Université de Montréal

A Unified Framework for the Comprehension of Software's Time Dimension

par
Omar Benomar

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Thèse présentée à la Faculté des études supérieures et postdoctorales
en vue de l'obtention du grade de Philosophiæ Doctor (Ph.D.)
en Informatique

Février, 2015

Résumé

Les logiciels sont de plus en plus complexes et leur développement est souvent fait par des équipes dispersées et changeantes. Par ailleurs, de nos jours, la majorité des logiciels sont recyclés au lieu d'être développés à partir de zéro. La tâche de compréhension, inhérente aux tâches de maintenance, consiste à analyser plusieurs dimensions du logiciel en parallèle. La dimension temps intervient à deux niveaux dans le logiciel : il change durant son évolution et durant son exécution. Ces changements prennent un sens particulier quand ils sont analysés avec d'autres dimensions du logiciel. L'analyse de données multidimensionnelles est un problème difficile à résoudre. Cependant, certaines méthodes permettent de contourner cette difficulté. Ainsi, les approches semi-automatiques, comme la visualisation du logiciel, permettent à l'usager d'intervenir durant l'analyse pour explorer et guider la recherche d'informations. Dans une première étape de la thèse, nous appliquons des techniques de visualisation pour mieux comprendre la dynamique des logiciels pendant l'évolution et l'exécution. Les changements dans le temps sont représentés par des *heat maps*. Ainsi, nous utilisons la même représentation graphique pour visualiser les changements pendant l'évolution et ceux pendant l'exécution. Une autre catégorie d'approches, qui permettent de comprendre certains aspects dynamiques du logiciel, concerne l'utilisation d'heuristiques. Dans une seconde étape de la thèse, nous nous intéressons à l'identification des phases pendant l'évolution ou pendant l'exécution en utilisant la même approche. Dans ce contexte, la prémisse est qu'il existe une cohérence inhérente dans les évènements, qui permet d'isoler des sous-ensembles comme des phases. Cette hypothèse de cohérence est ensuite définie spécifiquement pour les évènements de changements de code (évolution) ou de changements d'état (exécution). L'objectif de la thèse est d'étudier l'unification de ces deux dimensions du temps que sont l'évolution et l'exécution. Ceci s'inscrit dans notre volonté de rapprocher les deux domaines de recherche qui s'intéressent à une même catégorie de problèmes, mais selon deux perspectives différentes.

Mots clés: Maintenance du logiciel, compréhension du logiciel, visualisation du logiciel, exécution de programme, évolution du logiciel.

Abstract

Software systems are getting more and more complex and are developed by teams that are constantly changing and not necessarily working in the same location. Moreover, most software systems, nowadays, are recycled rather than being developed from scratch. A comprehension task is crucial when performing maintenance tasks; it consists of analyzing multiple software dimensions concurrently. Time is one of these dimensions, as software changes its state with time in two manners: during their execution and during their evolution. These changes make sense only when analyzed within the context of other software dimensions, such as structure or bug information. Multidimensional analysis is a difficult problem to solve. However, there are certain methods that bypass this difficulty, such as semi-automatic approaches. Software visualization is one of them, as it allows being part of the analysis by exploring and guiding information search. The first stage of the thesis consists of applying visualization techniques to better understand software dynamicity during execution and evolution. Changes over time are represented by heat maps. Hence, we utilize the same graphical representation to visualize both change types over time. Other approaches that permit the analysis of a program's dynamic behavior over time involve the use of heuristics. In the thesis' second stage, we are interested in the identification of the programs' execution phases and evolution patterns using the same approach, i.e. search-based optimisation. In this context, the premise is the existence of internal cohesion between change events that allow the clustering in phases. This hypothesis of cohesion is defined specifically for change events in the code during software evolution and state changes during program execution. This thesis' main objective is to study the unification of these two time dimensions, evolution and execution, in an attempt to bring together two research domains that work on the same set of problems, but from two different perspectives.

Keywords: Software maintenance, program comprehension, software visualization, program execution, software evolution.

Contents

List of Tables

List of Figures

List of Appendices

I dedicate this thesis to my parents, Hafid & Chafika, my wife, Myrianna, and my kids, Mia & Kamil.

Acknowledgments

I thank my supervisors, Houari Sahraoui and Pierre Poulin, for guiding me throughout my thesis. It was an honor to have these two great professors as advisers. Their availability and generosity helped broaden my horizons in many domains. As a researcher, I was fortunate to learn from two professors who are different and complementary. For every idea, I had two perspectives that allowed me to advance my research and our discussions were always fruitful. Also, from a human perspective, I could not have chosen better supervisors to accompany me during my doctorate. I fed on their energy and good mood at difficult times, and their advices go beyond academic research.

I also thank my family. My parents, my sister, and my brother who provided me with limitless support as always. I thank my father, Hafid, for his wise advices, and my mother, Chafika, for her immense goodwill.

My wife Myrianna has given me immeasurable support and encouragement. Her kindness and compassion gave me the necessary strength to persevere. She played a vital role in allowing me to finish my doctorate. For all these reasons, I can not thank her enough or show her all the gratitude she deserves.

Finally, I thank my kids, Mia and Kamil, because they are the best loving, caring, and empathic children in the world.

Chapter 1

Introduction

1.1 Context

The development process of a software goes through multiple stages during its lifecycle. These stages are design, conception, implementation, validation and verification, deployment, and finally maintenance. Maintenance activity occupies the largest part of a software's life and also requires the most resources allocated to its development. Financial and human resource costs of software maintenance have been estimated by previous work [6, 19, 43, 54] to 50-80% of the total software development costs. IEEE standard [51] defines software maintenance as the modification of a software product after its deployment for bug correction, the improvement of some software attributes such as performance, or to conform the software to a changed environment. Hence, we can distinguish between four types of maintenance: corrective, adaptive, perfective, and preventive. The latter concerns the modification of software for easier future maintenance.

Nowadays, software is developed by dispersed teams (location-wise) over long periods of time. Although developers usually work on different modules forming a small subset of an entire software, it is important that they gain a good comprehension of the entire software to be maintained. This makes software comprehension an essential and necessary task for software maintenance. It is crucial to understand a program in order to better modify it. Program understanding represents more than 50% of maintenance effort [10, 20]. The difficulty of comprehension tasks is due to the need for the developer to build a mental representation of software. There are three types of software comprehension models. First, the *Top-Down* comprehension model lets the developer gain knowledge of the software due to prior maintenance tasks. This allows the developer to formulate hypotheses about the investigated code and identify modules (packages, classes, methods, etc.) or code fragments involved in a use-case scenario or implementing a particular software functionality. The hypotheses are then evaluated in an iterative process to be refined, accepted, or rejected.

Second, developers have sometimes limited knowledge of the studied system, for instance in the case of new developers joining a development team. Software analysis and its comprehension

have to start from the source-code level and acquire higher-level knowledge; it is the *Bottom-Up* comprehension model. It consists of building abstractions by regrouping lower-level information and forming a mental model based on these abstractions. For example, Pennington [44] suggests that developers construct a first mental image based on the control flow, and then a second mental image based on functional abstractions of the system.

Third, the integrated comprehension model is an aggregation of the two previously discussed models. In this model, developers utilize either model depending on their needs and the extent of knowledge for the considered parts of the system.

Therefore, we can facilitate the comprehension task by working on abstract representations of software. To this end, we are interested in the application of automatic and semi-automatic approaches to solve comprehension problems. We use software visualization as a semi-automatic approach, where the developer intervenes during the comprehension task. We also use heuristic search as an automatic approach to abstract low-level information and provide developers with high-level models of software. We are mainly interested in two domains: the comprehension of program execution and the comprehension of software evolution.

1.2 Research Problem

Software comprehension involves the analysis of several dimensions of the studied system, such as source code, program structure, bug management, etc. In addition to these information, developers must integrate in their comprehension model, the software's ability to change its state over time in evolution or execution; we will refer to it as software dynamicity. Software changes over time during its execution and during its evolution. These changes are usually meaningful only within the context defined by other software dimensions. The analysis of multiple software dimensions requires parallel processing of large amount of data. The interpretation of such information by developers is very challenging, especially if the raw data is presented without any processing.

The study of software dynamicity consists of analyzing software evolution on the one hand, and software execution on the other hand. Software evolution is the process of changing a software repeatedly over time by adding new functionalities and capabilities, or correcting mistakes. The comprehension of the evolution of a system facilitates its maintenance. The analysis of past changes applied to

a system helps to better modify it in the future by taking better and more advised decisions. Furthermore, a good comprehension of software evolution enables developers to identify critical moments in the development, that might explain present software problems. Another aspect of software dynamicity is the study of software execution. The analysis of the dynamic behavior of a system during its execution is essential for many comprehension tasks. Typically, dynamic analysis studies a program's execution traces generated during an execution scenario. Traces are usually prohibitively large to be interpreted by a human. Hence, they are abstracted in order to gain valuable knowledge about the system's dynamic behavior. Information, such as performance bottlenecks or feature location, can be deduced by analyzing the program execution.

Finally, the analysis of software dynamicity is challenging due to the changing nature of the information over time. Many execution comprehension problems share similar traits with evolution comprehension problems and vice versa. The execution comprehension research community has developed solutions to some of the comprehension problems it faces, that could be ported and used by the evolution comprehension research community. Also, the execution comprehension research community may benefit from advances in the evolution comprehension research community. However, there is little communication or sharing between the two research communities, and solutions developed in one community are used exclusively within its community. Identifying and modeling the commonalities of software dynamicity comprehension problems may help accomplish advances in both research communities.

1.3 Research Proposition

We propose to unify execution comprehension and evolution comprehension by using a common framework to solve comprehension problems in execution and in evolution. The framework defines the common characteristics of software dynamicity comprehension problems. It allows the definition and formalization of similar problems from the two research communities. The problems are then solved using similar techniques. First, we consider the use of a semi-automatic approach, software visualization, to solve the collaboration of entities over time. Second, we study the phase identification problem in software evolution and in program execution using heuristic search. The definition of comprehension problems using the common framework and the application of a similar resolution

technique, fall within our desire for the rapprochement of the two software dynamicity comprehension communities. Thus, our thesis can be summarized in the following statement:

"Software comprehension problems involving a time dimension should be studied using a common unified framework, which enables easier communication of solutions between evolution comprehension research community and execution comprehension research community."

1.4 Thesis Contributions

This thesis involves various contributions in software comprehension. These contributions are:

1. The central contribution of this thesis is the conception of a unified framework for software comprehension problems in the contexts of evolution and execution. The framework is typified by a simple meta-model describing time comprehension problems.
2. The framework is utilized to express the comprehension of entity collaboration in a generic manner for software evolution and software execution. This representation of entity collaboration analysis is defined in terms of software visualization.
3. Phase identification for the purpose of understanding software evolution and software execution is described using our unified framework. A search-based optimization technique is characterized based on our description of a phase identification problem.
4. Heat map visualization is used to understand developers' collaboration over the time period of software evolution. Heat maps represent developers' contributions to software. Developers' collaboration is represented by the aggregation of multiple heat maps.
5. Classes roles and contributions in the execution of use-case scenarios is analyzed with heat map visualization. Use-cases are depicted as heat maps where classes have different degrees of contributions. Classes' roles are determined using heat map comparisons.
6. Meta-heuristic search for the execution phase is used to determine the stages that program undergoes. The approach is based on object lives during the execution.

7. The identification of software evolution phases over time is performed with a search-based optimization on development activities in terms of source code changes.

1.5 Dissertation Organization

The remaining of the dissertation is organized as follows:

Chapter 2 covers related contributions in software comprehension. It presents work on comprehension problems in both considered contexts: execution and evolution. In Chapter 3, we describe the proposed unified comprehension framework. The framework is then used to express two comprehension problems in a generic manner and we apply a similar specific technique to solve each problem. Chapter 4 addresses the first comprehension problem considered: entity collaboration. We examine the use of software visualization to understand entity collaboration during execution and evolution. In Chapter 5, we consider the second comprehension problem: phase identification. We explain how to identify phases using a search-based technique. The phase identification problem is formulated as an optimization and solved using genetic algorithms. Finally, we conclude our dissertation in Chapter 6, where we discuss the implications of the unification of evolution and execution comprehension problems. We also describe other possible research directions based on our proposed framework.

Chapter 2

Related Work

In this chapter, we introduce existing work in the different domains covered by our research. There are various contributions in the literature that treat software comprehension at large and dynamicity comprehension in particular. In an effort to show the parallels between software evolution comprehension and software execution comprehension research, we organize the related work according to the research communities they belong to. First, we give an overview of related work in software evolution comprehension in Section 2.1. Section 2.2 presents previous work in software execution comprehension. In our unification approach, we formulate a meta-model for software comprehension problems involving time. We investigate developer collaboration in software evolution and class contributions to accomplish use-case scenarios, using the same approach, and we study the phase detection in software evolution and program execution by applying the same technique. Therefore, Sections 2.1 and 2.2 are both composed of three subsections. Sections 2.1.1 and 2.2.1 give an overview of how time is represented in other software comprehension problems. Sections 2.1.2 and 2.2.2 discuss contributions related to the first software comprehension problem considered in this thesis: collaboration analysis. Sections 2.1.3 and 2.2.3 present work related to the second software comprehension problem: phase identification. Finally, Section 2.3 concludes this chapter.

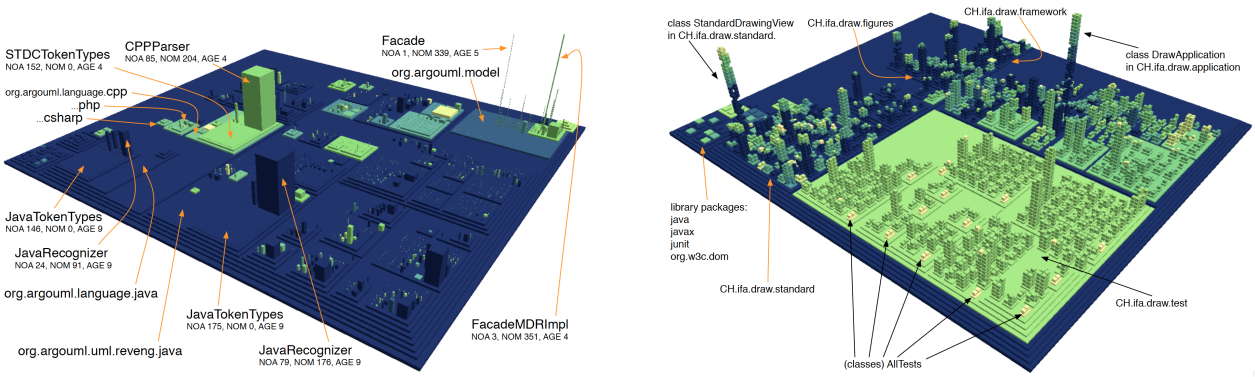
2.1 Software Evolution Comprehension

Software evolution comprehension problems come in various forms. The majority of them have to do with analyzing software evolution data to reveal higher-level information necessary to the comprehension process. Much research has been done in this domain. Here, we present how other contributions in software evolution comprehension represent time implicitly or explicitly. Then, we present research related to two specific software evolution problems: developer collaboration analysis and evolution phases identification.

2.1.1 Evolution Time Representation

Langelier et al. [40] use software visualization to study software evolution. The proposed software visualization shows the structure of the studied system in a treemap layout and represents classes as 3D boxes arranged over a 2D plane divided into regions that correspond to packages. The graphical characteristics of boxes are mapped to metrics computed on classes. The evolution of the system is visualized using animations. The authors introduce two animations to represent software evolution from one version to the next. First, an animation of a single class illustrates changes of the class between two versions. The graphical attributes of the box representing the class are linearly interpolated and in-between frames are created to ease the visual transformation of the class going to the next version. Second, the software representation might change between two consecutive versions and hence animation is used to achieve more spatial coherence in the system. There are two alternative animations for positioning classes on the 2D plane. On the one hand the static position animation, where a fixed position is computed for all classes with respect to all versions, i.e., any class that existed, exists, or will exist during the evolution is assigned a fixed position over time. On the other hand, the relative position animation, which is built on top of the static position animation but with a post-processing step that reduces space in early versions (fewer classes). The relative positioning of classes is always respected but empty spaces are collapsed to make classes closer. Therefore, the authors represent time during the evolution using scene animation. They provide means to navigate between versions while keeping visual coherence between views. Each view represents a version of the system, i.e., an instant in evolution time. Each version is displayed one after the other using animated transitions.

Wettel and Lanza [58] propose a software visualization based on a city metaphor to study the evolution of a software system. Their visualization provides three representations of time in software evolution. First, *Age Map* is used to represent the age of an artifact in the current version. A color scheme is mapped to the time since the last modification. It can be applied at different granularity levels as shown in Figure 2.1. This visualization allows users to distinguish between old parts of the system and recently changed ones to get the overall picture of the software evolution. Second, *Time Travel* permits the navigation back and forth between the different versions of the visualized system. It resolves the main drawback of the *Age Map* visualization, which flattens evolution information with respect to the current version. Each view represents an instant in time during the entire software



(a) Coarse-grained *age map* of *ArgoUML*. (b) Fine-grained *age map* of *JHotDraw*.

Figure 2.1 – *Age Map* visualization with different granularity levels [58].

evolution. It allows the analysis of the process of evolution of both the whole system and individual artifacts at coarse-granularity. Third, *Timeline* gives an overview of all the versions of a class in the system. For comparison, class versions are aligned along a time axis and color coded according to their age, as illustrated in Figure 2.2. This visualization allows users to gain insights into the evolution of a class at the method level of granularity.

Girba and Ducasse [31] present an explicit meta-model, *Hismo*, to represent software history as an entity. *Hismo*, illustrated in Figure 2.3, is based on three basic entities: *Snapshot*, *History*, and *Version*. The *Snapshot* entity encapsulates software artifacts whose evolution is studied. The *History* entity holds the set of versions of the studied system. The time dimension in this meta-model is represented by the entity *Version*. It has a time stamp, and adds the notion of time to the entity *Snapshot* by linking it to the entity *History*. The *Hismo* meta-model can be used to define history measurements for evolution analysis.

Wu et al. [60] present a visualization technique called *evolution spectrograph* to analyze software evolution. Their approach is based on sound spectrograph, which represents visually the frequency of content of sound and its variations in time presented in an XY graph. By analogy, the evolution spectrograph depicts how a spectrum of software components change over time. Software evolution is characterized in terms of time, spectrum, and measurements. The time dimension in the spectrograph can be represented by units of evolution events, such as the number of commits or releases, or by fixed periods of time. The spectrum dimension is the decomposition of software into smaller units.

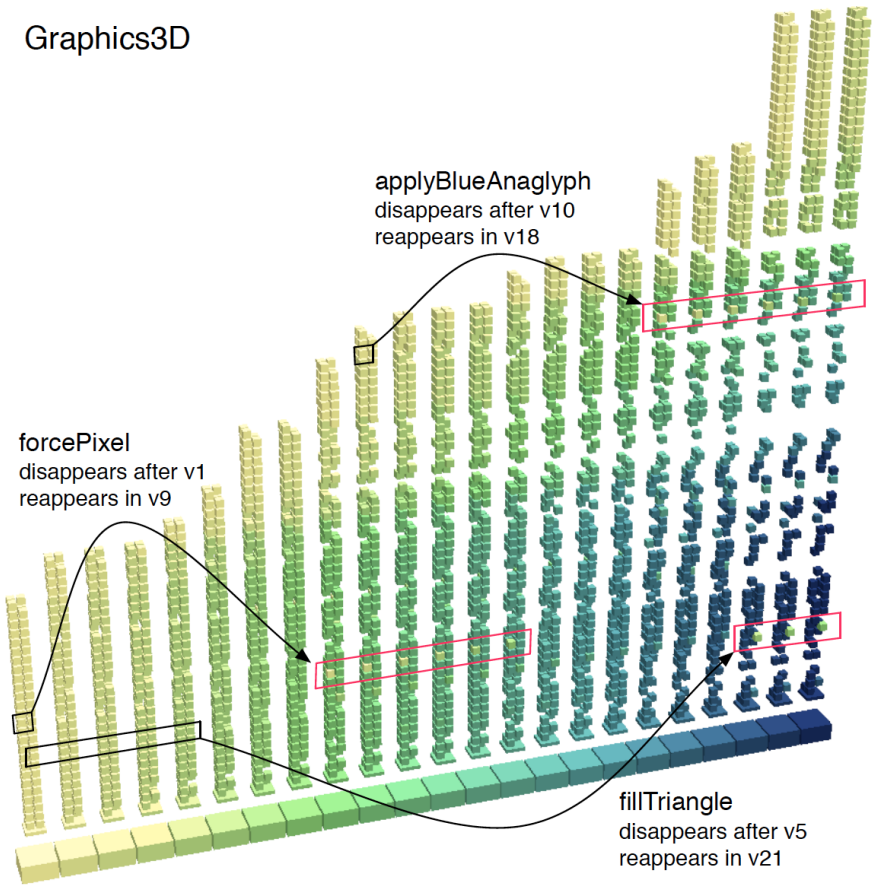


Figure 2.2 – *Timeline* visualization of class *Graphics3D* from *Jmol* [58].

The decomposition can be achieved at different levels of granularity depending on the studied subject. Measurements are computed for software units composing the spectrum at any point during the history of the studied system. For instance, Figure 2.4 illustrates the visualization used for the detection of sudden and discontinuous changes in *OpenSSH*. The spectrum is composed of source files of the system. Time is represented by a sequence of versions of the system. The measurements used for the analysis are *Fan In* (Figure 2.4(a)) and *Fan Out* (Figure 2.4(b)) of changed dependencies at the file level between two adjacent releases. The changed dependencies of the source files are color coded (red for change, green for no change) to visually analyze evolution.

The contributions discussed above study different problems in software evolution comprehension. The proposed approaches make use of the software time dimension in their respective studies of software evolution. Time is represented as a sequence of events occurring over the lifetime of a

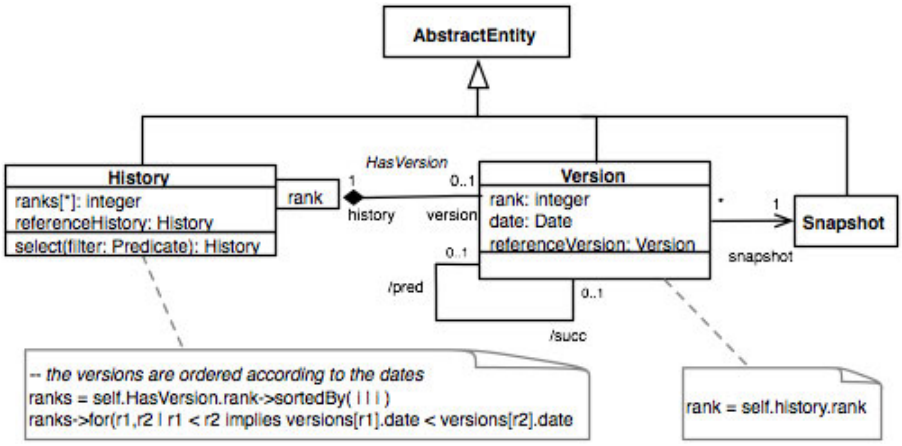


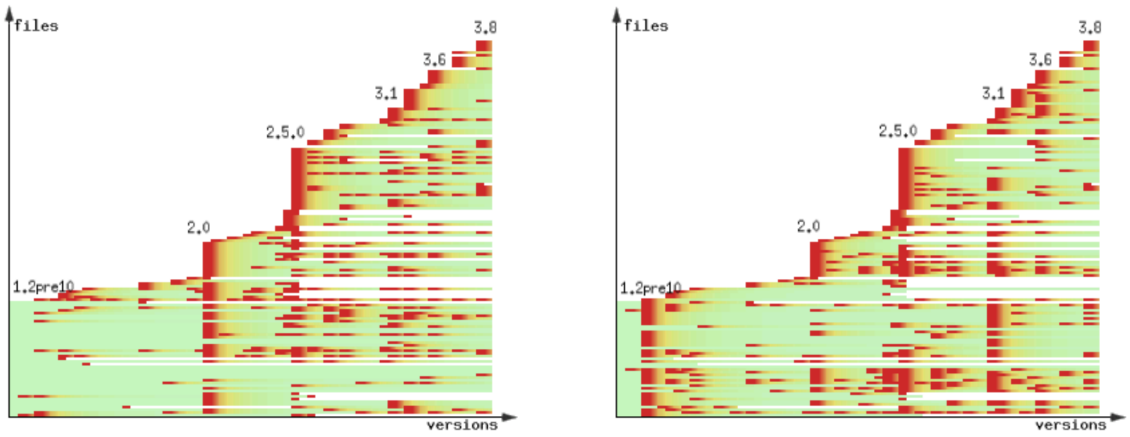
Figure 2.3 – *Hismo* is a meta-model for software evolution analysis [31].

system. These evolution events are represented using scene animation and navigation (see [40, 57]) or linearly aligned along an axis (see [60]). Time is also part of a higher-level abstraction modeling software evolution (see [31]).

2.1.2 Developer Collaboration Analysis

Begel et al. [11] propose a framework, *Codebook*, for mining software repositories to look for connections between developers. It exploits several information items such as source code, employee directory, discussion forums, mailing lists, etc. The authors automatically produce a graph representing people, artifacts, and their connections. The graph is based on a typical social networking graph based on an event model, i.e., an event happening at a node is reported to all nodes connected to the source node. By analogy, *Codebook* allows users to become "friends" with other people, or artifacts, enabling users to receive events concerning them. For example, closing a bug event is reported to all friends connected to the node representing the bug. One application of the framework is to search and find developers responsible for a particular code. *Codebook* links code to developers, to related bugs, and to emails where the code is mentioned. These relations allow developers to easily identify owners of a particular code.

Lungu et al. [42] present a platform for the analysis of super-repositories that is called *the Small Project Observatory* (SPO). The platform is a highly interactive web application with several interaction modes. Figure 2.5 shows the interface of the *Small Project Observatory*. SPO allows users



(a) Fan In of Changed Dependencies

(b) Fan Out of Changed Dependencies

Figure 2.4 – Evolution spectrogram visualization of *OpenSSH* [60].

to choose a perspective in the interactive view. The user can select and filter projects to be viewed, navigate between perspectives, or explore depicted projects. Filters can also be applied to alleviate the cluttering due to the amount of displayed information. Finally, detailed information on the view or on the selected elements is provided on demand in the panel at the right of the interface. SPO provides various perspectives to explore evolution of projects. For instance, the developer activity perspective depicts developers' contributions to a software. An activity line represents periods of time, where the developer is committing changes to the software repository. Another perspective concerns the representation of developers' collaborations using a graph as shown in Figure 2.6. This collaboration graph is constructed by associating nodes to developers and connecting the nodes if two developers make a sufficient number of modifications to the same project. The authors distinguished three types of developers using this perspective: loners, collaborators, and hubs. Loners work alone on projects. Collaborators work with others on few projects. Hubs collaborate on many projects.

Surian et al. [53] study developer collaboration by extracting patterns in a large collaboration network. They investigate the nature of connections between developers, and statistically characterize their collaboration clusters. The authors construct a collaboration network from *SourceForge.net* dataset and mine two types of patterns: high-level patterns and detailed topological patterns. High-level patterns are detected by computing different metrics from the collaboration network. Topological patterns are extracted using graph mining and graph matching techniques. The collaboration graph

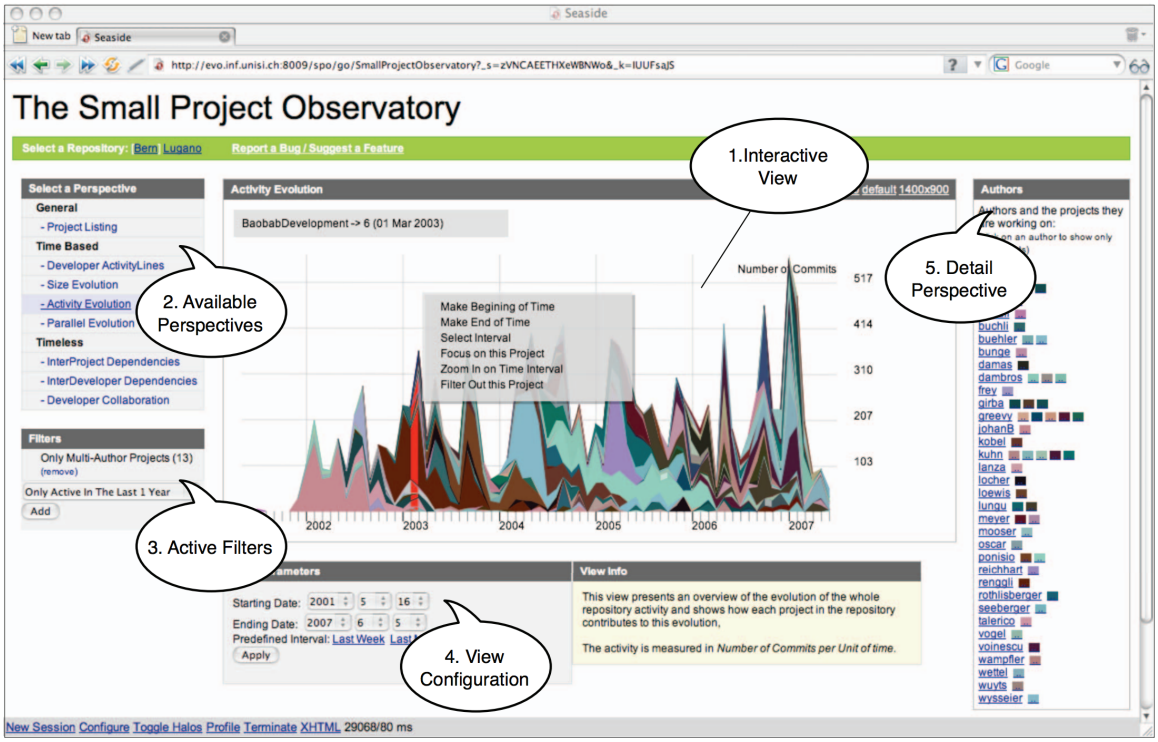


Figure 2.5 – The interface of the *Small Project Observatory* [42].

is composed of clusters representing connected components of the dataset, i.e., developers collaborating in *SourceForge.net*. First, mining is used to identify topological patterns in the small graphs of the collaboration network. Then, the identified patterns are used to find other patterns in the larger graphs of the collaboration graph using graph matching.

