Detection of Software Evolution Phases based on Development Activities

Omar Benomar, Hani Abdeen, Houari Sahraoui, Pierre Poulin, Mohamed Aymen Saied
DIRO, Université de Montréal, Canada
{benomaro, abdeenha, sahraouh, poulin, saiedmoh}@iro.umontreal.ca

Abstract—Software evolution history is usually represented at fine granularity by commits in software repositories, and at coarse granularity by software releases. In order to gain insights on development activities and on software evolution, the information on releases is too general, whereas the information on commits is prohibitively large to be efficiently processed by a developer. This paper proposes an automatic technique for the identification of distinct phases of evolution. Such software evolution phases are characterized by similar development activity in terms of changes to entities. Therefore, our technique decomposes software evolution history to assist developers identify periods of different development activities. Our analysis technique is a search-based optimization of the best decomposition of commits from the software repository using heuristics such as classes changed in each commit, and the magnitude/importance of these changes. To validate our technique, we applied it on the evolution history of five case studies covering multiple releases over several years of development. An interesting outcome of the evaluation is that our automatic decomposition of software evolution history recovered the original decomposition in software releases.

I. INTRODUCTION

A software system continuously evolves as it is maintained and enhanced over its lifespan [1]. To understand software evolution, we need to examine the dynamic behavior of a system over time [2]. A software repository records a trace of the evolutionary path taken during the realization of a software system, from all its previous versions to its current status. This evolution trace usually consists of commits, and often spans several years of development. On the one hand, commits represent atomic changes applied to software modules, and hence hold evolution information at a fine granularity. However, over a typical long period of development, the amount of information associated to commits is overwhelming. This makes any attempt to gain higher-level insights on the software evolution very challenging for a developer [3]. On the other hand, software evolution can also be represented by information collected at successive releases. A release event is a public event in software evolution, which is taken by the decision makers to set the boundaries of an iteration in software development [4]. Release notes, when available and rigorously documented, include information such as bug fixes, updated/new features, etc. However, information included in release notes is at a too coarse granularity [5].

Therefore, we are interested in techniques that can automatically describe the evolution process and provide a balance between the abstraction levels of commits and releases. The description should provide a periodical overview to help software managers better understand major development activities occurring in relevant periods of the software development process. For instance, a good overview should allow managers to answer the following questions: What type of changes better describes a given development period, and distinguishes it from other periods? What is the significance of changes during this period? Which classes/files most activities were performed on? How rapid or slow was the rhythm of development?

To ease automation of the recovery process and minimize human intervention, we must avoid to describe software changes using subjective terms, such as perfective, corrective, adaptive, feature addition, nonfunctional improvement, etc. We believe, as other researchers do (e.g., Kothari et al. [3]), that describing software changes using such terms is impossible in an accurate and automated fashion. Unlike existing work on understanding software evolution (e.g., [2], [6]–[8]), we seek an approach that describes software changes based on concrete facts, such as: the type of changes (e.g., add method, remove method, update method, add class, rename attribute or parameter, etc.) [9], the amplitude and significance of changes according to their potential impact on the source code [10], the classes/files involved in changes, the periodicity of changes (i.e., the rhythm of development) [7]. All this information can be automatically retrieved using information from the code and commits in source-control repositories.

To this end, we propose an approach that automatically analyzes the evolution of code from software repositories over the software’s lifespan, as well as the temporal rhythm of its development activities. It uses search-based optimization techniques to find the best decomposition of the evolution process into phases of development activities. In our approach, an evolution phase should delimit a period of time that is characterized by: (1) a dominant type of changes, that is different than the types of changes that characterize previous and next phases; (2) a regular rhythm of development activities; (3) a distinct set of classes undergoing changes over this time period, that is different than the set of classes changed in previous and next phases; and (4) a similar significance for the changes that all classes undergo through during the phase.

