Recovering Runtime Architecture Models and Managing their Complexity using Dynamic Information and Composite Structures

Soumia Zellagui, Chouki Tibermacine, Ghizlane El Boussaidi, Abdelhak-Djamel Seriai, Hinde Lilia Bouziane, Christophe Dony

Recovering Runtime Architecture Models and Managing their Complexity using Dynamic Information and Composite Structures

Soumia Zellagui[1], Chouki Tibermacine[1], Ghizlane El Boussaidi[2], Abdelhak-Djamel Seriai[1], Hinde-Lilia Bouziane[1] and Christophe Dony[1]
[1]LIRMM, CNRS and University of Montpellier, France
{zellagui,tibermacin, seriai, bouziane, dony}@lirmm.fr
[2]École de Technologie Supérieure, Montréal, Canada
Ghizlane.ElBoussaidi@etsmtl.ca

ABSTRACT

Program comprehension during software maintenance is a difficult task, hence the need to support it by recovering the as-built architecture of the system to be maintained. In this paper, we propose a method to recover runtime architecture models of object-oriented systems. The method combines static and dynamic analysis to recover an Object Graph (OG) and uses two techniques to manage the complexity of this graph.

CCS Concepts

•**Software and its engineering** → *Object oriented architectures;*

Keywords

Architecture Recovery; Static and Dynamic analyses.

1. INTRODUCTION

Software systems are maintained throughout their operational life. When a software undergoes major changes, the high level view of its structure and behavior, to which we refer as *software architecture*, is required. Such view helps supporting the developer in understanding and properly evolving the system. In practice, the continuous and cumulative changes undergone by the system increase its complexity and lead to a deviation from its intended architecture [7]. Thus, an architecture recovery process becomes inescapable for understanding the as-built architecture of the system before initiating any modifications.

A number of software architecture recovery approaches were proposed in the literature [4] but only few of them targeted the recovery of runtime architecture models [1,3], which are composed of the system's runtime entities (objects) and dependencies between them. Moreover, only few approaches provide strategies to manage the complexity of the recovered architecture models, especially in the case of large systems.

In this paper, we propose an approach that recovers the runtime architecture and helps managing the complexity of the recovered architecture. To do so, we combine static (source code) and dynamic (execution trace) analysis. In particular, we use static analysis to build OGs. Because the size of such graphs is most of the time very large, we refine them using information obtained through the analysis of execution traces. The information added to these graphs includes the lifespan of each object and its probability of existence at runtime. This information helps managing the complexity of the resulting refined graph by allowing to visualize only relevant parts of the graph according to the developer's concerns. Furthermore, to manage the complexity of the graph, we use the ownership model introduced in [2] to identify composite (internal) structures of objects in the graph. This enables to organize the refined OG into a hierarchy of composite structures/nodes that can be collapsed or expanded to hide or show their internal structure.

The paper is organized as follows. Section 2 presents a general overview of the approach which is defined as a multi-step process and then details each step of the process. We discuss briefly the results of a preliminary experiments in Section3. We expose the related works in Section 4 and we conclude in Section 5.

2. THE RECOVERY PROCESS

The proposed recovery process is depicted in Figure 1. The first step is a static analysis of the source code from which an initial OG, similar to the one introduced in [9], is recovered. Once the OG has been obtained, the dynamic analysis is prepared by instrumenting the source code of the system under study. The execution of the instrumented code using a set of test cases produces logs, *i.e.* execution traces. These generated traces are analyzed to extract information to refine the preliminary OG. The final step of the approach consists in managing the complexity of the refined OG using two techniques: i) exploiting lifespans and probabilities of existence, and ii) identifying the so-called composite structures, which enable to make the graph hierarchical and thus reduce its complexity. The refined hierarchical OG can be used by external tools for a visualization with a level of

SAC 2018, April 9–13, 2018, Pau, France
Copyright 2018 ACM 978-1-4503-5191-1/18/04…$15.00
https://doi.org/10.1145/3167132.3167420

detail. Developers can customize their visualization by focusing on the most durable objects or the most likely to exist at runtime. They can also focus on particular objects by unfolding hierarchical nodes to analyze their composite structure, or to visualize a high level view of the architecture (the graph hiding the internal composite structures). In the following, we discuss in detail the different steps of the process for recovering this refined hierarchical graph.

