Studying The Evolution of Software Systems Using Evolutionary Code Extractors

Ahmed E. Hassan and Richard C. Holt
Software Architecture Group (SWAG)
School of Computer Science
University of Waterloo
Waterloo, Canada
{aeehassa,holt}@plg.uwaterloo.ca

ABSTRACT
Software systems are continuously changing and adapting to meet the needs of their users. Empirical studies are needed to better understand the evolutionary process followed by software systems. These studies need tools that can analyze and report various details about the software system's history.

In this paper, we propose evolutionary code extractors as a type of tool to assist in empirical source code evolution research. We present the design dimensions for such an extractor and discuss several of the challenges associated with automatically recovering the evolution of source code.

1 INTRODUCTION

Software systems are continuously changing and adapting to meet the needs of their users. A good understanding of the evolution process followed by a software system is essential. This would permit researchers to build better tools to assist developers as they maintain and enhance these systems. Furthermore, it will pave the way for the investigation of techniques and approaches to monitor, plan and predict a successful evolutionary paths for long lived software projects.

We could study the evolution of a number of facets of a software project such as its requirements, its architecture, its source code, its bugs reports, or the interactions and communications among its developers. Each facet offers insight on a variety of issues surrounding the evolution of a software system. For example, studying the complexity of the source code or the number of reported bugs over time may give us a better understanding of how bugs are introduced in software systems. It may also assist us in building models to predict bugs and models to guide managers in allocating testing resources where they are needed the most [6].

To perform such studies a good record of these facets throughout the lifetime of a project is essential. For example, a record of the requirements of a software system since its inception till the current day is needed to study the evolution of its requirements. For some facets such as the requirements of a software system, such records rarely exist and if they do exist they tend to be incomplete or too high level. For other facets such as the features of a software system, they may be well documented in release notes, but it may be challenging and time consuming to recover them. For example, Anton and Potts have manually traced the evolution of features for telephony services [1]. Their study focused on telephony services in a single city due to the long time and resources required to distill and describe the evolution of features from the phone books.

We should focus on facets for which most projects have good historical records and which can be automatically analyzed with minimal effort. An empirical approach permits us to generalize our findings instead of associating them to peculiarities of specific systems. Luckily, a large number of software projects store artifacts generated throughout their lifetime in software repositories. For example, the source code and changes to it are recorded in a source control repository. The released versions are usually stored in release archives. Other repositories archive the mailing lists and emails among the project's developers. Bug tracking systems record various details regarding reported bugs and their fixes. These repositories provide a great opportunity for researchers to acquire empirical data to assist them in studying evolution.

To ensure that we can perform our studies on several software systems, we need tools that automatically recover data from these repositories and present the data in a standard format that is easier to process. This would permit researchers to focus on analyzing the recovered data instead of spending a large amount of time building tools to recover the data first.

The source code of a software project can be thought of conceptually as the DNA of the software. The source code encodes the software system's functionality. Studying changes to the various characteristics of the source code will help us understand the evolution of the software system. Moreover, there is a large body of research which demonstrates approaches to recover characteristics of the source code using automated techniques. Hence the source code is a very attractive facet of software project to study its evolution using automated techniques. In this paper, we argue the need for tools that could process the source code history of a software project and generate useful data automatically. We call such software tools *evolutionary code extractors*, as they extract the evolution of source code.

Organization Of Paper

The paper is organized as follows. Section 2 tackles the issue of describing the evolution of source code. We discuss several ways to describe the same change to the source code. We argue the need to choose descriptions which can be recovered automatically. Section 3 overviews the dimensions associated with studying and recovering the evolution of source code. The choices made by researchers along these dimensions influence the techniques used to build evolutionary code extractors. Section 4 highlights the challenges and complications that arise based on the choices along the dimensions presented in Section 3. Section 5 describes prior work which dealt with studying source code evolution. The prior work is presented and the design choices used by their extractors are explored using the dimensions presented in this paper. Section 6 concludes the paper with parting thoughts about evolutionary extractors and their benefits for studying software systems and validating our understanding of the evolution process followed by software systems.

2 DESCRIBING SOURCE CODE EVOLUTION

Describing the evolution of the source code boils down to describing the changes that occur to it. The simplest way to describe source code changes is by describing its effect on the code size (the addition and removal of lines of code).

