Dynamic Object Process Graphs

von Jochen Quante

Dissertation

zur Erlangung des Grades eines
Doktors der Ingenieurwissenschaften
– Dr.-Ing. –

Vorgelegt im Fachbereich 3 (Mathematik und Informatik)
der Universität Bremen
im September 2008

Gutachter: Prof. Dr. Rainer Koschke
Prof. Dr. Jürgen Ebert

Summary

An Object Process Graph is a view on the control flow graph from the perspective of a single object. It contains the uses of the object and the paths through the control flow graph that connect these uses. Such a graph can be extracted by static or dynamic program analysis.

This thesis addresses dynamic extraction of Object Process Graphs and the applications of these graphs. In a first step, methods for online and offline construction of Dynamic Object Process Graphs are presented. These methods are shown to be applicable even for large and interactive programs. The second step is the further transformation and application of the extracted graphs.

Dynamic Object Process Graphs can be used as the basis for protocol recovery by transforming them to protocol automata. A comparison of the results to several traditional dynamic protocol recovery approaches demonstrates their quality. Another application is the visualization of Dynamic Object Process Graphs for supporting program understanding. Several case studies illustrate their potential, and a controlled experiment was performed to assess their general utility for this purpose.

Altogether, this thesis shows that and how Dynamic Object Process Graphs can be extracted efficiently and used effectively. Similar to program slicing, Object Process Graph extraction is an enabling technique with applications in many reverse engineering tasks.

Zusammenfassung

Ein *Object Process Graph* ist eine Sicht auf den Kontrollflussgraphen aus der Perspektive eines einzelnen Objekts. Er enthält im Wesentlichen die Benutzungen des Objekts sowie die Pfade durch den Kontrollflussgraphen, die diese Benutzungen verbinden. Ein solcher Graph kann durch statische oder dynamische Programmanalyse extrahiert werden.

Diese Arbeit beschäftigt sich mit der dynamischen Extraktion von *Object Process Graphs* und deren Anwendungen. Im ersten Schritt werden Methoden zur Online- und Offline-Extraktion von *Dynamic Object Process Graphs* vorgestellt. Ihre Anwendbarkeit – auch für grosse und interaktive Systeme – wird gezeigt. Im zweiten Schritt werden die Graphen für verschiedene Anwendungen weiterverarbeitet und transformiert. Eine Anwendung von *Dynamic Object Process Graphs* ist die Rekonstruktion des Protokolls einer Komponente. Dies wird durch Transformation der Graphen zu Protokollautomaten erreicht. Deren Qualität wird in einem Vergleich mit traditionellen dynamischen Protokollrekonstruktionsverfahren gezeigt. Eine weitere Anwendung ist die Visualisierung von *Dynamic Object Process Graphs* zur Unterstützung des Programmverstehens. Mehrere Fallstudien illustrieren das diesbezügliche Potential, und es wurde ein kontrolliertes Experiment durchgeführt, um die Eignung für das Programmverstehen genauer zu untersuchen.

Insgesamt zeigt diese Arbeit, dass und wie *Dynamic Object Process Graphs* effizient extrahiert und effektiv genutzt werden können. Ähnlich wie das Programm-Slicing stellen sie eine Basistechnik mit Anwendungen in vielen Bereichen des Reverse Engineering dar.

Acknowledgements

First of all, I would like to thank my advisor, Rainer Koschke. I learned a lot from him about scientific work and scientific writing in general as well as reverse engineering techniques and methodology in particular. He also pointed me to the topic of this thesis at a very early stage, which enabled its completion in a reasonable amount of time, and contributed many good ideas. He has been a great advisor; it has been a priviledge to work with him.

I also want to thank Jürgen Ebert, who agreed to be my secondary advisor. I would specially like to thank him for initiating (along with Franz Lehner) the Workshop series on Reverse Engineering in Bad Honnef, which is an ideal event for meeting reengineering people from science and practice. This workshop provided valuable inspiration and contacts to me.

Thanks to Gunther Vogel for providing the static OPG extraction tools. Also thanks to all the Bauhaus people who have built this excellent reengineering infrastructure over the years. Thanks to my colleagues at the University of Bremen for interesting discussions and stimulation, and in particular to my office mate Thilo Mende for proof-reading this thesis (and for the cookies). Thanks to Lutz Prechelt and Leif Kruse, who gave me some useful hints in the preparation phase of the experiment, and to the 27 students who voluntarily participated. Thanks to Renate Klempien-Hinrichs, who helped me with several graph transformation issues. Also many thanks to all the people I met at various conferences for the fruitful discussions and inspirations.

