# On Feature Traceability in Object Oriented Programs

Giuliano Antoniol, Ettore Merlo
Computer Engineering Department
École Polytechnique
Montreal, Canada
antoniol@ieee.org,
ettore.merlo@polymtl.ca

Yann-Gaël Guéhéneuc, Houari Sahraoui
GEODES – Group of Open and Distributed Systems, Experimental Software Engineering
DIRO, University of Montreal, Canada
guehene@iro.umontreal.ca,
sahraouh@iro.umontreal.ca

## ABSTRACT

Open-source and industrial software systems often lack up-to-date documents on the implementation of user-observable functionalities. This lack of documents is particularly hindering for large systems. Moreover, as with any other software artifacts, user-observable functionalities evolve through software evolution activities. Evolution activities sometimes have undesired and unexpected side-effects on other functionalities, causing these to fail or to malfunction. In this position paper, we promote the idea that a traceability link between user-observable functionalities and constituents of a software architecture (classes, methods. . . implementing the functionalities) is essential to reduce the software evolution effort. We outline an approach to recover and to study the evolution of features—subsets of the constituents of a software architecture—responsible for a functionality.

## Keywords

Feature traceability during evolution.

## 1. INTRODUCTION

Evolution of implementation and evolution of functionalities characterises the life of any software system. Successful systems operate for decades and often outlive the hardware and operational environments for which they were conceived, designed, and developed originally. Source code of industrial systems often evolves without the documentation being updated because maintaining consistency and traceability between high-level abstractions, functionalities, and software constituents is costly and time-consuming. Documentation updates are also frequently neglected due to time and evolution pressure. High-level documentation, such as requirement or design documents, is often absent in open-source systems and no effort is pursued to provide traceability information. Yet, open-source systems are now common, *e.g.*, most ADSL routers, modems, and fire-walls, run customised versions of the open-source Linux operating system.


*TEFSE'05,* November $8^{th}$, 2005, Long Beach, California, USA.


Meanwhile, software evolution involves costly and tedious program comprehension activities to identify and to understand data structures, functions, methods, objects, classes, and—more generally—any high-level abstractions required by maintainers to perform the evolution. As of the year 2005, source code browsing is required during software evolution (and maintenance) because missing or obsolete documentation leads maintainers to rely on source code only. Source code browsing is resource consuming as the size and the complexity of systems increase.

An alternative to source code browsing is the automated recovery of *higher-level abstractions beyond those obtained by examining the system itself* [7], such as program features. We define a program feature as a micro-architecture, which is a subset of a program architecture grouping data structures, fields, classes, functions, and methods participating in the realisation of a user-observable functionality in a given scenario. The scenario details the conditions and steps of realisation of the functionality. For example, in a ADSL router, setting the user's name and password corresponds to one feature, as is adding a new fire-wall rule.

In this position paper, we support the idea to recover program features automatically, *i.e.*, to build traceability links between source code and user-observable functionalities, and to maintain traceability links among subsequent releases of a same feature as well as among different features of a given release. We define a traceability link as an association between a micro-architecture and a user-observable functionality. A traceability link can be used to highlight differences among features in a release or among releases for a given feature. Indeed, information on a feature evolution, along with the rationale for evolution (bug lists, user requests), is essential to identify fault-prone features. Information on feature interactions is essential to avoid undesired side-effects on unmodified features during evolution.

Recovering feature, *i.e.*, identifying micro-architectures responsible for a functionality, and maintaining traceability links among releases and among features require the development of several technologies. We need to resort to:

- Static and dynamic analyses.
- 2D and 3D visualisation.
- Information retrieval and computational linguistics.