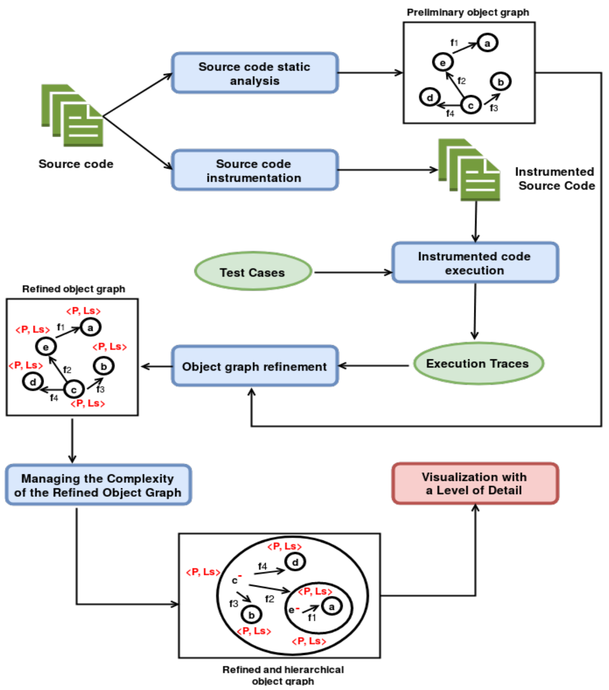
Preliminary object graph
Source code static analysis
Source code
Source code instrumentation
Instrumented Source Code
Test Cases
Instrumented code execution
Refined object graph
Execution Traces
Object graph refinement
Managing the Complexity of the Refined Object Graph
Visualization with a Level of Detail
<P, Ls>
Refined and hierarchical object graph

Figure 1: Process for creating the refined hierarchical object graph

- **Source code static analysis:** the purpose of this first step is building a preliminary OG. An OG is a directed labeled graph that represents the structure of a given software system in terms of objects. In this graph, nodes denote objects and an edge between two nodes *n1* and *n2* indicates that there exists in the source code an assignment through which *n1* references *n2* via one of its fields. Edges are labeled with fields names.

  In order to build this OG, the flow of objects must be captured from the source code. To do so, we rely on an Object Flow Graph (OFG) built using the method described in [9]. An OFG is a directed labeled graph in which nodes represent objects or program variables[1] (fields, local variables, methods' parameters or methods' arguments), and edges represent assignments between these variables. To build this OFG, we use three kinds of statements: allocation, assignment and invocation sites.

  Objects in the OFG are collected from allocation sites and the flow of each object is inferred by analyzing the statements in which the reference of this object is used. Once the OFG is obtained, the OG can be recovered by analyzing the output sets of the OFG nodes that correspond to fields.

- **Source code instrumentation & instrumented code execution:** we instrumented the code by automatically adding statements that produce execution traces which report the following: i) system start and end timestamps. ii) object creation: creation timestamp, object identifier, the position (class name + line number) of the allocation site responsible for creating the object and the object hashcode. iii) object destruction: destruction timestamp, by overriding the *finalize()* method if it does not exist, and the hashcode of the destroyed object. Then, to generate execution traces, the instrumented code is executed using a set of test cases.

- **Object graph refinement:** the purpose of this refinement is to add two kinds of labels on nodes of the OG: probabilities and lifespans. Labels added on nodes are of the form <probability, lifeSpan>. For each object, its probability is the ratio of the number of occurrences of this object in execution traces to the total number of execution traces. Lifespans are measured using the creation and destruction timestamps, which exist in the execution traces. First, we measure the average creation and destruction timestamp for each node from all the execution traces where it exists. Then, we scale each timestamp to the total lifespan of the application (the difference between the system's start and end time). The lifespan of a node is a range, having as a minimal value the scaled creation timestamp and as a maximal value the scaled destruction timestamp.

- **Managing the Complexity of the Refined Object Graph:** to manage the complexity of the refined object graph, we use and combine two techniques. The first technique exploits the information available in our refined graph, namely the object lifespan and probability of existence. The second technique aims at identifying the composite structures of objects in the previously recovered graph.

  Having an OG that includes the lifespan of each object and its probability of existence at runtime, we can help manage the complexity of the OG by allowing to visualize only relevant parts of the graph according to the developer's needs. In fact, developers can set thresholds for the values of the information added to the graph in order to focus, for instance, on objects that are the most durable or the most likely to exist at runtime. In general, we expect the objects that constitute the GUI to be the most durable; i.e., they are created when the software system is launched. Conversely, depending on the complexity of the application domain of the system, some domain-specific (business) objects may be more or less durable depending on the importance of the object in the domain.

  The second technique identifies composite structures in the refined OG based on the owners-as-dominators

[1]We are only interested by those which are typed by user-defined classes/interfaces

ownership model [2]. In this model, an owner object (the composite) should dominate an owned object (component), that is, an object cannot be exposed outside of the boundary of its owner. In other words, all access paths to the owned object should pass through its owner. This technique enables to build the composite structures of objects in the form of hierarchical nodes in the graph, which helps managing the complexity of the graph.

