserted code, exactly as in Figure 2 (see Section 3) Correlating the lower left image (A) with the content evolution images from Figure 2, we can easily see that a major benefit of migrating to the new version 2.0 of the ROBOCOP framework was to decrease the number of mandatory interfaces that a component must implement to be compliant with the framework.

CVSscan allows also more in-depth analysis of the re-factoring a component passes through. This allows us, when browsing the code, to separate framework-induced code changes from those attempting to achieve a better design for the component itself. Figure 4 depicts such a case. We use here a line-based mapping to show the evolution of a component's code over 10 versions. The same color scheme as in Figure 2 and Figure 3 is used to display line changes. In Figure 4, we can quickly see an abrupt change performed in version 8. At a first look, we believe to see the addition of several component interfaces (blue stripes, A) and deletion of some of the existing ones (blue stripes, D). However, a closer analysis of the image (B) shows that all function declarations from version 7 are also found in version 8, and the actual code deletions refer only to parts of the implementation (function bodies). Moreover, the newly introduced functions (A) are not interface implementations. Using a classical code editor, we can easily investigate the declarations of the newly added functions and realize they do not have a ROBOCOP signature. Hence, the major re-factoring performed in version 8 does not change the component interface but is rather an attempt to *factor out* common implementation code (C) in order to make the component code more readable.

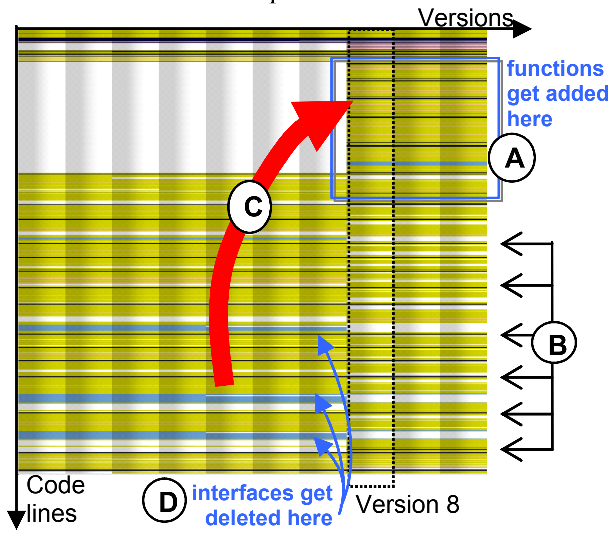

Figure 4: ROBOCOP component re-factoring: factorizing common functionality

5 Assessment of framework performance

An important decision factor in choosing a certain component framework as base for a software system is the set of quality attributes it offers. Sometimes it is the case that different component frameworks claim to meet the same set of nonfunctional requirements. For example, both Koala [6] and PECOS [9] allegedly offer similar benefits concerning testability, resource utilization, and availability of a computational model. In such cases, further evaluation of the frameworks is needed to assess their *performance* with respect to each quality attribute and identify relevant quality margins. To do this, we map the performance to graphical patterns in the evolution of a linear code structure and use CVSscan to visualize these patterns.

Figure 5 shows two CVSscan snapshots displaying the performance of the ROBOCOP System Integrity Manager (SIM) module running on a given terminal. The SIM module [5] is a part of the ROBOCOP framework in charge with remotely maintaining the system integrity of already deployed systems. One of the main activities of this module is to search for defective components and replace them with good-functioning ones according to a set of predefined policies. While this is one of the main features of the ROBOCOP framework, care has to be taken when building the replacement policies to avoid inconsistencies.

In Figure 5, we depict the evolution of a set of components running on a ROBOCOP terminal (client), managed by a SIM using two different replacement policies (*Policy 1* and *Policy 2*). The horizontal axis represents the time. The vertical axis represents the component set. In other words, whereas the examples presented in the previous sections displayed code lines versus time, we now display components versus time. Color encodes component change: yellow (lighter) = component changed by the SIM; green (darker) = component remains unchanged. The lower colored strip in each image shows the policy that generated a given change: light green (lighter) = *Policy 1*; blue (darker) = *Policy 2*.

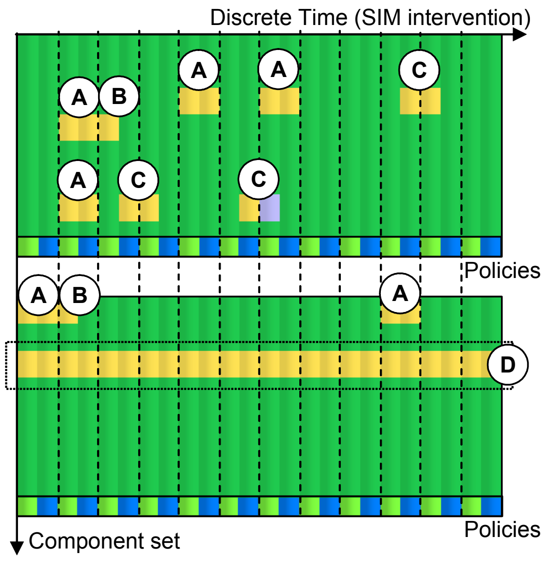

Figure 5: ROBOCOP system integrity manager: consistent(top) and inconsistent (bottom) policies