Recently, several authors (see, for example, [3, 8]) addressed the problem of identifying feature in object-oriented systems. We concur with previous contributions that these

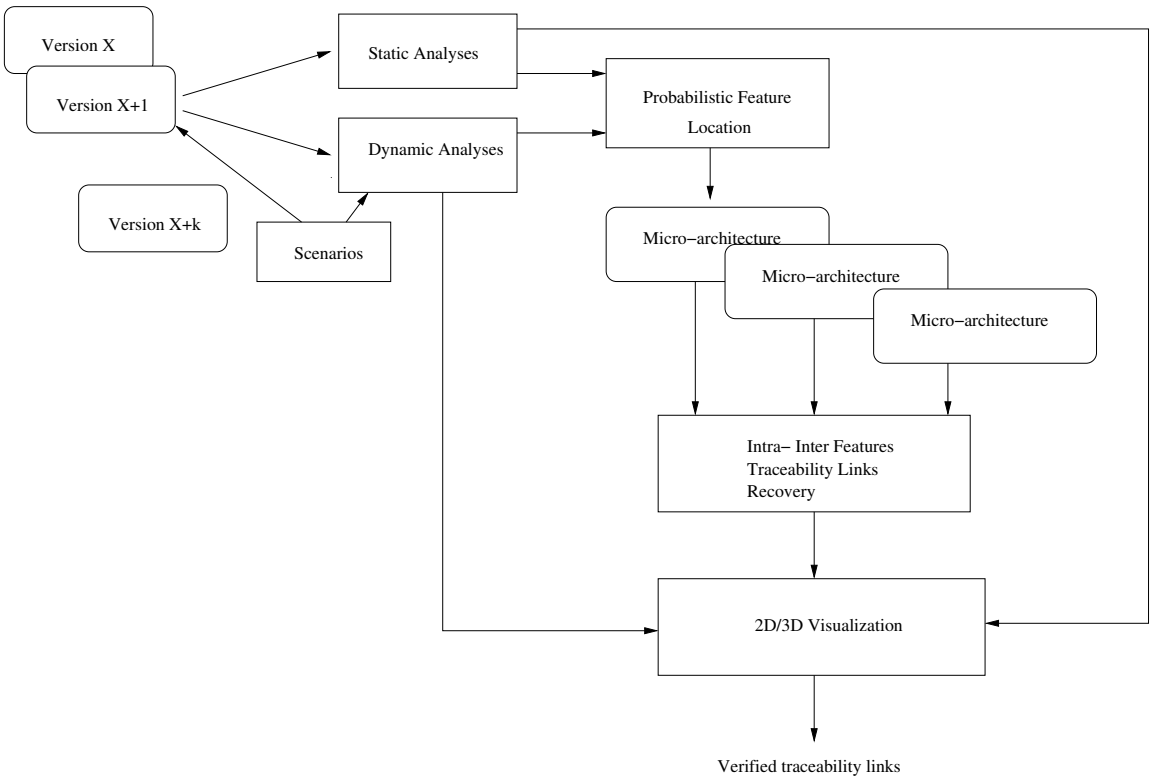


**Figure 1: Outline of the process of traceability recovery.**

systems should be the target of studies for feature identification and evolution, because of the wide adoption of the object-oriented paradigm. Also, we agree that both static and dynamic analyses are needed. We believe, in addition to mapping features and software constituents, that tracing feature evolution across releases and identifying feature interactions reduce the program comprehension effort and, thus, decrease the maintenance and evolution costs.

## 2. OUR APPROACH IN A NUTSHELL

We borrow from a previous work [3] key definitions and equations. A feature links a program architecture with the program dynamic behaviour. Thus, a first step is to recover the program architecture. Second, a subset of the program architecture, a micro-architecture, must be identified as participating in the realisation of a functionality. Third, intra- and inter-feature relationships across releases are studied to highlight feature evolution. Feature evolution and feature traceability are not addressed in previous work nor is, to the best of our knowledge, features to micro-architectures traceability and evolution.

We assume that the source code is available and that a compiled version can be exercised under different scenarios. Data collected from the realisation of a functionality, under different scenarios, is used to filter static data, thus relating files, functions, classes, and methods, with functionalities and scenarios. Clearly, this filtering process is an unavoidable preliminary step to study feature interaction and evolution and to highlight differences among features.