Then, we transform the above set of heuristics into metrics that will be used in our decomposition approach of software evolution in order to classify detected evolution phases in eight categories. Our classification should enable software managers to understand the major development activities that characterize a time period, and detect recurrent patterns in
the software development process. For instance, managers can distinguish between releases consisting of fewer phases characterized by important changes and slow development, from releases consisting of many phases characterized by less important changes and rapid development. They can also identify the location and duration of phases of slow development, and how software changes during these phases. Then, they can identify periods when development activities and rhythm did not change much, but classes involved in development activities changed.

The paper is organized as follows. Section II presents related work. The phase identification approach is detailed in Section III, while the proposed classification of phases is described in Section IV. Our approach is evaluated in Section V before concluding in Section VI.

II. RELATED WORK

Related work closer to ours is about analyzing software evolution for understanding evolution processes and identifying common evolution phases. Xing and Stroulia [6] present an approach for understanding evolution phases and styles of object-oriented systems. The authors use a structural differenting algorithm to compare changing system class models over time, and gather the system’s evolution profile. The resulting sequence of structural changes is then analyzed to gain insight about the system’s evolution. Barry et al. [7] 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, and sequence analysis is applied to reveal similar patterns in time. Bennett and Rajlich [11] introduce a staged model for software life cycle. The model comprises five distinct stages: initial development, evolution, servicing, phase out, and close down. Each stage is defined by specific characteristics, such as minor corrections and enhancements during the servicing stage, and major changes during the evolution stage. The authors also propose a versioned staged model where each version of the system follows the staged model.

Kemerer et al. [2] use sequence analysis for mining patterns of software evolution from change logs. In their approach, change events are categorized into 30 categories that are based on (1) three subjective classifications for changes (maintenance tasks: corrections, adaptations, and enhancements); (2) six basic categories for changed modules (data handling, logic, structure, computation, user interface, and module interface); and (3) three types of modifications that can be performed in “enhancement” events (add, change, or delete). To categorize change events, only the text/comments describing change logs are used, regardless of changed entities in the code, and the granularity or amplitude of changes. Hindle et al. [8] propose an approach to classify large commits into five categories of maintenance tasks: corrective, adaptive, perfective, feature addition, and non-functional improvement. In a previous study [12], they state that large commits are usually associated to perfective tasks, while small commits are associated to corrective tasks. However, Kothari et al. [3] argue that it is not possible to categorize software changes in a confident and automated fashion using the aforementioned specific categories of maintenance tasks. They propose an automated approach that examines the temporal evolution of source code to identify change clusters. These change clusters classify code-change activities as either a software maintenance or a new development. To identify change clusters, their approach first identifies a subset of canonical changes that best represent the modification activities in a given time period. Then, using clustering techniques, it classifies all other changes in the time period as belonging to one of the identified canonical clusters. The authors use change clusters to identify trends in the development process.

Unlike the above mentioned studies, and regardless of evolution phases and maintenance tasks, some studies analyzed structures in commit histories, or in release-to-release iterations of development. Biazzini et al. [13] analyze the topology (structure) of commit histories in de-centralized version control systems using the repository Metagraph. They model commit history as a directed acyclic graph whose nodes are commits and edges are parent/child relations. Hindle et al. [4] analyze the behavior of projects around the time of release. They use four classes to categorize release revisions: source code, testing, build, and documentation. Then, they use these classes to identify behaviors that occur before, during, or after releases, that they call release patterns.

From other perspectives, a family of research papers focuses on analyzing commits for understanding intents of implementations [14], identifying commits that contain tangled changes [15] or peripheral modifications [16], providing further insights on the nature of commits [17], [18], predicting latent software bugs in commits [19], or identifying the commit window of a release [20]. Another body of work analyzes commits and code changes to detect logical and evolutionary coupling between software entities [21], [22], or for using topic models to study software evolution [23].

III. PHASE IDENTIFICATION

The process of software evolution typically involves various development activities. Identifying which part of an evolution is related to which development activity can help managers in their comprehension and maintenance tasks. However, identifying this relation between software evolution and development activities is challenging, since source-control repositories store only explicit information about the software evolution (revisions or commits). In this work, we aim at approximating this relation using heuristic search. Before introducing the heuristics that we are going to use, we first define related concepts. Then we describe how we gather the necessary data for our algorithm. Finally, we present the genetic algorithm as an implementation of the heuristics to identify software evolution phases.
A. Basic Definitions

The notions used in our approach are derived from a meta-model unifying software comprehension problems in the contexts of evolution and execution, by Benomar et al. [24]. They are defined as:

**Event:** An evolution event corresponds to *one day* of development in the history of a software. All commits of the event are considered as part of the same revision.