We are interested in ways to describe source code changes that can be automatically recovered and which are richer than simply describing the addition and removal of lines. For example, even though terms such as perfective, corrective, and adaptive are usually used to describe changes to the code; it is not possible to confidently and accurately describe the evolution of a software system in an automated fashion using these terms. We would require a large number of heuristics, human intervention, and intuition to rank changes to source code accordingly. In short, we seek approaches that provide a balance between the richness of the recovered descriptions and the ease of automating the recovery process.

Consider a developer who is asked about her/his activities in the last few days, a number of replies are possible. Each reply describes the activities performed at a specific level of detail and in respect to particular characteristics of the software system. For example, the developer working on a text editor software system may say: "*I added support for saving a text file, I also fixed a bug in the layout engine used by the editor.*" This reply describes change at the feature level.

Instead if we were to ask the developer to elaborate more on her/his changes and their effect on the source code ($i.e.$ to describe her/his changes to the source code), we are bound to get an even larger and more diverse number of replies which describe the same exact change work from different perspectives. The following is a list of possible replies.

1. *I changed 5 lines in the source code.*
2. *I added 3 lines in file main.c. I also commented out 2 lines from file layout.c.*
3. *I added 1 line in the main() function, 2 lines in the init() function, and removed 2 lines from the refreshLayout() function.*
4. *I got the main() function to call function init() and I added a check in the init() function to make sure the filename is set before I call refreshLayout(). Also in the refreshLayout() function, I no longer check if the filename is set.*

The first reply deals with changes to the size of the overall system. The second reply is more specific, it specifies the location (files) of these changes. The third reply is even more specific than the second reply as it maps the changes to the exact function (source code entity) where they occurred. Finally, the fourth reply describes the change using its effect on the call dependencies between the code entities. Table 1 summarizes the developer's replies.

Reply #	Characteristic	Level Of Detail
1	Size (LOC)	System
2	Size (LOC)	File
3	Size (LOC)	Function
4	Structural (Call Dep.)	Function

Table 1: Classifying Developer's Replies About Code Changes

3 THE DIMENSIONS OF CODE EVOLUTION

In the previous section, we showed that a simple change could be described in a number of ways. Each way focuses on a particular characteristic of the source code at varying levels of details. Researchers studying the evolution of source code need to build tools (evolutionary code extractors) to recover and describe this evolutionary process. We believe that there are a number of design dimensions which they should address before they embark on building these tools. In this section we focus on these design dimensions and list the choices associated with each dimension.

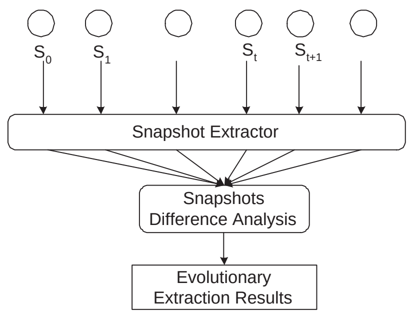
S_0
S_1
S_t
S_{t+1}
Snapshot Extractor
Snapshots Difference Analysis
Evolutionary Extraction Results

Figure 1: Recovering the Evolution of Source Code

Frequency of Snapshots

The source code of a software is continuously changing. We need to determine the frequency at which we should observe the code. Consider Figure 1, conceptually to monitor the evolution of the source code we need to decide on a number of historical snapshots of the system's source code. We then need to define some characteristics of these snapshots and study the differences between consecutive snapshots along these characteristics. The frequency of observations/snapshots determines the number of snapshots and their moment in time. Several methods exist to define snapshots:

- **Event based**: Source code progresses through different events throughout the lifetime of a project. For event based snapshots, we would use project events to determine the snapshots. Examples of these events are:
  - *Change*: A code change is simply the addition, removal or modification of a single line to a software system. Using a Change frequency would produce the largest number of snapshots, due to the large number of changes that occur throughout the lifetime of a software system.
  - *ChangeList*: A changelist is the grouping of several code changes to represent a more complete view of a change. For example, a changelist may contain two changes – one change is the addition of a function f2() and the other change is the addition of a call of function f2() in function f1(). These two changes might be required to implement a specific feature or fix a particular bug.
  - *Build*: A build represents the grouping of several features. Builds are usually done to merge the various features that have been developed by the team members. Builds may be requested by the project lead as an indication of achieving development milestones or to track the progress of a project towards the final release. Nightly builds, release candidate builds, and feature-complete builds are examples of builds.
  - *Release*: A release represents the grouping of a large number of features. The release is sent to customers and users.
- **Time based**: Time based snapshots are independent of the project and source code state. Instead they are done based on calendar time, such as weekly, monthly, and quarterly snapshots.