Bhattacharya et al. [17] investigate the use of a graph-based characterization of a software to capture its evolution, and analyze developer collaborations at the commit and bug-fixing levels. The authors propose two types of graphs based on two data sources: the source-code repository and the bug-tracking system. They distinguish between source-code-based graphs and developer-collaboration graphs. For instance, the module collaboration graph is a source-code-based graph that captures the communications between modules like a call graph but with a coarse-grained representation. The module collaboration graph helps in the comprehension of software components' communications. The developer-collaboration graphs are built using either bug information or commit information. The bug-based developer-collaboration graph represents bug assignment between developers, i.e., nodes are developers and a directed edge denotes a bug assignment between a pair of developers.

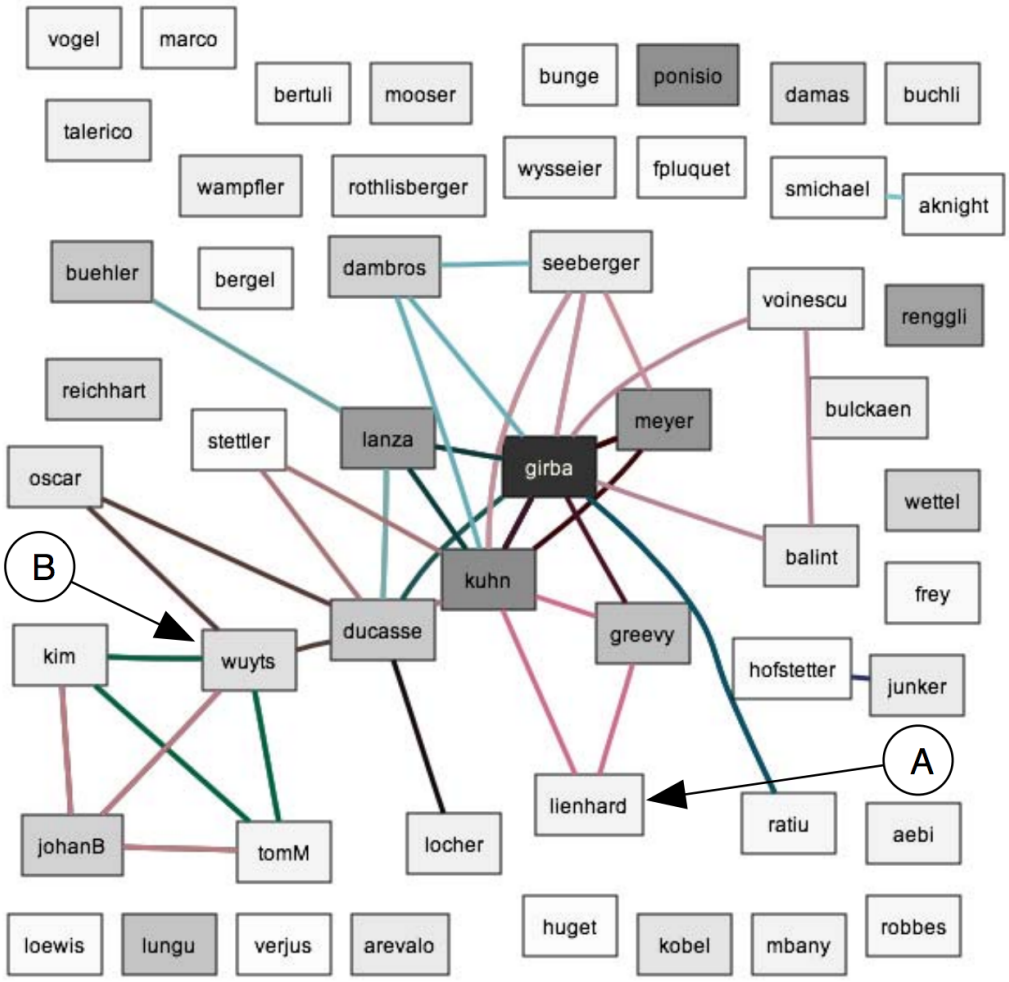
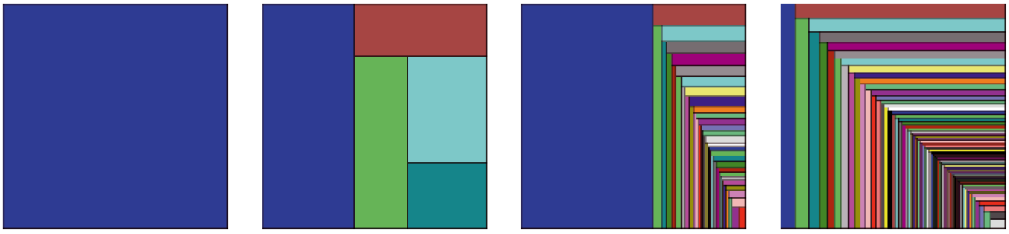


Figure 2.6 – The developer collaboration graph perspective [42].

The commit-based developer-collaboration graph shows how developers collaborate in events other than bugs. An undirected edge is added between two nodes (developers) if they work on a same file. Several graph-metrics are computed to analyze and characterize evolution, such as graph diameter, node ranks, average degree, or clustering coefficient.

D'Ambros et al. [27] use a data model combining system repository log files and bug reports of the studied software to analyze its evolution. The two data sources are parsed and the extracted information is stored in a database for analysis. The authors propose an approach consisting of four software evolution analysis types: developers' effort distribution, change coupling, trend analysis, and hot-spot detection. We are interested in the developers' effort analysis type, which concerns the

study of developers' contributions to a system during its evolution. The approach utilizes *Fractal Figure* visualization to represent developer-related information of a given software artifact. A fractal figure is composed of a set of rectangles of different sizes representing developers. Colors of rectangles are used to differentiate developers and their area is mapped to a structural (LOC) or evolutionary metric (number of commits performed by the developer). Figure 2.7 illustrates different development patterns. The visualization allows users to easily figure out how an artifact has been developed by visually identifying development patterns. For instance, detecting a major developer for an artifact is easily done by searching for the pattern in Figure 2.7(c). It is also useful for assessing the development-team formation by correlating development patterns with bug-related information.



(a) One developer **(b)** Few balanced developers **(c)** One major developer **(d)** Many balanced developers

Figure 2.7 – Development patterns based on *Fractal Figure* visualization [27].

2.1.3 Evolution Phases Identification

Xing and Stroulia [61] present an approach for understanding evolution phases and styles of object-oriented systems. The authors use a structural differencing algorithm to compare changing system class models over time and build the system's evolution profile. First, they classify structural modifications into five categories of change activities: active, rapid developing, restructuring, slow developing, and steady going. The number of structural modifications, such as class additions or method movements, are computed for an entire system's evolution and described with statistical descriptions (using quartiles). Then, each version of a studied software is associated with a change activity according to the structural modifications it undergoes. As a result, the evolution of the software is described using the change activities performed from one version of the system to the next.

Barry et al. [9] propose a method to identify software evolution patterns. The pattern identification is based on software volatility information. Volatility is approximated by computing the amplitude, dispersion, and periodicity of software changes at regular intervals in the software history. Each period is defined by a volatility class using the three metrics. For instance, if a period is characterized by low amplitude, long periodicity, and low dispersion of the changes, it is considered as least volatile because it has occasional small modifications occurring in a well-behaved pattern. Volatility classes are ordered by their degree of volatility, and vectors of these classes (one per period of time) are used to describe the software evolution. Volatility vectors are compared to a volatility vector representing a completely stable system, and stability distances are computed. Finally, vectors are grouped according to their stability distances to reveal evolution patterns.

Bennett and Rajlich [12] introduce a staged model for software life cycle. The model comprises five distinct stages: initial development, evolution, servicing, phase out, and close down. During the initial development, the first version of a software is developed. The main outcomes of the initial stage are the system's architecture and the knowledge acquired by the development team. The evolution stage's goal is to adapt the software to changing requirements and environment. Evolution stage allows developers and users to benefit from their past experience with the application. During this stage, important changes both in size and impact on the system, are made without deteriorating the software architecture. The next stage is servicing where the system is no longer evolving, and only small changes such as patches and wrappers are performed on the software to extend its life. During the phase out stage, no changes are applied to the system, but it may still be available to users. Finally, in close down stage, the system is no longer available to users who are redirected to a newer version. The authors also propose a variation of the staged model where, after initial development, each version of a system follows a different model.

D'Ambros and Lanza [23] use visualization to analyze software entities' evolution at different granularity levels. The authors represent bug-related information along side commit-related information. This production-related information is visualized using rectangular areas, representing time intervals, and color coded to show development activity, and sequentially ordered along a time axis to represent an entity's evolution in the software history. In order to improve visual scalability, the authors aggregate sequences of rectangles and define three types of entity evolution phases: stable, high stable, and spike, to help with visual scalability. Figure 2.8 illustrates the visualization of entity

evolution. The authors also define evolution patterns to characterize entity evolutions. for instance, artifacts can be persistent if they are alive for a large part of the system lifetime. There is also the day-fly entity, which lives for short periods of time with respect to the system's lifetime. If an entity was introduced in the system (first commit) after the first bug affecting it was reported, the entity is assumed to be introduced for fixing the bug. Furthermore, the stabilization-evolution pattern is associated with a living (still being committed) entity that had an intense development for revisions and bugs followed by stable phases. An addition-of-features pattern is characterized by an increase in the number of commits of the entity, as well as in the number of bugs related to it. An increase in the number of commits accompanied by a decrease in the number of bugs suggests a bug-fixing evolution pattern. Finally, the evolution pattern representing refactoring or code cleaning is characterized by an increasing number of commits while the number of bugs remains constant and low. These evolution patterns are easily recognizable using the proposed visualization.

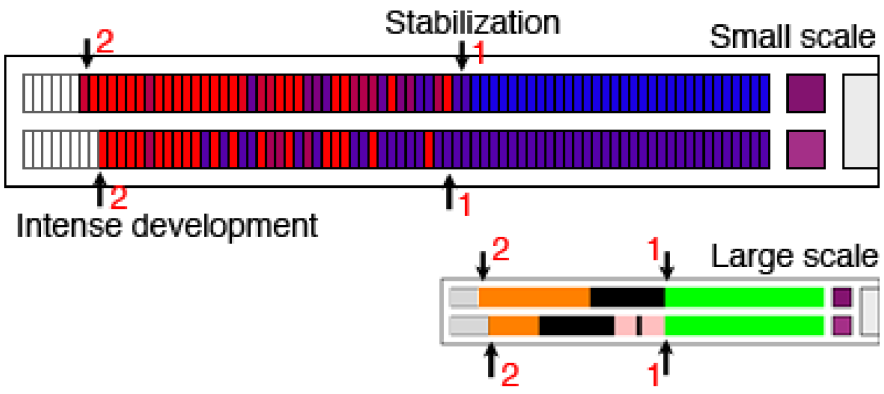


Figure 2.8 – Visualization of entity evolution with discrete time (top) and phases (bottom) [23].

2.2 Software Execution Comprehension

Execution comprehension is usually based on dynamic analysis techniques. It consists of understanding the behavior of a program by observing its execution. Software class contribution or involvement in the implementation of a use-case scenario is a popular research problem in the research community. Another such problem is the detection of execution phases that a program goes through at runtime. Here, we introduce contributions in software execution comprehension to illus-

trate how time is represented in this context. Then, we present some contributions that tackle the aforementioned execution comprehension problems.

2.2.1 Execution Time Representation

De Pauw et al. [25] propose an execution pattern visualization to help maintainers analyze a program execution. The execution patterns are constructed from interaction diagrams, where a diagram is represented as a tree to reveal execution patterns. The tree representation is unidirectional in both X and Y axes, and execution time follows the Y axis whereas messages between objects are laid out along the X axis. Execution events are depicted sequentially along the time axis, much like many behavioral diagrams. Figure 2.9 illustrates the execution patterns' visualization. The visualization is interactive and users can search for execution patterns using different criteria, such as the involvement of a class, an object, or a message name. It also provides means to simplify the visualization by collapsing, flattening, or underlaying the tree representation. Complementary information about the execution is presented to the user in form of charts or other diagrams. The user can use these techniques to visualize program execution step by step along the vertical axis.

Renieris and Reiss [49] represent data for temporal-trace execution using two views: a spiral view shows the complete trace, and a linear view highlights parts of the execution trace. They also propose ways to coordinate these two views and to take advantage of their respective strengths. Figure 2.10 illustrates the visualization's linear view. Function calls are depicted with horizontal bars representing activation time of the functions. Depth of function calls is represented on the vertical axis. The user can zoom in and out for viewing at different levels of detail. Each function call is assigned a color according to a predefined order. The authors present the spiral view to achieve better visual scalability of the execution traces. In this view, time is represented as a spiral that wraps the linear representation around itself. The two visualizations are connected by making their time windows centered around the same instance. Finally, the visualization is linked to source code and allows users to click on a view to show the corresponding source code and call stack.

Bohnet et al. [18] propose a technique for visualizing pruned execution traces, that supports programmers in comprehension tasks. The technique generates a linear view of a pruned trace that shows call similarities. Pruning of traces preserves the time ordering inherent to the original execution trace and represents it on the X axis. (see Figure 2.11). First, function calls are classified according to their

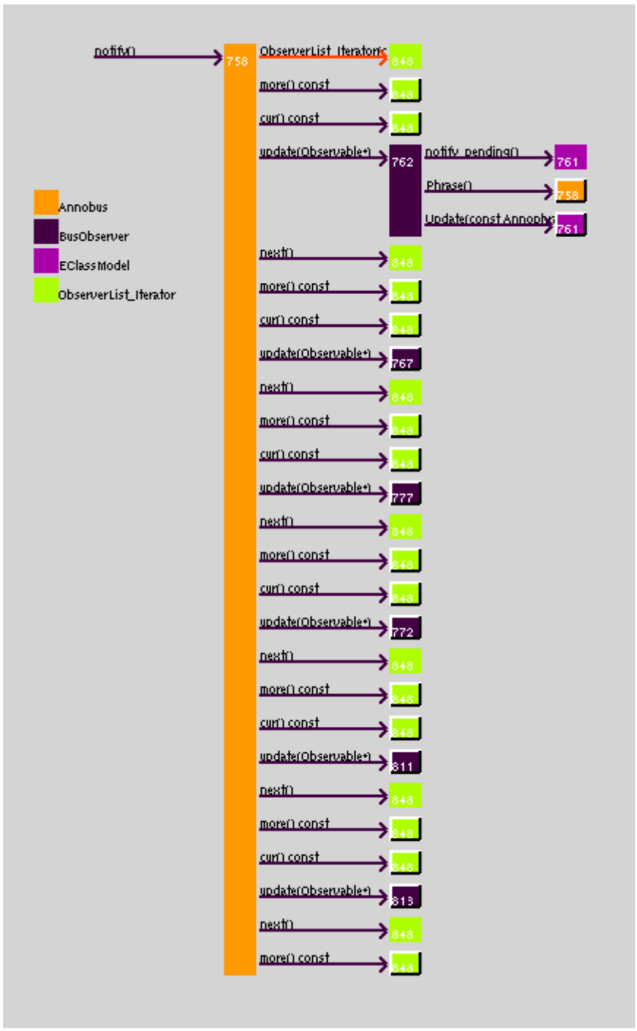


Figure 2.9 – The execution patterns visualization [25].

type: coordinating call, worker call, control delegating call, etc. Then, similarities between calls are detected and visualized with an emphasis on repetitive behaviors. The approach computes a similarity metric between two calls by calculating the number of distinct active functions over the maximum number of executed functions for the two calls. Two calls are considered similar if the similarity metric is higher than a threshold defined interactively by the user. Call similarity is used to compute repetitive patterns in the execution.

Kuhn and Greevy [38] propose to represent an execution trace as signals in time, and use signal processing techniques to analyze it. The authors consider a feature trace, consisting of a sequence

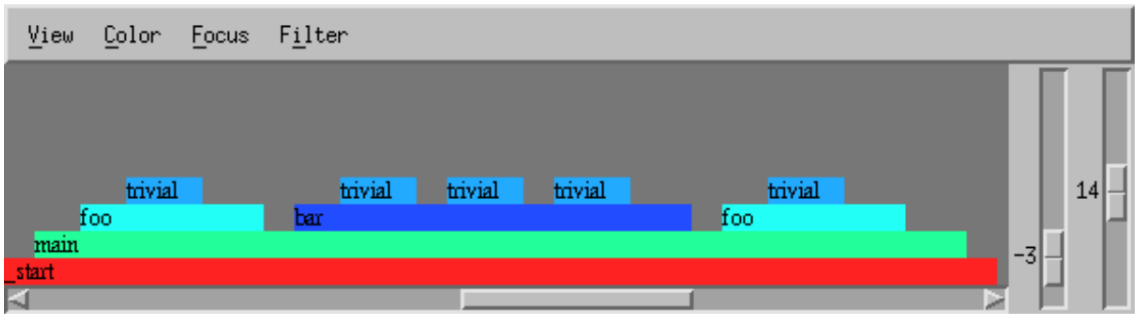


Figure 2.10 – The linear view of program execution visualization [49].

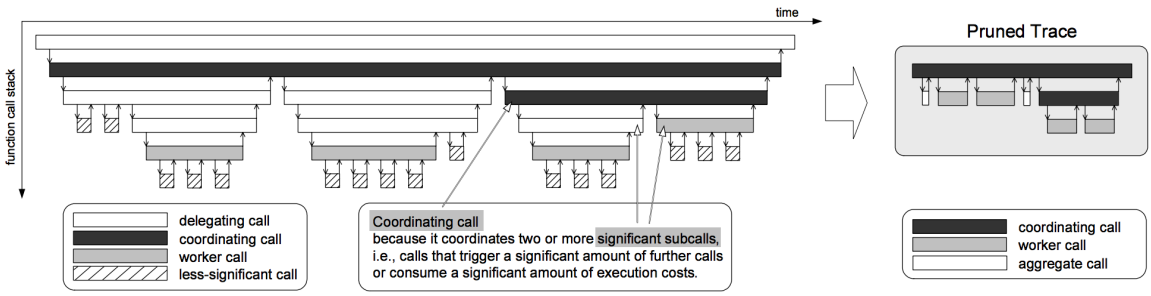


Figure 2.11 – Visualization of an original execution trace (left) and a pruned trace (right) [18].

of execution events (method calls), as a signal composed of monotone sub-sequences. The signal is cut when the depth of the call stack drops, i.e., when the execution reaches an event without children and resumes at its parent in the call graph. This results in sub-sequences of execution events where the nesting level either increases or is constant. These monotone sub-sequences of execution are then summarized by compressing them into one execution chain, saving a considerable amount of space. Figure 2.12 illustrates how the execution trace is transformed into a signal. The resulting signals, representing feature traces as time series, are visualized as time plots and color coded to show additional information, such as concepts of single feature classes. In order to compare different features, the authors use a dynamic time wrapping technique to measure similarity between their corresponding signals.

Dugerdil and Alam [26] visualize program execution using a city metaphor, and the software's time dimension is represented by animation. The authors split execution traces into contiguous fixed-duration segments of execution events, and count the number of class occurrences in these segments. For every execution segment, each class is assigned one of three colors (red, green, blue) to represent

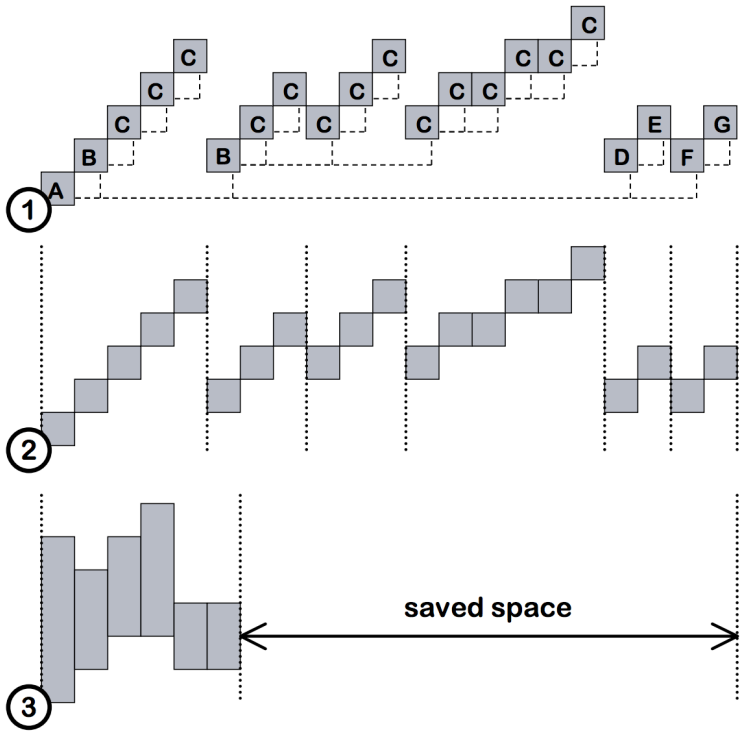


Figure 2.12 – Conversion of an execution trace into signal [38].

its degree (high, medium, low) of involvement in the execution period defined by the segment, as shown in Figure 2.13. The execution information encapsulated in segments is then visualized using scene animation in two views: macroscopic and microscopic. In the macroscopic view, each frame represents class occurrences in one execution segment. Buildings corresponding to classes are illuminated using associated colors. In the microscopic view, the authors focus on individual execution events (method calls) composing the segments. The method calls are represented by solid pipes linking buildings of caller/called classes. Figure 2.20(b) illustrates one frame at the macroscopic view.

Lienhard et al. [41] propose a dynamic analysis technique from the object flow perspective. An object flow is a visible path of its reference during execution. Object references are represented as a meta-model composed of an execution part and a static part, as shown in Figure 2.14. In the execution part the *Activation* entity, representing a method execution, denotes the execution time where the object reference is visible during program execution. The object flow analysis, proposed in this approach, is visualized using two views: inter-unit flow and transit flow. Inter-unit flow visualization allows users to track object transfers between classes by constructing a graph where nodes represent

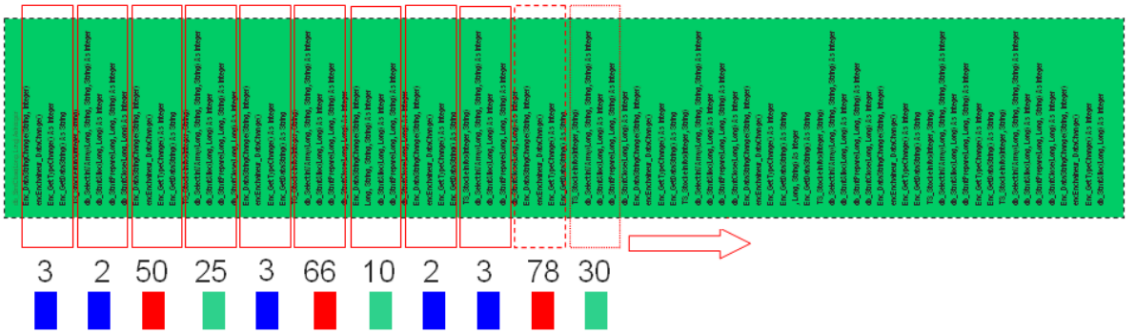


Figure 2.13 – Decomposition of an execution trace and mapping of colors to numbers of class occurrences [26].

classes and edges represent transfers of object references. The transit-flow view provides a detailed view of an object reference's visible path inside a given class. The approach considers object references as first class entities, and uses visualization to help answer questions such as how classes exchange objects, or which classes act as object hubs.

In the aforementioned contributions, time is represented as a sequence of execution events or as fixed-length periods. Elements of a sequence can be are aligned on an axis (Y axis [25] or X axis [18, 38, 49]), or they are used to generate frames representing states at different periods of time (see [26]). They can also be part of a higher-level model representing program execution (see [41]).

2.2.2 Entity Collaboration Analysis

Wong et al. [59] define three metrics to quantify the relation between software components and program features. First, the authors characterize features and software components using the same representation. A software component is represented by the set of basic blocks contained in it, i.e., sequences of consecutive statements or expressions with no control transfer present in the component's source code. A feature is represented by the basic blocks executed when it is exercised. Second, the proposed metrics make use of this common representation to measure disparity between a feature and a component, the concentration of a feature in a component and the dedication of a component to a particular feature. Disparity is computed as the number of basic blocks exclusive to either the component or the feature, divided by the total number of basic blocks of the component and the feature. Disparity is higher if the feature and the component share fewer basic blocks. Feature concentration

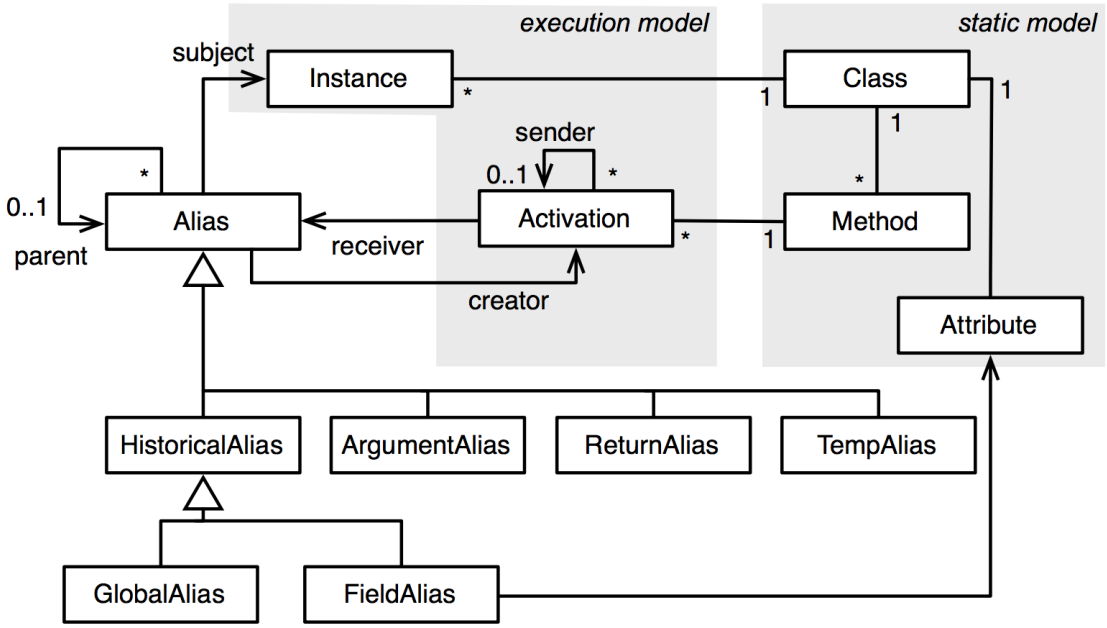


Figure 2.14 – The object flow meta-model [41].

is computed as the number of common basic blocks between the feature and the component divided by the number of blocks in the feature. A feature is more exclusively implemented by a software component if a majority of its basic blocks are also the component's blocks. Dedication is computed as the number of common basic blocks between the component and the feature, divided by the number of blocks in the component. A component is dedicated to the implementation of a feature if a majority of its basic blocks are also the feature's blocks. The quantitative approach helps in determining collaboration between components to achieve a feature.

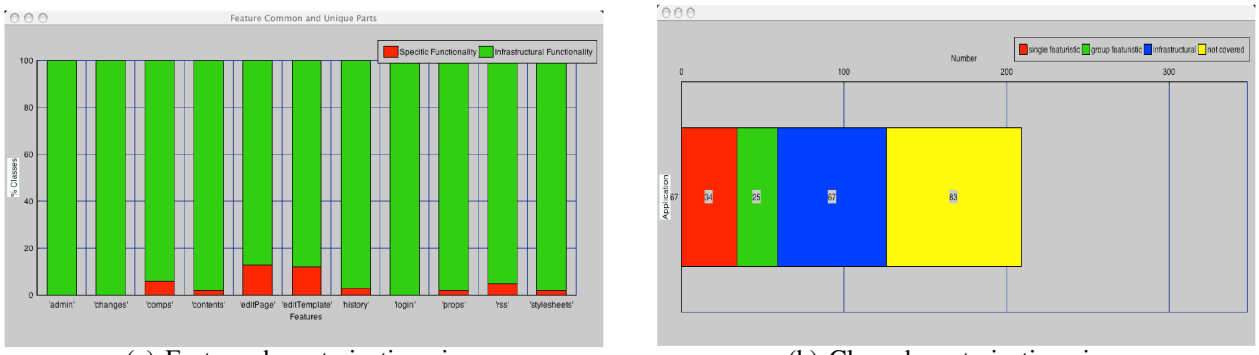
Zhang et al. [63] propose an approach to identify use cases from source code. The approach is based on the idea that use cases are separated in the source code by branching statements. The authors construct a *branch-reserving call graph* (BRCG), a call graph augmented with branching information. Relations within the BRCG are labeled as either sequential or branching. However, not all branching statements are used to differentiate between use cases in source code. The approach includes an automatic pruning technique that removes unimportant branching statements. This is done by assigning an importance weight to each node in the BRCG according to its type, depth, and number of direct sub-nodes. Nodes with weights lower than a chosen threshold are discarded. The

pruned BRCG is then traversed to generate possible execution traces by following the different paths of the remaining branching statements. Finally, the generated traces are analyzed and processed to remove repetitions and inadequate call sequences and construct a use-case model relating use cases to source code. The mapping of a use case to source code helps maintainers understand how functional requirements correspond to source code.

Zaidman et al. [62] study the application of web-mining techniques to execution traces for program comprehension. Their approach begins with the definition of precise execution scenarios representing program functionalities. Then, execution traces are collected using nonselective profiling and a compacted call graph is derived from the dynamic call graph of the traces. Nodes of the compacted call graph represent classes of the studied system, and their edges represent directed messages between class instances. Weights assigned to edges indicate the number of messages between two classes. The compacted call graph is used to identify coordinating classes and classes providing small functionalities. To this end, the authors apply the HITS web-mining algorithm on the compacted call graph. The HITS algorithm [37] identifies hubs and authorities on the web. Hubs are pages that refer to other pages containing information, and authorities are pages containing useful information. For each class, i.e., node in the compacted callgraph, the algorithm computes two metrics to measure its *authority* and its *hubiness*. A good hub is one that is related to good authorities, and a good authority is one referred by many good hubs. The authors claim that hub classes are good candidates for beginning the comprehension process because they play a pivotal role is a system's architecture.

Greevy and Ducasse [32] present an approach to explicitly map features to classes. The approach rests on the characterization of features and classes using specific metrics. First, a selected subset of features are exercised and their execution traces are collected. A fingerprint of each feature is computed and consists of information about classes and methods involved in the implementation of the feature, for instance, the sets of classes and methods involved in the feature execution trace, the set of classes and methods exclusive to the feature, i.e., not involved in any other feature, and the sets of classes and methods involved in other features as well. Feature-fingerprints are used to characterize features as either disjoint or completely, tightly, or loosely related. The authors also characterize classes of the studied system in a similar manner. They measure, for each class, the number of appearances in a feature execution trace and the number of features referencing the class. Classes are then characterized as infrastructural, single-feature, group-feature, or non participating classes ac-

cording to values of the aforementioned metrics. Characterizations from the two perspectives (feature and class) are then visualized as shown in Figure 2.15(a) for features and Figure 2.15(b) for classes. Finally, the mapping between features and common classes implementing them is illustrated in Figure 2.16.