**Trace:** An evolution trace is the software evolution process divided into a sequence of consecutive events. It is important to distinguish an evolution trace in this paper from the more familiar concept of an execution trace.

**Entity:** An entity is a software class on which changes are applied during evolution.

**Activity:** An activity relates to changes that an entity undergoes from its first commit to its last.

**Phase:** An evolution phase is a period of time in a software’s life-cycle that is characterized by a certain development style, different than the development styles before and after that period.

**Cut position:** A cut position is a time stamp of a commit, indicating a transition between two successive phases.

**Phase identification solution:** A solution is a set of cut positions, that decomposes the evolution trace into phases.

B. Mining Software Repository

We use information from the evolution history in the software repository to identify evolution phases. We first acquire the evolution history log (e.g., an *svn* log), and then we construct the evolution trace by segmenting the software evolution into individual events combined by a per-day unit. An event might therefore consist of several commits, in which case, they are all aggregated. Events without any commit are discarded because we are interested in periods of time where there is an actual (recorded) development activity.

From the resulting evolution trace, we identify the entities involved in each event, i.e., the entities affected by the event’s commits. Then, for each entity in each event, we compute the entity’s changes from one revision to the other in terms of ‘types’ and ‘significance level’. To compute entity changes, we use *ChangeDistiller* [9], a tool that applies the change distilling technique based on a tree-differencing algorithm for fine-grained source-code change extraction. Basically, it takes as input two versions of an entity source code, constructs their abstract syntax trees (ASTs), and extracts detailed change information, such as types and significance levels of changes. The notion of significance level is taken from the work of Fluri and Gall [10]. Change types are classified according to their change impact and functionality preserving or modifying property; they are divided into five categories: *Crucial, High, Medium, Low,* and *None*. For instance, renaming a parameter in a method signature may have high impact on other source-code entities. However, it preserves the functionality of the method, and hence, parameter renaming is assigned a *Medium* significance level.

C. Heuristics

Our technique for identifying evolution phases relies on assumptions about the locality, nature, and dispersion/frequency of changes applied to software modules during their evolution. This is outlined as follows:

- Changes performed within an evolution phase should be of similar nature, and ideally, they should have the same magnitude and importance. Moreover, the type of changes performed on entities must differ from one evolution phase to the previous/next phases.
- The cadence of development must be the same throughout an evolution phase. It could be rapid or slow, but should be relatively constant within a phase.
- Two successive evolution phases should not involve the same complete set of entities.

D. Phase Identification Algorithm

Identifying evolution phases boils down to determining the best decomposition of the events in an evolution trace according to the heuristics outlined in Section III-C. Unfortunately, an exhaustive search is prohibitively large because the search space explodes relative to the number of events, and an evolution trace has hundreds to thousands of events. In fact, the number of possible solutions is the combinatorial \( \binom{n-1}{m-1} \) for \( n \) evolution events and \( m \) phases in a particular solution. Moreover, \( m \) is unknown and bound to \( 1 \leq m \leq n \), since at one extremity, the entire evolution trace can be a single phase, and at the other extremity, each event can be a phase. Building up from our work on identifying program execution phases [25], we formulate the problem of identifying evolution phases as an optimization problem and use a global-search genetic algorithm to find a near-optimal solution.

The optimization algorithm starts by creating an initial population of randomly generated solutions, and in an iterative process, produces a new population/generation of solutions. These solutions are derived from the solutions in the previous iteration, using crossover and/or mutation operators. In the following, we detail the main aspects of the algorithm.

