Object Based Dynamic Model Extraction

Philipp Bouillon
FernUniversität in Hagen

April 1, 2005

Abstract

We describe the outline of a method to retrieve a model from various program runs of object-oriented software which can then be used to find bugs in the program, re-design or refactor it, test the program against the model, perform regression testing and check if new code violates the model. When completed, the model extraction and testing will be implemented as an Eclipse plug-in and the programmer is informed of deviations from the model.

1 Introduction

Today's computer programs are tremendously complex. Still, a common task is to further improve and extend those programs and to add new features or to adapt the program to match new regulations. Most of the time, many people are involved in the implementation process of the software, so it is imperative to aid the programmers in understanding and extending existing program code.

Usually, design documents lead programmers to relevant parts of the code where they can then implement their changes. More often than never, however, those design documents are out of date and do no longer reflect the actual state of the software. Thus it becomes necessary to provide an automatic means to infer needed information from a program.

In this paper, we are going to present the outline of a method to retrieve *information* on a program by the means of dynamic analysis. The process is fully automated and, when finished, will provide the programmer not only with a better understanding of the program at hand but with a formal *model* of that program which use is not limited to program understanding but can be applied to re-designing, refactoring, reverse engineering, debugging, and more.

It should be noted that our approach is tailor-made for object-oriented programs since we relate calling events happening during the program execution to specific objects. Thus, it is also clear that our approach is a *dynamic* analysis of various program runs.

To obtain our model, we use different approaches which we then combine into one consistent description of a program. The first approach we use is already described by Ammons in [1] who uses machine learning on dynamic traces to create *Probabilistic Finite State Automatons* (PFSAs) that can represent the control-flow graph of a program.

A second approach can be found in the work of Dallmeier et al. [2] where sequences of calls are extracted from Java programs. Those sequences are collected for several program runs and then compared. Differences between the sequences are considered as a possible bug in the program.

Combining and comparing these two approaches is one of our first goals to create a model which is a formal representation of the program in question.

The remainder of the paper is organized as follows: Section 2 gives an overview of the proposed system and the steps that are necessary to obtain and use a model. Section 3 describes the current state of the project while Section 4 ends the paper with references to related work and a conclusion.

2 The Big Picture

Before we can think about creating a model of some sort, we first have to create a pool of data from which to extract that model. We have decided to dynamically analyze object-oriented programs and extract their *event-traces*. From the recorded data, we can then build our model.

Informally, an event-trace is a sequence of method-calls and return statements which are all attributed to a specific *object* (as opposed to the general class of that object).

It would now be great if we could just take an event-trace and analyze all data stored in it with respect to a certain *class* C (by combining all the information we have for objects of C). Unfortunately, event-traces become immensely large even for short programs, because thousands of objects from hundreds of classes may call thousands of methods...

Thus, the event-traces must be broken down into smaller sets from which we can then start our analysis. One way of breaking down the traces is again described by Ammons who picks out *scenarios*, where a scenario is a (short) sequence of calls that perform a certain task (like locking, modifying and unlocking a file). For a complete definition of scenarios and their extraction from event-traces—which is not a trivial task—see [1].

In [2], the event-traces are split into smaller sequences by sliding a variable *window* over the complete trace and comparing only the sequences of length

n.

Both approaches can be used effectively to break down the huge event-trace of a program run, but we are of course not limited to those two. Other approaches will be considered and tested.

After breaking down the event-traces into smaller sets, those sets can be grouped and *normalized*. We might, for example, have various sets that all describe the modification of a file, only for different files. On the other hand, we might have a certain call-sequence in a number of sets and a slightly different sequence in one other set, which might be an indication of a bug in the program.

In the next step, we can finally begin to analyze the obtained data and derive a *model* from it. Ammons now used Machine Learning on his scenarios to produce so called *Probabilistic Finite State Automatons* which can tell, how probable a call to method A is after a call to method B. Thus, assuming for a moment that the analyzed program is correct, the PFSA gives us a first model of a certain part of the program: The most probable sequence in which methods have to be called. In addition, with the PFSAs, a programmer has a testing profile, giving her an estimate on how much of the program has been tested, since the probability for each path in a program execution is stored in the PFSAs.

To refine our model and to make it more general, we are currently researching various analysis models, different from the PFSAs used by Ammons. Our hope is to have a set of different, yet similar, views of a program which can be combined to one single model which gives the programmer valuable insights of the structure and the calling concepts used in the program.

Once the model is in place, it can help the programmer in several ways. Besides program-understanding, the model can be used for *testing* as a newly written part of the program can be checked against the model where violations indicate possible bugs in the program. Furthermore, the programmer gets hints for the architecture of his program. If, for example, a class is used *either* for writing data, *or* for reading data, i.e. two objects exist: One for reading, one for writing, the programmer could consider refactoring the class into two separate classes. So, the model can help *re-design* an existing program. Another possibility for the use of the model might be to incorporate the model into the program and use run-time checks to verify the program against the model. Thereby, the developers can gain hints if something in the program does not work according to their plan and thus the debugging becomes easier.

Instead of relying solely on the dynamically created model, the programmer can be guided even more by comparing the dynamic model with statically created model. Thus, we can even give guidance while the programmer is typing the program, assuming that the model is precise enough.

Ultimately, the model-extraction, the check of the model against currently entered code, and the visualization of any deviations from the model will be implemented as an Eclipse plug-in. The vision being a programming aid which automatically tells the programmer when she is about to make a mistake and informing her of a possible correction. Of course, depending on the model, the plug-in is not restricted to bug prevention or fixing, but can also be used to refactor or re-design existing code if it turns out that a class is used in a number of different ways.

3 Current State of the Project

At the time of this writing we can extract event traces from Java programs and analyze those traces with the *Strauss* specification miner [1]. The results need to be verified, interpreted and incorporated into an Eclipse plug-in, but first, other means of specifying a model will be evaluated.

So far, we have been testing our program with toy projects written in Java to prove the concept. The results gained with those toy programs only show that the work done by Ammons is applicable to object-oriented programs. To actually put the PFSAs into a model interpretable by an Eclipse plug-in, we will have to integrate Ammons work into Eclipse and then combine it with our algorithms.

The next steps will involve a combination of the two approaches by Dallmeier et al. and Ammons, as well as the search for new alternatives in breaking down event-traces into useful chunks of information.

Next to machine learning, other approaches of *interpreting* the data obtained by the trace-analysis will be discussed, evaluated and integrated into our algorithm.

4 Conclusion and Related Work

Although for now, we are only extending the work by Ammons and Dallmeier, we hope that an integration of the two sketched methods will provide any programmer in an object-oriented language with a precise model which will allow for an easier extension of existing programs as well as a simplification of debugging. First results of toy programs show that our idea is realizable although we do not know yet, what the model will look like.

References

[1] Glenn Schatzman Ammons. *Strauss: a specification miner.* PhD thesis, University of Wisconsin, 2003. Supervisor-Rastislav Bodik.

[2] Valentin Dallmeier, Christian Lindig, and Andreas Zeller. Lightweight defect localization for java. In *Proc. ECOOP 2005 – 19th European Conference on Object-Oriented Programming*, Glasgow, Scotland, July 2005.