(a) Feature characterization view.

(b) Class characterization view.

Figure 2.15 – The characterization views from the two perspectives for *SmallWiki* [32].

2.2.3 Execution Phases Identification

Pirzadeh et al. [46] exploit the sequence of method calls during program execution to reveal phase shifting within an execution trace. They claim that executed methods tend to disappear as the program enters a new execution phase. They construct a set of methods invoked as the program executes. This working set of methods is updated at regular intervals of execution events, i.e., after a fixed number of method invocations called chunks. The methods are ranked according to their prevalence, which is used to decide if they are disappearing from the execution, i.e., if a phase shift is occurring. The prevalence of a method takes into account its frequency, the chunk number of the method's first appearance, the current chunk number, and the chunk size. The ranked methods are compared after each chunk to the original working set (of the execution's first chunk) and a phase transition is detected when difference between the two working sets exceeds a given threshold. Once a phase shift is detected, the next step is to identify the exact chunk where the new execution phase begins. This is the chunk from which a majority of methods start to fade. To locate this chunk, the authors compute the mid-rank value of each method in the execution, i.e., the midpoint between the method's highest

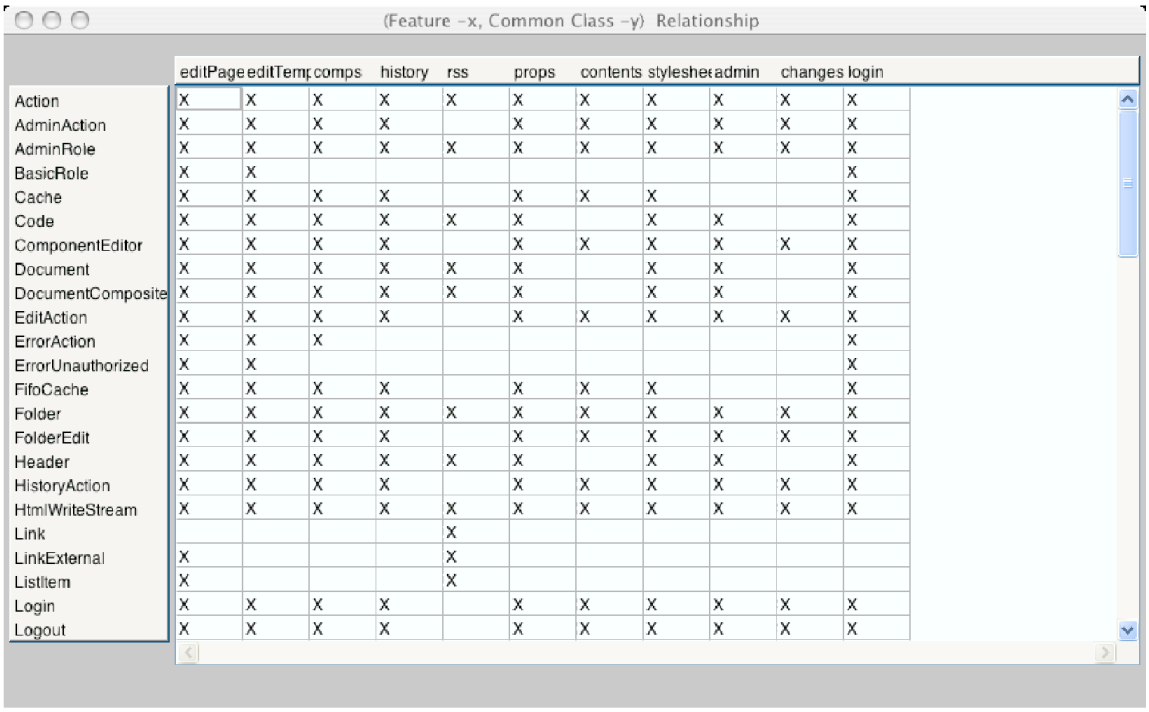


Figure 2.16 – The view of the correlation between infrastructural classes and features of *SmallWiki* [32].

and lowest ranks in all chunks. Then, they identify, for each method, the chunk where the mid-rank value is closest to the method's rank in that chunk. The chunk where most methods ranks are closest to their mid-rank values is the location of the phase transition. For example, Figure 2.17 illustrates an execution trace where a phase transition is detected at chunk 8 and the exact start of the new phase occurs after chunk 6, where all methods of the original working set (A, B, C) disappear from the execution trace.

The work by Watanabe et al. [56] employs information from objects to detect phase transition. An execution trace is collected according to a use-case scenario and processed to retrieve the objects' information used to identify feature-level phases. The authors use the assumption that different functionalities use different sets of objects. The authors keep an updated list of least-recently-used objects, called LRU cache. A new phase is detected if the change frequency of the LRU cache is higher than a given threshold. For each method call of an execution trace, the cache is updated after and its change frequency is computed based on the ratio of added objects within a window of execution events preceding the method call. If a phase transition is detected, the starting event of the new phase

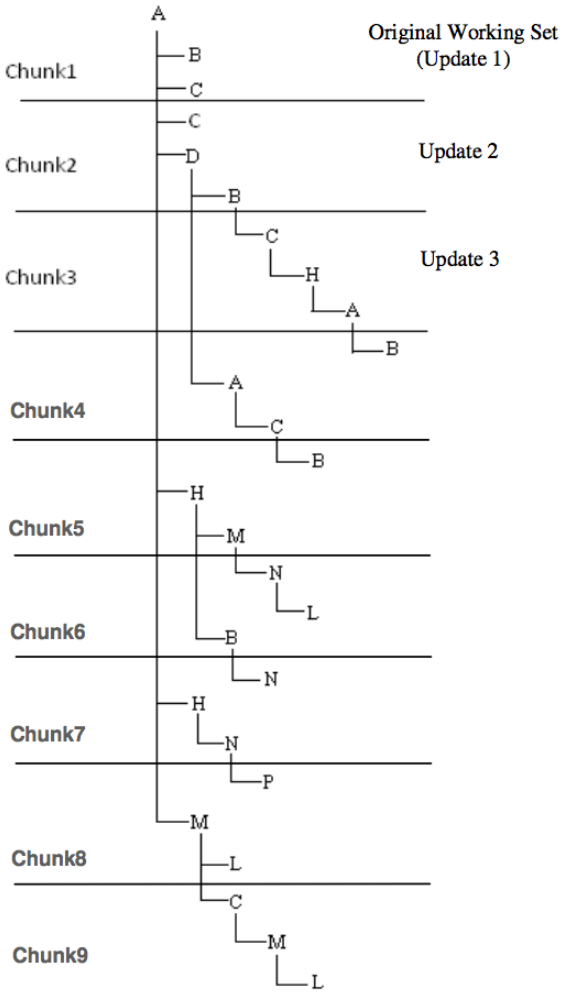


Figure 2.17 – An example of a call trace divided into chunk of size three [46].

is searched, among a given number of events prior to the phase transition, for one with the lowest call stack depth. The phase detection algorithm used in this work has four parameters: LRU cache size, the window size for change frequency computation, frequency threshold, and number of execution events considered for the identification of a phase's exact starting event. The resulting phases may represent the features exercised in the use-case scenario or parts of them.

Cornelissen et al. [21] apply visualization techniques to identify and understand program features. They propose two views of execution traces: (1) a massive sequence view as a UML-based view, and (2) a circular bundle view that utilizes hierarchical edge bundles to represent dependencies occurring during execution. Figure 2.18 depicts the tool *Extravis* that implements the two views. The circular

view shows the software's hierarchical structure and the relations between software entities during execution. The method calls are represented as hierarchical edge bundles with color codes for the direction or the chronology of calls. The massive sequence view shows execution events on the vertical axis and individual method calls horizontally. The tool allows the user to select a subset of consecutive execution events and zoom for a more detailed view. The presented visualizations help users understand program execution by exploration, feature location, and feature comprehension. For instance, the massive sequence view helps in detecting major phases of a program's execution by visual analysis of similarities between execution patterns. Then the user can locate a particular feature by zooming in a phase in the massive sequence view. Finally, the identified feature can be further investigated in the circular view.

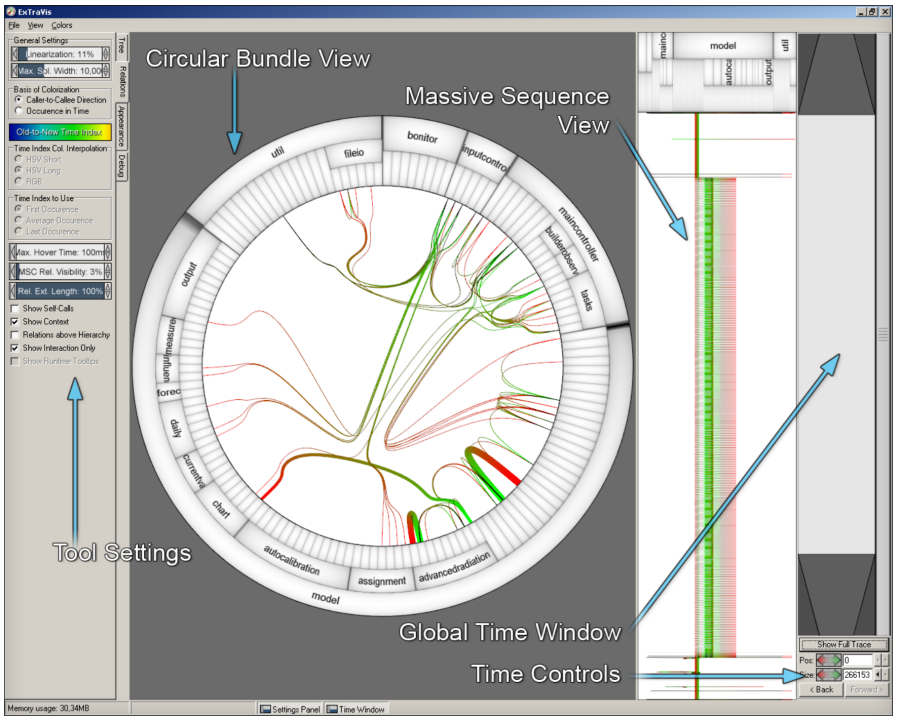


Figure 2.18 – Visualization of an execution trace of the system *Cromod* using *Extravis* [21].

Reiss [48] proposes an automatic approach to detect program execution phases. TDynamic data is collected and processed during program execution (online processing). Execution metrics are computed, at regular intervals of time, for each class or group of classes. Therefore, each execution period is characterized by the number of method calls per class. Reiss determines a phase switch when the

similarity value between a number of successive execution periods' share similar dynamic data. A vector representing the number of methods calls for each class is constructed for each execution period, and a cumulative window vector is also constructed by adding and normalizing the vectors in a given window preceding the current execution period. Each cumulative window vector is compared with a cumulative vector representing the entire current phase using the dot product. If the result is higher than a given threshold, the current window is considered as part of the current phase, otherwise, a phase transition is occurring. In the case of phase transition, the current cumulative window vector is compared to vectors representing previously detected phases for possible similarity, which means that a previously seen execution phase is happening again. If the current cumulative window vector is neither similar to the current phase nor to any previous one, a new phase is starting. Phases are displayed using JIVE [47], as shown in Figure 2.19. The horizontal scroll bar, used to browse over the program execution time, encodes colors representing phases. Color changes when the user navigates from one phase to another, but remains unchanged when navigating within an execution phase.

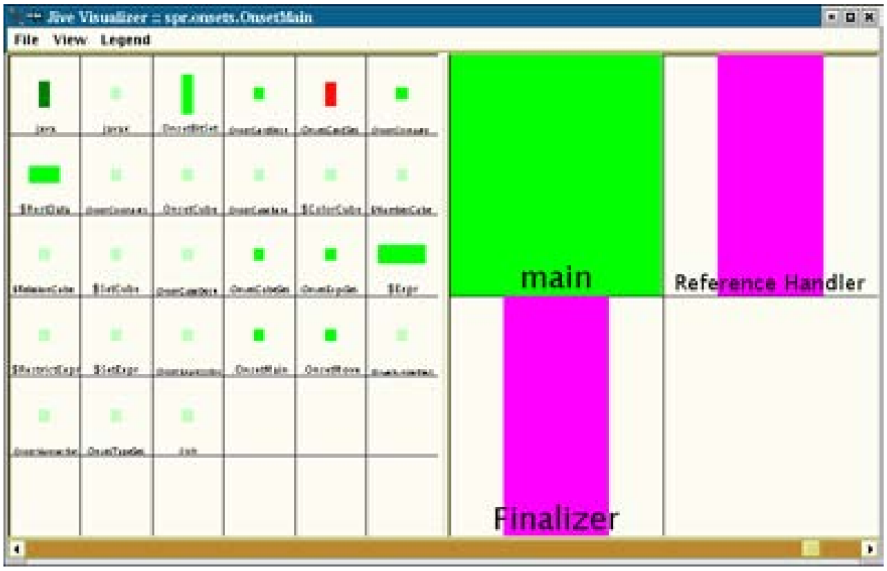
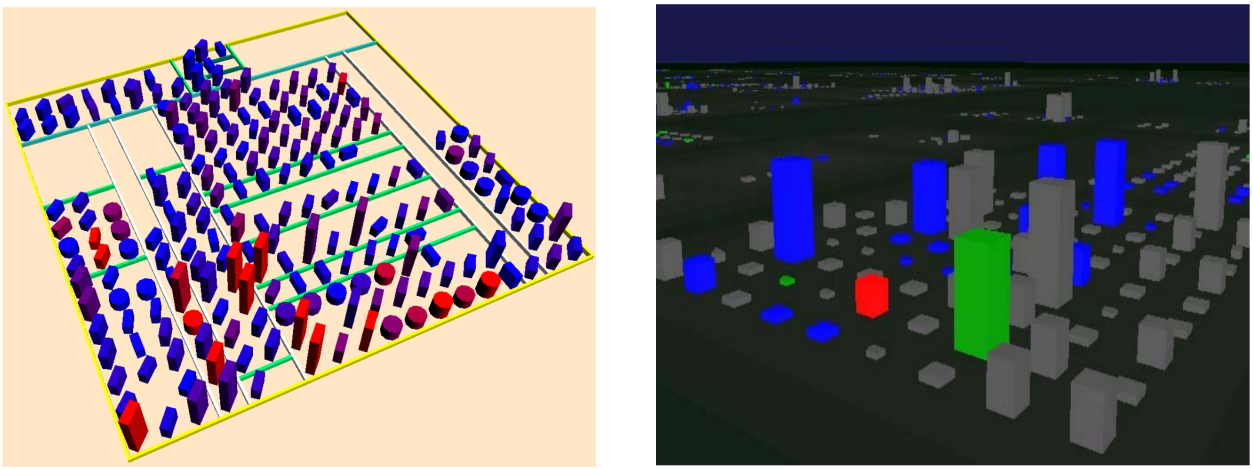


Figure 2.19 – Visualization of an execution of the set-theory game *OnSets*. The execution is in the phase *Computing final configuration possibilities*, which is assigned an orange/brown color on the horizontal scroll bar [48].

Asadi et al. [7] identify concepts in execution traces using genetic algorithms. The proposed approach consists in five steps: trace collection, trace simplification, textual analysis of source code, and search-based concept location. First, execution traces are collected by instrumenting the studied system, according to execution scenarios with cohesive steps. Second, traces are pruned and compressed by removing the execution events that represent noise, such as mouse tracking methods in GUI applications and too-frequently called methods. Repetitions of method calls are also removed using a Run-Length-Encoding (RLE) algorithm to keep only one occurrence of any repetition. Third, source code of the remaining methods is textually analyzed using Latent Semantic Indexing (LSI) and is used in the evaluation of the solutions' fitness during optimization. Solutions are represented by bit-strings where 0 represents any method invocation and 1 represents the last method invocation of an execution segment. For instance, bit-string 000100001001 represents an execution trace containing 12 method invocations, split into three distinct execution segments with 4, 5, and 2 method calls respectively. The genetic algorithm searches for a near-optimal solution by maximizing execution segments' textual similarity and minimizing textual similarity between consecutive segments. Segments' cohesion is the average cohesion of textual similarity between source code of any pair of methods that compose it and the segments' coupling is the average similarity between a segment and all other segments in the trace. The obtained solution represents a mapping between concepts and execution scenarios.

2.3 Summary

The contributions presented in this chapter treat software comprehension in two different contexts: software evolution and software execution. They may appear different in nature at first, but when examined closely, they share many commonalities. For instance, the work in [40] and [26] use software visualization with the same metaphor (see Figure 2.20) to analyze software evolution and software execution respectively. Moreover, they both represent time, evolution time for the former and execution time for the latter, using animation to depict changes that software classes undergo over time. Other examples that show commonalities between the two software comprehension fields are the two considered software comprehension problems: collaboration analysis and phase identification. First, collaboration comprehension can be abstracted as understanding how different software entities or actors interact with each others to contribute to software over time. For example, see [59]



(a) *VERSO* visualization for evolution analysis [40]. (b) *EvoSpaces* visualization for execution analysis [26].

Figure 2.20 – Similar software representations used for time representation using scene animation.

for an understanding of classes' collaborations in implementing software feature at runtime, and [42] for developers' contributions and collaborations throughout software history. Second, phase identification contributions, both in evolution comprehension and in execution comprehension, approach the problem as determining periods in time (evolution or execution) where software exhibits similarities. We can see this, for example, in [56] for the detection of execution phases, and [9] for the identification of software evolution patterns. Finally, we can clearly observe, from the existing work presented in this section, that the two research communities share many similarities at different levels: (1) their representations of software's time dimension, (2) the comprehension problems faced by each community, and (3) the techniques used to solve these problems. These observations motivate our thesis statement, which suggests that unifying the two research communities is possible based on a common framework. Such unification could help facilitate the application of solutions from one community to problems in the other. Thus, the main goal of this thesis is to propose a common framework for software comprehension problems that involves understanding changes during evolution and execution.

Chapter 3

Unified Comprehension Framework

As mentioned earlier, the dimension of time in software comprehension appears in both program execution and software evolution. Much research has been devoted to the understanding of either program execution or software evolution, but these two research communities have developed tools and solutions exclusively in their respective context. In this chapter, we explain how a common comprehension framework should apply to the time dimension of software. We formalize this as a meta-model that we instantiate and apply to the two different comprehension problems.

3.1 Introduction

There is a consensus that program comprehension is a major challenge in software maintenance [22]. The comprehension task involves large resources dedicated to maintenance [45]. This has led to the development of many tools to support program comprehension by the maintenance research community. Such tools are designed to assist developers understand certain aspects of software.

Software changes over time in two ways: during its execution and during its evolution. On the one hand, dynamic analysis studies program execution by inspecting data collected during the execution of a program in order to reveal insights about the actual behavior of the program during its execution. It is precise because it analyzes genuine changes that a program undergoes during its execution time. On the other hand, developers often also need to understand the evolutionary path of their software. This understanding allows them to make informed decisions about upcoming software maintenance tasks. Evolution comprehension involves the analysis of software history to understand the changes it sustained over time. There are many comprehension problems associated with program execution, such as identifying periods of time where a program executes in a similar manner, or class contributions to a particular execution scenario. There are also various comprehension problems with regards to software evolution, such as identifying developer collaborations during the evolution and the detection of periods of time where a software evolves in a similar way. Each of these comprehension problems has been tackled in a distinct manner within its research community. For instance,

Bhattacharya et al. [17] investigate the use of a graph-based characterization of software to capture its evolution, and analyze developer collaborations at the commit and bug-fixing levels. In the execution analysis research community, Dugerdil and Alam [26] present a program execution visualization tool that relies on the segmentation of execution traces for information reduction. The resulting reduced traces are visualized in 3D for analysis. In the evolution analysis research community, Wettel and Lanza [58] use the same visualization, the city metaphor, to study software evolution. Also, Barry et al. [9] study the problem of evolution comprehension by revealing evolution patterns of software. They classify evolution of several software systems according to their change volatility. From the execution perspective, the problem of phase identification arises when understanding execution parts where the program performs a high-level feature, such as file I/O operations, sending emails, etc. For instance, Watanabe et al. [56] propose an approach to detect execution phases that relate to the execution of program features.

These comprehension problems have several commonalities, even though they have been studied by two separate research communities that rarely share each other's solutions. There is a clear differentiation of time dimensions of software comprehension in both research communities. However, the representation of time for comprehension problems has also shared common concepts through many contributions. For instance, we can distinguish between two distinct software visualization communities. First, execution visualization, interested in the depiction of execution states of software over time, and understanding its dynamic behavior (see [24, 49]). Second, evolution visualization, interested in the representation of software changes from one version to another (see [40, 58]). These two communities treat the visualization of software's time dimension in a disconnected manner. However, they share common concepts of software's time dimension representation. This differentiation is also reflected in research for the comprehension of software execution and evolution using automatic approaches. On the one hand, some approaches study program execution by applying automatic algorithms to reveal valuable information from execution traces (see [7, 46]). On the other hand, some automatic approaches investigate the comprehension of software evolution to gain insights on how software evolves during its lifetime (see [61]). Although these contributions rely on the representation of time in software, there is no explicit model for its representation, and if there is one (see [31]), it is only for one of the time dimensions.

Our goal in this chapter is to propose a common comprehension framework to reconcile software's time dimensions. The framework typifies common grounds for representing time in software along two dimensions: execution and evolution. The framework is described using a meta-model for the comprehension of software's time dimension.

3.2 Unified Comprehension Framework

As explained earlier, the comprehension of many aspects of software during its execution and during its evolution requires some representation of time. We propose to formalize a time-comprehension framework that offers a new unified perspective for many common problems studied in both execution and evolution research communities. Such framework enables the efficient application of solutions and advances from one community to the other, and vice versa. The originality of this comprehension framework lies in the idea that many approaches in program execution understanding and software evolution comprehension already make use of similar models, often implicitly, to solve their respective problems. To the best of our knowledge, the unification of comprehension problems of different natures in appearance, but similar in reality, has never been addressed before.

In order to express the unification of time dimensions in software comprehension, we establish a unified meta-model, as depicted in Figure 3.1. The meta-model is based on our study on how time is represented in software comprehension problems both in evolution and execution and describes the main components present in many software comprehension problems involving time. The *sequence* is the main component of comprehension problems. It represents the entire studied period of time, with a *start time*, an *end time*, and *events*. The sequence contains *events* that appear as actions occurring periodically over time. An *event* has a time stamp, is triggered by a *subject*, and has an impact on *objects*. *Subjects* and *objects* constitute the two *entity* types involved in the comprehension process. *Entities* are characterized by *properties*, that are modified by *changes* introduced over time by *events*.

3.3 Application: Collaboration Comprehension

Another software comprehension problem comes in the form of understanding collaborations between entities over time. It can also apply to execution and evolution. We instantiate our meta-model to represent the problem, and use the instance model to solve it. Each entity has a certain activity

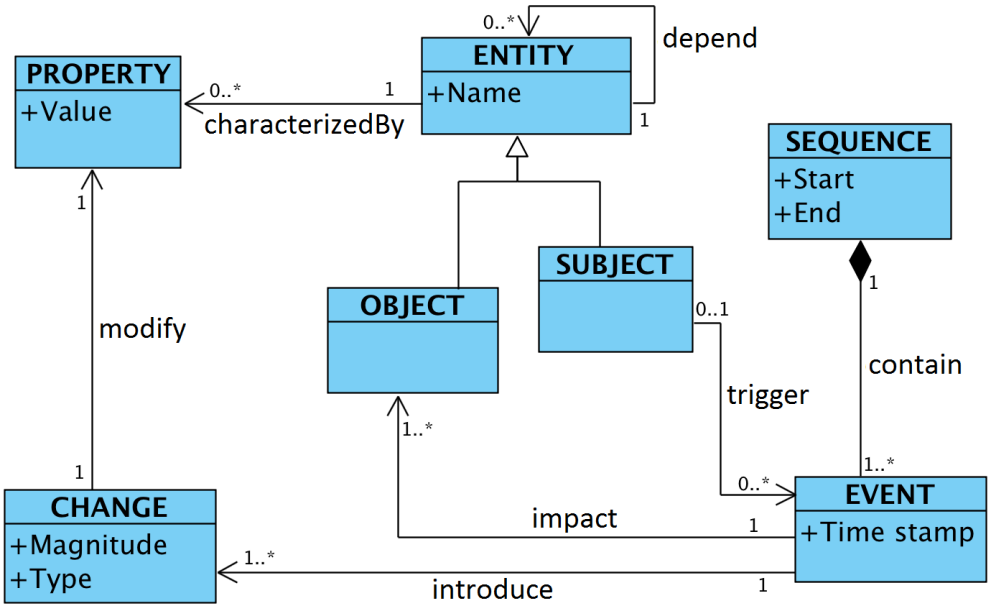
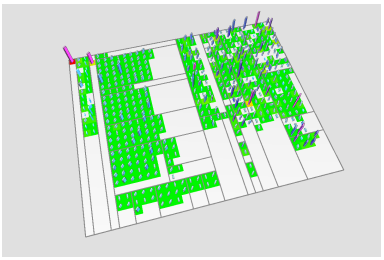


Figure 3.1 – The unified meta-model for software's time dimension comprehension.

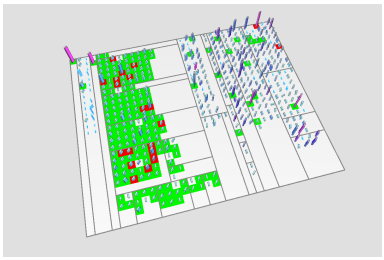
over time, that can be represented by the change introduced by the event it triggers (in the case of subjects), or by the event it is impacted by (in the case of objects). The events have time stamps and hence can be aggregated over time to illustrate the common activities of the entities. We use the same semi-automatic approach, i.e., software visualization using the same metaphor (heat distribution), to understand collaboration of developers in software evolution, and also the contributions (collaboration) of classes in the execution scenarios that they are involved in. The visualization used in both execution and evolution uses the city metaphor combined with heat maps. Entity contributions are visualized using heat maps on top of our software visualization. We provide tools to aggregate and compare multiple heat maps: First, color weaving allows combining different heat maps into one. Second, flipping the order between overlaid and aligned heat maps helps to visually reveal differences. Finally, multiple opened heat maps allow visual exploration of differences and similarities between colored elements.

3.3.1 Evolution: Developer Collaboration

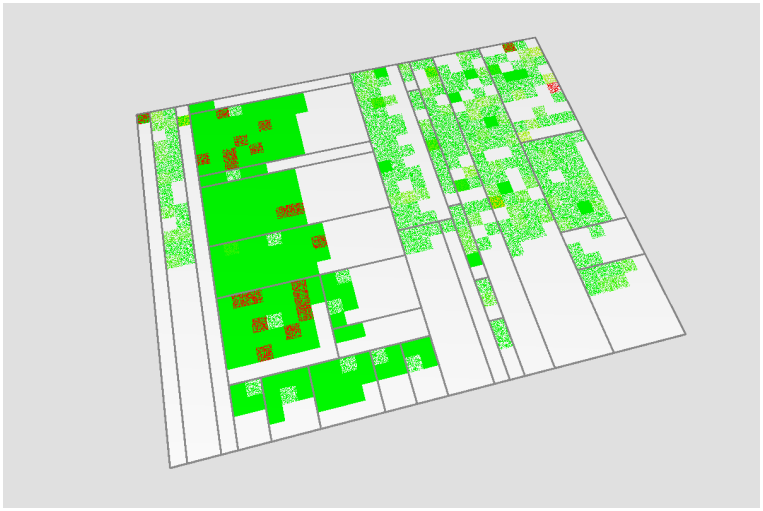
In evolution comprehension, *developers* are the subjects who commit changes to the software. Each *commit* is an instance of an event that introduces a change in the source code of a class. *Classes*



(a) *mrfloppy* contribution.



(b) *mtnygard* contribution.



(c) Combined contributions.

Figure 3.2 – The contributions of developers *mrfloppy* and *mtnygard* to *JHotDraw* 6.0.1.

are objects in our meta-model and their *source code* is a property. We use a visual representation of software to show the contributions of each developer to software code. These contributions are depicted in the form of heat maps where the intensity of changes is mapped to a gradient of colors. Then, the techniques of collaboration analysis mentioned earlier allow us to easily answer questions that developers ask [30], such as "who is working on what?", or "who changed this class?".

Figures 3.2(a) and 3.2(b) show heat maps that represent the contributions of developers *mrfloppy* and *mtnygard*, respectively, to *JHotDraw* version 6.0.1. The contributions of these two developers are combined to bring out their collaboration. The combination of the two heat maps uses color weaving, and the resulting heat map shows their common activities, as illustrated in Figure 3.2(c). Chapter 4 explains the approach in more details and presents the results of a case study.

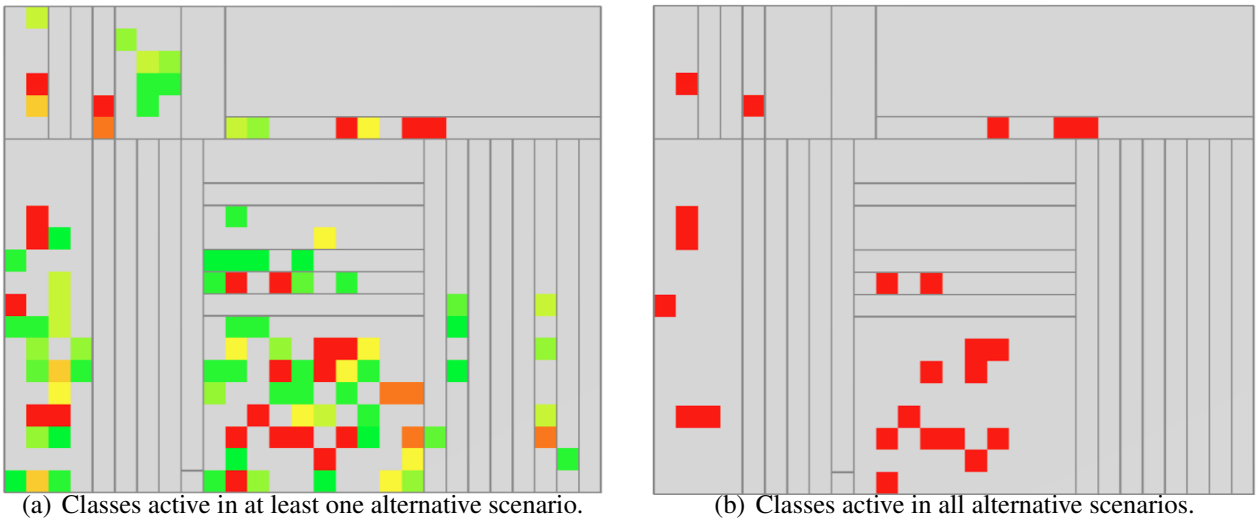
3.3.2 Execution: Class Contribution

The comprehension tasks considered in the execution problem concern class contributions within a use-case scenario. Several classes intervene during execution of a use-case scenario. We use our heat map visualization to understand the collaboration of classes in the execution. We could thus easily identify core classes in use-case scenarios, relate classes to specific features, and study how classes collaborate to the execution of a use-case scenario.