E. Solution Encoding

The one-day events ordered along the time dimension must be organized in sets of consecutive events, each set constituting one phase. A solution thus decomposes the evolution trace into evolution phases. The goal of the optimization algorithm is to find the best cut positions. Figure 2(a) schematizes solutions A and B for an evolution trace consisting of \( n \) events (days). Both solutions have the same global length, as each one represents the entire evolution trace. Solution A is a decomposition in 7 phases delimited by 6 cut positions. Solution B is a decomposition in 5 phases with 4 cut positions. A solution can be represented by a vector of its cut positions, where the vector length indicates the number of identified phases in the solution. Therefore, solutions can be vectors of different lengths.
F. Initial Population

Our algorithm starts by creating a diversified random initial population of $N$ solutions, with different numbers of phases at different cut positions. Identical solutions are easily eliminated, thanks to the vector representation. The number of phases of the solutions is bound in the initial population. However, our technique is not limited by a fixed number of phases or even a subset of phases’ numbers. Therefore, the number of phases may exceed the maximum number of phases generated in the initial population, during the search using our operators (crossover and mutation). Finally, only solutions with a fair fitness are incorporated into the initial population, to allow the search to start form a good overall set of individuals.

G. Fitness Function

To convert the proposed heuristics from Section III-C into measurable properties, we developed four metrics that are combined into the fitness function. The search process is guided by this function towards a solution that best satisfies those heuristics. Note that to simplify the notations in this section, we denote phases by $i$ and $j$.

**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 1 illustrates the concept of phase entity coupling with a trivial example. On the one hand, the entity $A$ is subject to 4 modifications in the phase $i$ and 2 modification in the phase $j$. Hence, it is more important to phase $i$. On the other hand, the 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 phase entity coupling as entities in phase $i$ are not shared with entities in phase $j$. The phase coupling is to be minimized in order to satisfy our heuristics. The coupling between two successive phases $i$ and $j$ is computed as:

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

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 1, the entity coupling value is evaluated to 0.45, which is explained by the fact that 50% ($\frac{5}{10}$) 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 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 common 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 of 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. [7], 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 [10]. 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 the 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:

$$\text{mean}_i(c) = \frac{1}{n_i} \sum_{k} n_{ki}(c)$$

$$\text{var}_i(c) = \frac{1}{n_i} \sum_{k} [n_{ki}(c) - \text{mean}_i(c)]^2$$

$$\text{CICh}(i) = \frac{1}{5} \sum_{c=1..5} \frac{1}{\text{var}_i(c)}$$

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 $c$, $\text{mean}_i(c)$ is the mean of the number of changes of significance level $c$, and $\text{var}_i(c)$ 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. [7]. Similarly to the phase change-importance cohesion, we compute the development-rate cohesion metric (to maximize) as the inverse of the change-importance cohesion, we compute the development-rate cohesion metric of a phase $i$ is:

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

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

Fitness Function: For a given solution $s$, coupling metrics $\text{ECp}$ and $\text{CTCp}$ are computed for each cut position (two consecutive phases) in the solution, and the coupling values for the solution, $\text{SECp}$ and $\text{SCTCp}$, are computed as the average of all coupling values associated to the solution’s cut positions. Similarly, the solution cohesion metrics, $\text{SCiCh}$ and $\text{SDRCh}$, 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:

$$\sqrt[4]{\text{S}(1-\text{ECp})(s) \times \text{S}(1-\text{CTCp})(s) \times \text{S}\text{CICh}(s) \times \text{S}\text{DRCh}(s)}$$

H. 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 evolution trace. These cut positions act as the individuals’ chromosomes and are used for crossover between two parent solutions to produce two child solutions that will populate the next generation. A solution may also be modified by mutating its chromosomes (cut positions).

Elitism: To create the new population in a given generation, we first automatically add the current 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 the tournament selection strategy. A fixed number of candidate parents are randomly chosen from the current population and the best solution among them is retained for reproduction. For two parent solutions selected, we apply the crossover and mutation operators with certain probabilities, where the crossover probability is higher than the mutation probability at the start of the search. Probabilities are adapted during the search if the algorithm gets stuck at the same solution during a certain number of iterations, in order to inject more new genetic material with mutations instead.