Thanks to my parents for their ongoing support of almost everything I do. Thanks to my children Rebecca and Thomas, who were both born during the work on this thesis and who were always there for a nice distraction from work. Finally, a big thank you to my wife Kathrin for always supporting me in the decision to go back to University, which meant moving to a different city, giving up her job, and living a different life.

Jochen Quante, September 2008

Contents

Part I

Prelude

Chapter 1

Introduction

1.1 Problem and Context

"Software is everywhere in modern civilization. Software is in your mobile phone, on your home computer, in cars, airplanes, hospitals, businesses, public utilities, financial systems, and national defense systems. Software is an increasingly critical component in the operation of infrastructures, cutting across almost every aspect of global, national, social, and economic function. One cannot live in modern civilization without touching, being touched by, or depending on software in one way or another." [159]

Grady Booch estimates that there must be about 800 billion source lines of code (SLOC) in the world, with an additional 30 billion SLOC newly produced per year [20]. Although this is only a very rough guess, it is probably the right order of magnitude and illustrates the importance of software. A large portion of this vast amount of code is still running and has to be *maintained*.

1.1.1 Software Maintenance

Software maintenance is the *"modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a changed environment"* [84]. If software is *not* maintained, it becomes useless after some time, because the environment is continually changing. This effect is called *"software aging"* by David Parnas [139]. It was first described by Lehman and Belady and is known as their *"law of continuing change"* [109]. Another one of their laws states that if the system *is* adjusted, that is, maintained, *"ignorant surgery"* [139] leads to steadily *increasing complexity*. What does that mean? Maintenance is often performed by people who are not familiar with the original design concepts – be it due to lack of information, lack of time, or other reasons. Therefore, such changes are often performed in a way that is inconsistent with the original developers' ideas. Also, these maintainers often only have a local view on the system and do not understand the impact of their changes, introducing new and possibly subtle errors. And even if the changes are performed with the system's architecture in mind, there often is no time to do the changes in a correct and consistent way.

As more and more of this kind of changes occur, a system gets increasingly less comprehensible and more complex, and therefore each further change becomes even more expensive. Obviously, the combination of these two Lehman's rules means that software ages inevitably and has to be replaced at some point – if no counteractive measures are taken.

According to some (rare) studies, software maintenance accounts for up to 80% of a product's total lifecycle cost [134]. This is an astonishingly high share at first glance, but is comprehensible when regarding our considerations about software aging. However, this fact is still neglected in practice. Software maintenance is usually performed by software *developers* who never had any training about software *maintenance* – although such training could presumably make maintenance more effective and less error-prone [174]. Other studies have shown that about 50% of the time in software maintenance is spent for gaining an understanding of the system to be modified alone [54]. This means that techniques that help to understand a software system – or certain aspects of it – have a great potential to reduce maintenance costs. Corbi [33] gives an overview of the associated challenges.

1.1.2 Software Reengineering

This is where **Software Reengineering** comes into play. This discipline develops tools and methodologies which support a maintainer in his work. The goal of these tools is to provide a better understanding of a system, which should result in more consistent changes, a higher productivity, and fewer introduced errors – and the aging process may be slowed down. For example, techniques like *refactoring* [55] or *architecture compliance checking* [131] can be helpful for the latter. When a system finally has to be replaced, Software Reengineering techniques help in the migration process. But let us first define what Software Reengineering and related terms mean.