Classes are instances of objects in our meta-model. Events are represented by the *execution of a method* in a class, and are characterized by the instant of execution. Each event introduces a *change in the execution state*. In order to help comprehend class contributions during execution, we generate heat maps that depict the class activity in the use-case scenario. Hence, a heat map shows the mapping of a color gradient to class activity for each use-case scenario. We also provide a mean to aggregate class contributions of several alternate use-case scenarios into one heat map to identify core classes, i.e., classes that intervene in many alternate use cases. Other comparison options are also available, as explained in Section 3.4. For instance, to identify core classes, we generate a heat map aggregating several alternative use-case scenarios. The colors are mapped to the number of alternative use-case scenarios involving a class activity, i.e., the red color is associated to a class active in all alternative scenarios and the green color to a class active in only one scenario. Figure 3.3 illustrates how we identify core classes of the use-case scenario *Inbox actions* by generating the heat map aggregating eleven alternate use-case scenarios (Figure 3.3(a)) and filtering red classes (Figure 3.3(b)). These classes, in red, are active in all use-case scenarios involving the actions performed by clicking the email's inbox buttons. Chapter 4 presents the approach and results of the heat map visualization of the classes collaboration problem.

3.4 Application: Phase Identification

The phase identification problem arises in comprehension problems for both execution and evolution. Concepts of the unified meta-model are used to model the problem. Phases are detected by partitioning the sequence into several event sequences. Each event sequence must satisfy an internal cohesion property. Phase cohesion may be quantified by metrics that compute similarity of events, entities, and their properties within a phase. Also, consecutive phases, i.e., subsequent parts in the



(a) Classes active in at least one alternative scenario. (b) Classes active in all alternative scenarios.

Figure 3.3 – Heat map representing the aggregation of eleven alternative use-case scenarios of *Inbox actions* use case.

trace, must be as dissimilar as possible to ensure low coupling between phases in terms of entities involved and their properties. As explained in Section 3.1, findings from two different communities (see [9, 56]), which do not co-reference each other although they were published only a few years apart, both tackle the phase identification problem. To highlight the proximity between the two communities, we consider the resolution of phase identification problems from the two communities, using our meta-model and a similar technique. We model phase identification as an optimization problem, solved as a partition of a trace into sub-sequences of events according to some heuristics, which are translated to metrics to evaluate the solution's quality.

3.4.1 Evolution Phases

The *trace* of a software is represented by its evolution history, which is composed of several commits; *commits* correspond to instances of the events in our meta-model. The problem of the identification of evolution phases becomes therefore one of partitioning the evolution history into periods of time that are similar in terms of classes changed and of the nature of changes that these classes underwent. A *class* is an instance of an object in our meta-model. Class changes are used to evaluate the phases' cohesion and coupling. We use a mining strategy to collect the changes reported

by the commits on the source code of the classes. Here the *source code* of a class at any given commit (time) refers to its property in our meta-model. We use the classification proposed by Fluri and Gall [28], where change types are affected by an ordinal scale, according to their change impact and functionality preserving/modifying characteristic, that corresponds to five categories: *Crucial, High, Medium, Low*, and *None*. To collect the change data, we use *ChangeDistiller*, a tool that applies the change distilling technique of Fluri et al. [29]. Hence, we gather the significant levels for every change between any two consecutive commits (i.e., events in our meta-model) of a class. The problem is expressed as an optimization problem, and a meta-heuristic is used to find a good solution, i.e., a good decomposition of the evolution history (trace). Phase cohesion is computed as the similarity between changes to the same class within a phase. Phase dissimilarity is computed as the ratio of different classes between a phase and its subsequent. We applied our phase detection technique on the evolution history of five systems: *ArgoUML*, *JFreeChart*, *ICEfaces1*, *ICEfaces2*, and *ICEfaces3* that span long developments periods and several thousands commits. Results show that our algorithm was able to rediscover all cut positions (i.e., events marking the beginning of a phase) due to releases, without any information about releases. Therefore, a change in evolution phases exists from one release to the next one. We also identified a number of phases within a release development period, which suggests that the process of building a software release also goes through distinct evolution phases. Chapter 5 details the application of the phase identification approach, and presents evaluation results.

3.4.2 Execution Phases

We consider the entire execution trace of a particular use-case scenario of a studied software. Here, the sequence in our meta-model is instantiated by the execution trace. The trace contains execution events, such as class instance creations, method calls, and method returns. The goal is to decompose the execution trace into phases representing the executions of high-level features. Phase identification is formulated as an optimization problem, which is solved using a meta-heuristic algorithm, where a solution is a decomposition of the trace, and the quality of a solution is based on metrics of phases' cohesion and coupling. Phase cohesion is computed as the similarity between class instances involved in the execution phase. Phase dissimilarity is computed as the dissimilarity between class instances involved in two successive phases. We applied this approach for the phase

identification of seven execution scenarios with two software: *JHotDraw* and *Pooka*. The results were compared with a manually tagged reference. For each execution scenario, events marking the beginning of the execution of a new feature were recorded and used as reference for the evaluation. We evaluated our solutions with the reference by computing precision and recall, both in terms of phases and in terms of events involved in the phase. We detected execution phases with event precision over 80% and phase precision over 65%. The details of the application of the execution phase detection algorithm are covered in Chapter 5.

3.5 Summary

We presented a common framework that unifies software's time dimension comprehension both in execution and evolution. To this end, we formalized a meta-model that encompasses the abstract concepts required for the comprehension problems both in program execution and software evolution. Then, we considered two different comprehension problems that involve software's time dimension, and used an instance of our meta-model to solve them. Each problem was described in a generic manner for both execution and evolution, and solved using the same technique: a meta-heuristic search for the phase identification problem, and heat map visualization for the collaboration analysis problem.

This novel idea suggests that understanding software's time dimension can greatly benefit from the same comprehension framework for both execution and evolution. Another example of a common problem that could profit from our framework, is the detection of events that introduce defects in software over time. In the execution context, it corresponds to the debugging problem; in the evolution context, the problem comes back to identifying one or more past events in the evolution history that introduced the defect. Consequently, the proposed unification idea opens new perspectives for comprehension problems from both communities, by allowing the application of solutions from one community onto problems particular to the other.

Chapter 4

Collaboration Comprehension

Interactive software visualization offers a promising support for program comprehension, including program dynamicity. We present, the extension of an existing visualization tool with heat maps to explore time and other software dimensions. To this end, we first illustrate how our framework is used to unify the two main software dynamicities, execution and evolution. Then, this unified framework is exploited to define a visualization environment based on heat maps. We illustrate our approach on two comprehension tasks: understanding the behavior of programmers during the evolution of an application, and understanding class contributions in use cases. The case studies show that the heat map representation contributes to answer, more easily, many of the questions important to program comprehension.

4.1 Introduction

Software visualization environments are increasingly used as software comprehension tools by the maintenance research community. Visualization of multi-dimensional data helps program comprehension by involving human analysts in data exploration without overwhelming them. Unlike automated tools, visualization allows free exploration without a predefined and hard-coded process. Several dimensions may be explored simultaneously, such as structure, quality, and bug tracking.

At the same time, much effort has been dedicated to consider the time dimension in program comprehension. For many tasks, it is essential to understand the dynamicity of a program, for example, from the point of view of execution or evolution. Nevertheless, it is difficult to represent efficiently the time dimension in a visualization tool. It is even more difficult when the time representation is combined with the representation of other dimensions. Roughly speaking, three types of approaches may be used to represent the dynamicity of a program: (1) different snapshots, corresponding to different time steps, displayed side by side (e.g., [58]), (2) an animated sequence displaying the program's state changes (e.g., [40]), and (3) aggregation of the data into a single view (e.g., [55]).

The existing research consider execution and evolution dynamicities as two different problems, irrespective of the approach used to represent them. Consequently, tools proposed in one community are usually not reused in the other. Nevertheless, the two dynamicity problems present similarities in many aspects, which suggests that their unification is possible.

In this chapter, we propose a first step towards the unification of execution and evolution software dynamicities. This unified framework is exploited to define a visualization environment based on heat maps. Heat maps are commonly applied on existing representations to display the intensity of a particular phenomenon with respect to the represented entities. We extend an existing visualization environment, VERSO [39], in which different dimensions are already represented. In our extension, heat maps are used either to visualize basic properties related to time or combinations of such properties. Our adaptation of the heat-map metaphor is not straightforward. Indeed, heat maps are commonly used on concrete representations where the entities' positions are meaningful, such as in meteorology. In our context, software is intangible. It is intended to be understood by humans and computers, and has no concrete reality outside of these purposes.

We illustrate our approach on two comprehension tasks: understanding the behavior of programmers during the evolution of an application, and understanding class contributions in use cases. These case studies show that a metaphor based on heat maps contributes to answer, more easily, many of the questions important to program comprehension.

4.2 Unifying Time Dimensions

Software dynamicity shows itself in two dimensions: execution and evolution, i.e., software changes over time while executing, and while being developed. This differentiation between software's time dimensions is echoed in the software visualization communities. We can distinguish between two visualization communities in the representation of software dynamics. First, execution visualization is interested in the depiction of the execution states of software and understanding its dynamic behavior (see [21, 49]). Second, evolution visualization is concerned with the representation of software changes from one version to another (see [40, 58]). These two communities treat software dynamicity visualization in a disconnected manner. There exist similarities in the two visualizations, which suggests their possible unification.

4.2.1 Examples of Software Dynamicity Problems

Here are two examples of software dynamicity problems, one for each time dimension stated above.

4.2.1.1 Software Execution Problem

Consider the task of understanding and analyzing classes' roles in different execution scenarios. We collect data about multiple executions that represent use cases with their main scenario and alternative scenarios. The main scenario describes the *normal* execution of the use case, while alternative scenarios detail the possible extensions and special cases of the main scenario. Each execution scenario brings into play several classes that contribute to its fulfillment. Classes intervene in an execution scenario at different degrees depending on their roles in the software. For instance, core classes are triggered in a majority of scenarios but with low frequencies and generally at the beginning of an execution. On the contrary, specialized classes appear in specific scenarios with high frequencies. When analyzing classes' contributions in program execution using visualization, one must consider the representation of the time aspect with respect to classes interventions in the execution. Also, the aggregation of different execution scenarios in one visualization is key to detect differences or similarities, such as core classes.

4.2.1.2 Software Evolution Problem

The second example of a software dynamicity concerns the study of developers' collaborations and contributions during software evolution. Usually, software development projects are run by teams of developers. Each developer contributes to the software with a certain degree. Also, several developers may concurrently contribute to software development at different periods of time. Developers operate on software classes and perform changes on them from one version to another. The developers' changes differ in importance, size, and frequency. The visualization of developers' contributions must take into consideration the evolutionary behavior of software as well as the sequence of changes made by developers on software entities. Furthermore, the visualization has to provide a mean of comparing and combining multiple developers' contributions.

4.2.2 Dynamicity Representation Framework

Software dynamicity as defined earlier can be described using a representation framework that defines its key elements. Here is such a framework:

Event: It is an action that occurs periodically during the time dimension. An event is triggered by subjects over time and causes the overall state to change.

In Example (1), an execution event occurs when a method in a class is executed. It is triggered by the execution of an instruction and causes the execution state to change. In Example (2), an evolution event is defined as a change in the software structure. It is triggered by developers who change the software at a certain time of its life cycle. The event is characterized by the importance of the change, its size, and the time of the change (version).

Entity: There are two types of entity: An entity that triggers the event (subject) and an entity that undergoes the event (object). Usually, the entity of interest is the subject as it contributes to the software. Subjects' contributions are evaluated on objects and both entities are central in the representation.

In both examples, the objects are the software classes, while the subjects are the use cases (1) and the developers (2).

State: It is the state of the software at a given moment. The subjects cause events that impact on objects. The change incurred by the objects results in an overall state change.

The execution event in Example (1) is triggered by the use-case execution (entity) that changes the execution state (call stack, objects and variables, etc.). In Example (2), the developers (subjects) contribute to a software by modifying its classes' (objects) code. Hence, the software structure (state) changes due to the event.

Entities' contribution over time: Each entity contributes to a certain degree to a software and it does that at different moments of the time scale considered.

The classes' appearances in the different use cases are an example of entities' contributions in Example (1). Developers changes to the software's classes over time constitute the entities' contributions in Example (2).

Aggregation of entities' contributions: The entities' contributions give valuable information for the analysis tasks at hand, but one often needs to combine several entity contributions to be able to answer analysis questions involving several entities' contributions.

In Example (1), the core classes' identification brings into play several classes' contributions (one for each scenario) that have to be aggregated to identify classes appearing in most use cases and early during execution. In Example (2), consider the task of studying the developers' collaboration and determining which classes are subject to contributions by several developers. Several developers' contributions must be aggregated to identify classes changed by multiple developers at a given time.

4.3 Visualizing Dynamicity with Heat Maps

Our primary goal is to analyze how different subjects (e.g., use-case scenarios or developers) contribute over a time period (e.g., execution or evolution) to the state change (e.g., execution time or code) of a given large-scale software. In our setting, we are interested in changes made to the states at different levels: classes, packages, and system. In the remainder of this section, we first show how heat maps allow displaying state changes. Three important aspects are discussed, in particular:

— *Integration with an existing visualization metaphor* to add dynamic information to other displayed software dimensions.

— *Choice of color schemes* to ease the perception of dynamic information despite the size of the studied software.

— *Package and class placement* to use a heat map to highlight regions of interest rather than the coloration of individual classes.

The second part of this section is dedicated to the combination of multiple heat maps to perform analysis that involves many subject contributions. Finally, the last part details the navigation features that help analysts in their tasks.

4.3.1 Representing Entities' Contributions

Entities' contributions, e.g., classes' participation in an execution scenario or classes' code change made by a developer during the evolution, are represented using heat maps. A heat map offers a convenient technique to visualize the software's time dimension, as it adds a visualization layer on top of the actual software visualization.

Integration with *VERSO*

In our work, we use *VERSO* as the visualization basis, and extend it with heat maps to represent the entities' contributions over time. As mentioned earlier, *VERSO* provides a visualization of a program (packages, classes, relationships between classes, etc.). A heat map augments this representation by applying a color gradient on the software visualization, which adds different information from the one already conveyed by *VERSO*. The heat-map visualization is orthogonal to *VERSO*'s visualization. It can be used on other software visualizations with minor modifications when 3D elements are placed on a plane (2D).

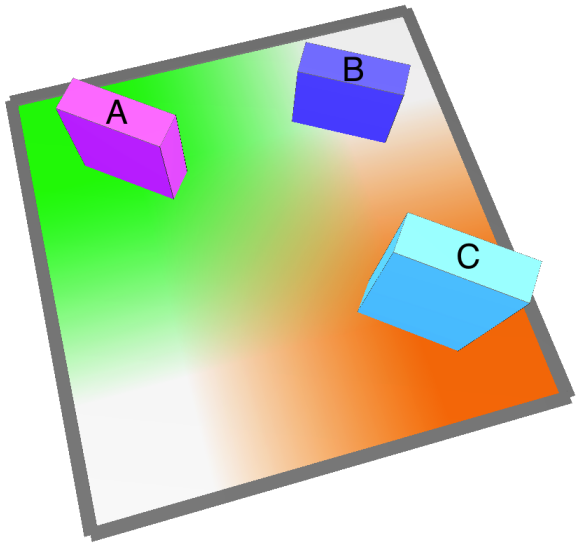


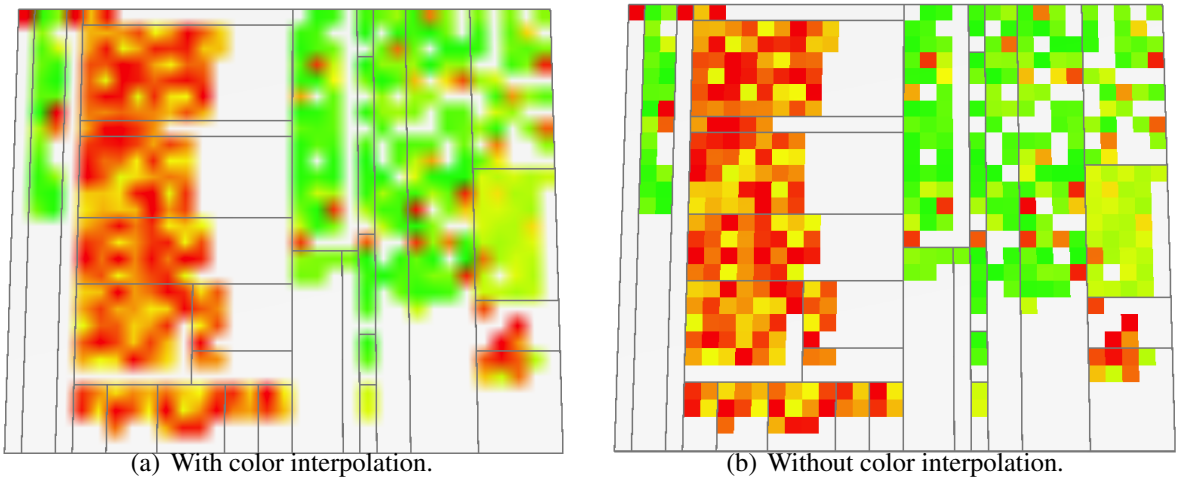
Figure 4.1 – The heat map is applied on the entire square region representing a package with three classes: A, B, and C. It shows that the state of B does not change, while the state of A changes with a lesser degree (green) than the one of C (red).

Given an object-oriented program composed of a hierarchy of packages, where each package contains classes, the root package represents the rectangle encompassing the entire scene and determines the size of the heat map. A heat map covers the entire rectangle of the root package and conveys the proper dynamic information about each class contribution by associating the class' data to a corresponding color, as shown in Figure 4.1. Hence, we consider the implementation of heat maps as color textures (2D array of color pixels, known as "texels" for texture elements). This implementation choice comes naturally considering the graphical properties that we want to visualize. A data array containing the information to be visualized is computed and used to generate a color texture with the appropriate colors at the corresponding coordinates. As stated earlier, heat maps are meant to represent the time dimension of a software, and as such, we introduce two ways of representing this dimension. First, the colors of the heat map may represent the time or age of a certain information, e.g., time since the last change made by a given developer to the classes. On a color scheme representing the heat map, color intensity represents the age of the last changes. Second, heat maps may represent accumulation events' effects triggered by the same subject. For example, color intensity represents the entire execution time used by the methods of each class. In some cases, we want heat map colors to be "plain" colors, for instance, to compare two specific colored regions. In other cases, the visualization requires color interpolation to give a visual impression of the phenomenon as a whole or to infer a sense of continuity in the dataset. We use bi-linear color interpolation between regions (see Figure 4.2). Color interpolation helps to highlight regions of interest by blurring the edges of individual adjacent colored cells.

Colors

A color texture is represented by an array containing the different computed colors. These colors must reflect the data distribution that they represent. For this matter we considered several color systems that have different visualization characteristics.

Color scales for uni-variate data, as used in heat-map visualization, should respect some desired visual properties [50], such as colors chosen to visualize ordered data values must be perceived as following the same order. We use the analogy of heat to produce this perception of order. Colder colors (green) are lower than warmer colors (yellow) which are lower that hot colors (red). The distance between two colors should also be representative of the distance between the two corresponding data



(a) With color interpolation. (b) Without color interpolation.

Figure 4.2 – Interpolation of heat-map colors.

values. Furthermore, clearly separated data values should be represented by distinguishable colors, and closer data values should correspond to variations of the "same" color. The color visualization should not introduce perception side effects. It should not create fake boundaries, i.e., if data values do not have boundaries, the colors should not suggest that there are some. Also, the color visualization must not convey a certain organization of data if it is not present in the data itself. For example, the arrangement of colors should not suggest clusters if the corresponding data do not aggregate as such.

Element Placement

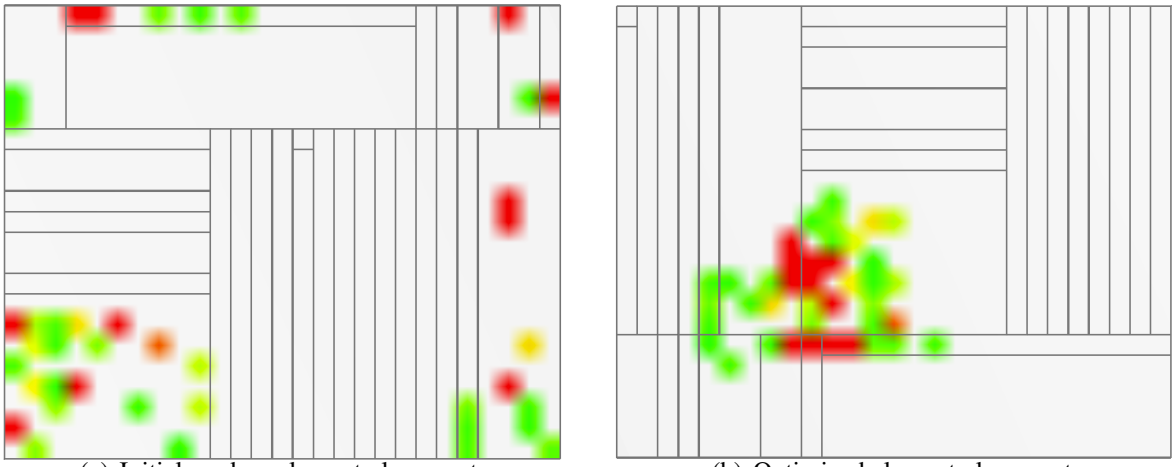
Usually heat maps are used on representations in which concerned elements have meaningful positions. In the case of software, the positions of classes and packages are not absolute and are determined to show a given property, such as the software architecture. The *Treemap* layout algorithm used in *VERSO* arranges software entities on a rectangular region. The algorithm operates on a tree structure to subdivide the rectangle into smaller regions representing software elements (packages and classes). It processes the software elements starting from the tree root and recursively traverses the tree, but the processing order of the node's children is random, or in alphabetical order, or in size of sub-trees, which either can be irrelevant to an observer. Therefore, we take advantage of the two

degrees of freedom (sub-packages and classes) to augment our heat-map visualization by enforcing particular element placements according to the visualization needs. In order to take full advantage of the heat-map visualization, we group classes that are part of the heat map, i.e., classes that have a data value to be displayed in the heat-map visualization. This element placement allows for a better analysis of the heat map because interesting elements that have similar values, and therefore similar colors in the heat map are put closer together. This was motivated by the fact that when analyzing a heat map, it is easier to have the interesting software elements closer to each others rather than scattered over the entire base rectangle. Hence, we consider element placements that minimize distances between all interesting software elements.

The search space covers all possible permutations of the sub-packages, and for each package, all possible classes positions within their parent packages. The size of this search space is prohibitively large and results in a combinatorial explosion. Therefore, a brute force or an exhaustive search would be very inefficient. Moreover, we do not necessarily need the optimal solution to element placement, as our goal is to improve the heat-map visualization. Therefore a near-optimal solution should prove satisfactory for our goal. For this matter, we view element placement as an optimization problem that could be solved using a meta-heuristic algorithm.

In order to search for a layout that minimizes distances between classes with similar colors in a heat map, we use a simulated annealing (SA) algorithm [36]. SA is a local search meta-heuristic inspired by the metal annealing process of metallurgy, where a crystalline solid is heated and then cooled down according to a cooling schedule until it reaches its optimal energy state, and thus is free of defects. For layout optimization, we start by an initial layout, and using a pseudo temperature with a cooling scheme, we simulate iteratively the state change by exploring neighboring solutions. The generation of a neighboring solution from a current one combines two strategies. First, we randomly choose a level in the software package tree structure, from which we select two sibling packages. We swap those two sibling packages' position, which does not alter the software structure (first degree of freedom). Then, we randomly choose two packages, not necessarily the ones chosen in the first stage. We select two classes from each chosen package, and we swap their relative positions within their parent package (second degree of freedom). Each candidate solution is evaluated using an objective function that computes the sum of the relative Euclidean distances between the classes involved in the heat map, and is compared with the current solution. Solutions that improve the fitness

are automatically accepted. Those with a fitness deterioration could be accepted with a probability that decreases along with the cooling process and the level of deterioration.

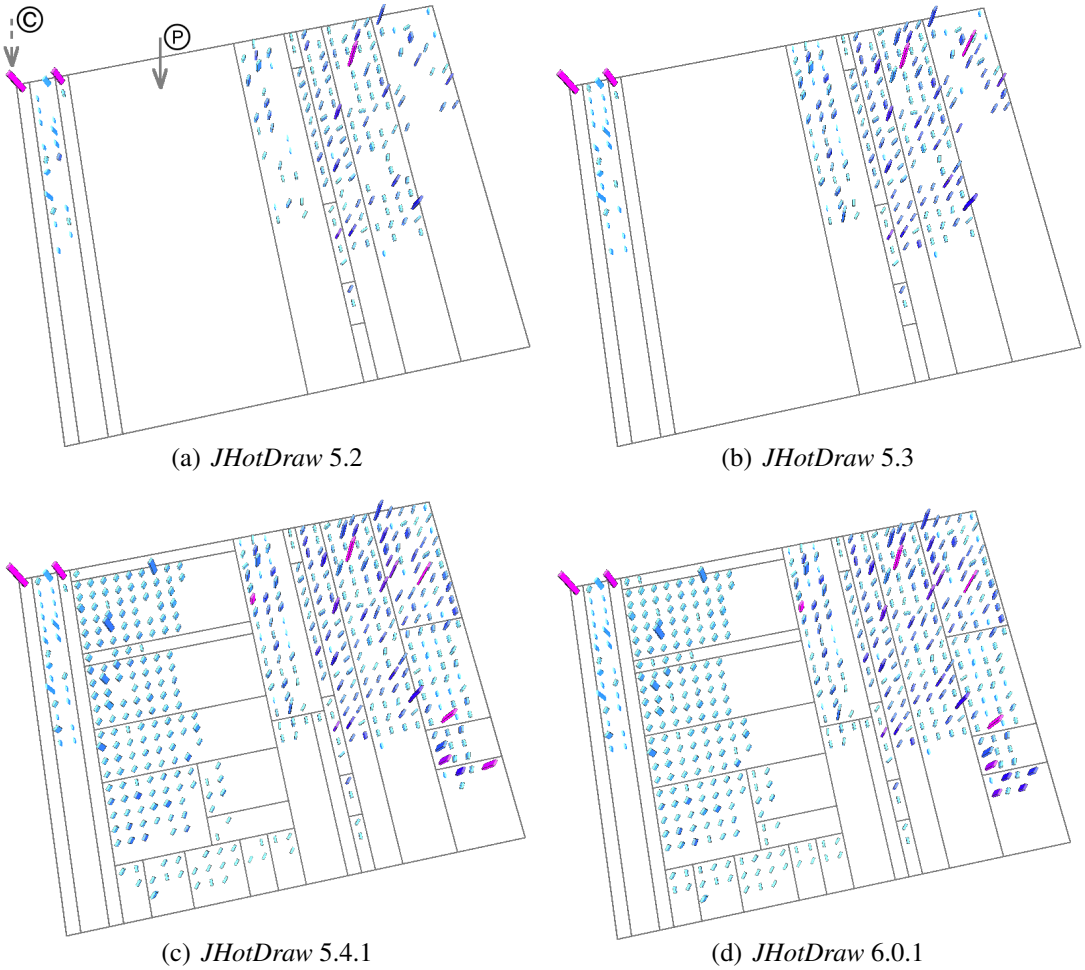


(a) Initial random element placement. (b) Optimized element placement.

Figure 4.3 – Element placement optimization starts with an initial solution that is then optimized by swapping sibling packages and sibling classes within packages (not necessarily the swapped packages).

Figure 4.3(a) displays a heat map of the method executions per class when using *Pooka* [2] system for a use-case scenario, where element placement is done with Treemap and random order of the sub-packages and classes. Figure 4.3(b) illustrates the same data with the optimization of element placement. The heat-map visualization is easier to analyze when classes involved, with the same degree, in the execution scenario are closer to each others. For instance, to compare two classes with similar colors, we should minimize distance between them, thus reducing visual scanning effort and potential visual perturbations. In addition, it becomes easier to perform an action of zooming on features if the area of interest is smaller.

Another placement issue is when a heat map displays data related to evolution. Indeed, software structure tends to change over time as from one version to another, classes and packages may be removed, added, or modified. The representation of these changes should not alter the visual coherence of the overall navigation and interaction with the scene. For instance, if a class is removed and another one is added at a given version, the added class should not be positioned at the location of the removed class to avoid confusion. The same is true for entire packages. To have consistent heat



(a) *JHotDraw* 5.2 (b) *JHotDraw* 5.3

(c) *JHotDraw* 5.4.1 (d) *JHotDraw* 6.0.1

Figure 4.4 – Fixed positions layout for four versions of *JHotDraw*.

maps, we use a fixed position layout [40] where all software elements remain at the same position during the visualization of each version. The elements' positions are computed for all versions at the beginning of the visualization by constructing a virtual tree representing all elements that exist at any version. The virtual tree contains the hierarchy level of the elements that is used for the treemap layout explained above. Figure 4.4 illustrates the fixed position layout computed for four versions of *JHotDraw* [1]. Package Ⓟ, which contains tests, is added only in version 5.4.1 and hence appears only in the last two versions, but its space is reserved since the initial version. Also, class Ⓒ remains in the same position throughout the entire *JHotDraw* evolution.

4.3.2 Aggregation of Entities' Contributions

For some comprehension tasks, it is necessary to analyze two or more heat maps corresponding to the contributions of different subjects. For example, one could execute different scenarios of the same use case and study the involvement of the classes for all the variations of the use case. To this end, we utilize three different strategies:

Multiple windows: Each subject's contribution is represented by a heat map, and rendered on a separate scene and window. The analysis is done by visually comparing the different scenes where the different contributions are depicted. Several interaction features are provided to facilitate navigation between the different windows (see Section 4.3.3).