If we were to build an evolutionary extractor, we would conceptually have to process each snapshot using a *snapshot extractor*. The snapshot extractor would determine the characteristics of each snapshot. Then we would perform a snapshots difference analysis. This analysis would determine changes in the studied characteristics between each pair of consecutive snapshots (see Figure 1).

The choice of which snapshot frequency to use is dependant on the type of analysis that will be performed on the recovered data. If we were to study the average number of functions that must be changed to implement a feature then a changelist frequency may be the most appropriate choice.

The choice of the frequency of snapshots determines the number of snapshots which will be studied. If release snapshots are used then we will have a smaller number of snapshots in comparison to using change snapshots. The number of snapshots affects the performance of an evolutionary extractor. The larger the number of snapshots, the more time is required to perform the analysis.

Data Source

When studying the evolution of living creatures throughout time, we are usually limited by the availability of fossils of these creatures. Or if we are able to monitor the creatures as they evolve we are limited by how often we monitor them. Whereas for studying the evolution of source code, we have a much richer fossil history. Source code control systems, which are available for many long-lived software systems, store each change to the source code. Hence, we can track the evolution of the source code at a very high frequency (*change frequency*). If we were to draw a parallel to monitoring the evolution of a living creature, the data stored in the source control repository is equivalent to the creature informing the researcher monitoring it each time it is about to evolve/change. This is clearly not possible in living creatures but luckily possible in source code due to the detailed records kept by source control systems.

Alternatively, we can deploy tools to monitor and record the developer activities during code development as done by [12]. This later approach may be used when source control repositories are not accessible. Or it can be used when additional details, not available in the source control repository, are needed. Furthermore, project release archives which store a copy of released software may be used to study the evolution at the release frequency.

The Characteristics of Code

As we seek to describe the evolution of source code we need to define a set of characteristics and techniques to measure these characteristics. We can then describe the evolution of the source code in terms of these characteristics and change to them. For example, the size of the source code (*i.e.* the lines of source code) is a characteristic which can be measured easily. We can then compare the evolution of the system from one snapshot to the next.

In this subsection we cover a number of possible characteristics. The choice of characteristics to monitor is dependant on the research performed and the ease of recovering such characteristics from the source code. For example, recovering the number of lines of the source code is easier and less resource intensive than recovering the current dependency structure of the source code. We chose to focus on the static aspect of the source code instead of its dynamic and behavioral aspects due to the complexity associated with recovering and describing behavioral changes to source code.

We can describe the characteristics of source code using two general approaches:

- **Metric**: Metric approaches define measures to describe the current state of the source code. The simplest measures are metrics such as the Lines Of Code (LOC) or the number of defined functions. Other elaborate metrics such as complexity metrics could be used as well. Using metric approaches we can track changes in the value of the metrics (characteristics) over time.

- **Structural**: Structural approaches describe the current structure of the code. They could describe the dependency structure of the code such as '*function_1* depends on *function_2*', or they could describe the include dependencies such as '*file_1* includes *file_2*'. Using structural approaches, we could track changes in the structure of the source code, such as the addition of new functionality and its effect on the dependencies among the various parts of the source code. For example, we would expect once a function is added, other functions will be changed to call (depend on) it.

Some characteristics are cumulative, such as the number of functions, in the sense that characteristic measures derived based on a high frequency snapshots (such as change frequency) could be combined to study the same characteristic at the release level ($i.e.$ the number of functions that exist per release). This is usually not possible for a large number of characteristics such as complexity metrics. It may be beneficial to recover the evolution of source code using the most number of snapshots (change level) then to abstract the data for less frequent snapshots (release level). Using this approach the recovered evolutionary data could be used for a variety of studies based on the desired level of frequency.