We can see different patterns emerging when the SIM replaces a faulty service: A = both policies generate a replacement; B = only *Policy 1* generates a replacement; C = *Policy 1* generates a replacement immediately after a replacement generated by *Policy 2*; D = both policies generate continuously replacements. While pattern A may indicate that *Policy 2* comes as an enhancement of *Policy 1*, as it corrects possible flaws not covered by the latter, pattern C may indicate a problem introduced by SIM, as every replacement performed by *Policy 2* is on a latter occasion overridden by *Policy 1*. Moreover, pattern D may indicate the presence of an inconsistency between the two policies, as the decision about changing a component is continuously retaken at every SIM intervention by both policies. Currently, ROBOCOP system developers are considering the use of CVSscan to help them while assessing the SIM module performance for a given set of policies.

6 Conclusions

We have presented several techniques for assessing several aspects of the component development process, using the CVSscan source code visualization tool. We validated our approach, both in terms of the CVSscan tool intuitiveness and the correctness of the concrete findings we obtained with it. For this, we analyzed ROBOCOP, a complex third party component framework, and discussed our findings and methodology with its developers. Two issues emerged. First, our concrete findings (e.g. "component interfaces stayed unchanged for the following 'x' versions") were confirmed by the developers as known, correct facts. Second and more interesting, some findings led us to hypotheses (e.g. "the patterns seen here denote code re-factoring") that were *new* for the developers, but, at further detailed code inspection, were found correct. In other words, our visualization lets one see known information and also discover new facts about a given component structure and implementation.

Our approach is relatively generic. CVSscan's line-based code model currently uses the UNIX-like `diff` provided by the CVS repository to compare code. Although this makes CVSscan applicable to any type of source code, a weak point is the accuracy of the `diff` operator used to compare component versions. The visualization accuracy depends on the heuristics behind this operator, which can lead to data misinterpretations, e.g. when too many changes occur between consecutive versions. However, CVSscan can also support a syntactic, instead of line-based, code model, where the central element is a component's interface, defined e.g. as a list of methods. Once a `diff` mechanism is implemented for such a model, e.g. by syntactic interface comparison via method signatures, CVSscan can be straightforwardly used. Instead of a code line, users would see now a method, or even a whole component. The CVSscan tool, as well as several example datasets, is available at:

www.win.tue.nl/~lvoinea/soft/CVSscan_setup.exe

So far, we only focused on the evolution of individual components. As future direction of research, we plan to extend our approach with higher-level overviews, such as whole-project evolution visualizations, to enable inter-component evolution analyses on entire systems. Our final aim is to integrate CVSscan in a toolset for component visualization and analysis and make it effectively and efficiently available to the component based software engineering process.

7 Acknowledgments

This research was part of the ITEA project *Space4U*, whose aim is to define a component based framework for the middleware layer of high volume embedded appliances (http://www.win.tue.nl/space4u)

8 References

[1] Articus Systems, *Rubus OS - reference manual*, 1996

[2] S.G. Eick, J.L. Steffen, E.E. Sumner, "Seesoft - A Tool For Visualizing Line Oriented Software Statistics", *IEEE Trans. on Software Engineering*, IEEE CS Press, Vol. 18, N. 11, Nov. 1992, pp. 957– 968

[3] ITEA, *ROBOCOP: Robust Open Component Based Software Architecture for Configurable Devices Project -- Framework concepts*. Public Document V1.0, May 2002, available online at: http://www.hitech-projects.com/euprojects/robocop/

[4] A. Möller, M. Åkerholm, J. Fredriksson, and M. Nolin, "Evaluation of Component Technologies with Respect to Industrial Requirements", *Proc. of the 30th EUROMICRO Conference (EUROMICRO'04)*, IEEE CS Press, Rennes (France), 31 August – 3 September 2004, pp. 56 – 63

[5] J. Muskens and M. Chaudron, "Integrity Management in Component Based Systems", *Proc. of the 30th EUROMICRO Conference (EUROMICRO'04)*, IEEE CS Press, Rennes (France), 31 August – 3 September 2004, pp. 611 – 619.

[6] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee, "The Koala Component Model for Consumer Electronics". *IEEE Transactions on Computers*, 33(3), 2000, p.p. 78– 85

[7] L. Voinea, A. Telea, M. Chaudron, "Version Centric Visualization of code Evolution", to appear in *Proc. of the IEEE Eurographics Symposium on Visualization (EUROVIS'05)*, IEEE CS Press, available online at http://www.win.tue.nl/~lvoinea/cvsv.pdf

[8] L. Voinea, A. Telea, J.J. van Wijk, "CVSscan: Visualization of code evolution", to appear in *Proc. of the ACM Symposium on software Visualization (SoftVis'05)*, ACM Press, NY, USA, available online at http://www.win.tue.nl/~lvoinea/cvss.pdf

[9] M Winter, T Genssler, "Components for Embedded Software – The Pecos Approach*", Proc. Second International Workshop on Composition Languages*, In conjunction with 16th European Conference on Object-Oriented Programming (ECOOP), Malaga (Spain), June 11, 2002