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

Omar Benomar Houari Sahraoui Pierre Poulin
Dept. I.R.O., Université de Montréal
{benomaro, sahraouh, poulin}@iro.umontreal.ca

Abstract—The dimension of time in software 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 paper, we claim that 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.

I. INTRODUCTION

There is a consensus that program comprehension is a major challenge in software maintenance [1], and consequently, comprehension tasks tend to involve large resources dedicated to maintenance [2]. 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 in 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 a software. This understanding allows them to make reasoned 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 in time where a program executes in a same manner, or class contributions to a particular execution scenario. There are also various comprehension problems associated with software evolution, such as identifying developer collaborations during the evolution, or periods of time where 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. [3] investigate the use of graph-based characterization of a software system to capture its evolution, and analyze developer collaborations at the commit and bug fixing levels.

In the execution analysis research community, Dugerdil and Alam [4] 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, Lanza et al. [5] use the same visualization, based on the city metaphor, to study the evolution of a software system. Barry et al. [6] also study the problem of evolution comprehension by revealing evolution patterns of a software. They classify evolution of several software systems according to their change volatility. From the execution perspective, the problem of phase identification arises in understanding execution parts, where a program performs a high-level feature, such as file I/O operations, sending emails, etc. For instance, Watanabe et al. [7] 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, this differentiation reflects in the software visualization communities, where we can distinguish between two distinct communities. First, execution visualization, interested in the depiction of execution states of software over time, and understanding its dynamic behavior (see [8], [9]). Second, evolution visualization, interested in the representation of software changes from one version to another (see [10], [5]). These two communities treat software’s time dimension visualization in a disconnected manner. However, they share common concepts of software’s time dimension representations. 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 [11], [12]). On the other hand, some automatic approaches investigate the comprehension of software evolution to gain insight on how software evolves during its lifetime (see [6]). Although these contributions rely on the representation of time in software, there is no explicit model for its representation, and if there is one (e.g., [13]), it is only for one of the time dimensions.

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

The rest of this paper is organized as follows. Section II details the proposed unified meta-model of software’s time comprehension. Section III illustrates the use of the comprehension framework for the phase detection problem. Section IV shows the application of our approach to the collaboration comprehension problem. Finally, Section V concludes.

II. UNIFIED COMPREHENSION FRAMEWORK

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 the proposed 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 nature 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 1. The meta-model 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. Events of a sequence appear as actions that occur periodically in 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.

III. 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 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 I, contributions from the two different communities (see [7], [6]), 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 formulate phase identification as an optimization problem, solved as a partition of a trace into sequences of events according to some heuristics, which are translated to metrics to evaluate the solution’s quality.

A. 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 evolution phase identification 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 undergo. 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 classes. Here the source code of a class at any given time (commit’s time stamp) refers to its property in our meta-model.

We use the classification proposed by Fluri and Gall [14], where change types are assigned an ordinal scale, according to their change impact and functionality preserving/modifying characteristic, that correspond 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. [15]. Hence, we gather the significance levels for every change between any two consecutive commits (i.e., events in our meta-model) of a class. The formulated optimization problem is solved using a meta-heuristic to find a good solution, i.e., a good decomposition of the evolution history (sequence in our meta-model). Phase cohesion is computed as the similarity between changes to the same class within a phase in terms of significance level and size of changes. Another phase cohesion metric used is the similarity in development rate, using mean time between commits. Phase dissimilarity is computed as the ratio of different classes being involved in a phase with respect to its subsequent phase. We applied our phase detection 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. Results show that our algorithm was able to rediscover most 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, which suggests a change of development activity. 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. The identified phases are characterized by comparing the value of the computed metrics for a phase, to the value of the same metric for the entire evolution history. For instance, release JFreeChart 1.0.18 is composed of two evolution phases. First, there is a phase of less important changes, rapid development, and similar types of changes. Then, the release’s second phase is characterized by a slower development and more variety in change types. Details of the approach are reported by Benomar et al. [16].

B. Execution Phases

We consider the entire execution trace of a particular usage 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 creation, method call, and method return. The goal is to decompose the execution trace into phases representing executions of high-level features. Phase identification is solved by optimization 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 to the phase identification of seven execution scenarios from two softwares: 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%. Benomar et al. [17] detail the application of the execution phase detection algorithm.

IV. 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 instantiated our meta-model to represent the problem, and used the model to solve it. Each entity has a certain activity 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, 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 between aligned heat maps helps to visually reveal differences. Finally, multiple opened heat maps allow visual exploration of differences and similarities between colored elements. Benomar et al. [18] provide details on this approach and report results of the application of heat-map visualization to collaboration comprehension problem in the execution and evolution contexts.

A. Evolution: Developer Collaboration

In evolution comprehension, developers are the subjects who commit changes to a software. Each commit is an instance of an event that introduces a change in the source code of a class. Classes 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 a software system. These contributions are depicted in the form of heat maps where intensity of changes is mapped to a gradient of colors. Then, the techniques of collaboration analysis mentioned earlier allowed us to easily answer questions that developers ask [19], such as “Who changed this class?”, “Which class has been changed most?”, etc.

Figures 2(a) and 2(b) show heat maps that represent the contributions of developers dnoyeb and mrfloppy, respectively, to JHotDraw 5.4.1. The contributions of these two developers are combined to bring up their collaboration. The combination of the heat maps uses color weaving, and the resulting heat map shows common activity, as illustrated in Figure 2(c).

B. Execution: Class Contribution

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 collaboration between classes in an execution. We could thus easily answer questions such as:

A. Which classes are the core of a use-case scenario?
B. Which classes are specific to a certain feature?
C. How do classes collaborate to the execution of a use-case?

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 (time stamp) of execution. Each event introduces a change in the execution state. In order to answer questions like those stated above, we generate heat maps that depict class activity in a use-case scenario. Hence, a heat map shows the mapping of a color gradient to class activity for each use-case. We also provide a mean to aggregate class contributions of several alternate use-cases 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 IV. For instance, to answer Question B, we generate heat maps for two use-case scenarios,
where one of them includes an extra feature. Flipping between the two heat maps or comparing them side-by-side reveals the classes contributing to the extra feature. Figure 3 illustrates how we identify classes intervening in the execution of feature add sender to address book by comparing two alternate use-case scenarios of Pooka in Figures 3(a) and 3(b).

![Fig. 3: Comparison of two alternate executions of a use-case. Classes in the blue rounded box are active in (b), but show no activity in (a).](image)

V. CONCLUSION

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

This novel idea suggests that understanding software’s time dimension requires the same comprehension framework for both execution and evolution. Another example of a common problem that could benefit 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.

REFERENCES