3. PRELIMINARY EXPERIMENTS

A prototype of the approach was implemented using Spoon [6] which is a library for source code analysis and transformation. The visual output of our approach is generated using GraphVIZ[2] which is an open source graph visualization software.

We applied our approach on the Jext[3] open source Java project using 15 scenarios that exercise the main functionalities described in Jext documentation.

For lack of space, we don't discuss all the details of this preliminary experiment. To summarize, using the composite structures technique only, the OG displays 12 nodes at the highest level and 203 nodes when fully expanded. Combining the composite structures with our technique for managing the OG's complexity, the number of nodes is reduced in a more substantial way. For example, if the user chooses to display only the objects that have a probability of one and a lifespan greater than 5%, the number of nodes of the OG is reduced to 8 at the highest level and to 139 when fully expanded (i.e., a total reduction of 46%).

4. RELATED WORK

A large body of research exists for supporting the reverse engineering of the static and/or dynamic information needed for architecture recovery. In this section, we discuss the works that are the closest to our work

Both Spiegel et al [8] and Abi-Antoun et al [1] proposed static analysis techniques, named Pangaea and SCHOLIA respectively, in order to recover OGs of Java systems. The analysis process in SCHOLIA is neither object, nor polymorphism nor flow sensitive. Flow sensitivity refers to the ability to capture information by considering conditional statements (if-else statements, loops, etc). Polymorphism sensitivity means that method invocation context is taken into account and separate information is computed for different invocations of the same method. Object sensitivity means that objects are identified by their allocation points instead of class names. On the other hand, the process in Pangaea is object and polymorphism sensitive. The aspect of mitigating the complexity of the recovered graphs is not taken into account in Pangaea, whereas, in SCHOLIA, architectural extractors (developers) use ownership domain annotations to annotate the Java code, then they use static analysis to extract a hierarchical Ownership Object Graph (OOG). Each of the works of de Brito et al [3] and Flangan et al [5] recovers OGs dynamically. The analysis process in the two techniques is object, polymorphism and flow sensitive. To promote the scalability of OGs, de Brito et al [3] use the summarization by domain and Flangan et al [5] apply some abstractions such as: defining ownership and containment relations between objects. Wang et al [10] proposed an automatic recovery technique based on hybrid analysis. The static analysis is used to build the OG. This graph is then enhanced with dynamic profiling information such as allocation frequency on nodes. Thereafter, this information is used to reduce the OG to a tractable size. The analysis process in this approach is object and flow sensitive.

Our approach combines a static and dynamic analyses. It promotes automation and avoids to developers their involvement in the recovery process. The proposed process is object, polymorphism and flow sensitive, which guarantees a more "precise" analysis of the source code, compared to existing works which consider these aspects only partially. The recovered OG is refined with both: i) new labels and ii) a hierarchical structure, in contrast to existing works. These additional features enable to reduce the complexity of this graph.

5. CONCLUSION

In this paper, we proposed an approach to recover refined and hierarchical OGs of Object Oriented (OO) software systems. As proposed, these OGs have the following distinguishing features: i) Nodes are labeled with lifespans and probabilities of existence that allow a visualization with a level of detail. ii) They support the collapsing/expanding of objects to hide/show their internal structure. Future works will surely regard the application of the approach to several systems and in the migration of OO systems towards component based ones.

[2]https://www.graphviz.org/

[3]https://sourceforge.net/projects/jext/

6. REFERENCES

[1] M. Abi-Antoun and J. Aldrich. Static extraction and conformance analysis of hierarchical runtime architectural structure using annotations. In *OOPSLA*, 2009.

[2] D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. *ACM SIGPLAN Notices*, 1998.

[3] H. de Brito, H. T. Marques-Neto, R. Terra, et al. On-the-fly extraction of hierarchical object graphs. *Journal of the Brazilian Computer Society*, 2013.

[4] S. Ducasse and D. Pollet. Software architecture reconstruction: A process-oriented taxonomy. *In Trans on Soft Eng*, 2009.

[5] C. Flanagan and S. N. Freund. Dynamic architecture extraction. In *FATES & RV*. 2006.

[6] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier. Spoon: A library for implementing analyses and transformations of java source code. *Software: Practice and Experience*, 2015.

[7] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *SIGSOFT Software Engineering Notes*, 1992.

[8] A. Spiegel. *Automatic distribution of object oriented programs*. PhD thesis, 2002.

[9] P. Tonella. Reverse engineering of object oriented code. In *ICSE*, 2005.

[10] L. Wang and M. Franz. Automatic partitioning of object-oriented programs for resource-constrained mobile devices with multiple distribution objectives. In *ICPADS*, 2008.