![Fig. 2. (a) Solution A (resp. B) represents the decomposition of the evolution history into 7 (resp. 5) phases. (b) The resulting offspring solutions of the crossover operator applied to the parent solutions in (a).](image-url)

**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 is at the same location in both solutions,
and it produces two parts for each solution. The upper part of solution A is combined with the lower part of solution B to form a child solution (AB), and conversely, the upper part of solution B is combined with the lower part of solution A (BA). The two siblings share the new random position which was not present in their parents, and receive portions of their parents’ chromosomes. This is illustrated in Figure 2(b) with the new cut position indicated in blue.

**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 in the solution. The second mutation strategy consists in merging two successive phases into a single one, where we randomly select one cut position and discard it. The third mutation strategy randomly changes a cut position, i.e., the boundary between two consecutive phases is re-located within the phases. This results in the alteration of existing phases without changing the number of phases in the solution. Figure 3 illustrates the three mutation strategies applied to solution A from Figure 2(a).

**IV. 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 III-C. 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 what development activities happened in it, managers need to compute the metrics that we outlined in Section III-C 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 [9]. 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. [7]. 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 1. 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 descriptions of the values for each criterion. However, this would increase the number of phase types and thus, it would complicate the classification.
V. Evaluation

We evaluate our approach from different perspectives:

- **Stability of our algorithm**: We evaluate the stability of the solutions produced by our search-based phase identification technique. To this end, for a software system and a period of evolution, we assess the similarity between solutions produced in different runs by analyzing the distance between cut positions of the resulting solutions.

- **Comparison with time stamps of releases**: As a release represents the decision to demarcate the end of a development iteration from the beginning of the next one [4], it is reasonable to consider the time stamps of releases as important reference cut positions of the software evolution. For our solutions, we define a cut position as a “release cut position” if it is very close to a release time stamp. Ideally, our approach should always find cut positions that are identical to the cut positions due to releases.

- **Comprehension of software releases**: We use the classifications proposed in Section [IV] to describe the identified phases in terms of development activity characteristics. We use the sequence of identified phases to understand the development activity leading to a software release. As the description of software evolution with the proposed classification is represented as a sequence of phase types, we use sequence visualization for comprehension tasks.

A. Data

**Studied Systems**: We applied our evolution phase identification technique to describe development activities of the following five systems: ArgoUML [26], JFreeChart [27], ICEfaces1, ICEfaces2, and ICEfaces3 [28]. The studied systems are different in size, and have different development periods. For each system, Table [II] 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 the last release date. This is indicated in Table [II] 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.

<table>
<thead>
<tr>
<th>System</th>
<th># commits</th>
<th># classes</th>
<th># releases</th>
<th>Period</th>
</tr>
</thead>
<tbody>
<tr>
<td>ArgoUML</td>
<td>9150</td>
<td>2748</td>
<td>15+1</td>
<td>10/09/03 - 31/07/14</td>
</tr>
<tr>
<td>JFreeChart</td>
<td>3000</td>
<td>1482</td>
<td>13</td>
<td>19/06/07 - 31/08/14</td>
</tr>
<tr>
<td>ICEfaces1</td>
<td>3916</td>
<td>1371</td>
<td>8+1</td>
<td>22/02/07 -11/11/14</td>
</tr>
<tr>
<td>ICEfaces2</td>
<td>882</td>
<td>1088</td>
<td>4+1</td>
<td>19/07/10 - 30/05/12</td>
</tr>
<tr>
<td>ICEfaces3</td>
<td>2418</td>
<td>1827</td>
<td>5+1</td>
<td>18/11/11 - 24/01/14</td>
</tr>
</tbody>
</table>

**Evolution Data**: The evolution data are 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 [9]. The tool takes as input two versions of the same class and returns all the changes between them, with each type and significance level [10]. 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.

B. Setting

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.

C. Stability and Similarity with Reference Solutions

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 7. 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. In Figure 4, this gives the associations $\{(c_{b1}, c_{a2}), (c_{b2}, c_{a3}), (c_{b3}, c_{a4}), (c_{b4}, c_{a4}), (c_{b5}, c_{a6})\}$.