Figure 1 shows the key steps of the process of identifying features and studying their evolution. Static and dynamic analyses are used to extract data from executions of several releases of a system, following given scenarios. These data are analysed using a filtering and a probabilistic approaches to identify features. The features are micro-architectures, which are in turn analysed to identify traceability links, depending on the releases. The micro-architectures and traceability links are displayed to ease understanding and to confirm and to improve the previous analyses. Indeed, with available technologies and tools, user interaction is necessary to confirm identified features and traceability links. We aim at reducing the effort required to perform feature identification and evolution analyses but we do not believe that maintainers are not be needed to study feature evolution.

## 3. FEATURE IDENTIFICATION

For each needed technology, we provide a rationale for its adoption and we outline foreseeable results.

### 3.1 Static Analyses

Static analyses aim at obtaining compact representations of a system, such as metrics profiles, class diagrams, call graphs, dependency graph, and architectural views.

For example, metric profiles are used to study macro-evolution [4, 24] from a high-level point of view. Metric profiles are also use for scalable clone detection approaches. Recently, these approaches were extended to object-oriented systems [23]. Other approaches to identify duplicated code snippets were developed for procedural code [5, 6, 16, 18, 22], but are, in general, not well suited for object-oriented

systems. The ability to identify duplicated code can also be used to verify whether or not changes in classes, methods, or functions are limited to their structures or also impact their behaviours.

Static analyses are essential to recover a system architecture *as is* (directory and files organisations, file and subdirectory dependencies, and other representations). Unfortunately, static analyses only are imprecise. In particular, analysing programming languages such as C or C++ poses several challenges. Besides the intrinsic language peculiarities (*e.g.*, `union`, `struct`, `class`, function pointers...), preprocessor directives must be managed. Preprocessor directives are a usual way to obtain portability when developing in C or C++. Analysing multi-platform source code where preprocessor directives are platform-dependent is similar to projecting the source code onto a given hardware/software configuration. Thus, two approaches are possible to analyse multi-platform source code:

- Pre-process and analyse the code sources with different configurations.
- Construct a fictitious *reference* configuration assuming that each preprocessor `#ifdef` condition is *true*.

The first approach is feasible only for small or medium size systems, for which the entire software configuration to preprocess and to analyse is available. The second approach assumes that, very often, only the *then* part of a `#ifdef` is present and that the *then* branch almost always contains more code than the *else* branch. However, a fictitious reference configuration does not characterise a specific platform. Nevertheless, the later approach is well-suited for studying software evolution and for recovering high-level traceability links among releases.

Extracting information about methods and class relationships from object-oriented source code is difficult. Relationships may have degrees of imprecision due to intrinsic ambiguities. Given two classes and a relation between these, there are ambiguities due to implementation choices and to relationship characteristics (association or aggregation, unidirectional or bidirectional). Pointers, references, templates (e.g., `list<tree>`), and arrays (e.g., `Heap a[MAX]`) can represent both associations and aggregations. Recently, studies [13, 17] highlighted that no two approaches agree on the extraction of relationships from object-oriented source code.

We do not consider the limitations of static analyses as severe issues. We acquired and developed several parsers and tools to analyse systems statically with reasonable precision. In particular, we developed our own C++ parser, which manages the previous degrees of imprecisions, to generate AOL files. AOL files are higher-level representations of object-oriented systems (classes, methods, relationships) simple to handle programmatically. Also, we extract dynamic data, which compensates for the imprecisions of the static analyses.

## 3.2 Dynamic Analyses

Dynamic analyses are a necessary source of data to link functionalities with software constituents and, thus, to identify micro-architectures responsible for the specific implementation of functionalities. We follow a previous approach [3] inspired by Wilde [10] in which we extract and filter dynamic data during feature identification.