Level of Detail

The level of the detail of the characteristics for a snapshot varies. Some snapshot extractors recover information at the system level such as the number of lines of the whole system, whereas other extractors can recover details at the function level such as the number of lines of each function. Or for structural characteristics some extractors recover the interaction between source code files, such as 'file $x.c$ calls file $x.h$'. Whereas other extractors report information at a lower more detailed level, such as 'function $f1$ calls function $f2$'. Also some extractors detail information about the internals of a function, such as '$func1\_for\_loop\_1$ calls function $f2$'.

The level of details for a snapshot defines the level at which the evolution of the source code can be described. It also limits the type of analysis that could be performed on the recovered data. The more detailed the extracted data, the more complex it becomes to develop a snapshot and an evolutionary extractor to generate this type of information, we believe there are a number of detail levels:

- **System Level**: At the system level a single value is generated for each snapshot of the studied system such as the total number of lines or the total number of files in the software system. Developing snapshot extractor for this level is usually easier than the other more detailed levels.
- **Subsystem Level**: At the subsystem level, the source code is divided into a small number of subsystems. Metric values for each subsystem or structural information about the interaction between these subsystems are generated by the snapshot extractor. For example, the source code of an operating system may contain four subsystems: a Network Interface, Memory Manager, Scheduler, and File System subsystems. A metric approach may track the size of each of these four subsystems. A structural approach may track the dependencies between these four subsystems.
- **File Level**: At this level, the extractor reports changes at the file level, such as the number of functions in a file or the number of lines in it. For example, an evolutionary extractor would detail information such as on Feb 2, 2004 file $x.c$ had 10 lines changed in it: 5 lines added and another 5 removed.
- **Code Entity Level**: At the code entity level, the snapshot extractor describes the snapshot based on code entities such as functions, classes, macros, or data types. For example, it could report the number of lines in a function, or the dependencies between the function in the source code. This data could be used during the snapshots difference analysis to report the addition of a new call to a function or the removal of a data dependency from another function. At this level of detail, the concepts of a function and data type renaming are possible. For example, it may be expected from an evolutionary extractor to report that a function was renamed. We believe that the detection of renaming of a source code entity versus the addition and removal operation of two sperate entities should be done as a post extraction step using techniques such as the ones described in [13].
- **Abstract Syntax Tree (AST) Level**: The AST level represents the lowest level of detail for information produced by an extractor. At this level, the snapshot extractor produces the AST of the source code. The AST is studied during the snapshots difference analysis (see Figure 1) to report changes to the internals of entities such as the addition/removal of new fields in a data structure, or if-branches and case statements inside functions.

The level of detail in the extracted information limits the type of analysis possible. It may also complicate the development of the extractor, for example AST level evolutionary extractors are much harder to develop as they require the development of snapshot extractors which can parse the source code and produce very low level details about its characteristics. In contrast, a system level evolutionary extractor is much simpler to develop as it does not need to perform detailed analysis of the source code snapshots.

4 CHALLENGES AND COMPLEXITY

In an ideal situation, we would develop extractors that would describe the evolution of the source code at the most detailed level, the AST level, and at the highest frequency (change frequency). Unfortunately this is a rather hard problem and developing such an extractor would be too complex and time consuming.

As researchers approach the problem of building an evolutionary extractor, they must decide on the choices along the dimensions, discussed in the previous section. The benefits and limitations of each decision are weighted using many criteria. The most important criteria we found in our work are the needs of the research for which the extractor is being developed, the time allocated for the project, and the funding at hand. We have developed several source code extractors for many programming languages in the last few years and we found that this engineering approach is paramount for the success of such projects due to the unsurmountable effort and challenges surrounding the development of the most suitable and practical extractor [7]. We cover a few of the challenges associated with developing evolutionary code extractors.

Robustness and Scalability
When studying the structural evolution of source code, we need to develop an extractor that can analyze the source code to determine structural dependencies among source code entities. An approach which is based on a text book grammar will fail to parse legacy systems, due to the variety of dialects of programming languages and the multitude of proprietary compilers extensions. This variety complicates building an extractor. The developers of snapshot extractors could adopt various approaches to deal with the complexity of parsing legacy software systems. Some developers may choose to have their extractors recover gracefully when such extensions are processed, others may choose to specialize their parsers for such peculiarities using a variety of parsing techniques such as island grammars, robust parsing, and precise parsing [11]. The choice of techniques to use is influenced by the peculiarities of the studied software systems. An evolutionary extractors should be robust and recover gracefully with no user intervention to permit the analysis of several snapshots in an automated fashion.