Flipping: The heat maps representing the different contributions are all displayed in the same scene. However, only one heat map is rendered at any given time. The comparison of heat maps is done by switching the displayed heat map, only the heat map that is displayed on the surface plane is changed, the scene remains the same.

Color weaving: To represent multi-variate data (multiple heat maps), we tested color blending and color weaving. Both techniques use multiple color scales, one for each variable (heat map) to be visualized. Color blending consists of mixing the color values of the represented variables, thus resulting in one computed color. However, issues with color blending are the identification of individual variables and the resulting colors might not be meaningful. Color weaving is performed by representing the individual colors side by side in a higher-frequency color texture. We explored the use of color weaving to represent multi-variate data, as it performs better than color blending when the dimensionality of the data increases [34]. Figure 4.5 gives a close-up view of two different heat maps combined in the same view. The resulting texture is obtained by subdividing each individual region of a software element into a number of texels (100×100 texels, Figure 4.5 right) and by introducing noise with the location of the two colors of the same region in the combined heat maps.

4.3.3 Navigating in Views

VERSO provides an interactive visualization environment. Most software maintenance tasks are too complex to be completely automatic. Hence, human intervention is often needed during analysis

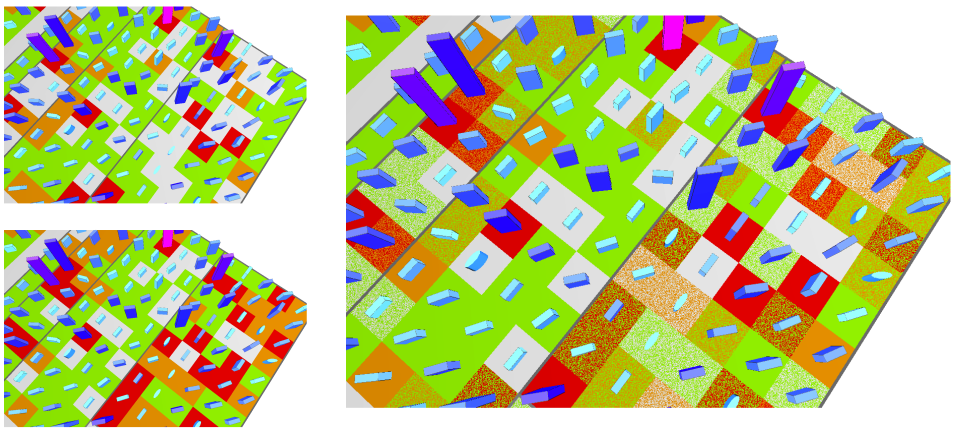


Figure 4.5 – The two heat maps on the left are combined in the heat map on the right using color weaving with a high-frequency color texture.

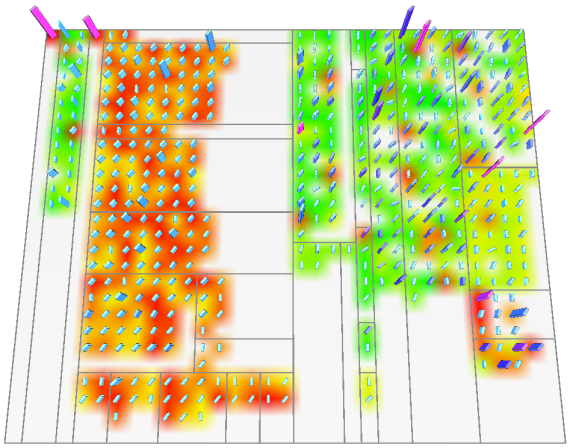
tasks. Heat-map visualization follows the same principle and allows the analyst to navigate within the 3D scene. Some navigation and interaction features are:

Camera: The 3D scene camera allows the user to change the point of view from anywhere in the scene, as well as zooming in and out of it. In the multiple windows view, a camera synchronization feature helps maintaining visual coherence between windows for comparison sake.

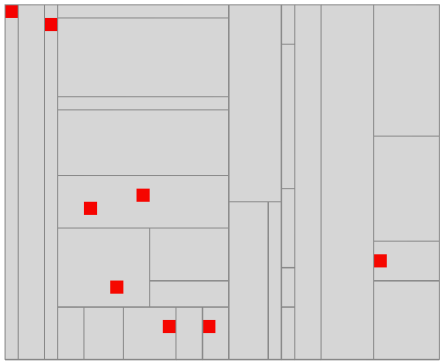
Color-scale manager: This feature permits to filter a heat map within a subset of interesting values. It also allows the user to re-map a range of values to a wider color range in order to better distinguish between closer values. Figure 4.6 illustrates these features.

Histogram: The color-scale manager interface provides a histogram of the values displayed in the heat map. This histogram gives an intuition about the distribution of heat-map data. It also gives the user extra information about high concentrations of values that may need to be filtered and re-mapped in order to be analyzed separately.

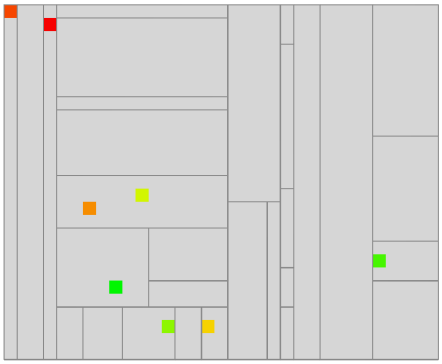
Navigation: Switching between several heat maps must be as simple and as fast as possible, because it is one of the most frequent operation used during exploration and analysis of data. This is associated with the up/down arrow keys and its result is instantaneous. We also use left/right arrow keys to switch between versions of software.



(a) No filtering.



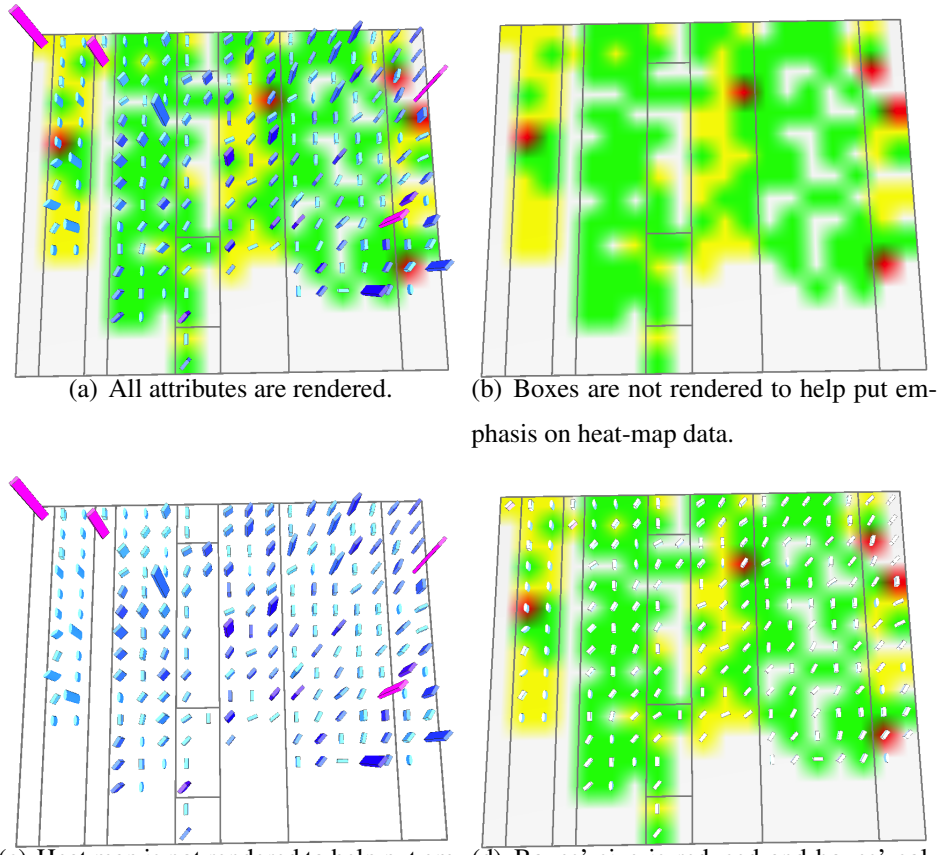
(b) Filtering.



(c) Remapping on a wider color gradient.

Figure 4.6 – The color gradient on the rectangle represents the time/age of changes accomplished by developer *mrfloppy* in *JHotDraw* 6.0.1.

Scene clearing: Despite our efforts to make the scene less cluttered, we sometimes need to clear the scene where all attributes are rendered (see Figure 4.7(a)) in order to better visualize specific graphical elements. For this purpose, the user may hide the 3D boxes and display only the heat map (see Figure 4.7(b)), and vice versa (see Figure 4.7(c)). The user can also keep some contextual information given by the 3D boxes desaturated colors and fixed heights, as shown in Figure 4.7(d).



(a) All attributes are rendered.

(b) Boxes are not rendered to help put emphasis on heat-map data.

(c) Heat map is not rendered to help put emphasis on 3D boxes data.

(d) Boxes' size is reduced and boxes' colors are desaturated to help put emphasis on heat-map data.

Figure 4.7 – Options for scene clearing and color interpolation.

4.4 Illustrative Case Studies

To illustrate the use of our heat-map visualization, we discuss two case studies involving two time-based software comprehension tasks. The first case study concerns the evolution of *JHotDraw* [1] software and the second one targets the analysis of *Pooka* [2] software's features. These illustrative case studies are an attempt to show the usefulness of our visualization technique in answering comprehension questions. A more rigorous study involving users would be more appropriate and will be considered in future work.

4.4.1 Evolution Comprehension

4.4.1.1 Objective

To understand some aspects of software evolution, an analyst needs to combine time-related information with other software properties. In *VERSO*, we used heat maps to represent time information of software evolution in addition to static software information mapped to 3D boxes. For the sake of illustration, tasks are defined as questions to answer for program comprehension.

4.4.1.2 Tasks and Data

During maintenance tasks related to software evolution, developers often ask recurrent questions. Fritz and Murphy [30] determined several such questions by interviewing professional developers. The developers' questions are organized by domains of information from which answers can be found, such as source code, bug reports, test cases, etc. Because we visualize program changes over time, we are interested in two domains: source code and change sets. Here are some of these developers' questions, related to these two domains, that we answered using our visualization technique:

A) Who is working on what?

B) What are coworkers working on?

C) Who changed this class?

D) What is the most popular class?

E) Who is working on the same classes as I am?

F) What classes have been changed?

G) Which class has been changed most?

We considered four versions of *JHotDraw* (5.2, 5.3, 5.4.1, and 6.0.1), a Java GUI framework for technical and structured graphics. The numbers range from 14 to 36 packages, and from 171 to 498 classes. The SVN logs of each version were extracted and parsed to retrieve information about the contribution of the different developers in each version. There are eight developers who contributed to *JHotDraw* over the four considered versions. The collected data have been organized into a matrix, where the first dimension represents developers, and the second represents versions. Such an organization permits to visualize a developer's contribution over several versions of a software with the

possibility to compare multiple contributions of the developers. The visualization techniques offer a way to represent the developer's contribution under different facets. For example, we can visualize the importance of the contribution as the proportion of changes made to different classes. We can also visualize the recency of the contributions. The modifications can be filtered by type: all changes, additions, modifications, and removals.

4.4.1.3 Analysis

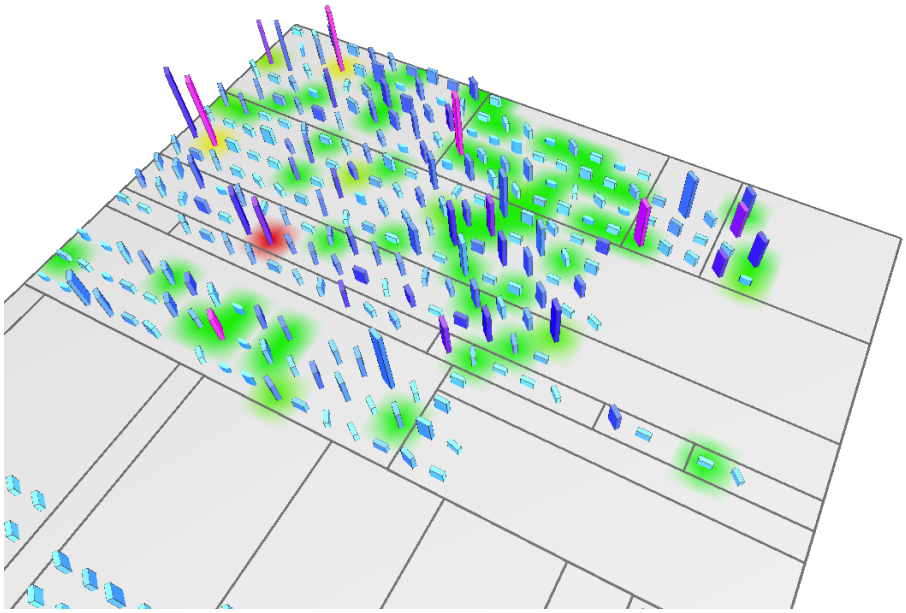
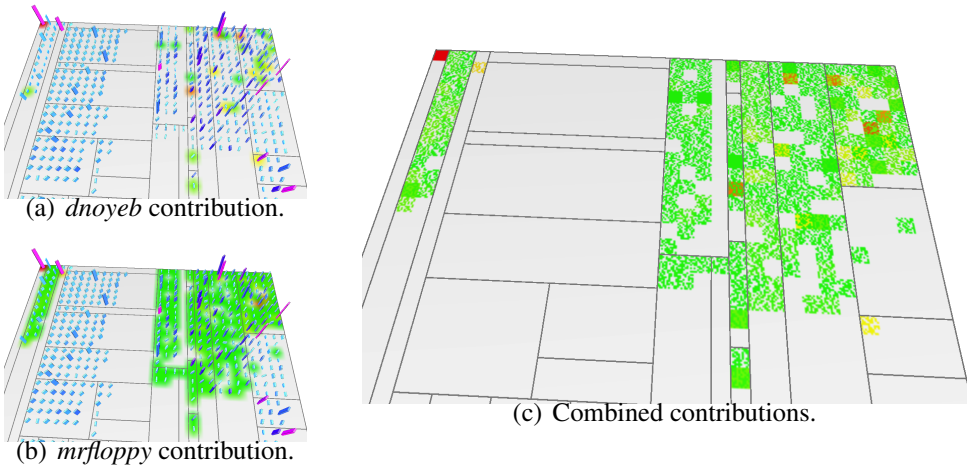


Figure 4.8 – *ricardo_padilha*'s contributions to *JHotDraw* 6.0.1.

We used our heat-map visualization within the *VERSO* framework to answer selected questions by generating heat maps representing the software's degree of changes, which corresponds to the extent to which the software differs between two time stages. The information about change is organized by developer and by version of the software, hence, a heat map corresponds to one developer who worked on a particular version of *JHotDraw*. Figure 4.8 shows the changes made by developer *ricardo_padilha* in *JHotDraw* 6.0.1. The changes include additions, modifications, and removals of classes. The colors represent the importance of the changes made. For instance, *ricardo_pardilha* made the most changes to class *org.jhotdraw.figures.TextFigure*, which appears in red in the heat map of Figure 4.8. To answer most of the questions with respect to one developer, we generate heat maps



(a) *dnoyeb* contribution.

(b) *mrfloppy* contribution.

(c) Combined contributions.

Figure 4.9 – The combination of heat maps representing the contributions of developers *dnoyeb* and *mrfloppy* to *JHotDraw* 5.4.1.

representing the developer's contributions. These heat maps answer question A. Question B can be answered by aggregating all these heat maps and visualizing the resulting heat map, which would include the contributions of all developers and their importance. For question E, we compare the heat maps one by one with the heat map representing the developer asking the question in order to identify classes in common. For instance, developers *mrfloppy* and *dnoyeb* have 26 classes in common in version 5.4.1. Figure 4.9(c) illustrates the contributions of these two developers to *JHotDraw* 5.4.1. The analysis is done by comparing the two heat maps. The comparison can be achieved either by generating two scenes side by side displaying each of the heat maps, as shown by Figures 4.9(a) and 4.9(b), or by flipping interactively between the two heat maps on the same scene, and noticing the differences, or by combining the heat maps in the same view by weaving their colors (see Figure 4.9(c)). We used a coarse-grained texture in this case (compared to Figure 4.5 right) to highlight classes in common, where there are two colors in the class region, in comparison to classes that do not appear in both heat maps, where the background color is weaved with one color. Questions D, F, and G can be answered in the same manner with the generation of a heat map that represents the aggregation of all changes made by all developers. This resulting heat map would show the most popular classes in red (for question D). Filtering this heat map to show only modifications will answer question F (all classes with a heat-map color) and question G (classes with a color red). Finally, to answer ques-

tion C, one can generate a heat map for each developer representing the age of changes rather than their amplitude. Then, the heat maps are compared side by side or by flipping. The responsible one is the developer with the heat map exhibiting the more red color for this class.

4.4.2 Execution Comprehension

4.4.2.1 Objective

We consider program execution from the perspective of time, and visualize it using heat maps. These are used as an exploratory technique, as well as for specific tasks, such as feature location or identification of core classes in alternative executions. Our goal is to help the user gain insights into a software without up-front knowledge, and also get insights into the software's features by performing alternative use cases.

4.4.2.2 Tasks and Data

We used our visualization technique to analyze features from *Pooka*. We performed a visual analysis of execution traces to understand classes' participation in different use cases. The tasks taken for the case study are inspired by the case study reported by Cornelissen et al. [21].

We collected the execution traces of software *Pooka*, an email client written in Java, using the Javamail API. There are 301 classes organized in 32 packages. We used a custom extraction agent written in C that utilizes the jvmti API to listen to events triggered at the method's entry or exit. Each time a method of our considered software is called during execution, the agent captures the entry event and returns information about the method, such as its parent class, the thread within which it is executed, its signature, a time stamp, etc. The same information is collected when the method exit event is captured. This information is then processed to be aligned for each method to produce a call graph with the execution time of methods. The execution time is relative, in the sense that nested methods' execution times are included in the outer method. The call graph generated includes multiple execution threads and can be traversed in the same order as the events were triggered during execution. It can also be traversed by following a particular execution thread. System method calls are filtered out as well as unnecessary events, such as mouse hovers, panels repainting, etc. After filtering, we organized the traces by use case; each use case regroups several alternative execution traces. We

recorded 37 traces of user interactions with *Pooka* and organized them in three main use cases: "*read mail*", "*inbox actions*", and "*search mail*". For example, use-case "*read mail*" includes an execution trace of opening an email without attachments, one of opening an email with attachments, and one of opening an email and searching for a word within the email, etc.

Several information types can be visualized using the techniques presented in this chapter. For instance, we can represent the age of each class in the execution stack. We can visualize the activity (cumulative execution time) of each class in an execution trace. Finally, we can visualize the occurrence of classes over several execution traces. These visualizations are then used to compare different use cases and execution traces.

4.4.2.3 Analysis

We generate a heat map for each execution trace representing an alternative use case. The resulting color gradient reflects the appearance of the class in that particular use case. Hence, every heat map shows the classes participating in the use case and its contribution to its execution. A color red indicates high activity within the use case, while a color green indicates low activity. For instance, "*open an email, add the sender to the address book, and close the email*" triggers 31 classes in seven different packages. Each class has a different level of activity when this use case is executed. Figure 4.10 shows class activity during execution of the use case. We can see the 31 classes contributing to the use case. There are eight classes with high activity (red) and the class *net.suberic.pooka.gui.dnd.DndUtils* is the one with the least activity in this particular use case. Another important task that our technique helps the user to perform is the identification of core classes that appear in several alternative use cases. This task is achieved by aggregating multiple alternative use cases and computing a heat map representing the number of class occurrences. The heat map indicates the number of alternative use cases where the classes appear. Core classes appear generally in several executions of the use case. The heat-map visualization technique enables the user to narrow down the search for classes responsible for minor changes in executions. For instance, consider the two alternative use cases: "*open email and close it*" and "*open email, add sender to address book, and close email*". We generate two heat maps representing the class activity of the two use cases, and we compare them with the techniques stated above. The result shown in Figure 4.11 indicates clearly that, for example, classes *FileResourceManager*, *ResourceManager*, *VcardAddressBook*, and *Vcard* (circled in blue) are active

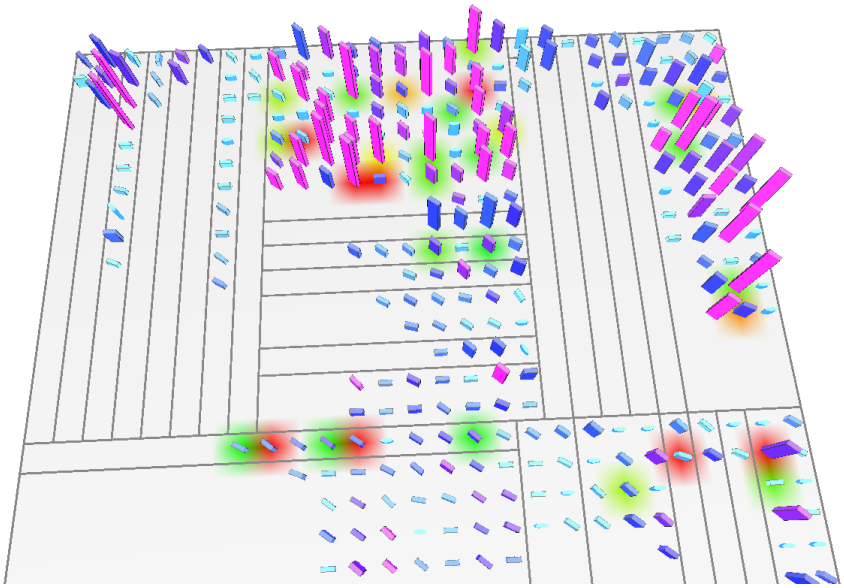


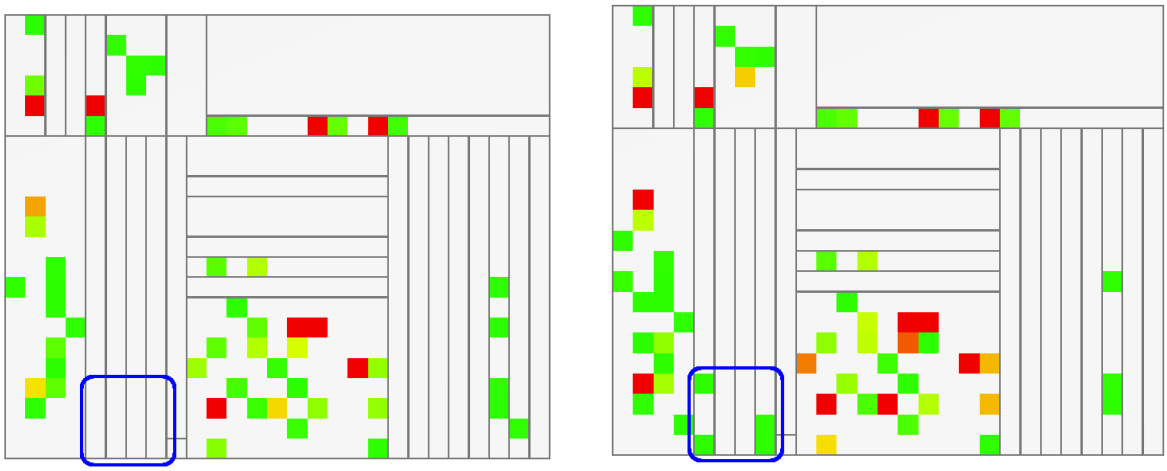
Figure 4.10 – Class activity in the use case "*open email, add sender to address book, and close email*".

in Figure 4.11(b), but not in Figure 4.11(a), which suggests that they are responsible, with a few others, for the treatment of "*add sender to the address book*". To identify core classes, we generate heat maps representing the classes' activity in the alternative executions. Then, we aggregate them and concentrate on classes that appear in red, i.e., classes active in most alternative executions.

4.5 Summary

In this chapter, we explored the visualization of software dynamicity. We consider software changes that occur during execution and evolution. The heat-map metaphor used in our visualization technique is suitable for the representation of a software's time dimension. Hence, in an attempt of unification, we applied the heat-map metaphor to the representation of software execution and evolution. Heat-map visualization allows the distinction between software's dynamicity and its overall context. It allows the developer to analyze the software's temporal dimension and to keep in mind the general context that is often needed to understand it.

In order to illustrate the use of our visualization approach, we performed two case studies on two different systems for the two software dynamicity aspects. The case studies illustrate that heat maps permit to answer practical questions and offer a simple and precise way about where to start the search



(a) "*open email and close it*".

(b) "*open email, add sender to address book, and close email*".

Figure 4.11 – Comparison of two alternative executions of a use case. Classes in the blue rounded square are active in Figure 4.11(b), but have no activity in Figure 4.11(a).

for an answer. Consequently, they corroborate our claim that the software's time dimension studied may be represented and analyzed using the same visualization metaphor.

Our studies have also revealed that there are still open issues when representing time aspects on top of other software properties. The color spaces used for the heat-map visualization are device-dependent models, and as such, do not relate well with the way colors are perceived. In particular, Euclidean distance between two color values in the color space should be proportional to their perceptual distance. This property is present in device-independent color spaces such as CIE LAB and CIE LUV, which are perceptually more uniform. Even though we did not feel penalized by this limitation, we plan to utilize more perceptually coherent color spaces, but still not completely satisfactory, in order to augment heat-map visualization. Another issue is the challenging problem of layout stability while working on software evolution. Our layout algorithm works when we consider previous versions up to the current version. However, for future versions, our algorithm would completely re-arrange the elements to consider a recent version. This would result in a major change in classes' positions from one version to another, and temporal coherence in the scene could be affected. We plan to tackle the layout stability issue in future work to preserve a certain coherence when the represented software's structure evolves. There are two main approaches to follow: (1) improve the stability of the

Treemap layout by using *Voronoi Treemaps* [8], which gains more stability by relaxing the rectangular subdivisions and allowing arbitrary shapes, (2) use another layout technique such as *EvoStreets* [52], which is also more stable due to its use of an incremental approach to incorporate new changes to the software's structure.

Chapter 5

Phase Identification

Identifying phases allows developers to have higher-level abstraction information from low-level data to help in software comprehension. The identification of execution phases and evolution phases is difficult because of the amount of data to be processed. Indeed, understanding a program from its execution traces is extremely difficult because a trace consists of thousands to millions of events, such as method calls, object creations and destructions, etc. Nonetheless, execution traces can provide valuable information, once abstracted from their low-level events. Furthermore, software evolution history is usually represented at fine granularity by commits in software repositories, and/or at coarse granularity by software releases. In order to gain insight on development activities and software evolution, information on releases is too general, whereas information on commits is prohibitively large to be efficiently processed by a developer. We propose to identify phases by casting our approach as an optimization problem and detect phases using a meta-heuristic search algorithm. First, we detect feature-level execution phases based on events collected from traces of program execution. We search through dynamic information provided by the program's execution traces to form a set of phases that maximizes similarity within a phase and dissimilarity of consecutive phases with respect to defined properties of execution events. Second, we use an optimization approach to find distinct software evolution phases that are characterized by similar development activities. A development activity is approximated by properties of the changes applied to the system during its lifetime. To validate our technique, we applied and evaluated our search algorithms on different execution scenarios of *JHotDraw* and *Pooka* for execution phase identification. For evolution phases' detection, we applied our technique on the evolution history of five systems: *ArgoUML*, *JFreeChart*, *ICEfaces1*, *ICEfaces2*, and *ICEfaces3*, each one spanning long development periods and several thousands of commits.

5.1 Introduction

Program comprehension is a difficult and resource-consuming task. It is also a necessary step in many software maintenance activities performed by developers. To address this issue, a grow-

ing program-comprehension research community actively develops techniques and tools to support maintenance. One such family of techniques deals with dynamic analysis, which helps in understanding behavioral aspects of the analyzed program. Another family of techniques deals with analyzing source-code repositories to comprehend software evolution.

5.1.1 Execution Phases

Dynamic analysis shows to developers information from a different perspective to help them better grasp how a program executes. This execution comprehension is crucial when analyzing a program because for many problems, dynamic analysis is more precise than static analysis, which relies on source code. However, this comes at a much higher cost in complexity. Typically, a program execution produces an execution trace that records execution events such as method calls and returns, object creations and destructions, etc. Usually, an execution generates thousands to millions of such events. This is prohibitively too large for a developer to even just look at, in order to gain a better understanding of the program. Fortunately, not all execution events need to be considered to grasp the dynamic behavior of a program. In fact, developers get a better idea of the execution when they get the "big picture" of the runtime information. For all these reasons, one should focus directly on useful parts of execution traces that relate to software functionality in order to reduce such complexity. Ideally, this abstraction should be established automatically by identifying in the execution, the phases that correspond to software functionality.

We propose an automatic approach for the detection of high-level execution phases from previously recorded execution traces, based on object lives, and without specifying parameters or thresholds. Our technique is simple and based on the heuristic that, to a certain extent, different phases involve different objects. We apply a meta-heuristic to implement our heuristic. In particular, we utilize a genetic algorithm to search for the best decomposition of an execution trace into high-level phases according to objects triggered during the program execution. To the best of our knowledge, it is novel to use a search-based approach for high-level phase detection. We used *JHotDraw* and *Pooka* as case studies for the evaluation of our approach, and identified execution phases on seven use-case scenarios of these two open-source software.

5.1.2 Evolution Phases

Software changes over time. Its source code is constantly updated and evolves over its lifetime. Software repositories record a trace of the evolutionary path taken to realize a software. An evolution trace usually consists of many commits, and spans several years of development. The commits represent atomic changes applied to software modules and hence hold evolution information with fine granularity. On the one hand, the amount of information for software development over several years makes any attempt to gain higher-level insights on the evolution of the software development very challenging for a developer. On the other hand, software evolution can also be represented by information about the different releases. Release notes, when rigorously documented, include information such as bug fixes, updated features, new added features, etc. Release notes are not always available and even when they are available the information included is usually at a too coarse grain to deliver a good picture of the evolution stages of the studied software. A good way for developers and managers to gain insights about how software evolved in the past is to have a high-level representation of its development according to the activities performed over time. Our temporal analysis technique is a search-based optimization of the best decomposition of software repository commits using heuristics such as the classes changed in each commit and the magnitude/importance of these changes. We used *ArgoUML*, *JFreeChart*, *ICEfaces1*, *ICEfaces2*, and *ICEfaces3* as case studies for the evaluation of our approach.