Reverse engineering has its origin in the analysis of hardware [27]. There, it is applied to recover a product's internal design when only the end product is available. The purpose usually is to improve an own product or to analyze a competitor's or an adversary's product. **Software Reverse Engineering** (SRE) is the application of this idea to software: based on the available artifacts of a system, such as its code, it is the task of trying to recover any unavailable documentation, that is, raising the abstraction to a higher level. However, in contrast to the hardware domain, it usually focusses on one's own code. It is basically the inversion of the normal software development activities, which are called **Forward Engineering** for differentiation. The main goal of SRE usually is to gain a sufficient design-level understanding of a system to aid maintenance, strengthen enhancement, or support replacement [27]. **Software Reengineering** additionally includes the subsequent alteration or transformation of the system. Figure 1.1 illustrates the relation of the different terms. Conceptually, Reverse Engineering also includes recovering the specification from a given design. However, this is

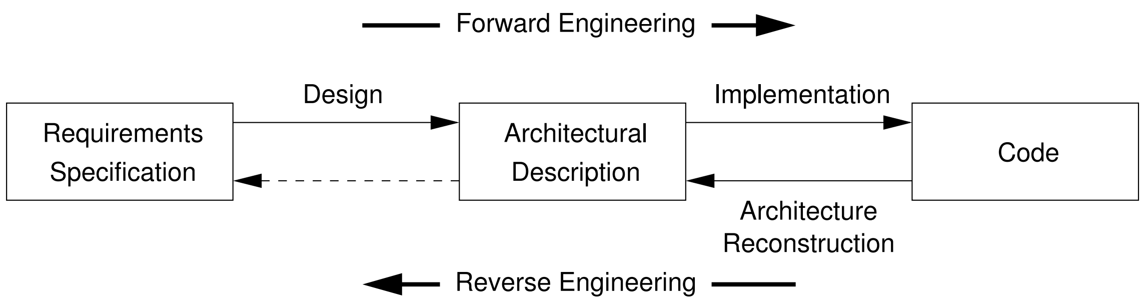

Forward Engineering
Design
Implementation
Requirements Specification
Architectural Description
Code
Architecture Reconstruction
Reverse Engineering

Figure 1.1: Forward and Reverse Engineering. The combination of both is Software Reengineering.

hardly ever explicitly done in practice. The requirements specification will rather be recovered by looking at the visible behavior of the running system.

The anticipated design-level understanding of a given system cannot be obtained from the design documentation alone, because the latter is often out of date, or it does not exist at all. The only reliable source of information regarding the structure of a system is the system itself. SRE therefore usually tries to recover the system's design based on the code (*static analysis*) or on observations of the running system (*dynamic analysis*). Of course, any additional information that is available may be used and can be helpful in this process, but the system itself makes up the only information that is always available and reliable.

1.1.3 Software Architecture and its Recovery

Software Architecture is *"the fundamental organization of a system, embodied in its components, their relationships to each other and to the environment, and the principles guiding its design and evolution"* [86]. This is one thing that SRE tries to recover: the fundamental organization of a system. It is impossible to grasp the information about a system's organization all at once for non-trivial systems. Therefore, we need to look at the system from different **viewpoints** to describe it in its entirety [104]. Only the sum of different **views** (that is, instances of viewpoints) gives the full picture. Depending on the task at hand, we can then use the view that is best suited for that task. For example, if we want to gain a rough overview of a system, we will like to have a high-level logical view which contains just the basic design elements and their relationships (*conceptual view*). If we are to implement or test part of that system, we will need more detailed information, in particular interface definitions, which could be found in a view that shows the decomposition of the system into modules (*module view*) [80].

As motivated above, the architecture of a system is often not available and has to be reconstructed. Certain views are easy to reconstruct: in many cases, information about which routines exist and what their signatures are can be extracted from source code without much effort. For some programming paradigms, such as object-oriented programming, it is also easy to detect modules or classes and their syntactic interfaces, because these are explicitly described in the code – but for others, such as C, it is a hard problem to find out which routines and attributes belong together [96]. However, object-orientation also has its drawbacks: it makes programs harder to understand [211].

The module view may be easily available, but it contains information at a quite low level. A larger system may consist of thousands of modules. In order to just get an overview of the basic design of a system, it is infeasible to look at all the individual modules and their interdependencies: you will not see the wood for the trees. It is therefore necessary to recover more abstract views. However, this is a challenging problem: what are meaningful aggregations of modules? Also, the modules as defined by syntactical units may not be identical to the real *logical components* – groups of related elements with a unifying common goal or architectural concept. In other words, the modularization may be suboptimal. *Software clustering* techniques are one approach to tackle these problems. Unfortunately, the clustering criteria cannot be exactly defined; therefore, support of a human expert is indispensable [96, 129].