We instrument and generate a trace of the execution of a system, given a scenario. We associate events in the execution trace with a functionality using a relevance index, a ranking quantifying the probability that an event is relevant to the functionality under study. We concur with Wilde [10] that the use of set operations must be avoided. Unfortunately, avoiding set operations imply using thresholds and maintainer interactions to validate identified features. Let $\mathcal{F}$ be a set of scenarios exercising a functionality of interest and $\overline{\mathcal{F}}$ a set of scenarios not exercising the functionality. Execution of scenarios in $\mathcal{F}$ produces a set of intervals $\mathcal{I}^*$ containing events relevant to the functionality under study. We mark these intervals as relevant via *Start / Stop* signals. Scenarios in $\overline{\mathcal{F}}$ always produce intervals in $\mathcal{I}$, intervals irrelevant to the functionality. However, intervals $\mathcal{I}^*$ ($\mathcal{I}$, respectively) may contain irrelevant (relevant) events. Indeed, any scenario is likely to decompose in a few intervals in $\mathcal{I}^*$ surrounded by many intervals in $\mathcal{I}$. If $N_{\mathcal{I}^*}$ and $N_{\mathcal{I}}$ are the overall numbers of events in the two sets $\mathcal{I}^*$ and $\mathcal{I}$, then the frequency of $e_i$ in $\mathcal{I}^*$ is $f_{\mathcal{I}^*}(e_i) = \frac{N_{\mathcal{I}^*}(e_i)}{N_{\mathcal{I}^*}}$ and its frequency in $\mathcal{I}$ is $f_{\mathcal{I}}(e_i) = \frac{N_{\mathcal{I}}(e_i)}{N_{\mathcal{I}}}$. The relevance index is:

$$r(e_i) = \frac{f_{\mathcal{I}^*}(e_i)}{f_{\mathcal{I}^*}(e_i) + f_{\mathcal{I}}(e_i)} \quad (1)$$

Equation 1 is a *renormalization* of Wilde's equation [10], where events are re-weighted by population sizes to make events comparable directly.

Depending on the required details, tools, and available space, complete execution traces could be kept. Technologies to compress traces were proposed in the literature [14] and used for different purposes, see for example [8]. We experimented with processor emulation on the Mozilla web-browser, using Valgrind, with satisfying results. We also compared processor emulation with profiling techniques and found that processor emulation collects more accurate data from with little performance over-head.

## 3.3 2D and 3D Visualisation

One of the most recent technology to support maintainer activities is visualisation [20]. Visualisation allows representing on a single view complex information. Visualisation is a good cognitive support to the understanding of systems. Furthermore, animations can be used to help maintainers in grasping relevant information on evolution readily. Once functionality are linked with micro-architectures, micro-architecture evolution can be studied with the support of 2D and 3D visualisation techniques [9, 12, 19].

For example, Ghoniem applies adjacency matrices to ease the visualisation and the understanding of the evolution of large constraint problems [11]. We apply this technique to the visualisation and the understanding of the interaction among objects during executions of object-oriented systems. We predict that this technique could also be useful for the evolution of features across releases.

Another example is the use of 3D representations to visualise features and to assess the overlapping among features and their evolution. We succinctly describes a visualisation technique presented elsewhere by one of the author [19]. Classes of a system are represented as 3D boxes, which metric values characterise height, colour, and twist, to highlight the roles of the classes in the system. For instance, kernel packages contain a large proportion of complex classes