5.2 Phase Identification

A program execution typically involves various functionalities. Knowing which part of the execution (phase) belongs to which functionality helps maintainers to focus on this part during their comprehension and maintenance tasks. However, there are no explicit links in a program (source code or execution) between functionalities and execution events. Equivalently, software goes through different development stages (phases) and undergoes various changes during its lifetime. Mapping periods of time to development activities helps developers and managers get a better idea of the software evolution.

The goal of our work is to explore various heuristics that approximate phases. Before introducing these heuristics, we first define the related concepts. Then we present the implementation of heuristics using a genetic algorithm to search for phases.

5.2.1 Unified Comprehension Framework

Several concepts utilized in our approach in the contexts of execution and evolution come directly from our unified comprehension framework. We present the basic definitions of these concepts in both contexts, indicated below as (1) execution, and (2) evolution.

Execution event / Evolution event: (1) An execution event is an action that occurs periodically during execution, e.g., a method call, a method return, an object creation. It encapsulates information such as the class of the triggered method, the instance of the class, and the method name. (2) An evolution event is defined as one day of development in the history of the software. All commits of the event are considered as part of the same event.

Object / Entity: (1) An object is the instance of the class concerned by the execution event. Objects are uniquely identified. (2) An entity is a software module on which changes are applied during evolution.

Object life / Entity activity: (1) An object begins its life when it is created and the end of its life is approximated by its last appearance (reference) in the execution trace. (2) The activity of a software module is represented by the modifications that it undergoes within a certain period of time. It spans from the first change to the module to its last change in the evolution history.

Execution trace / Evolution trace: (1) A trace is the sequence of all execution events representing the entire execution scenario. (2) An evolution trace is the history of software evolution divided into a sequence of consecutive days.

Execution phase / Evolution phase: (1) An execution phase is a sequence of consecutive execution events. It is a portion of the entire execution trace. (2) An evolution phase represents a time period in software history. The consecutive events during this period of time constitute the phase.

Cut position: (1) A cut position is the location of the first event of a phase in the trace. (2) A cut position or cut point is an instant in time when there is a switch in evolution phases. It is represented by the first commit of an evolution event.

Phase identification solution: (1) A solution is a set of cut positions that decomposes the execution trace into phases. (2) A solution is a set of cut positions that decomposes the evolution history into phases.

5.2.2 Heuristics

We base our phase detection technique on heuristics that characterize phases we want to identify. From the unification point of view, this part of our approach is the most different in the two contexts: execution and evolution. However, the heuristics have also commonalities in a sense that they lead to the notions of phases' internal cohesion and external dissimilarity.

5.2.2.1 Execution Heuristics

Our approach for execution phase identification stands on assumptions concerning the activities of objects during their lifetime. Our rationale is based on the role of objects during program execution and is outlined as follows:

— Objects collaborating in a same phase often have overlapping lifetimes. For instance, if two objects collaborate, one of them cannot end its lifetime before the other object's life begins.

— Two successive execution phases should not have many active objects in common, otherwise it would suggest that the program is still within the same phase.

— Many objects are created at the beginning of an execution phase and destroyed before the end of the phase. Some objects come from previous phases.

— Not all objects active in a phase are representative of this phase. Such objects are more general and are indifferently used during the execution. Other objects characterize the phase as they are only triggered during this particular phase.

— A phase does not begin between two successive method-call events, between two successive method-return events, or even between a method-call event and a method-return event. A phase

switch occurs only when the program's execution exits a method and enters a method, i.e., between a method-return event and a method-call event.

5.2.2.2 Evolution Heuristics

Our evolution phase identification technique rests on assumptions concerning the nature of changes applied to software modules during their evolution and is outlined as follows:

— Two successive evolution phases should not involve changes to the same set of modules, otherwise it would suggest that the software evolution is still in the same phase.

— Changes performed on an entity within a phase are of the same nature. Each software module undergoes changes of the same magnitude and importance within an evolution phase.

— The types of changes performed on software modules differ from one evolution phase to the other.

— The development rhythm is the same throughout an evolution phase. It could be fast, or slow, but should be relatively constant within a phase.

— The smallest period of time representing development activities is one day. In an evolution history usually spanning several years, phases can be divided as consecutive development days.

5.2.3 Detection Algorithm

We approach phase identification as an optimization problem. We consider phases as subsets of the execution/evolution events contained in the execution/evolution trace. The phase detection problem then becomes one of determining the best decomposition of the trace's events.

5.2.3.1 Search Space

The search space is composed of all possible solutions of the phase identification problem. During the optimization process we explore the search space to improve the quality of our solution. The search space is closely related to the solutions and thus we describe the execution phase identification search space and the evolution phase identification search space separately.

Execution: Considering that an execution trace contains n events (possibly in the order of hundreds of thousands), and that a particular execution contains any number m of phases, the number of possible solutions is C_k^l, where $k = m - 1$ is the number phase shifts in the trace, and l ($0 \leq l \leq (n/2) - 2$) is the number of positions in the execution trace where a phase shift can occur. The number of possible solutions C_k^l depends on the shape of the call tree. The wider the tree is, the larger l will be.

To understand the effect on l, consider the two extreme (unlikely) cases, as shown in Figure 5.1. First, a call tree composed of a single branch coming out of the root node (Figure 5.1(a)), this branch consisting of a sequence of method calls followed by a sequence of method returns. In this example, $l = 0$ because there is no method return followed by a method entry in the entire trace. Second, a call tree with a depth of 1 (Figure 5.1(b)), where each method call is immediately followed by a method return. In this second example, $l = (n/2) - 2$ because there are as many potential phase shifting positions as the number of pairs of method return/call (in this order), minus the root method call (at the beginning of the trace) and the root method return (at the end of the trace).

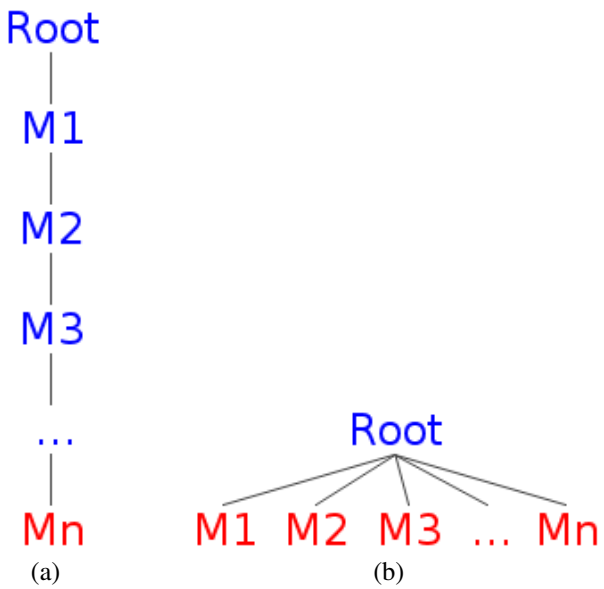


Figure 5.1 – Example of a call tree with (a) a sequence of successive method calls followed by a sequence of successive method returns, and with (b) each method call followed by a method return, except for the root call.

The number k of phase switches during execution is the number of phases minus 1. For instance, an execution phase containing two phases has one phase switch. The problem of phase detection considers therefore the number of possible combinations of k from l. As mentioned earlier, the number of events n can be very large and l remains also large despite the fact that $l \leq (n/2) - 2$. This results in an exceedingly large search space to investigate exhaustively. Hence, we rely on a meta-heuristic search, in our case a genetic algorithm, to find a solution to our phase identification problem.

Evolution: The number of possible solutions, i.e., decompositions of the evolution trace, is C_{m-1}^{n-1}, where n is the number of evolution events in the history (in the order of hundreds or thousands), and m is the number of phases in a particular solution. The number of phases m, representing the periods of time in the evolution where development activities are similar, is unknown and hence is bound by the number of events in the entire trace (each event is a phase), and the entire history itself being a single phase, i.e., $1 \leq m \leq n$.

5.2.3.2 Search Algorithm

As explained earlier, the search space is prohibitively large for an exhaustive approach. Therefore, we consider a meta-heuristic search and use a genetic algorithm to find a near-optimal solution representing the decomposition of the execution or evolution trace into phases that provide a higher abstraction than the fine-grained information of method calls/returns or commits, and that help developers get an overview of the considered time period.

The algorithm starts by creating an initial population of solutions, i.e., different trace decompositions. In the first generation, these solutions are generated randomly. Then in an iterative process, every iteration produces a new generation of solutions derived from the previous ones. For each iteration, we push the "fittest" candidates to the next generation (elitism), and then we generate the rest of the solutions composing the next generation by combining/modifying existing solutions using crossover and/or mutation operators. The fitness of a solution is computed using a function implementing the heuristics stated earlier.

The details about the main aspects of our algorithm are presented in the following sections.

5.2.3.3 Solution Encoding

A solution is a decomposition of the execution trace into different chunks representing phases. Our algorithm searches for the best cut positions to decompose the trace into phases. These cut positions in the trace are events where a phase shift occurs. Figure 5.2 (left) schematizes two solutions: A and B. Solution A illustrates a decomposition of the trace of n events into seven evolution phases, with six cut positions representing phase switching. Solution B divides the trace into five phases, with four phase-shifting cut positions. The solutions have the same combined length as each one represents the entire execution or evolution history considered. Therefore, each solution is solely characterized by the cut positions that correspond to the first events of each phase of the solution. We simply represent a solution by a vector of integers containing the cut positions, and construct the phase of a particular solution based on its cut positions when needed. This vector representation of our solution maps perfectly with the genomic representation of solutions in genetic algorithms, where each vector of positions corresponds to a chromosome or genotype of our solution. The vector size indicates the number of phases, and therefore, it can have different sizes as we do not limit our search to a fixed number of phases, and assume it is unknown prior to the application of the detection algorithm.

5.2.3.4 Initial Population

At the beginning of the search algorithm, we create an initial population of solutions, i.e., integer vectors containing phase cut positions. In order to diversify the population's individuals, we generate N solutions as follows:

1. We randomly choose cut positions in the trace, within the number of events in the trace. The positions must be valid phase-shifting positions in the execution context, i.e., a cut position is a method call (start of a phase) AND its preceding event must be a method return (end of a phase). In the evolution context, all events are valid cut positions.
2. The positions of the selected events are then sorted in ascending order, because events are ordered in time, and phases are successive. For any two or more equal cut positions, only one of them is conserved. Then we construct integer vectors of the cut positions to represent the solutions.
3. In order to vary the number of phases in an initial population of N individuals, we generate five solutions with two phases, the next five solutions with three phases, and so on. In total, we produce

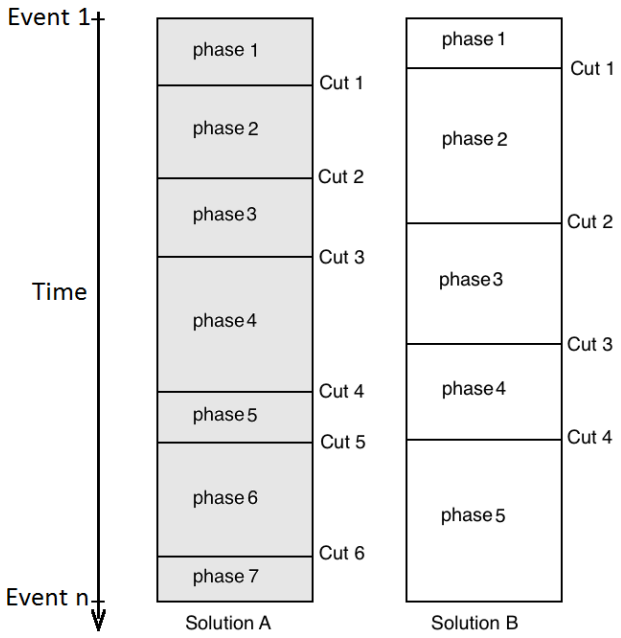


Figure 5.2 – Solution A represents the decomposition of the trace into seven phases. Solution B represents the decomposition of the trace into five phases.

solutions containing different numbers of phases from two phases to $N/5$ phases. However, our technique is not bound by a fixed number of phases or even by a subset of phases' numbers, and therefore the number of phases can exceed $N/5$ during the search using our operators (crossover and mutation).

4. Finally, only solutions with a fair fitness are incorporated into the initial population. This is to allow the search to start from a fair overall set of individuals.

5.2.3.5 Fitness Function

Our approach rests upon the heuristics explained in Section 5.2.2. To convert the proposed heuristics into measurable properties, we developed metrics that are combined into the fitness function. The search process is guided by this function towards a solution that best satisfies those heuristics. As mentioned earlier, the heuristics for the execution phases' detection are different from those of the evolution phases' detection. There are different ways to translate the proposed heuristics into measurable properties that allow an evaluation of the quality of a phase detection solution. We use three metrics in the execution context and four metrics in the evolution context. The metrics are combined

to model the fitness function of our problem in the two contexts. Again, while the metrics are different from one context to the other, they are based on our general framework with the common notions of phase internal cohesion and external dissimilarity or decoupling. Note that to simplify the notations in this section, we denote phases by i and j.

Execution

Phase Object Coupling: Two successive phases are coupled if they share objects. An object is shared by phases if its lifetime covers them. Figure 5.3 illustrates eight different cases that can occur when computing coupling over two phases. These cases differ in the way object lifetimes are distributed over the two successive phases. Some of them are more desirable than others, and therefore, are given larger weights when computing the result. The object-coupling metric of a phase is a linear combination of the number of objects per category. Here are the details and rationale behind our weight affectation, starting from the most desirable to the least desirable distribution of object lifetimes. Weights range from 1 to 6 according to the object's lifetime. There are eight categories of object lifetimes and some of them are similar in terms of phase coupling, so they are assigned the same weight. We illustrate the different cases using the example of Figure 5.3. We refer to two consecutive phases as the first phase and the second phase (phase i and phase j in Figure 5.3).

First, objects that are included in one phase have a weight of 6, i.e., they are created and destroyed within the same phase. This is the ideal case since each phase would involve a different set of objects, e.g., $Obj1$ and $Obj2$.

Second, we assign a weight of 5 to objects that are destroyed in the first phase or created in the second phase, e.g., $Obj6$ and $Obj7$. This resembles the first category, except for the fact that objects are not created/destroyed within the first/second phase respectively. It is a good category because the two successive phases do not share objects.

Third, there are objects that are created in the first phase and destroyed after the second phase, such as $Obj5$. Here we have two sub-categories, one of which is more desirable than the other. If the object is not active in the second phase, i.e., the object is not involved in any event, we assign a weight of 4. Although the two phases share the object, according to our earlier definition, the second phase does not use it. However, if the object is involved in the second phase, we assign to it a weight of 2

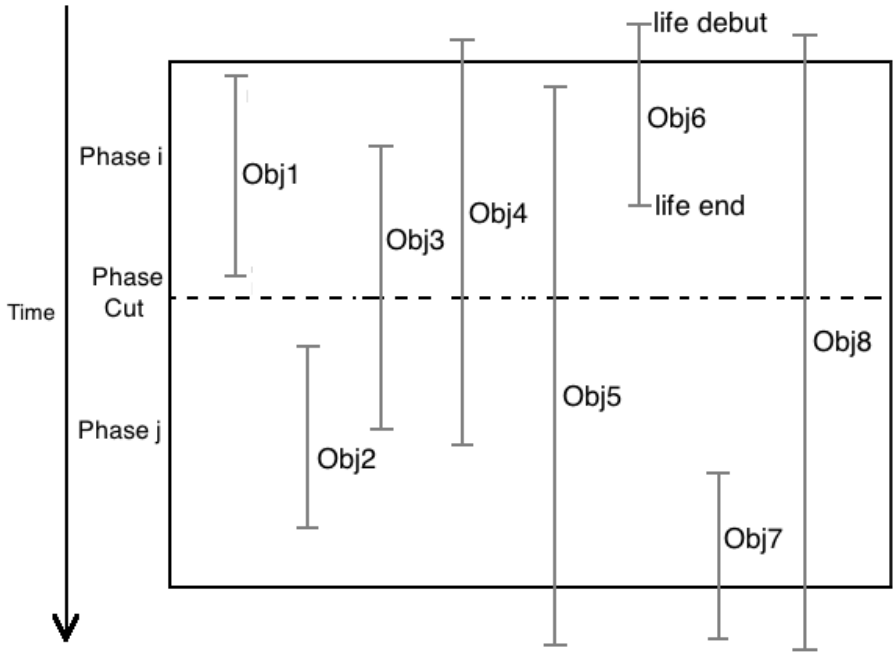


Figure 5.3 – Phase coupling with the lifetimes of the objects.

because the object is active in the two consecutive phases, which probably means that the cut position is not appropriate. There is also the contraposition of the previous case, in which we used the same weights' values, where an object is destroyed in the second phase and created before the first phase, e.g., $Obj4$.

Then, there is the case of an object created before the first phase and destroyed after the second phase, e.g., $Obj8$. The object could be involved in the first phase only, the second phase only, or both. Following the same principle as the two previous cases, we assign a weight of 3 when the object is involved in one phase only, and a weight of 2 if it is active in both phases.

Finally, the less desirable case is when the object is created in the first phase and destroyed in the second phase, such as $Obj3$. Here we assign the lowest weight of 1 because we probably should merge the two phases and hence remove the cut position.

The coupling between two successive phases is computed as the number of objects in each category, multiplied by its corresponding weight. Formally:

$$Coupling(i,j) = \frac{\sum_k (w_k|OC_k|) - [\min(\{w_k\}) \sum_k (|OC_k|)]}{[\max(\{w_k\}) - \min(\{w_k\})] \sum_k (|OC_k|)} \tag{5.1}$$

where OC_k is the set of objects of category k, and w_k is the weight affected to the objects of category k. The coupling for a solution is the average coupling on the successive phase pairs.

Object Similarity: This metric calculates the similarity between objects of two successive phases. We construct for each phase the list of distinct active objects. Then, we compute the number of objects in common between the two successive phases. The number of common objects in each phase is divided by its total number of objects. The object similarity is taken as the average between the two resulting numbers of the two phases

$$Obj(i,j) = 1 - \frac{1}{2}\left(\frac{|DO_i \cap DO_j|}{|DO_i|} + \frac{|DO_i \cap DO_j|}{|DO_j|}\right) \tag{5.2}$$

where DO is the set of distinct objects in the phase. The object similarity of a solution is the average similarity of every two consecutive phases.

Thin Cut: The thin cut represents the number of objects that are divided, in terms of their respective lifetimes, by the cut position. For each cut position, we compute the number of objects that are active before and after the cut position. The resulting number is then normalized by the number of objects in the entire trace. For example in Figure 5.3, there are four objects divided by the cut position: $Obj3$, $Obj4$, $Obj5$, and $Obj8$. Therefore, the result would be $4/M$, where M is the number of objects created before the given cut. Formally:

$$Cut(pos_i) = 1 - \frac{|CO_i|}{|TO_i|} \tag{5.3}$$

where CO_i is the set of objects that are created before cut position pos_i and destroyed after it, and TO_i is the set of objects created before cut position pos_i. The thin cut of a given solution is simply the average score of each position.

Fitness Function: The fitness function of a solution in the execution phases detection problem, is defined as follows:

$$fitness(sol) = \frac{a \times Coupling(sol) + b \times Obj(sol) + c \times Cut(sol)}{a+b+c} \tag{5.4}$$

where solution *sol* to be evaluated consists of cut positions in the execution trace, and a, b, and c are the weights affected to each component. The weights can be used to give a different importance to each metric discussed above. In our approach, we empirically set their values equal to one for equal importance.

Evolution

Phase Entity Coupling: We define coupling between two successive phases as the sharing of software modules. A software module, or entity, is shared by two successive phases if it undergoes changes in both of them. Moreover, the number of occurrences (changes) to an entity within a phase denotes its importance to this particular phase. Therefore, coupling between phases is weighted by the importance of the entities to each phase. Figure 5.4 illustrates the concept of phase entity coupling with a trivial example. On the one hand, entity A is subject to 4 modifications in phase i and 1 modification in phase j. Hence, it is more important to phase i. On the other hand, entity B is more important to phase j since it is changed 5 times in phase j and twice in phase i. We also have entities C and D that are, respectively, only present in phase i and phase j. This is the ideal case with respect to phase entity coupling as entities in phase i are not shared with entities in phase j. Phase coupling is to be minimized in order to satisfy our heuristics. Coupling between two successive phases i and j is computed as:

$$ECp(i,j) = \frac{1}{2}\left[\frac{\sum_k \min\left(n_{ki}, n_{kj}\right)}{\sum_k n_{ki}} + \frac{\sum_k \min\left(n_{ki}, n_{kj}\right)}{\sum_k n_{kj}}\right]$$

where k is the global index of the entity changed in phase i or j, and n_{ki} and n_{kj} are the number of times entity k is changed in the respective phase. In the example of Figure 5.4, the entity coupling value is evaluated to 0.45, which is explained by the fact that 50% $\left(\frac{2}{4}\right)$ of the entities in those phases are modified only in one of the two phases (C and D). Also, the shared entities between these two phases are more important to one phase than the other. Hence, the coupling value (0.45) is less than 0.5. Phase coupling is normalized between 0 and 1.

Phase Change-Type Coupling: Development activity can be characterized by the types of changes performed in evolution phases. We define change-type coupling as the number of com-

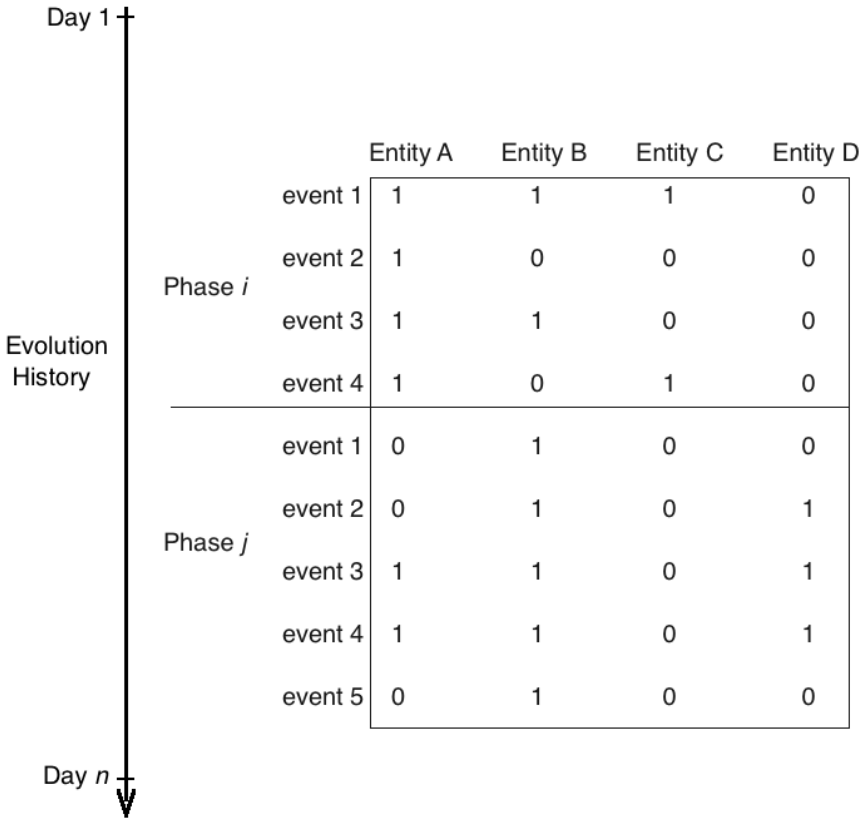


Figure 5.4 – Phase entity coupling with entity changes over evolution events. A value of 1 indicates that the column entity is changed in the row event. A value of 0 indicates that the column entity is unchanged in the row event.

mon change types between two successive phases. An evolution phase may include several changes of different types, and thus we have to find those that best describe the development activity in this phase. To this end, we first compute the ratio of all changes of one type with respect to all changes of all types for the entire software history, that we call the global ratio for this change type. We have one such global ratio for each change type. Then, for each evolution phase we compute the same ratio for each change type with respect to all changes occurring in the phase, that we call the local ratio. We represent the evolution phase by the change type that has a larger local ratio than its global ratio, i.e., the change types that are more specific to the phase. Having computed the representative sets of change types for each phase of our solution, we calculate the number of common change types between two consecutive phases to measure the phase change-type coupling as:

$$CTCp(i,j) = \frac{1}{2}\left[\frac{|CT_i| \cap |CT_j|}{|CT_i|} + \frac{|CT_i| \cap |CT_j|}{|CT_j|}\right]$$

where CT_i and CT_j are the sets of representative change types of phases i and j respectively. The range of the phase change-type coupling is $[0,1]$.

Phase Change-Importance Cohesion: To ensure that each entity in a phase undergoes changes that have similar significance, the cohesion metric measures the similarity between the significance of changes that software entities undergo within the phase. It is inspired by the amplitude metric used by Barry et al. [9], which measures the size of a change. In our approach, we believe that defining the nature of source-code changes by their significance level is more precise and meaningful than using the traditional text differentiation between source-code entities [28]. To encode the importance of changes in an evolution phase, we characterize the committed entities in that phase by a vector representing the number of changes of each significance level: [*#Crucial, #High, #Medium, #Low, #None*]. Having computed the map of significance level with related changes for each entity in a phase, we can calculate the phase change-importance cohesion. To this end, we consider all constructed vectors of changes, and compute the variance of the number of changes related to each significance level, i.e., one variance per significance level. By considering each significance level separately, we ensure that the phase entities undergo similar changes both in terms of amplitude (number of changes) and significance (level).

By definition, variance decreases when changes are similar. Thus, we take the inverse of the variance (to maximize the value) as the cohesion metric for each significance level. The phase change-importance cohesion metric is simply the arithmetic average of the cohesion values, related to the five significance levels. Formally, the change-importance cohesion metric of a phase i is:

$$\begin{aligned} mean_i(c) &= \frac{1}{n_i}\sum_k n_{ki}(c) \\ var_i(c) &= \frac{1}{n_i}\sum_k [n_{ki}(c) - mean_i(c)]^2 \\ CICh(i) &= \frac{1}{5}\sum_{c=1..5}\frac{1}{var_i(c)} \end{aligned}$$

where c denotes the five significance levels mentioned before, phase i delimits many variables with it as a subscript, such that n_i is the number of entities (in phase i), $n_{ki}(c)$ is the number of changes to entity k with significance level c, $mean_i(c)$ is the mean of the number of changes of significance level c, and $var_i(c)$ is the variance of the number of changes of significance level c. The last equation above should modify the value 5 if the number of significance levels changes.

Phase Development-Rate Cohesion: An evolution phase should follow a "constant" development rhythm. Hence, we are interested in evaluating the regularity of commits within a phase, i.e., without consideration for their duration. To evaluate the speed of development within a phase, we use the average time (computed from each commit's time stamp) between commits of the phase. To evaluate the regularity of commits within a phase, we compute the variance of the elapsed time between consecutive commits. Our definition of the phase development-rate cohesion metric is based on the dispersion measure described by Barry et al. [9]. Similarly to the phase change-importance cohesion, we compute the development-rate cohesion metric (to maximize) as the inverse of the variance. Formally, the development-rate cohesion metric of a phase i is defined as:

$$DRCh(i) = \frac{1}{var(t_{c_a,c_{a+1}})} \quad \forall c_a \in i$$

where $var(t_{c_a,c_{a+1}})$ is the variance of time intervals between every two consecutive commits c_a and c_{a+1} in phase i.

Fitness Function: For a given solution s, coupling metrics ECp and $CTCp$ are computed for each cut position (two consecutive phases) in the solution, and coupling values for the solution, S_{ECp} and S_{CTCp}, are computed as the average of all coupling values associated to the solution's cut positions. Similarly, the solution cohesion metrics, S_{CICh} and S_{DRCh}, are computed as the average of cohesion values of the solution's phases. The resulting fitness function must evaluate the quality of solution s during the search process. As the solution that we are looking for should comply with all the previously outlined heuristics, we must maximize the following formulation for the **fitness** of the

four aforementioned metrics, i.e., as the geometric mean of these metrics:

$$fitness(s) = \sqrt[4]{S_{(1-ECp)}(s) \times S_{(1-CTCp)}(s) \times S_{CICh}(s) \times S_{DRCh}(s)}$$

5.2.3.6 Genetic Operators

Each iteration of the genetic algorithm involves the creation of the next generation of individuals from the current one. As explained above, the solutions are represented by a vector of cut positions in the execution trace. These cut positions act as the individuals' chromosomes and are used for crossover between two selected parent solutions to produce two child solutions to populate the next generation. A solution is also modified by mutating its chromosomes (cut positions).

Elitism: To create the new population in a given generation, we first automatically add the two fittest solutions. This ensures that no good solutions will be lost during the search. The two fittest solutions are also considered for reproduction to generate other solutions as explained next.

Selection: To select candidate (or parent) solutions from the current population, we use two selection strategies. The first strategy, used in the execution phases detection approach, is the roulette-wheel technique. It consists of assigning selection probabilities to the current population's solutions prior to the selection process. The selection probabilities are proportional to the quality (fitness score) of the solution. The better a solution is, the more chances it has to be selected to crossover and produce a child solution. The second selection strategy, used in the evolution phases' detection, is the tournament selection, where a fixed number of candidate parents are randomly chosen from the current population. Then, the best solution among them is retained for reproduction. When two parent solutions are selected, we apply crossover and mutation operators with certain probabilities, with crossover probability higher than mutation probability in the beginning of the search. The probabilities are changed during the search if the algorithm is stuck in the same solution for a long time (measured in number of iterations). At the beginning of the search, we give more opportunity to the combination of existing genetic material (crossover). Then, when we reach the limit of combinations, we give more opportunity to injecting new genetic material (mutation).

Crossover: We use a single-point crossover. To perform a crossover between two solutions, we randomly pick a new cut position independently from the two solutions' cut positions. The new cut position at the same location in both solutions produces two parts for each solution. The top part of solution A is combined with the bottom part of solution B to form child solution AB. Conversely, the top part of solution B is combined with the bottom part of solution A for child solution BA. Figure 5.5 illustrates the resulting offsprings from the application of crossover on solutions A and B from Figure 5.2. The new random cut position is shown as a blue line dividing both solutions. According to the cut position, offspring AB is formed from phase 1, phase 2, phase 3, and the top part of phase 4 of solution A (with respect to the new cut position), and the bottom part of phase 3 of solution B, phase 4, and phase 5 of solution B. The offspring BA is a result of combining phase 1, phase 2, and the top part of phase 3 of solution B, and the bottom part of phase 4 of solution A, phase 5, phase 6, and phase 7 of solution A. The two siblings share the new random position, which was not present in their parents in this case, and receive portions of their parents' chromosomes. In this sense our crossover operator may introduce new genetic material with the one inherited from the parents.