An alternative to clustering is *filtering*: the information is reduced to those artifacts that are probably most relevant for the currently examined aspect of a system. Unfortunately, this bears the risk that too much information is filtered out. Apart from automatic approaches, both techniques – clustering and filtering – are also employed in interactive *Software Visualization* environments.

Visualization is *"the process of transforming information into a visual form, enabling users to observe the information. The resulting visual display enables the scientist or engineer to perceive visually features which are hidden in the data but nevertheless are needed for data exploration and analysis."* [59] The visualization of software is another important technique for software architecture recovery: information about artifacts of a system is visualized in some way, and the interpretation and abstraction is left to the human user. Also, when higher-level abstractions have been built automatically, the results have to be presented to the user, which is a visualization as well.

1.1.4 Problem Statement

We have seen that information about modules and their interdependencies can be recovered from the source code in many cases. However, the interface of a component consists of more than just exported declarations. According to Parnas [140], the interface also covers the assumptions that a using component is allowed to make about it, and the assumptions that the component makes about its used components. For example, this covers pre- and postconditions for routines or restrictions on allowable parameters. Another important aspect of the interface is the set of allowable sequences of routine calls. In this thesis, we call this the *protocol* of the component. Whereas the exported declarations are the *syntactic* interface, the protocol is part of the *semantic* interface. This is another aspect of a software's architecture that may need to be recovered.

```
typedef struct { ... } Stack;    /* Stack data structure          */

Stack *create();                 /* creates a new Stack instance  */
void   init  (Stack *s);         /* initializes a Stack           */
void   push  (Stack *s, Item e); /* put e on top of the Stack     */
Item   pop   (Stack *s);         /* remove+return the top element */
```

Figure 1.2: Interface of a Stack component.

Definition: The **protocol** of a component defines the allowed sequences of routine calls that may be invoked on it or on an instance of it.

As an example, take a look at the synactic interface of a Stack component as shown in Figure 1.2. Let us assume the usual semantics for a stack. When using this component, you have to be aware of several restrictions: a Stack object first has to be created and initialized (`create`, `init`) before any of the other routines may be called. Additionally, pop may only be called if there is at least one element on the Stack, or, in other words, if the total number of pushes is higher than the number of pops. Otherwise, there would be no element to remove from the Stack – an error which may lead to weird subsequent errors if it is not handled in an adequate way. The source of this error can be very hard to find. It is easy to avoid such errors if the protocol for the component is defined and checked for. This example shows that there can be sequence restrictions and also more complex usage restrictions for a component. The protocol of the component describes these restrictions.

Unfortunately, defining such a protocol is not supported by most programming languages. A component's protocol therefore often remains unspecified. Even if it *is* specified, this is usually only done in an informal way, either in the documentation or as comments in the code. When the protocol of a component is available, it is very useful for automatically checking if the component is used correctly in a given application. It can also be used to apply state-based testing techniques to the component itself to check if it is implemented correctly. When the protocol is *not* available, systematic testing of stateful components is hardly possible. Therefore, it is absolutely desirable to have a formal specification of the protocol. And it would be of great help if the protocol could be automatically or semi-automatically extracted from the code. This reconstruction is called **protocol recovery**.

There are two possible approaches to this problem [100]. One can look inside the implementation of the component to find out about its protocol, which may work when defensive programming has been used (*glass-box understanding*). The alternative is to look at how the component is being used by one or several applications (*black-box understanding*). If the applications use the component in a correct way, this should also give a good picture about the allowable ways of using the component. One contribution of this dissertation is a new dynamic black-box protocol recovery technique.

1.2 Approach

As introduced above, the **protocol** of a component (in the sense of this thesis) describes the allowable sequences of operations that may be applied to it. However, a component is not purely static – its *state* has to be considered when talking about the protocol. When a component has no state, the allowable order of operations is usually unrestricted. It is therefore more interesting to regard the protocol of a dynamic **object** instead of a static component. From a general perspective, we could define any allocated region of storage to be an object. The protocol for such an object would not be of much help because memory only has the operations `read` and `write`. Therefore, let us take an object-oriented view and additionally regard the routines that operate on that memory as belonging to the object. Routines that belong to one particular object are called its **atomic methods**.