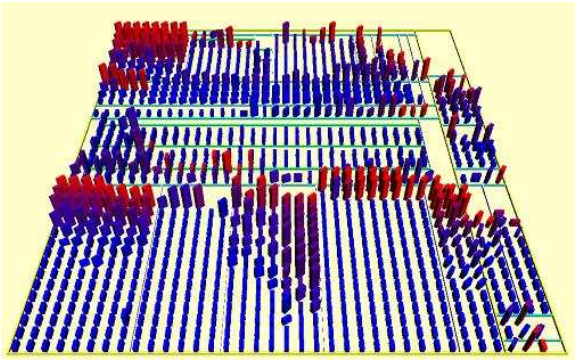

**Figure 2: Visualisation of metric values and roles.**

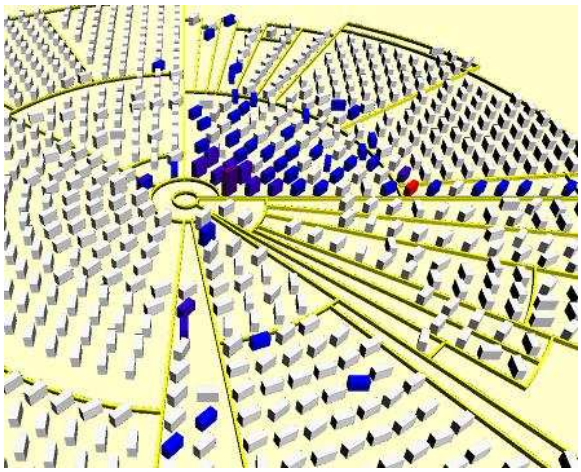

**Figure 3: Visualisation of overlaps among features.**

with high coupling. If we consider that a class is a 3D box where complexity is mapped to the height and coupling to the colour (blue to red), these classes are big and red. Similarly, utility packages contain a large proportion of complex classes with medium-to-low coupling (big purple). Without additional (semantic) information, we have a first idea on the roles of packages, which eases understanding. Figure 2 displays the 3D representation of a system, where high, twisted boxes are kernel classes, while the low, straight boxes are data classes.

A second use of 3D visualisation is highlighting overlaps among micro-architectures. Filters can be applied to a representation to highlight the constituents of a micro-architecture exercised in more than one functionality. Figure 3 shows the 3D representation of a system with few classes implementing several functionalities are highlighted.

Finally, visual analysis can be applied to analyse the evolution of a system and to compare this evolution against external information. For instance, when evolving from one release to another, we can display differences among representations. Figure 4 shows the 3D representation of the evolution of classes represented by their metric values across several releases. Using external data (bug corrections, feature implementation...), we can link this information with past evolution activities. We can also trace the evolution of micro-architectures and of the corresponding functionalities.

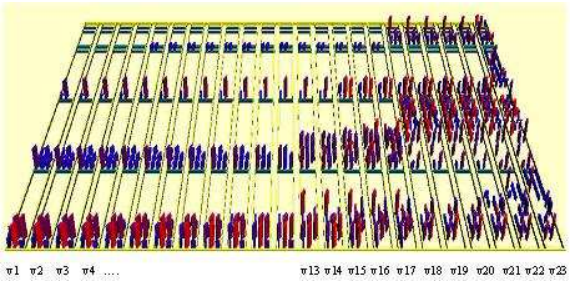


**Figure 4: Visualisation of feature evolutions.**

## 3.4 Information Retrieval

Information retrieval and computational linguistic technologies have been widely adopted in the traceability recovery community [21, 15, 2]. We concur with previous work that these technologies are essential to identify traceability links and to assess their accuracy.

We support the hypothesis that information retrieval technologies are useful to highlight micro-architectures across releases and, thus, to support the study of feature evolution. These technologies must be compared with graph and sub-graph matching approaches or hybrid approaches where similarity scores are defined among vertices of graphs [1]. Comparison must be performed in terms of accuracy and of effort required to post-process identified traceability links.

Since multiple technologies for traceability links are available, it is important to combine different sources of data, weighing the data with its (dis)advantages *wrt.* the specific approach performance. Thus, for example, vector space results could be combined with graph matching approaches to reinforce conjectures on traceability links and to decrease the required effort to assess their accuracy.

# 4. FEATURE EVOLUTION

We envision several studies of feature evolution across releases of a same system, depending on the original hypotheses. The following subsections highlight possible hypotheses.

## 4.1 Functionality Existing across Releases

A first study could focus on functionalities which exist across several releases of a system. Interesting questions to answer to assist maintainers in understanding the implementation are:

- Does the micro-architecture associated with the functionality changes across releases?
- Do the changes come from users' requests for *this* functionality or from other evolution activities?

Functionalities existing across releases and being modified are observable software improvements potentially. Maintainers observe both the modification of the functionality and the evolution of the functionality implementation. We believe that traceability links are useful to ease program comprehension and to promote bug-free evolution.