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 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.

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 4 $S_a$ 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 III 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.

<table>
<thead>
<tr>
<th>System</th>
<th>Stability</th>
<th>Reference Similarity</th>
<th>Reference Recall</th>
</tr>
</thead>
<tbody>
<tr>
<td>ArgoUML</td>
<td>0.80</td>
<td>0.80</td>
<td>0.94</td>
</tr>
<tr>
<td>JFreeChart</td>
<td>0.90</td>
<td>0.79</td>
<td>0.85</td>
</tr>
<tr>
<td>ICEfaces1</td>
<td>0.82</td>
<td>0.87</td>
<td>1.0</td>
</tr>
<tr>
<td>ICEfaces2</td>
<td>0.86</td>
<td>0.76</td>
<td>1.0</td>
</tr>
<tr>
<td>ICEfaces3</td>
<td>0.90</td>
<td>0.83</td>
<td>1.0</td>
</tr>
</tbody>
</table>

TABLE III  EVALUATION OF THE EVOLUTION PHASE IDENTIFICATION.

D. Software Release 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. We give examples of how to characterize software releases using the phase classification proposed in Section IV. For illustrating purposes, we propose two representations of phases and releases in Figures 5 and 6.

Figure 5 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 6 is another visualization of the data in Figure 5; 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 6 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 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 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...
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 releases of ArgoUML and all releases of ICEFaces1. The noticeable difference between the aforementioned releases is that the number of transitions between evolution phases, which involve important changes (red family of phases) and less important changes (green family of phases), is much larger in the ArgoUML releases than in the ICEFaces1 releases. Focusing on the beginning of releases, Figure 6 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. Precisely, 12 releases begin with evolution phases from the red family of phases (A, B). Releases that begin with slow development rhythm involving less important changes are rare. In our study, only 3 releases begin with phases from the gray family (N, M). Hence, we think that these are exceptional cases. This leads us to identify different categories of releases.

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

- **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.

- **Slow development:** As opposite 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 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 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 2.1.0Beta2, 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 slow development rhythm. In our sample, these releases are considerably less frequent than rapid development releases. Figure 6 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.

Other analyses may be performed on sequences describing software releases. For instance, a similar approach to Figure 6 can be used to find similarities between releases in their last phases or intermediate phases. Other sequence analysis techniques can also be used to estimate the probability of transitions between the different types of evolution phases.

VI. CONCLUSION

We present an approach to identify evolution phases undergone by software. Phases, as defined by the development activities performed during software evolution, are identified using a search-based genetic algorithm. Detecting phases in software evolution history is formulated as an optimization problem of finding the best decomposition of the evolution history, that maximizes certain heuristics in phases. The heuristics describing development activities include similar development rates, and similar sets of entities modified with comparable changes in importance and types. The identified evolution phases are then characterized based on their development activities, and used to represent the software evolution as a sequence of phases. We showed that our algorithm converges to similar solutions, which indicates its stability, and we compared the computed solution to the decomposition of software evolution in terms of releases. The results illustrate that after most releases, the development activity changes, and that development leading to software releases is composed of several stages. Finally, we analyzed several releases of different software, and found similarities with respect to the development activities leading to those releases.

REFERENCES


[15] K. Yamauchi, J. Yang, K. Hotta, Y. Higo, and S. Kusumoto, “Studying software evolution history is formulated as an optimization problem of finding the best decomposition of the evolution history, that maximizes certain heuristics in phases. The heuristics describing development activities include similar development rates, and similar sets of entities modified with comparable changes in importance and types. The identified evolution phases are then characterized based on their development activities, and used to represent the software evolution as a sequence of phases. We showed that our algorithm converges to similar solutions, which indicates its stability, and we compared the computed solution to the decomposition of software evolution in terms of releases. The results illustrate that after most releases, the development activity changes, and that development leading to software releases is composed of several stages. Finally, we analyzed several releases of different software, and found similarities with respect to the development activities leading to those releases.

REFERENCES