As motivated in the previous section, the protocol is often unspecified and has to be recovered first. Quite a number of *protocol recovery* approaches have been published in the past; Chapter 9 gives a complete overview. Very few of these approaches examine the component's code: the majority belongs to the black-box understanding class. These black-box approaches either use static or dynamic analysis. They are either based on the concrete sequences of operations that are or may be applied to the object, or they abstract from the object's state. The latter track how the object's state – that is, the contents of its memory area – changes and how these changes are related to atomic method calls. The resulting state automata describe the impact of modifying method calls on the object's state; read-only method calls are not considered and cannot be considered. These approaches therefore produce a kind of protocol that is different from what we are interested in. For example, the fact that `is_empty` is always called before `pop` for a given stack object cannot be recovered.

Approaches that are based on analyzing concrete sequences of operations on an object try to deduce the general protocol of the object from these sequences. Regular grammar inference techniques are usually applied for this. Using such an approach has several advantages: the necessary interface-level information is easy to extract, the instrumentation is quite lightweight, and regular grammar inference techniques have been studied for a long time and are readily available. On the other hand, these approaches have to speculate whether sequences of identical operations result from a loop or not, because they do not take into consideration the context in which each operation is called. Therefore, they tend to overgeneralize. One effective approach to prevent this is to query the user in the generalization step. However, this interaction is quite expensive, since a lot of queries are necessary even for simple protocols [201].

Figure 1.3 shows an example for a protocol recovery approach that is based on interface interactions only. We regard a simple Stack object (depicted in the center) as introduced above. The activity diagram on the left-hand side illustrates the sequence of operations that is applied in a given program run which uses this object. This sequence is always linear, that is, it does not have any branches.

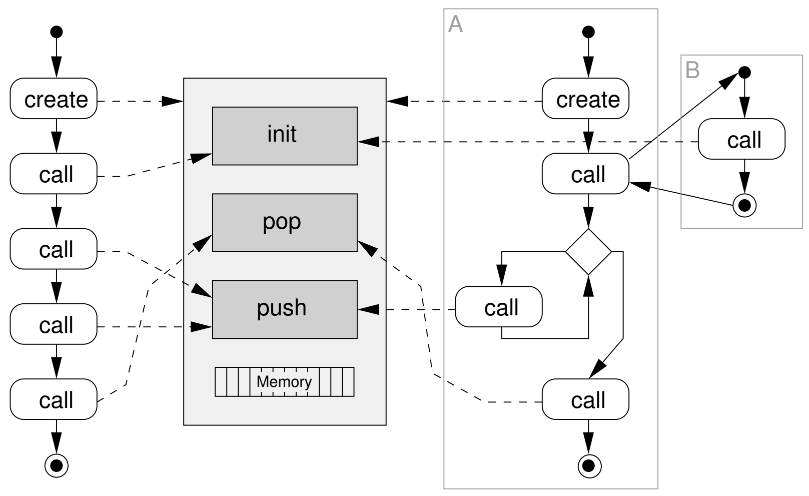


Figure 1.3: Pure sequence of operations called on an object (left) compared to the OPG representation (right), illustrated on a simple Stack object (center). A and B denote routines.

Also, the call sites are anonymous, which means that we do not know *who* is using the object; we just know *that* it is used. This kind of information is the input to traditional dynamic protocol recovery approaches.

To improve the basis for protocol recovery, I propose to widen the perspective with respect to the object: let us allow the object to find out from where it is used and how these usages are connected through control flow. This information can be represented by a data structure called **Object Process Graph** (OPG). The right-hand side of Figure 1.3 shows the corresponding OPG. From that diagram, we can additionally identify control structures such as branches (the diamond demarks a decision), loops, and routine calls which affect how the program gets to that point. Based on that graph, we can find out for sure whether an operation is called from within a loop or not, and this in turn should improve the quality of a protocol that is recovered on this extended basis.

The OPG from the example is quite small and clear, and it would look just the same even if the use of that Stack object was embedded into a much larger program. It is a quite meaningful subgraph of the control flow graph, and in this regard it is the result of automatic filtering. The OPG may even give a good impression about the application's general structure, if the object is of central concern for the application: it shows the locations in the program where the object is used and how these uses relate to each other. If object and function names are meaningful, this can give additional hints for understanding a program. Altogether, OPGs have the potential for being a good starting point for program comprehension.