Furthermore, the scalability of an extractor is another hard challenge, as extractors are expected to analyze large software systems which may contain several million lines of code. This is complicated more by the fact that this analysis must be performed for each snapshot of the code and there could be thousands of snapshots. For example, examining a million line of source code at the change frequency would conceptually require the extraction of the characteristics associated with the source code after each change. This may require that the analysis of a million lines of code is repeated thousands of times, such an approach becomes infeasible and impractical. Instead developers of evolutionary extractors must develop more elaborate techniques to deal with this challenge.

An ideal goal for a snapshot extractors is to have the extraction process require no more time that the compilation time of the software system. In contrast for evolutionary extractors, even meeting this goal would require too much time and would make using an evolutionary extractor infeasible and impractical to study long lived software projects. Incremental extraction techniques similar to incremental compilations may assist in speeding up evolutionary extractors.

Accuracy
Ensuring the high accuracy of the extractor output is another challenging task. Given the size of the software systems extracted and the techniques used to recover from different system peculiarities, extractors have the tendency to miss some relations (*false negatives*), or in some cases add superfluous ones (*false positives*). Accurate extractors require rather complex language grammars and adopt several elaborate techniques to recover from errors and correctly identify information. [2] and [10] have shown empirically the difficulty faced by already well established extractors in ensuring this accuracy. Evolutionary extractors would use similar techniques, therefore we expect that they would have to face similar challenges.

To make matters worse, when dealing with extraction over an extended period of time, the adopted approaches have to deal with unrelated entities having similar names appearing and disappearing throughout time.

The Changing and Unstable Nature of Source Code
An evolutionary extractor conceptually performs its work by analyzing each snapshot then comparing the generated information for each snapshot. As pointed out, this may be too time consuming. Furthermore, this would require the source code to be in some compilable stable state to permit the snapshot extractor to process it. This is not possible for example, when studying source code evolution at the change frequency – a developer may add a call to a function before she/he defines the function. Therefore, intelligent approaches are needed to analyze un-compilable and incomplete source code. Alternatively, we may choose to use less frequent snapshots that are more likely to be complete such as nightly builds or code-complete builds. These builds are less likely to cause the snapshot extractors to fail.

Development Time
Another challenge associated with evolutionary extractors is the time needed to develop them. An ideal solution would be to adopt a regular source code extractor and modify it. In particular, for each change in the project we can rerun the extractor and compare the output of the extractor run before and after the change. Unfortunately, as highlighted in the previous subsection this may not be the optimal solution as the source code may not be compilable. Therefore evolutionary extractors must either be built from scratch or built by adopting regular extractors and enhancing them to deal with many of the aforementioned challenges. Clearly reusing already developed extractors would speed up the development time but may limit the type of analysis and may negatively affect the performance of the evolutionary extractor. On the other hand, building an extractor from scratch may provide the most flexible approach but would require a longer development time.

5 PREVIOUS WORK

In this paper we advocate the use of evolutionary extractors. We present several critical dimensions based on which such extractors should be designed. A number of evolutionary extractors have been built by many researchers studying the

Reference #	Snapshot Frequency	Data Source	Characteristics of Code	Level of Detail
[8]	Release	Release Archives	Metric (LOC)	System
[9]	Release	Release Archives	Metric (LOC)	System/Subsystem
[13]	Release	Release Archives	Structural (Call/Data Dep.)	File
[4]	Changelist	Source Control	Structural (Co-Change)	File
[14]	Changelist	Source Control	Metric (Change)	File

Table 2: Summary of Evolutionary Extractors Design Choices

evolution of software systems. Although the term evolutionary extractor was not used by these researchers, the type of analysis performed by them fit well into our definition of an evolutionary extractor. In this section, we overview their work and present it using the design dimensions for evolutionary extractors presented in this paper.

Work by Lehman *et al.* [8] tracked the evolution of the size of the source code, to perform such analysis evolutionary extractors that used code metrics at the system level monitored changes to the size of each release. Godfrey and Qu [9] developed evolutionary extractors that use metrics at the system and subsystem level to monitor the evolution for each release of Linux. In addition, Qu and Godrefy [13] developed evolutionary extractors that track the structural dependency changes at the file level for each release of gcc.