Mutation: We mutate an individual in three different manners, depending on a certain probability. The first mutation strategy splits one phase into two, by generating randomly a new cut position and inserting it at the correct location in the solution. The second mutation strategy consists in merging two successive phases into a single one, where we randomly select one of the cut positions and discard it. Finally, the third mutation strategy randomly changes a cut position, i.e., the boundary between two consecutive phases is repositioned.

This results in the alteration of four existing phases (the previous and subsequent phases for each cut point) without changing the number of phases in the solution. Figure 5.6 illustrates the three mutation strategies applied to solution A from Figure 5.2. Phase 4 in solution A was subdivided into two phases (phase 4' and phase 4") with the insertion of a new cut position to produce mutant A1. Mutant A2 is the result of removing the fifth cut position of solution A, which resulted in the merging of phase 5 and phase 6. The third cut position of solution A was altered resulting in the modification of phase 3 and phase 4 in mutant A3 (phase 3' and phase 4').

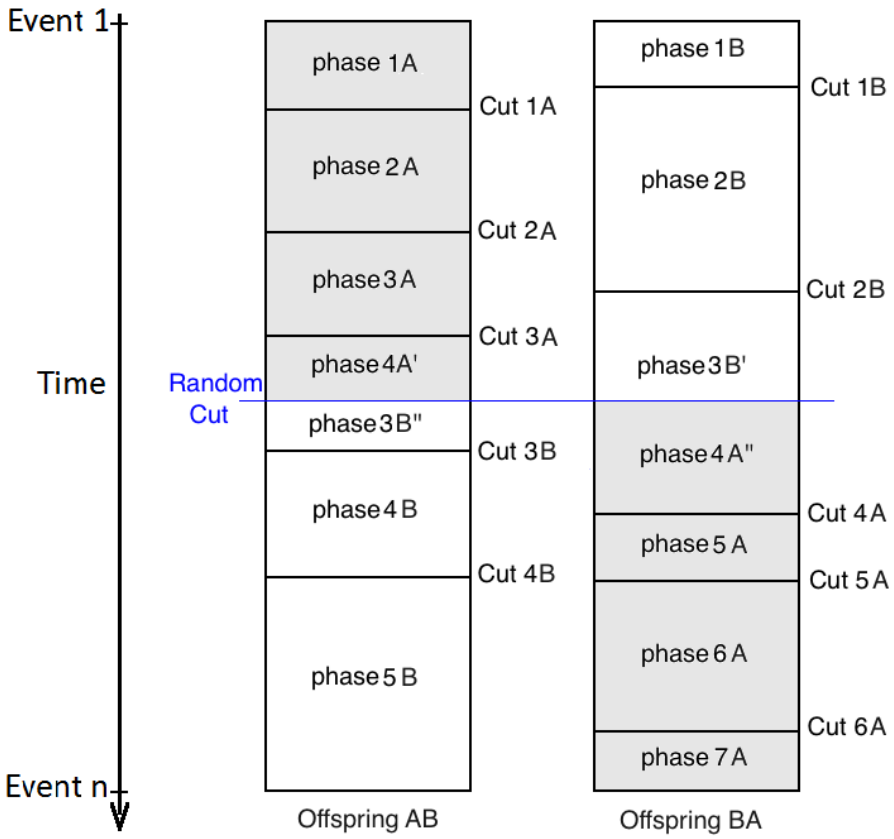


Figure 5.5 – The result of the crossover operator applied to the solutions of Figure 5.2.

5.3 Case Studies: Execution Phase Identification

To assess our approach, we apply our phase detection algorithm on three scenarios of *JHotDraw* [1] and four scenarios of *Pooka* [2]. This section introduces the settings of our case studies and the choices made while evaluating our approach. In particular, we evaluate the accuracy of our technique in detecting high-level execution phases based on object usage and lifetime.

5.3.1 Settings

5.3.1.1 Execution Data

Our phase detection technique takes as input an execution trace. This trace is constructed by monitoring the execution of a program and by recording its events. We use an implementation in C of the jvmti API to listen to the *JVM* for method entries and exits. Each execution event is described by

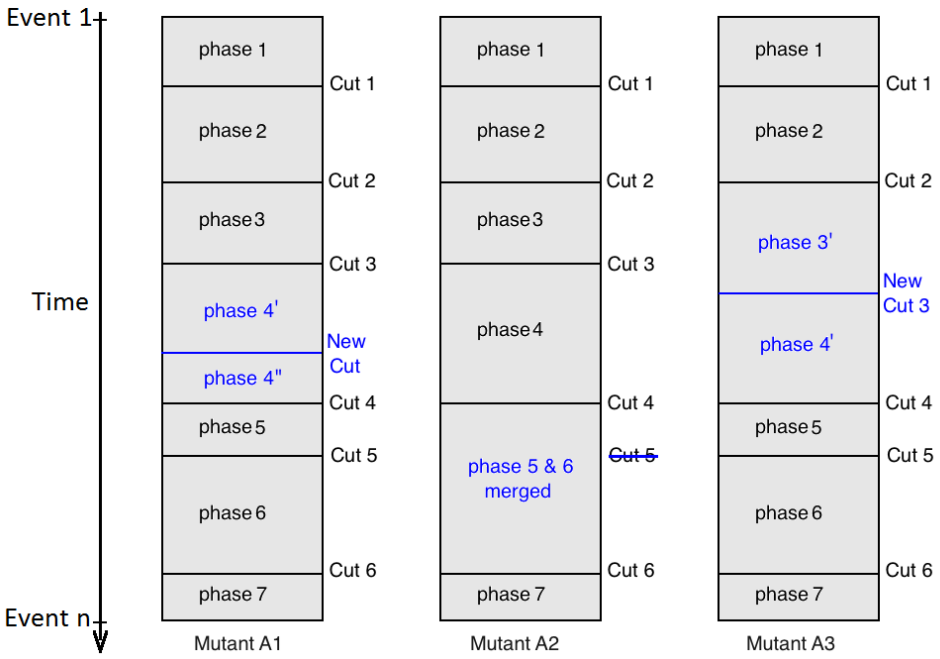


Figure 5.6 – The result of the mutation operator applied to solution A of Figure 5.2.

its type (method entry or exit), the class name with the method triggered, a time stamp, the method arguments, the return type, the object unique id, and the dynamic object in the case of polymorphism or dynamic dispatch. To determine the reference that our solutions will be compared to, the tracer allows us to record event ids on demand. These ids serve to determine the beginning and ending events of each external functionality, i.e., boundaries of their corresponding phases.

We extracted three execution traces, one for each of three scenarios of *JHotDraw* [1], an open-source Java GUI framework for technical and structured graphics. It contains 585 classes distributed over 30 packages. We also executed four scenarios of *Pooka* [2], an open-source email client written in Java and using the *Javamail* API. It contains 301 classes organized in 32 packages.

The execution traces, presented in Table 5.I, differ in size and in performed tasks. For each scenario, we defined the beginning and ending events of each phase in the trace, as explained above. These events serve as reference for evaluating our phase detection approach. Table 5.I summarizes the information about the extracted execution scenarios, which were used during the evaluation.

For example, execution scenario J1 contains three execution phases: an initialization phase, a phase representing the opening of a new file, and a phase representing the drawing of a rectangle with

Scenario	Description	Nb of events
J1	*Init ; New file ; Draw round rectangle;*	66983
J2	*Init ; Open file ; Animation ; New file; Window tile;*	100151
J3	*Init ; Open file ; Delete figure ; Write html text area;*	105069
P1	*Init ; Open email ; Reply to sender ; Consult Help;*	79506
P2	*Init ; Change theme ; Send new email;*	101710
P3	*Init ; Search inbox ; Delete email;*	99128
P4	*Init ; Get new emails; Window tile;*	63162

Table 5.I – Description of the seven execution scenarios.

rounded corners. In the second phase (*'New file'*), *JHotDraw* opens a window canvas for drawing, on which we draw a rounded rectangle figure. In terms of cut positions, an ideal solution for this scenario would be to have two cut positions: one at the end of the initialization phase, and the other at the end of the *'New file'* phase.

5.3.1.2 Algorithm Parameters

Our genetic algorithm uses parameters that may influence the resulting solution. We present here the values empirically chosen for our evaluation.

The **initial-population size**, which stays constant throughout the iterative process, affects both the algorithm's performance and efficiency [33]. We start our search with a population of 100 solutions. Solutions are generated randomly (see Section 5.2.3.4), but only those having a fitness value of at least 0.5 are incorporated in the initial population. This allows our search algorithm to start from a population of a reasonable quality.

For the **selection strategy**, the roulette-wheel technique is used. We also use the **elitist strategy** that incorporates the two fittest solutions directly to the next generation. Regarding the **genetic operators' probabilities**, we start with a crossover probability of 90% and a mutation probability of 10%. As a **termination criterion**, we fixed the number of generations to be produced to 10$\times$ the

size of a population, i.e., 1000 generations. Finally, for the **fitness function definition**, we utilized a combination of three metrics (see Section 5.2.3.5).

5.3.2 Evaluation

We ran our algorithm on the seven described scenarios. As our algorithm is probabilistic, each scenario was processed six times; the best solution was retained.

As discussed in Section 5.3.1.1, the beginning and end positions (i.e., events) of each phase is recorded during the tracing process. We used these positions as an oracle to evaluate our solutions. Precision and recall were used to assess our solutions. We computed the precision for a phase in terms of events, as explained by Asadi et al. [7], as well as for the recall. They are defined formally as:

$$precision_{event}(DE, AE) = \frac{|DE \cap AE|}{|DE|} \tag{5.5}$$

$$recall_{event}(DE, AE) = \frac{|DE \cap AE|}{|AE|} \tag{5.6}$$

where DE is the set of detected phase events and AE the set of actual phase events. The event precision and recall of a solution are simply the averages of the phases precision and recall, respectively.

We computed the precision and recall in terms of phases, by comparing the detected phases with the actual phases, as done by Watanabe et al. [56]. A phase is considered detected if it has event precision of at least 75% with the corresponding oracle phase.

$$precision_{phase} = \frac{|Detected \cap Actual|}{|Detected|} \tag{5.7}$$

$$recall_{phase} = \frac{|Detected \cap Actual|}{|Actual|} \tag{5.8}$$

All execution scenarios include an initialization phase. There are many object creations during this phase, many of them remain active in subsequent phases. Based on the objects' lifetimes, our approach fails to detect the initialization phase of several scenarios. The results of Table 5.II consider the initialization phase, which penalizes them. The results between parentheses do not include the

Scenario	$\textbf{precision}_{event}$	$\textbf{recall}_{event}$	$\textbf{precision}_{phase}$	$\textbf{recall}_{phase}$
J1	0.85 (0.94)	0.38 (0.57)	0.66 (1.0)	0.50 (0.66)
J2	0.89 (0.96)	0.58 (0.59)	0.80 (1.0)	0.60 (0.75)
J3	0.82 (0.92)	0.64 (0.64)	0.66 (1.0)	0.20 (0.50)
P1	0.91 (0.95)	0.71 (0.71)	1.0 (1.0)	0.60 (0.75)
P2	0.93 (0.96)	0.69 (0.61)	1.0 (1.0)	0.75 (0.66)
P3	0.94 (0.99)	0.32 (0.32)	0.83 (1.0)	0.33 (1.0)
P4	0.96 (0.99)	0.37 (0.37)	1.0 (1.0)	0.66 (1.0)

Table 5.II – Summary of the evaluation results of the seven scenarios.

initialization phase in the calculations and are clearly better, which suggests that the rest of the phases are correctly detected (see $precision_{phase}$).

To understand the lower results for the initialization phase, we further analyzed object creations, destructions, and first use, i.e., first method call after the object creation. We found that many objects are created during the initialization and are used for the first time in subsequent phases. Figure 5.7 illustrates well this fact. Many object creations (top curve) happen during the initialization phase (phase 1). The figure also shows a peak of first-time object uses (middle curve) at the beginning of phase 2. This suggests high coupling between phases 1 and 2, which is penalized in our algorithm and thus this cut position, if ever encountered, would probably be discarded during the search.

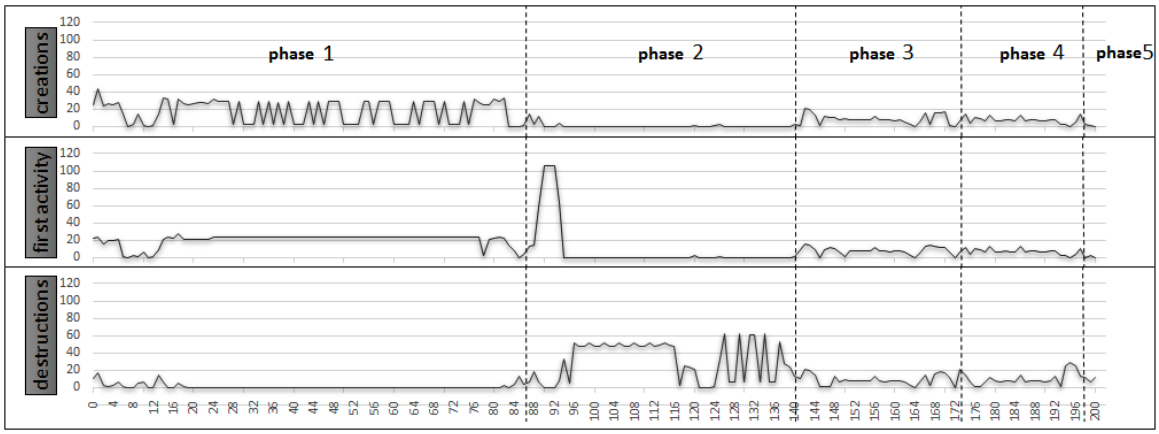


Figure 5.7 – The objects' creations, first activity, and destructions in the trace of an execution scenario.

Finally, for sanity check, we compared our results to random search. To have a similar setup than our algorithm, we generated the same number of solutions (100 individuals $\times 1000$ generations), i.e., $100,000$ random solutions for each scenario. Each time, we selected the best solutions from these, according to the same fitness function. The results of the Wilcoxon test showed that our algorithm performs significantly better than random search for $precision_{event}$ ($p = .018$), $precision_{phase}$ ($p = .027$), and $recall_{phase}$ ($p = .043$). However, the difference for $recall_{event}$ is not statistically significant ($p = .271$).

5.3.3 Discussion

When developing our approach, we made several decisions concerning the detection heuristics and their implementation. The results of our experiments showed that these decisions could still be improved upon.

We considered that all objects utilized by a program to accomplish an execution phase contribute equally to phase detection. However, objects are different in terms of lifetime, number of uses, and execution pattern. Some objects are created at the beginning and destroyed at the end of the execution. Others are more specialized and have shorter lifetimes. Apart from their lifetimes, objects also differ in the way they are used in the execution. An object may be executed sparsely from its creation to its destruction, or it can be used regularly and very frequently. Another aspect of object execution is the regularity with which an object appears in the execution trace. These object-execution properties should be further investigated to define object-execution profiles that will be much more representative of execution phases.

Our single-point crossover strategy also introduces mutation in the form of the new common cut position of two-parent individuals. This strategy preserves most of the parents' phases and possibly creates new phases. This crossover strategy is consistent with our execution phase definition, which states that a phase is a portion of the trace. Another possible crossover strategy is a uniform one, where we generate two child solutions from two parents by selecting their cut points. Here no new cut position is introduced, and all the parents' cut points are inherited. However, the resulting individuals may end up with no phases from the parents because one parent's phases could be further segmented by the other parent's cut positions. Although we opted for our first strategy, we believe

that more sophisticated crossover operations could reduce the mutation factor while preserving the completeness and consistency of trace decomposition.

The choice of fitness function metrics is an important decision when using the approach. We tried several metrics to evaluate the fitness of our solutions. Some of them gave better results in some particular scenarios. We chose the metrics configuration that gave the best results on average for all scenarios. Therefore, we believe it is important to investigate the relationship between the nature of the functionalities involved in a scenario and the metrics. For example, some metrics favor solutions with few phases while others tend to orient the search towards solutions with more phases.

Finally, the fitness function is computed as a combination of three metrics (Equation (5.4)). The factors were weighted equally, on the same domain ($[0, 1]$), but they have in practice different magnitudes. Taking the simple average could favor some metrics over others. To alleviate this, we can use a multi-objective search algorithm for which the magnitude of single objectives is not important.

5.4 Case Studies: Evolution Phase Identification

We applied our evolution phase identification technique to describe development activities of the following five systems: *ArgoUML* [3], *JFreeChart* [5], *ICEfaces1*, *ICEfaces2*, and *ICEfaces3* [4]. The studied systems are different in size, and have different development periods. For each system, Table 5.III gives, for the development period, the number of commits, the number of classes involved in the commits, the dates for this period, and the number of official releases. It is worth mentioning that *ICEfaces* is one software developed in three separate software repositories. The development team opted for different repositories when substantial changes were made to the API in terms of architecture and technology. We consider each sub-project as a separate system since our approach takes as input software repository information. Furthermore, *ArgoUML* and the three *ICEfaces* projects continued to undergo changes after the last official release, i.e., the software repository contains commits after their last release date. This is indicated in Table 5.III as "+1" in the number of releases of these projects. In *ArgoUML*, the time period represents the development process that will lead to *ArgoUML* 0.36 (not yet released). In the *ICEfaces* projects, the time period represents only servicing, i.e., changes to support the final officially released version of the application.

System	# commits	# classes	# releases	Period
ArgoUML	9150	2748	15+1	10/09/03 - 31/07/14
JFreeChart	3000	1482	13	19/06/07 - 31/08/14
ICEfaces1	3916	1371	8+1	22/02/07 - 11/11/14
ICEfaces2	882	1088	4+1	19/07/10 - 30/05/12
ICEfaces3	2148	1827	5+1	18/11/11 - 24/01/14

Table 5.III – Evolution information of the five studied systems.

5.4.1 Setting

5.4.1.1 Evolution Data

The evolution data is collected from software repositories; they include commit dates, committed entities, types of changes that entities underwent, and importance of each change. We queried the repository for the history log, from which we gathered information about the commits and entities involved. We gathered the changes and their importance using *ChangeDistiller* [29]. The tool takes as input two versions of the same class and returns all the changes between them, with each type and significance level [28]. We recovered the source code of every class version and passed it to *ChangeDistiller* for abstract syntax-tree differentiation. The data collected was then fed to our algorithm.

5.4.1.2 Algorithm Parameters

The parameters used for our phase identification algorithm are as follows. At the beginning, the algorithm creates an initial population of 200 solutions. The population size remains the same for every iteration. As an *elitist strategy*, we directly incorporate the two fittest solutions from the current generation to the next generation. We used the tournament selection technique as *selection strategy*. The *genetic operators' probabilities* are set to 70% for crossover and 30% for mutation. However, if the best found solution is not improved during 100 successive iterations, then the probability values are switched to 70% for mutation and 30% for crossover. This should allow the optimization process to avoid local optima by exploring other regions in the search space. Once the best solution is

changed/improved, we switch the probabilities back to their original values. As *termination criterion*, the algorithm keeps running until no better solutions can be found for 200 iterations.

5.4.2 Stability and Similarity Evaluation

In order to evaluate the stability of our solutions, and compare them to reference solutions, we define a distance metric between two solutions, e.g., S_a and S_b in Figure 5.8. Let us arbitrarily choose S_b as the one with fewer phases. Since solutions may have different numbers of phases, to compare two solutions, we associate to each cut position in S_b a corresponding cut position in S_a that is its closest cut position. This gives the associations $\{(c_{b1}, c_{a2}), (c_{b2}, c_{a3}), (c_{b3}, c_{a4}), (c_{b4}, c_{a4}), (c_{b5}, c_{a6})\}$ in Figure 5.8.

The metric computes the distance between the cut position and its correspondent in terms of the number of days separating them, e.g., d between c_{b1} and c_{a2}. This distance is normalized by the length of the matching phase of the cut position, e.g., l. A matching phase is the phase, in the second solution, that encloses the cut position of the first solution, i.e., the time stamp of the cut position in the first solution is within the time period defining the phase in the second solution. Therefore, the value of the distance metric is within the range $[0, 1]$. The distance between two solutions is then calculated as the average of distances between the respective cut positions. For instance, a value of 0 means that the cut positions of one solution have the same time stamps as their corresponding cut positions in the other solution. A value of 0.25 suggests that, on average, a cut position of one solution is at a "quarter" of the distance of its matching phase, from its corresponding cut position in the second solution. For the sake of clarity, we show and analyze *Similarity* between solutions, which is defined as one minus the distance value; thus we are striving to find a value closer to 1.

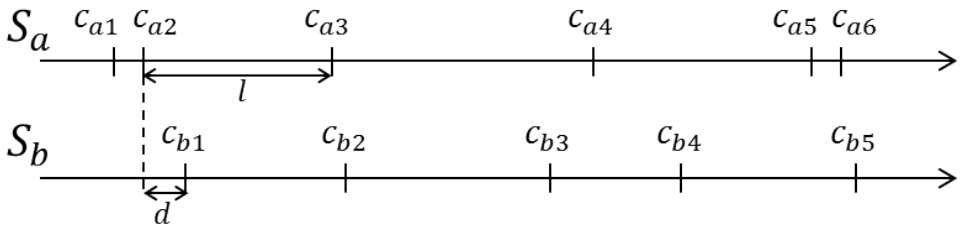


Figure 5.8 – Comparing solutions S_a and S_b with 6 (resp. 5) cut positions.

We ran our algorithm five times for each of the five systems, and thus obtained five solutions for each system. First, we evaluate the stability of our algorithm as the average pairwise similarity between the five solutions.

For each system, we also compare our best solution, in terms of fitness function, to the reference solution created from the official releases. For this purpose, in addition to the similarity metric, we also compute the *recall* of our solutions as compared to their reference solutions. In general, our solutions contain more cut positions than the reference solutions. Thus, we define the recall of a solution as the normalized number of reference cut positions that have a unique corresponding cut position in our solution, over the total number of reference cut positions. Unlike the distance calculation, two reference cut positions should not be associated to the same cut position in our obtained solution. For example, if in Figure 5.8, S_a is the obtained solution and S_b is the reference one, both c_{b3} and c_{b4} are associated with c_{a4}. But as c_{b3} is the closest to c_{a4}, c_{b4} is considered as undetected, which results in a recall of 0.8 ($4/5$). We only compute the recall of a solution and not the precision because we are interested in knowing how many reference cut positions (software releases) we are able to detect. However, we do not claim that our approach will identify only software releases as evolution phases.

Table 5.IV summarizes the results of our evaluation. The algorithm converges to a similar solution after each run, i.e., the pairwise similarities between the obtained solutions are high. Furthermore, the best solution is very similar to the reference solution, and most of the release dates correspond to a unique cut position in the automatically obtained solution. The recall measure shows that our approach missed some releases of *ArgoUML* and *JFreeChart*, i.e., the value is lower than 1. An undetected release means that the development activity at the end of that release is similar to the one at the beginning of the following phase. For instance, we could not detect the shift between *ArgoUML* 0.32 and 0.32.1, as well as the end of the 1.0.7, 1.0.11, and 1.0.16 releases of *JFreeChart*.

5.4.3 Software Releases Comprehension

In this section, we show how the identified evolution phases can be used to describe the development activity leading to the release of a software version. First, we classify the phases identified by our approach and give examples of how to characterize software releases using the phase classifica-

System	Stability	Reference Similarity	Reference Recall
ArgoUML	0.80	0.80	0.94
JFreeChart	0.90	0.79	0.85
ICEfaces1	0.82	0.87	1.0
ICEfaces2	0.86	0.76	1.0
ICEfaces3	0.90	0.83	1.0

Table 5.IV – Evaluation of the evolution phase identification.

tion. For illustrative purposes, we propose two representations of phases and releases in Figures 5.9 and 5.10.

5.4.3.1 Phase Classification

The result of the phase identification algorithm is a decomposition of the evolution trace of the studied system into evolution phases. These phases are obtained by optimizing four metrics measuring the development activities according to the heuristics presented in Section 5.2.2. These phases represent abstractions over the software evolution, and help software managers in understanding the software evolution, i.e., by focusing on each phase separately and analyzing it as an abstraction over the development activities in that period. However, to characterize a phase and understand development activities that happened in it, managers need to compute the metrics that we outlined in Section 5.2.3.5 for that phase and interpret their values. For instance, although we know that each evolution phase is characterized by a relatively constant rhythm of development, managers may be interested in the characteristics of that rhythm, e.g., is it fast or slow? Hence, they need to compute the average of elapsed times between consecutive commits in the phase, and then to interpret it. In the same vein, although the entities in a phase undergo changes of the same significance, managers may be interested in characterizing the significance of the changes made in the phase, e.g., are they relatively important?

In order to help characterize and comprehend the identified evolution phases for a studied system, we propose a framework that classifies evolution phases according to the heuristics outlined in this paper. Precisely, we propose to classify evolution phases with regard to the following three criteria: (1) the importance of changes, (2) the development rate, and (3) the variety in change types. These criteria describe the development activity within an evolution phase.

Our classification approach is similar to the one proposed by Xing and Stroulia [61]. For each criterion, we define a measure, and compute it for each phase, as well as for the entire evolution trace. Then, the classification of a phase with respect to a criterion is based on the comparison between the values of the corresponding measure for the phase and for the entire evolution trace. More specifically, the classification of an evolution phase is performed as follows:

— *Importance of changes:* We compute the importance of changes done in the phase, using *ChangeDistiller*, and compare it to the average importance of changes in the entire evolution trace. The phase is labeled as undergoing "important changes" if its associated value is larger than the global average, and otherwise it is labeled as "less important changes" .

— *Development rate:* The average time between consecutive commits within the phase (the phase's commit-periodicity) is compared to the average time between consecutive commits in the entire evolution history (the overall commit-periodicity). This measure is directly inspired from the periodicity measure by Barry et al. [9]. The phase is marked as a "rapid development" evolution phase if its commit-periodicity is smaller than the overall commit-periodicity, and as a "slow development" phase otherwise.

— *Variety in change types:* Similarly, we compute the number of different types of changes done in the phase over the total number of types of changes performed during the entire evolution trace. This gives us a measure of the variety of types of changes carried out in the phase, which we compare to the average value with regard to all identified phases. Then, the phase is designated as having "different types of changes" if its value is larger than the total average, and "similar types of changes" otherwise.

Each of the three criteria qualifies an evolution phase in two ways, which yields a phase classification of $2^3 = 8$ categories, as presented in Table 5.V. For simplicity, we chose to classify the phases into only two categories with regard to each criterion (high or low). The classification may be based

on a more sophisticated statistical description of the values for each criterion. However, this would increase the number of phase types and thus, it would complicate the classification. For instance, associating to each criterion three categories, high, medium, and low, would result in $3^3 = 27$ phase types.

Class	Description
A	Important changes, rapid development, different types of changes
B	Important changes, rapid development, similar types of changes
C	Important changes, slow development, different types of changes
D	Important changes, slow development, similar types of changes
K	Less important changes, rapid development, different types of changes
L	Less important changes, rapid development, similar types of changes
M	Less important changes, slow development, different types of changes
N	Less important changes, slow development, similar types of changes

Table 5.V – Classification of the evolution phases.

5.4.3.2 Analysis of Software Releases

Figure 5.9 shows per release the identified evolution phases for the analyzed systems. Each horizontal rectangle represents a release (ordered from bottom to top), and releases are grouped by systems. Within each release, the identified phases are displayed as portions of rectangles whose widths are proportional to their durations in the release and whose colors denote the phase types, which correspond to the combinations of change importance (*important/less-imp*), frequency (*rapid/slow*), and uniformity (*different/similar*).

Figure 5.10 is another visualization of the data in Figure 5.9; it uses the same color-coding. Here phases are depicted with the same width, and thus the length of a release line is proportional to the number of phases that compose the release. In Figure 5.10, releases are sorted with respect to their similarities with phase types, starting from the first to the last phases. This figure captures two important pieces of information: (1) the variety of evolution phases that lead to a release, as well as

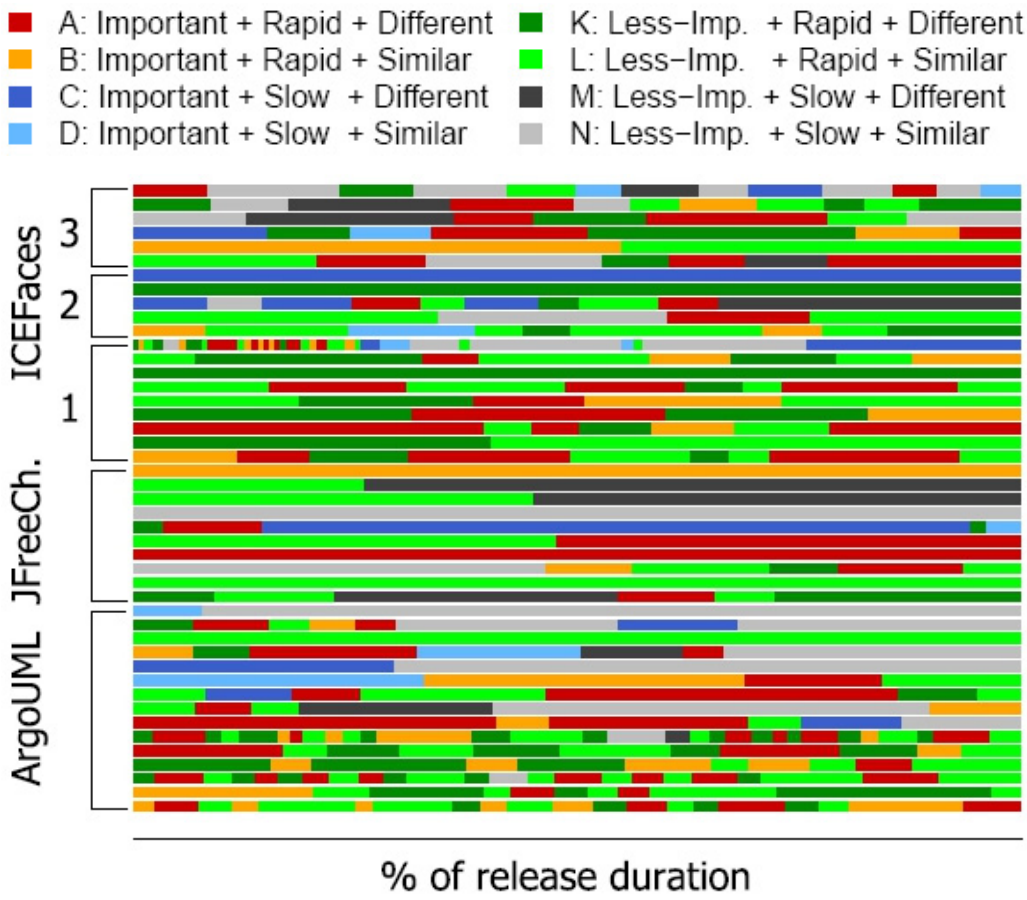


Figure 5.9 – Evolution phases per software release for all analyzed systems. Each horizontal line is a sequence of evolution phases, which represents one release. The duration of an evolution phase within a release is proportional to the release's duration. Classes from Table 5.V are color-coded as indicated at the top.

the transitions between those phases; and (2) the similarity between releases, especially with respect to the evolution phases in the beginning of release development process.