The basic idea of OPGs and a static technique for extracting them was proposed by Eisenbarth et al. [46]. However, the graphs that result from their technique become very large for objects with a complex usage profile. They often contain lots of infeasible paths due to the conservative assumptions that are necessary in any static analysis.

In this thesis, I describe how OPGs can be extracted by dynamic analysis, resulting in *Dynamic Object Process Graphs* (DOPGs). This reduces the graphs' size, avoids infeasible paths, and enables new applications that are impracticable with the static approach.

My hypothesis is that

1. OPGs can be extracted dynamically (feasibility),
2. DOPGs are a good basis for protocol recovery, and
3. visualized DOPGs can be helpful for program understanding.

1.3 Contributions

The contributions of my thesis are severalfold:

- **OPG meta-model**: I introduce a meta-model for OPGs. Such a model has not been explicitly defined before. The model contains support for multithreading and exception handling.
- **Dynamic OPG extraction techniques**: I describe in detail how OPGs can be extracted by dynamic analysis This particularly includes instrumentation of the program and transformation of traces to DOPGs. Exception handling and multithreading is considered in this approach. Apart from the basic extraction technique, I also introduce an online extraction technique which reduces the runtime overhead. This makes the approach applicable in practice.
- **Feasibility of approach**: The feasibility is shown in a case study that measures the runtime overhead of this approach. Also, DOPGs for a number of systems and objects are exemplarily studied.
- **Comparison to static technique**: In another case study, I quantitatively compare the OPGs resulting from dynamic tracing to those from static tracing.
- **Application for protocol recovery**: Building on others' research, I describe the OPG based protocol recovery process in detail and apply it to DOPGs.
- **Comparison to other dynamic protocol recovery techniques**: I compare the results of this protocol recovery approach to other existing dynamic approaches. As an additional indicator for this comparison, I introduce and use a new automaton difference metric.
- **Application for program understanding**: I report from a controlled experiment that I conducted to investigate whether visualized DOPGs help in program understanding. Procedure and results are discussed in detail in this thesis.

1.4 Project Context

The work on this thesis was performed as part of the **Bauhaus** project [97, 154]. Bauhaus started in 1997 as a research collaboration between the Institute for Computer Science of the University of Stuttgart and the Fraunhofer Institute for Experimental Software Engineering in Kaiserslautern (IESE). Today, it is a collaboration between the University of Stuttgart and the Software Engineering group at the University of Bremen. Since 2005, the commercial spin-off "Axivion"[1] is also part of the project.

The goal of this project is the development of methods and tools that support a maintenance engineer in his or her work. In particular, these methods aim at semi-automatically recovering different views on an existing system's architecture. The project is internationally well-known for its contributions in the areas of architecture reconstruction, feature location, clone detection, software clustering, product line consolidation, and protocol recovery.

The project has meanwhile created a strong infrastructure that supports analyses on different levels of granularity. The fine-grained intermediate representation (IML) is capable of representing programs from different languages and is equipped with control flow, data flow, and points-to analyses. The coarse-grained representation (RFG) contains information and dependencies at the interface level only, but is more appropriate for architectural analyses and can handle very large systems.

Traditionally, the Bauhaus project focussed on static analyses, with the exception of a combined static/dynamic feature location technique [46]. The work presented in this thesis is based on dynamic analysis and program transformation. Both techniques introduce new aspects to the Bauhaus project. On the other hand, it is a continuation of other people's work within the project: several researchers and students have also worked on the topics tracing, OPGs, and protocol recovery, and Gunther Vogel is currently finishing a closely related dissertation which covers static OPG extraction and protocol recovery issues [198].

[1]`http://www.axivion.com/`

1.5 Previously Published Content

A large share of the contents of this thesis has been previously published. Table 1.1 summarizes these publications and where their contents occur within this thesis.

Reference	Title	where published	Chapters
[148]	Dynamic Object Process Graphs	CSMR 2006	3, 6
[150]	Dynamic Object Process Graphs	Journal of Systems and Software	3, 5, 6
[146]	Online Construction of Dynamic Object Process Graphs	CSMR 2007	4
[149]	Dynamic Protocol Recovery	WCRE 2007	7
[147]	Do Dynamic Object Process Graphs Support Program Understanding? – A Controlled Experiment	ICPC 2008	8

Table 1.1: Previous publications of this thesis' content.