## 4.2 Appearing Functionality

In the case of a functionality appearing in a subsequent release of a system, interesting questions relate to the implementation of this functionality *wrt.* previously existing functionalities:

- Does the implementation of this functionality uses constituents in previously-existing functionalities?
- How well integrated is this functionality *wrt.* other functionalities? (How many previously-existing functionalities have been impacted by the introduction of this functionality?)

Appearing functionalities are likely to be the result of new code interacting with existing code and, thus, functionalities. The ability to locate functionality implementation, to identify constituents in common with existing functionalities helps in understanding the system, in ensuring its maintainability, and in avoiding unexpected side-effects. For example, when adding a new functionality, if existing classes and methods are modified, other functionalities may be negatively impacted. The probability of such undesired side-effects is higher if functionalities are not linked to source code and if their interactions are undocumented.

## 4.3 Disappearing Functionality

It is well known that functionalities disappear rarely, even when unused. As an example, Microsoft products include dozens of new functionalities in each new release, while no functionalities are ever removed. Nonetheless, we believe that removing functionalities should become more commonplace as good design and programming practices spread. Thus, interesting questions are:

- How does the micro-architecture associated with a disappearing functionality evolve?
- Does the removal of the functionality impact other functionalities through their micro-architectures?

The removal of a functionality must be documented and the impacted architectural constituents identified to reduce the impact of the removal on remaining functionalities. It is seldom the case that functionalities are implemented via completely decoupled micro-architectures. Thus, traceability links are beneficial to reduce required removal efforts and associated risks.

## 4.4 Changing Micro-architecture

Finally, it is possible that micro-architectures change while the user-observable functionalities of a system remain seemingly unchanged. Interesting questions are:

- Why did the micro-architectures change?
- What functionalities are potentially impacted by the changes?

The problem is somehow the opposite of the previous case. Indeed, maintainers should be able to navigate traceability links in both directions, from high-level user-observable functionalities to implementing micro-architectures, and vice-versa. This navigability is essential to any evolution activity. With a traceability link available, data on bug and number of changes can be related to functionalities easily. Thus, a relation between stability, quality, and frequency of changes can be drawn. We hope that such a relation may lead to understand better the pros and cons of different implementations and to identify more robust, more evolvable, and less error-prone micro-architectures.

# 5. CONCLUSION AND CURRENT WORK

In this position paper, we outlined an approach to identify features, to study feature evolution, and to provide feature traceability in object-oriented software systems. Several technologies, which are readily available, are needed in combination to study feature evolution. Challenges remain with respect to the amount of data to process and to need for maintainers interactions.

We are currently parsing and executing scenarios on several releases of the Mozilla web-browser to identify micro-architectures implementing functionalities, such as sending an e-mail or accessing a WEB page, and to study the feature evolution. We are also relating features across releases with external information, such as change requests. We will then apply our complete approach to the collected data to study feature evolution and assess our approach quality.

# 6. REFERENCES


[1] G. Antoniol, G. Canfora, G. Casazza, and A. D. Lucia. Maintaining traceability links during object-oriented software evolution. *Software - Practice and Experience*, 31:1–25, 2001.

[2] G. Antoniol, G. Canfora, G. Casazza, A. D. Lucia, and E. Merlo. Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering*, 28:970–983, Oct 2002.

[3] G. Antoniol and Y. Gueheneuc. Feature identification: A novel approach and a case study. In *Proceedings of IEEE International Conference on Software Maintenance*, Budapest, Sept 2005. IEEE Computer Society Press.

[4] G. Antoniol, U. Villano, E. Merlo, and M. D. Penta. Analyzing cloning evolution in the linux kernel. *Information and Software Technology*, 44:755–765, October 2002.

[5] B. S. Baker. On finding duplication and near-duplication in large software systems. In *Proceedings of IEEE Working Conference on Reverse Engineering*, July 1995.

[6] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proceedings of IEEE International Conference on Software Maintenance*, pages 368–377, 1998.