First Overview on Release-to-release Evolution: Looking at Figure 5.9, we can see that the green family of phases (K, L) and the red/orange family of phases (A, B) are considerably more visible/frequent than other phase types. This shows that the development rhythm of analyzed projects is rather rapid in almost all periods of software evolution. We can observe that for almost all analyzed releases, the green family of phases are frequently accompanied (followed or preceded) by phases from the red family, but rarely by blue or gray families of phases. This is mainly the case for the first seven

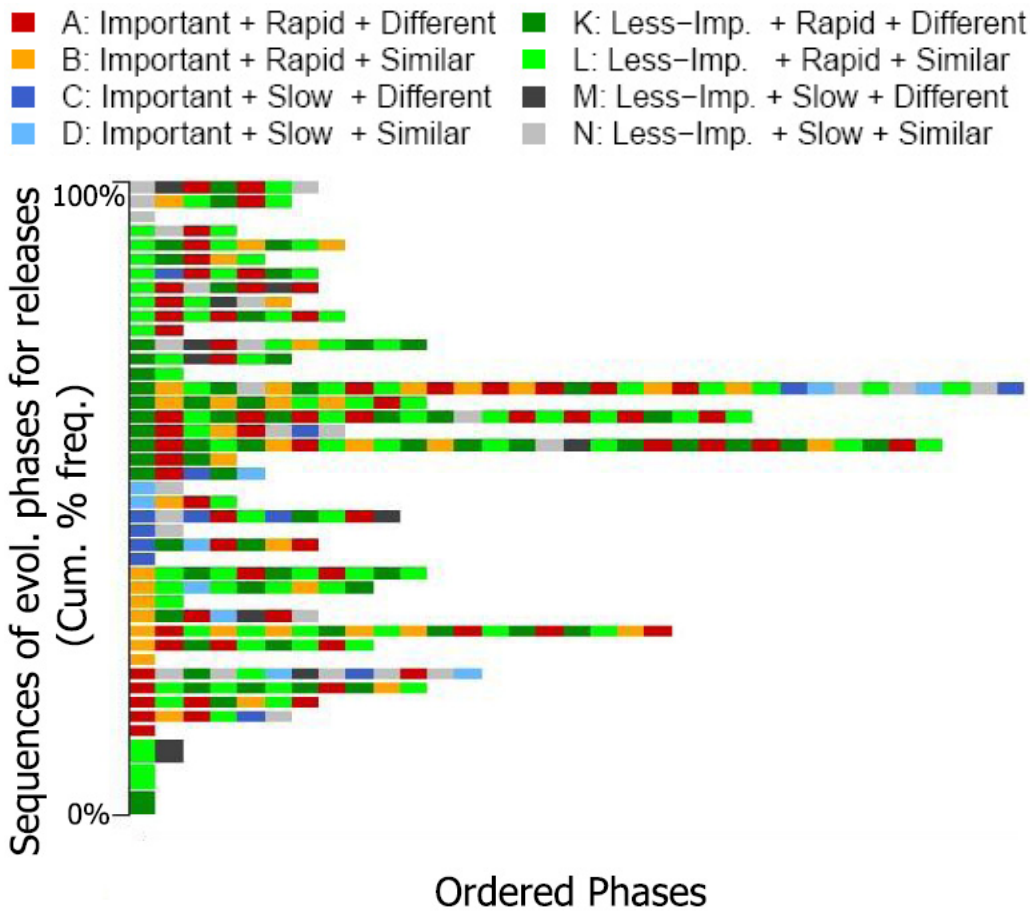


Figure 5.10 – Sequences of evolution phase types for all analyzed releases. Each line is a sequence of evolution phases, which represents a release. The sequences are ordered by similarity, independent of their respective software. Classes from Table 5.V are color-coded as indicated at the top.

releases of *ArgoUML* and all releases of *ICEFaces1*. The noticeable difference between the aforementioned releases is that the number of transitions between evolution phases that involve important changes (red family of phases) and less important changes (green family of phases), is much larger in *ArgoUML* releases than in *ICEFaces1* releases. Focusing on the beginning of releases, Figure 5.10 shows that the largest subset of analyzed releases are those that begin with rapid development phases involving relatively less important changes. More precisely, 24 releases out of 45 releases begin with evolution phases from the green family (K, L). In second place come the releases that begin with important changes in rapid development rhythm. More precisely, 12 releases begin with evolution phases from the red family of phases (A, B). Releases that begin with a slow development rhythm

involving less important changes are rare. Only 3 releases begin with phases from the gray family (N, M). Hence, we think that these are exceptional cases. This leads to the detection of different categories of releases.

Categories of releases: Based on the families of phases in release sequences, their numbers, as well as the number of transitions between phases, we can identify the following families of releases:

— *Rapid development:* Releases that are characterized by rapid development activities consist mainly of red and green families of phases (A, B, K, L). Such releases are the first 7 releases of *ArgoUML* (ordered bottom to top in the figure), all the releases of *ICEFaces1*, except the latest one (the servicing period), and the latest release of *JFreeChart*. Here, we observe that the development of *ArgoUML* and *ICEFaces1* was relatively rapid in the beginning, and then passed to phases of relatively slow development, while the development of *JFreeChart* became rapid in its latest (analyzed) release. Another example of such rapid development of releases is *ICEFaces3* 3.0.1, which consists of two evolution phases (B, L) depicted as an orange and then a green rectangle. This official maintenance release featured over 100 improvements and fixes during a development period of about a month with more than 60 commits. The first phase (B) of this release involved important changes, and the last phase (L) is composed of less important changes, which suggests a consolidation period before the release of *ICEFaces3* 3.0.1.

— *Slow development:* As opposed to rapid development of releases, these releases are characterized by a slow development rhythm throughout the release development cycle. These releases are consisting mainly of blue and gray families of phases (C, D, M, N). Such releases are releases *ArgoUML* 0.30.2 and 0.35.1, which have the following sequences of evolution phases, respectively: *{C-N}* and *{D-N}*. The main difference in release 0.30.2 is that *during a noticeable long period* of the release's development life cycle, the development activities were characterized by different types of important changes, i.e., phase C in 0.30.2 represents a period of 23 development days of the 78 days for the development of the release. An exceptional slow development in our sample is the servicing period in *ICEFaces2* after release 2.1.0*Beta*2, which consists of only one evolution phase: *{C}*. This indicates that 100% of the release development period (173 development days) is characterized by different important changes produced in a slow development rhythm. In our sample, these releases are considerably less frequent than rapid development releases. Figure 5.10

shows that the frequency of releases starting by rapid development phases (see releases that start with phases from the red family (A, B) or the green family (K, L)) is considerably higher than the frequency of other releases.

— *Arrhythmic development:* When the rhythm of development throughout the release development life cycle is not constant, switching between rapid and slow developments, we say that the release is characterized by an arrhythmic development. For instance, *ArgoUML* release 0.28 *{A-B-A-L-C-N}* is an arrhythmic development release that starts with a rapid development rhythm (the sub-sequence *{A-B-A-L}*) and ends with a slow development rhythm (the sub-sequence *{C-N}*). Other examples of arrhythmic development releases, are *ArgoUML* 0.34 *{K-A-L-B-A-N-C-N}*, and *JFreeChart* 1.0.18 *{L-M}*.

— *Complex mixture:* When arrhythmic development releases involve phases that are characterized by different natures of changes in terms of modified entities, importance of changes, and variety of types of changes applied, the release development becomes complex. Hence, it is difficult to characterize and understand such evolution periods. Fortunately, our analysis reveals that such complex mixture releases are not frequent, at least in our studied systems. Among the 45 analyzed releases in our study, we identified few complex mixture releases, and most of them are the servicing periods in the *ICEFaces* projects. For instance, the servicing periods after *ICEFaces1* 1.8.2 and *ICEFaces3* 3.3.0 are composed of several different types of phases because they represent diverse development activities needed to support the application after it is not considered anymore for further releases. The *ICEFaces1* 1.8.2 and *ICEFaces3* 3.3.0 servicing periods have the following sequences of evolution phases, respectively: *{K-B-L-K-N-B-K-L-A-L-B-A-B-A-B-A-K-A-L-B-A-L-B-L-C-D-N-L-N-D-L-N-C}* and *{A-N-K-N-L-D-M-N-C-N-A-N-D}*.

Other analyses may be performed on sequences describing software releases. For instance, a similar approach to Figure 5.10 can be used to find similarities between releases in their last phases or intermediate phases. Other sequence analysis techniques may be used to compute the probability of transitions between the different types of phases.

Appendix I presents the complete mappings of evolution phases to the software releases for all the studied systems.

5.5 Summary

We presented an automatic approach for identifying phases in the contexts of execution and evolution. Our approach is based on meta-heuristic search of the best decomposition of the considered period of time. It rests on specific heuristics for execution and evolution. We cast the problem of finding phases as an optimization problem and utilize a genetic algorithm to search for a good solution. In the execution context, our technique is based on object lifetimes and object collaborations. For evolution phase detection, we utilize software entities and changes that they undergo to evaluate phases. We evaluated our automatically detected execution phases by comparing them to manually detected phases. We ran our algorithm on seven different scenarios of *JHotDraw* and four scenarios of *Pooka*. The evolution phases are compared to official software releases for evaluation. We ran the phase detection algorithm on five software evolution histories: *ArgoUML*, *JFreeChart*, *ICEfaces1*, *ICEfaces2*, and *ICEfaces3*.

In an attempt to illustrate the commonalities between phase execution and phase evolution identification, our interpretation of the phase identification problem is the same in both contexts. We were able to identity phases using our approach with specific metrics for each context. In both contexts, the metrics measure the phases' internal cohesion and the coupling between two successive phases.

Among future research directions, the use of multi-objective search algorithms is one that could improve the computed results because both phase identification problems involve multiple combined metrics. Another research direction worth investigating is the study of other metrics to better model the heuristics of Section 5.2.2.

Chapter 6

Conclusion

6.1 Contributions

The time dimension is crucial in many software comprehension problems. A good understanding of how software changes over time aids to better grasp its complexity. One perspective on software's time dimension comprehension is to consider that software morphs over time in two ways: when its structure evolves and when its program executes. The main contribution of this thesis is the unification of software evolution and execution for comprehension. Our proposition begins with the definition of a common comprehension framework based on time analysis. The framework is then instantiated to model and represent two software comprehension problems. Each software comprehension problem is defined in the contexts of execution and evolution. Finally, we use similar techniques to analyze these comprehension problems in the two contexts.

The rationale behind our unification approach is the similarity between research problems in the execution comprehension community and in the evolution comprehension community. Despite this similarity, there is little communication between the two research communities. By establishing a unified model for the comprehension of software's time dimension in both contexts, we lay down the groundwork for better exchange of knowledge between the communities. Each research community has its strengths and has developed mature solutions to certain problems. Solutions from one community can be easily transposed to the other one using the unified model.

The first comprehension problem considered is the analysis of collaboration between different entities in software. This comprehension problem is characterized in the execution context as the contributions of classes in the accomplishment of use-case scenarios. In the context of evolution, it is the comprehension of developers' contributions to software development. We used software visualization to comprehend the entities collaboration problem in both contexts. Our visualization techniques included the representation of collaboration using heat maps.

The second comprehension problem is the identification of phases to abstract low-level information for comprehension purposes. In the execution context, the problem boils down to detecting phases

in execution traces that correspond to high-level features. The problem of evolution phase identification consists of determining periods of time in the evolution history, where software development is similar. This permits characterizing software evolution history with the different development phases it undergoes. We formulated the phase identification problem as an optimization one and applied search-based techniques to solve it in both contexts.

Besides our main contribution of the unified comprehension framework, we proposed advances to the state of the art in specific problems considered: class collaborations in use-case scenario analysis, developer contributions in software development understanding, high-level execution phase detection, and software evolution comprehension based on development activity phase identification.

Our work on the use of heat maps and software visualization for the analysis of class collaborations in execution, and developer contributions in evolution was presented at the *IEEE Working Conference on Software Visualization* [13]. The work on the application of meta-heuristic optimization for execution phase detection was published in the *Symposium on Search-based Software Engineering* [14]. An article presenting our evolution phase identification approach, with a search-based technique, has been accepted in the *International Conference on Program Comprehension* [15]. These findings to the specific problems considered in our unification approach, were published in their respective communities. This indicates the interest in our comprehension framework and consolidates our main research proposition concerning the unification. Finally, the research proposition of this thesis, a unified comprehension framework for execution and evolution problems involving software's time dimension, was accepted in New Ideas and Emerging Results track of the *International Conference on Software Engineering* [16], which suggests its originality in the software engineering community.

6.2 Future Perspective

In this thesis, we establish a new perspective on comprehension problems involving time analysis in program execution and software evolution. As mentioned before, each community has developed mature solutions, tools, and techniques that can be utilized by the other research community.

At the end of each chapter, we gave some shorter-term extensions for our work, applicable within the specific context of the corresponding chapter. In this section, we are looking at broader applications for future work.

Debugging a program execution is a well-known problem, that has been well studied in the execution comprehension community. With the unified framework, one can use debugging for software evolution, although there are certainly major challenges to this idea. First, when debugging a program, one can usually execute it as many times as wanted, with breakpoints, to analyze its state at certain instants of time before a bug occurs. This is impossible in software evolution for obvious reasons. However, thinking of software evolution as a repetition of development cycles, debugging would be to identify an "evolution bug" and correct it in the following cycles. Here, we mean by "evolution bug" an anomaly in the evolution process rather than a fault or error in the software itself. Breakpoints would be instants in the evolution where a snapshot of the development process state is rigorously studied to gain insights on the evolution path of software. The study can be carried out using existing evolution exploration tools, such as *Replay* [35].

Also, profiling is used in execution comprehension to assess resource consumption during program execution. It is used to identify bottlenecks and understand causes of peaks in resource consumption. The equivalent in evolution is the study of development process resource consumption. There exist methodologies and techniques to assess the resources needed and used during development, but most of these techniques are ad hoc and not automated. The evolution community would benefit from a profiling tool based on tools used in the execution community; and our unified framework can be utilized as starting point for such work.

Finally, there is much work on co-change analysis in evolution comprehension. It involves the analysis of software entities that change concurrently during software evolution. The co-change analysis problem can be translated to the execution comprehension by detecting objects that are manipulated concurrently during program execution. As in the execution context, this could help identify dynamic coupling between objects at runtime and deduce relations between objects to understand program execution using object co-executions.

Bibliography

[1] JHotDraw: a Java GUI framework. http://www.jhotdraw.org.

[2] Pooka: a java email client. http://www.suberic.net/pooka.

[3] ArgoUML: a UML modeling tool. http://argouml.tigris.org/.

[4] ICEfaces: a Rich Internet Application development framework. http://www.icesoft.org/java/projects/icefaces/.

[5] JFreeChart: a Java chart library. http://www.jfree.org/jfreechart/.

[6] L. J. Arthur. *Software Evolution: The Software Maintenance Challenge*. Wiley-Interscience, New York, NY, USA, 1988.

[7] F. Asadi, M. Di Penta, G. Antoniol, and Y.-G. Guéhéneuc. A heuristic-based approach to identify concepts in execution traces. In *Conference on Software Maintenance and Reengineering*, CSMR, pages 31–40, 2010.

[8] M. Balzer and O. Deussen. Voronoi treemaps. In *Proceedings of the Proceedings of the 2005 IEEE Symposium on Information Visualization*, INFOVIS '05, pages 7–, Washington, DC, USA, 2005. IEEE Computer Society.

[9] E.J. Barry, C.F. Kemerer, and S.A. Slaughter. On the uniformity of software evolution patterns. In *International Conference on Software Engineering*, pages 106–113, 2003.

[10] V. R. Basili. Evolving and packaging reading technologies. *J. Syst. Softw.*, 38(1):3–12, July 1997.

[11] A. Begel, Y. P. Khoo, and T. Zimmermann. Codebook: Discovering and exploiting relationships in software repositories. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 125–134, New York, NY, USA, 2010. ACM.

[12] K. H. Bennett and V. T. Rajlich. Software maintenance and evolution: A roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, pages 73–87, New York, NY, USA, 2000. ACM.

[13] O. Benomar, H. Sahraoui, and P. Poulin. Visualizing software dynamicities with heat maps. In *Working Conference on Software Visualization*, VISSOFT, 2013.

[14] O. Benomar, H. Sahraoui, and P. Poulin. Detecting program execution phases using heuristic search. In *Symposium on Search-Based Software Engineering*, Lecture Notes in Computer Science, pages 16–30. 2014.

[15] O. Benomar, H. Abdeen, H. Sahraoui, P. Poulin, and M. A. Saied. Detection of software evolution phases based on development activities. In *International Conference on Program Comprehension*, (to appear) 2015.

[16] O. Benomar, H. Sahraoui, and P. Poulin. A unified framework for the comprehension of software's time dimension. In *International Conference on Software Engineering*, (to appear) 2015.

[17] P. Bhattacharya, M. Iliofotou, I. Neamtiu, and M. Faloutsos. Graph-based analysis and prediction for software evolution. In *Proceedings International Conference on Software Engineering*, ICSE, pages 419–429. IEEE, 2012.

[18] J. Bohnet, M. Koeleman, and J. Döllner. Visualizing massively pruned execution traces to facilitate trace exploration. In *VISSOFT*, pages 57–64, 2009.

[19] D. Coleman, D. Ash, B. Lowther, and P. Oman. Using metrics to evaluate software system maintainability. *Computer*, 27(8):44–49, August 1994.

[20] T. A. Corbi. Program understanding: challenge for the 1990's. *IBM Systems Journal*, 28(2): 294–306, June 1989.

[21] B. Cornelissen, D. Holten, A. Zaidman, L. Moonen, J. J. van Wijk, and A. van Deursen. Understanding execution traces using massive sequence and circular bundle views. In *Proceedings of IEEE International Conference on Program Comprehension*, pages 49–58, 2007.

[22] B. Cornelissen, A. Zaidman, B. van Rompaey, and A. van Deursen. Trace visualization for program comprehension: A controlled experiment. In *International Conference on Program Comprehension*, pages 100–109. IEEE, 2009.

[23] M. D'Ambros and M. Lanza. Software bugs and evolution: a visual approach to uncover their relationship. In *Proceedings of the 10th European Conference on Software Maintenance and Reengineering, 2006*, pages 10 pp.–238, March 2006.

[24] W. De Pauw and S. Heisig. Zinsight: A visual and analytic environment for exploring large event traces. In *International Symposium on Software Visualization*, SOFTVIS. ACM, 2010.

[25] W. De Pauw, D. Lorenz, J. Vlissides, and M. Wegman. Execution patterns in object-oriented visualization. In *Proceedings of the 4th USENIX Conference on Object-Oriented Technologies and Systems*, COOTS'98, pages 16–16, Berkeley, CA, USA, 1998.

[26] P. Dugerdil and S. Alam. Execution trace visualization in a 3D space. In *International Conference on Information Technology: New Generations*, ITNG, pages 38–43, 2008.

[27] M. D'Ambros, Ha. Gall, M. Lanza, and M. Pinzger. Analysing software repositories to understand software evolution. In *Software Evolution*, pages 37–67. Springer Berlin Heidelberg, 2008.

[28] B. Fluri and H. Gall. Classifying change types for qualifying change couplings. In *International Conference on Program Comprehension*, pages 35–45. IEEE, Jan 2006.

[29] B. Fluri, M. Würsch, M. Pinzger, and H. Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Transactions Software Engineering*, 33(11):725–743, 2007.

[30] T. Fritz and G. C. Murphy. Using information fragments to answer the questions developers ask. In *Intl Conf. on Softw. Eng.*, pages 175–184, 2010.

[31] T. Gîrba and S. Ducasse. Modeling history to analyze software evolution: Research articles. *Journal of Software Maintenance and Evolution*, 18(3):207–236, May 2006.

[32] O. Greevy and S. Ducasse. Correlating features and code using a compact two-sided trace analysis approach. In *Proceedings of the 9th European Conference on Software Maintenance and Reengineering 2005*, pages 314–323, March 2005.

[33] J.J. Grefenstette. Optimization of control parameters for genetic algorithms. *IEEE Transactions Systems, Man and Cybernetics*, 16(1):122–128, 1986.

[34] H. Hagh-Shenas, V. Interrante, C. Healey, and S. Kim. Weaving versus blending: a quantitative assessment of the information carrying capacities of two alternative methods for conveying multivariate data with color. In *ACM SIGGRAPH 2006 Research Posters*, 2006.

[35] L. Hattori, M. D'Ambros, M. Lanza, and M. Lungu. Software evolution comprehension: Replay to the rescue. In *Proceedings of the IEEE 19th International Conference on Program Comprehension (ICPC), 2011.*, pages 161–170, June 2011.

[36] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[37] J. M. Kleinberg. Authoritative sources in a hyperlinked environment. *Journal of the ACM*, 46 (5):604–632, 1999.

[38] A. Kuhn and O. Greevy. Exploiting the analogy between traces and signal processing. In *Proceedings of the 22nd IEEE International Conference on Software Maintenance, 2006*, pages 320–329, Sept 2006. doi: 10.1109/ICSM.2006.29.

[39] G. Langelier, H. Sahraoui, and P. Poulin. Visualization-based analysis of quality for large-scale software systems. In *Proceedings IEEE/ACM International Conference on Automated Software Engineering*, pages 214–223, 2005.

[40] G. Langelier, H. Sahraoui, and P. Poulin. Exploring the evolution of software quality with animated visualization. In *Symposium on Visual Languages and Human-Centric Computing*, VLHCC, pages 13–20. IEEE, 2008.

[41] A. Lienhard, S. Ducasse, and T. Gîrba. Object flow analysis: Taking an object-centric view on dynamic analysis. In *Proceedings of the 2007 International Conference on Dynamic Languages:*

In Conjunction with the 15th International Smalltalk Joint Conference 2007, ICDL '07, pages 121–140, New York, NY, USA, 2007. ACM.

[42] M. Lungu, M. Lanza, T. Girba, and R. Heeck. Reverse engineering super-repositories. In *Proceedings of the 14th Working Conference on Reverse Engineering, 2007*, pages 120–129, Oct 2007.

[43] J. T. Nosek and P. Palvia. Software maintenance management: Changes in the last decade. *Journal of Software Maintenance: Research and Practice*, 2(3):157–174, 1990.

[44] N. Pennington. Empirical studies of programmers: second workshop. In *Empirical studies of programmers: second workshop*, chapter Comprehension strategies in programming, pages 100–113. Norwood, NJ, USA, 1987.

[45] T. M. Pigoski. *Practical Software Maintenance: Best Practices for Managing Your Software Investment*. Wiley, 1996.

[46] H. Pirzadeh, A. Agarwal, and A. Hamou-Lhadj. An approach for detecting execution phases of a system for the purpose of program comprehension. In *International Conference on Software Enineering Research, Management and Applications*, SERA, pages 207–214. IEEE, 2010.

[47] S. P. Reiss. Visualizing Java in action. In *Proceedings Symposium on Software Visualization*, SoftVis, pages 57–ff. ACM, 2003.

[48] S. P. Reiss. Dynamic detection and visualization of software phases. In *Proceedings of the International Workshop on Dynamic Analysis*, WODA, pages 1–6. ACM, 2005.

[49] M. Renieris and S. P. Reiss. Almost: exploring program traces. In *Workshop on New Paradigms in Information Visualization and Manipulation*, pages 70–77, 1999.

[50] S. Silva, B. Sousa Santos, and J. Madeira. Using color in visualization: A survey. *Computers and Graphics*, 35(2):320–333, 2011.

[51] IEEE Std. Standard for software maintenance, 1998.

[52] F. Steinbrückner and C. Lewerentz. Understanding software evolution with software cities. *SAGE*, 12(2):200–216, 2013.

[53] D. Surian, D. Lo, and E-P. Lim. Mining collaboration patterns from a large developer network. In *Proceedings of the 17th Working Conference on Reverse Engineering (WCRE), 2010*, pages 269–273, Oct 2010.

[54] J. Sutherland. Business objects in corporate information systems. *ACM Computer Surveys*, 27 (2):274–276, June 1995.

[55] L. Voinea and A. Telea. Multiscale and multivariate visualizations of software evolution. In *Proceedings ACM Symposium on Software Visualization*, pages 115–124, 2006.

[56] Y. Watanabe, T. Ishio, and K. Inoue. Feature-level phase detection for execution trace using object cache. In *Proceedings International Workshop on Dynamic Analysis*, WODA, pages 8–14. ACM, 2008.

[57] R. Wettel and M. Lanza. Visual exploration of large-scale system evolution. In *Proceedings of the 15th Working Conference on Reverse Engineering*, WCRE '08, pages 219–228. IEEE Computer Society, 2008.

[58] R. Wettel and M. Lanza. Visual exploration of large-scale system evolution. In *Proceedings of the 15th Working Conference Working Conference on Reverse Engineering*, pages 219–228, 2008.

[59] W. E. Wong, S. S. Gokhale, and J. R. Horgan. Quantifying the closeness between program components and features. *Journal of Systems and Software*, 54(2):87 – 98, 2000. Special Issue on Software Maintenance.

[60] J. Wu, R.C. Holt, and A.E. Hassan. Exploring software evolution using spectrographs. In *Proceedings of the 11th Working Conference on Reverse Engineering, 2004*, pages 80–89, Nov 2004.

[61] Z. Xing and E. Stroulia. Understanding phases and styles of object-oriented systems' evolution. In *Proceedings of the 20th IEEE International Conference on Software Maintenance, 2004*, pages 242–251, Sept 2004.

[62] A. Zaidman, T. Calders, S. Demeyer, and J. Paredaens. Applying webmining techniques to execution traces to support the program comprehension process. In *Proceedings of the 9th European Conference on Software Maintenance and Reengineering, 2005*, pages 134–142, March 2005.

[63] L. Zhang, T. Qin, Z. Zhou, D. Hao, and J. Sun. Identifying use cases in source code. *Journal of Systems and Software*, 79(11):1588 – 1598, 2006. Software Cybernetics.

Appendix I

Evolution Phases of the Case Studies

This appendix contains the evolution maps of the five studied systems. The phase types are described in Table 5.V.

ICEfaces1				
Release	**Date**	**Type**	**Phases**	**Nb phases**
1.6.1	31-08-07	Maintenance	B A K K A L L K L A A L	12
1.6.2	14-11-07	Maintenance	K K L L L	5
1.7.0	14-04-08	Feature	A A L A K B L A A	9
1.7.1	17-06-08	Maintenance	K A K B	4
1.7.2	08-10-08	Maintenance	L K K A B B L L L	9
1.8.0	02-04-09	Feature	L L L A A A L L L A A K L A A L	16
1.8.1	26-05-09	Maintenance	K	1
1.8.2	30-09-09	Maintenance	L K K K A L L B K L B	11

Table I.I – Evolution phases of *ICEfaces1*.

ICEfaces2				
Release	**Date**	**Type**	**Phases**	**Nb phases**
2.0.0	21-12-10	Feature	B L D L K L L L B L K	11
2.0.1	30-03-11	Maintenance	L L N A L L	6
2.1.0*B*	04-10-11	Maintenance	C N C A L C K L A M M M	12
2.1.0*B*2	04-11-11	Feature	K	1

Table I.II – Evolution phases of *ICEfaces2*.

ICEfaces3				
Release	**Date**	**Type**	**Phases**	**Nb phases**
3.0.0	03-02-12	Feature	L L A N K A M A A	9
3.0.1	27-03-12	Maintenance	B L	2
3.1.0	24-04-12	Feature	C K D A K M K B A	9
3.2.0	02-11-12	Feature	N M M A K A L N	8
3.3.0	16-04-13	Feature	K N M A N L B L K L K	11

Table I.III – Evolution phases of *ICEfaces3*.

JFreeChart				
Release	**Date**	**Type**	**Phases**	**Nb phases**
1.0.7 1.0.8a	14-11-07 07-12-07	Feature Maintenance	K L M A L K	6
1.0.9	04-01-08	Maintenance	L	1
1.0.10	09-06-08	Maintenance	N B L K A L	6
1.0.11 1.0.12	19-09-08 31-12-08	Feature	A	1
1.0.13	20-04-09	Feature	L L A A	4
1.0.14	20-11-11	Feature	K A C C K D	6
1.0.15	04-07-13	Feature	N N	2
1.0.16 1.0.17	13-09-13 24-11-13	Feature	L L M	3
1.0.18	03-07-13	Feature	L M	2
1.0.19	31-07-14	Maintenance	B	1

Table I.IV – Evolution phases of *JFreeChart*.

ArgoUML				
Release	**Date**	**Type**	**Phases**	**Nb phases**
0.16.1	04-09-04	Maintenance	B A L B L L L B L L L K B L B K A L K A K L B A	24
0.18.1	30-04-05	Maintenance	B B L L K K K L A K L A L L L K K K K L	20
0.20	09-02-06	Feature	K A A L K A K A L A K L L K N L A L A L A A K L L L A L L	29
0.22	08-08-06	Feature	K K B K K K B K K B L B L A L L	16
0.24	12-02-07	Feature	A A L K L K L L L K A A K K B L	16
0.26.2	19-11-08	Bug fix	K A A K L K K B A L L B L K B B B B K L L L K N M L K A A K A K A A A K K B L L L K A A L	45
0.28	23-03-09	Feature	A A B A A L C N	8
0.28.1	16-08-09	Maintenance	L A L M N B	6
0.30	06-03-10	Feature	L C A L L L A A A K L	11
0.30.1	06-05-10	N/A	D B B A L	5
0.30.2	08-07-10	N/A	C N N	3
0.32 0.32.1	28-01-11 23-02-11	Bug fix	B K A A D M A N	8
0.32.2	03-04-11	Maintenance	L L	2
0.34	15-12-11	Maintenance	K A L B A N C N	8
0.35.1 (dev.)	31-08-14	Maintenance	D N N N	4

Table I.V – Evolution phases of *ArgoUML*.