1.6 Thesis Outline

This thesis is organized in four parts, with the two middle parts containing the main contributions. In Part II, the dynamic extraction techniques are presented and evaluated, and in Part III, two applications of DOPGs are examined in more depth. Figure 1.4 sketches the general organization of the thesis.

The following Chapter 2 is an introduction to the notion of Dynamic Object Process Graphs. It contains the meta-model for this special kind of graph, summarizes an existing static extraction technique, and discusses the general differences between static and dynamic analysis.

The basic dynamic extraction technique is presented in Chapter 3, which starts Part II. As opposed to the constructive static technique, this approach is based on graph transformations. The necessary instrumentation and trace analysis are described in detail. To master the overhead that is caused by tracing, an online extension to the algorithm is introduced in Chapter 4. This chapter closes with a case study that analyzes the runtime overhead for different systems with the two approaches. Chapter 5 compares DOPGs to the corresponding graphs that have been extracted by static analysis. This case study completes Part II.

Part III starts with the presentation of several case studies (Chapter 6). They demonstrate the use and feasibility of the approach and sketch potential applications. These case studies are the basis for the deeper investigation of two main applications: Protocol recovery and program understanding. Protocol recovery is discussed in Chapter 7. Existing ideas for OPG based protocol recovery are

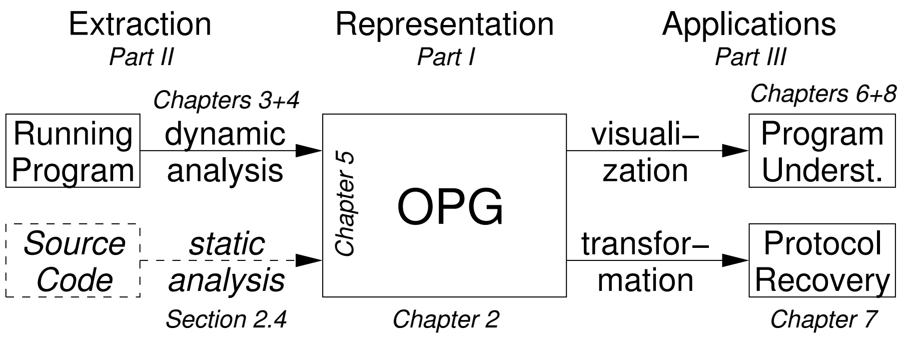


Figure 1.4: Thesis Overview.

extended and described, and in a case study, this technique is compared to other common dynamic protocol recovery techniques. Chapter 8 contains the description and discussion of a controlled experiment that was conducted to find out if DOPGs really support program understanding – a hypothesis that was raised as a result from the case studies.

Chapter 9 in Part IV contains an extensive overview of related work, and Chapter 10 concludes.

Chapter 2

Tracing and Object Process Graphs

In this chapter, I define what a trace is in the context of this thesis and show how sets of traces can be represented by Object Process Graphs. I give a constructive definition and a meta-model for these graphs. The basis for this has been provided by Eisenbarth, Koschke and Vogel in their papers about static trace extraction [45, 46]. In this chapter, it is formalized and extended to cover techniques such as multithreading and exception handling. I give a short summary of their static extraction technique and their experiences with it. A general discussion of static and dynamic analyses and their pros and cons finishes the chapter.

2.1 Traces for Individual Objects

Let us start with a motivating example. Consider the C program in Figure 2.1. It deals with two stacks `*s1` and `*s2`. We assume the usual semantics for stacks here. Function `read` reads a stack from a file, and `init` creates an empty stack. Although the program has passed all tests, how sure can we be that it does not cause a failure? And in fact, it contains a potential fault that is difficult to spot in the code: a violation of the stack protocol for variable `*s1`. Through the static analysis by Eisenbarth, Koschke, and Vogel, we can extract all sequences of operations

```
void main () {
  int i = 0;
  Stack *s1 = init();
  Stack *s2 = read();
  reverse(s2, s1);
  do {
    pop(s1);
    i = i + 1;
  } while (!empty(s1));
}
```

```
void reverse
  (Stack *from, Stack *to)
{
  while (!empty(from)) {
    push(to, pop(from));
  }
}
```

Figure 2.1: Example source code.

potentially applied to `*s1` – including those that violate the protocol. (We will lateron see how this can help to detect the error, and what the actual error in this example is.)