[7] E. Chikofsky and J. C. II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, Jan 1990.

[8] V. Dallmeier, C. Lindig, and A. Zeller. Lightweight defect localization for java. In *Proc. 19th European Conference on Object-Oriented Programming*, pages XX–YY, Glasgow, Scotland, July 2005.

[9] S. Ducasse and T. Gîrba. Modeling software evolution by treating history as a first class entity. In *proceedings of the workshop on Software Evolution Through Transformation*, pages 71–82, 2004.

[10] D. Edwards, S. Simmons, and N. Wilde. An approach to feature location in distributed systems. Technical report, Software Engineering Research Center, 2004.

[11] M. Ghoniem and J.-D. Fekete. Visualisation de graphes de co-activité par matrices d'adjacence. In É. Lecolinet and D. L. Scapin, editors, *actes de la* $14^e$ *conférence sur l'Interaction Homme–Machine*, pages 279–282. ACM Press, octobre 2002.

[12] M. Ghoniem, J.-D. Fekete, and P. Castagliola. A comparison of the readability of graphs using node-link and matrix-based representations. In M. Ward and T. Munzner, editors, *proceedings of the* $10^{th}$ *symposium on Information Visualisation*, pages 17–24. IEEE Computer Society Press, October 2004.

[13] Y.-G. Guéhéneuc and H. Albin-Amiot. Recovering binary class relationships: Putting icing on the uml cake. In D. C. Schmidt, editor, *proceedings of the* $19^{th}$ *conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM Press, October 2004.

[14] A. Hamou-Lhadj and T. Lethbridge. Compression techniques to simplify the analysis of large execution traces. In *Proceedings of the 10th International Workshop on Program Comprehension*, pages 159–168, Paris, France, Jun 26-29 2002.

[15] J. H. Hayes, A. Dekhtyar, S. Sundaram, and S. Howard. Helping analysts trace requirements: An objective look. In *Proceedings of IEEE Requirements Engineering Congerence (RE) 2004*, pages 249–261, Kyoto, Japan,, Sept 2004.

[16] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, July 2002.

[17] R. Kollmann, P. Selonen, E. Stroulia, T. Systa, and A. Zundorf. A study on the current state of the art in tool-supported uml-based static reverse engineering. In *Proceedings of IEEE Working Conference on Reverse Engineering*, pages 22–33, Richmond, Virginia, USA, October 29 - November 1 2002.

[18] K. Kontogiannis, R. D. Mori, R. Bernstein, M. Galler, and E. Merlo. Pattern matching for clone and concept detection. *Journal of Automated Software Engineering*, March 1996.

[19] G. Langelier, H. A. Sahraoui, and P. Poulin. Visualization-based analysis of quality for large-scale software systems. In T. Ellman and A. Zisma, editors, *proceedings of the* $20^{th}$ *internatinal conference on Automated Software Engineering*. ACM Press, November 2005.

[20] A. Marcus, L. Feng, and J. I. Maletic. 3d representations for software visualization. In *SoftVis '03: Proceedings of the 2003 ACM symposium on Software visualization*, pages 27–ff, New York, NY, USA, 2003. ACM Press.

[21] A. Marcus and J. I. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Proceedings of the International Conference on Software Engineering*, pages 125–135, Portland Oregon USA, May 2003.

[22] J. Mayrand, C. Leblanc, and E. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *Proceedings of IEEE International Conference on Software Maintenance*, pages 244–253, Monterey CA, Nov 1996.

[23] E. Merlo, G. Antoniol, M. D. Penta, and F. Rollo. Linear complexity object-oriented similarity for clone detection and software evolution analysis. In *Proceedings of IEEE International Conference on Software Maintenance*, pages 412–416, Chicago, USA, Sept 11-17 2004. IEEE Computer Society Press.

[24] E. Merlo, M. Dagenais, P. Bachand, J. S. Sormani, S. Gradara, and G. Antoniol. Investigating large software system evolution: the linux kernel. In *COMPSAC*, pages 421–426, Oxford England, Aug 26-29 2002.