Gall *et al.* [4, 5] have developed evolutionary extractors that track the co-change of files for each changelist in CVS. Zimmermann *et al.* [14] present an extractor which determines the changed functions for each changelist. Table 2 summarizes the design choices for each of the extractors developed by other researchers.

Draheim and Lukasz present a software infrastructure to study and visualize the output of evolutionary extractors, in particular they focus on visualizing metric evolutionary extractors using graphs [3].

6 CONCLUSION

Software practitioners and researchers have recognized the need to study the evolutionary process of software projects. The source code is an ideal facet of a software project to monitor and analyze as we can easily acquire various snapshots of it as it evolves. Furthermore, we can build tools – evolutionary code extractors – to automatically recover this evolutionary process. This recovered process could improve our understanding of software development and assist developers maintaining large long lived software systems.

In this paper, we advocated the need for such evolutionary extractors. We presented the various dimensions along which such extractors could be built. We discussed the challenges and complexities associated with the choices taken along these dimensions. These challenges complicate the development of such extractors, nevertheless we believe that a number of common extractors could be developed and reused within the research community to further empirical based understanding of software evolutionary processes. We also presented previous work which studies the evolution of code and attempted to classify these published extractors using the dimensions and choices we presented herein.

REFERENCES

[1] A. I. Anton and C. Potts. Functional paleontology: System evolution as the user sees it. In *Proceedings of the 25th International Conference on Software Engineering (ICSE 2001)*, Toronto, Canada, May 2001.

[2] M. N. Armstrong and C. Trudeau. Evaluating architectural extractors. In *Working Conference on Reverse Engineering (WCRE98)*, pages 30–39, Honolulu, HI, Oct. 1998.

[3] D. Draheim and L. Pekacki. Process-Centric Analytical Processing of Version Control Data. In *IEEE International Workshop on Principles of Software Evolution (IWPSE03)*, Helsinki, Finland, Sept. 2003.

[4] H. Gall, K. Hajek, and M. Jazayeri. Detection of logical coupling based on product release history. In *IEEE International Conference on Software Maintenance (ICSM98)*, Bethesda, Washington D.C., Nov. 1998.

[5] H. Gall, M. Jazayeri, and J. Krajewski. CVS Release History Data for Detecting Logical Couplings. In *IEEE International Workshop on Principles of Software Evolution (IWPSE03)*, Helsinki, Finland, Sept. 2003.

[6] A. E. Hassan and R. C. Holt. The Top Ten List: Dynamic Fault Prediction. *Submitted for Publication*.

[7] A. E. Hassan and R. C. Holt. Architecture Recovery of Web Applications. In *IEEE 24th International Conference on Software Engineering*, Orlando, Florida, USA, May 2002.

[8] M. M. Lehman, J. F. Ramil, P. D. Wernick, D. E. Perry, and W. M. Turski. Metrics and laws of software evolution the nineties view. In *Fourth International Software Metrics Symposium (Metrics97)*, Albuquerque, NM, 1997.

[9] Michael W. Godfrey and Qiang Tu. Evolution in open source software: A case study. In *IEEE International Conference on Software Maintenance (ICSM 2000)*, pages 131–142, San Jose, California, Oct. 2000.

[10] G. C. Murphy, D. Notkin, W. G. Griswold, and E. S. Lan. An empirical study of static call graph extractors. *ACM Transactions on Software Engineering and Methodology*, 7(2):158–191, 1998.

[11] S. Klusener and R. Lämmel. Deriving tolerant grammars from a baseline grammar. In *IEEE International Conference Software Maintenance (ICSM 2003)*, Amsterdam, The Netherlands, 2003.

[12] J. S. Shirabad. *Supporting Software Maintenance by Mining Software Update Records*. PhD thesis, University of Ottawa, 2003.

[13] Q. Tu and M. W. Godfrey. An integrated approach for studying architectural evolution. In *10th International Workshop on Program Comprehension (IWPC'02)*, pages 127–136. IEEE Computer Society Press, June 2002.

[14] T. Zimmermann, S. Diehl, and A. Zeller. How History Justifies System Architecture (or not). In *IEEE International Workshop on Principles of Software Evolution (IWPSE03)*, Helsinki, Finland, Sept. 2003.