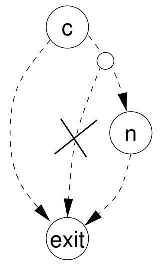
Each such sequence of operations is an *object trace*. A **trace** is *"a record of the execution of a computer program, showing the sequence of instructions executed, the names and values of variables, or both."* [85] We are specially interested in how a given object is used within a program. By object, I mean a certain memory area along with the associated operations on it. In practice, this can be a local or global variable or a variable allocated on the heap at runtime – or an instance of a class. An **object trace** is a sequence of operations applied to one specific object.

2.2 Definition of an Object Process Graph

As stated earlier, we want to "widen the perspective" of a regarded object: we want to know the locations of any object uses, and we want to know how those uses relate to each other. The data structure that contains the information about the latter is the interprocedural **control flow graph** (CFG). The CFG is a well-known concept from compiler technology. A CFG is constructed by creating one node for each statement in the code and connecting those nodes via edges according to the control flow of the application.

However, a CFG contains information about the entire program, which makes it very large and bulky. For understanding a program or getting information about the use of certain objects, this data structure is not adequate. An **Object Process Graph** (OPG) is a *projection* of the control flow graph that contains only those nodes that are *relevant* from the perspective of a given object. They reduce the CFG to the really interesting parts (with respect to that object) and allow us to reason about individual objects. The nodes of an OPG correspond to events in an object trace: the OPG describes a potentially infinite object trace in a finite closed form.

We now have to define what "relevance" means for a CFG node. This definition is in turn based on the notion of *control dependency*.



Definition: A node n of the CFG is **directly control dependent** on a node c iff c decides whether n is executed or not. This is the case when c has at least two successors, there is a path from c to n, there is a path from c to the exit node that does not pass n, and c is the last node with that property.

Definition: A node n of the CFG is **relevant** for an object obj iff

- it represents an operation on obj (creation, attribute access, or invocation of one of the object's methods), or
- there exists a relevant node that is control dependent on n.

```
Input: CFG, object
Output: OPG for that object

Mark all nodes of the CFG that represent operations on the given object.

while there is an unmarked node $n$ of which at least one of the marked
nodes is control dependent do
    mark $n$

foreach unmarked node $u$ do
    remove $u$, connect the dangling edges

Insert start, end, and return nodes.
```

Figure 2.2: Basic OPG construction algorithm.

This recursive definition of relevance can be used to construct an OPG from a given CFG and object. The algorithm is shown in Figure 2.2. Note that it is missing some details; it is just intended as an illustration of the general idea. Details about the construction algorithm are provided in the subsequent chapters.

An OPG is represented as a typed, attributed, and directed multigraph. "Typed" means that the nodes' labels are divided into classes, called types, and that edges of a certain type are restricted to be incident only to certain types of source and target nodes. "Attributed" means that attributes, such as numbers or text, can be attached to the nodes and edges. "Multigraph" means that there may be more than one edge that connects the same two nodes. Graphs of this kind have generally proven to be useful for Reverse Engineering and are known as TGraphs [42].

The meta-model for OPGs is shown as a UML class diagram in Figure 2.3. This model describes which node classes may be connected by which edges. Thick vertical arrows denote generalizations, while thin lines denote associations between classes. It should be noted that the classes do not require a lot of attributes besides the associations between them: the object that an OPG relates to is predetermined, so only method names and source locations (SLoc) remain.

Nodes can be roughly partitioned into two categories: nodes which represent atomic access or creation of the object (BasicNode), and nodes which are relevant for control flow (DecisionNode, Call/Entry/ReturnNode). Edges stand for intraprocedural (succ/true/false), interprocedural (call/entry/return), inter-thread (new_thread), or exceptional control flow (exception/exceptional_return). Tables 2.1 and 2.2 provide a summary of the different node and edge types and their meaning. An OPG is a subgraph of the corresponding CFG, therefore each node in the OPG represents a location in the program, and edges represent control flow between these locations.

All nodes of the OPG must be reachable from one unique start node *st*, that is, there must be a path from *st* to each other node. This implies that the graph is connected. Also, all referenced nodes and edges must be members of the OPG, that is, the graph must be closed. An OPG can more formally be defined as a

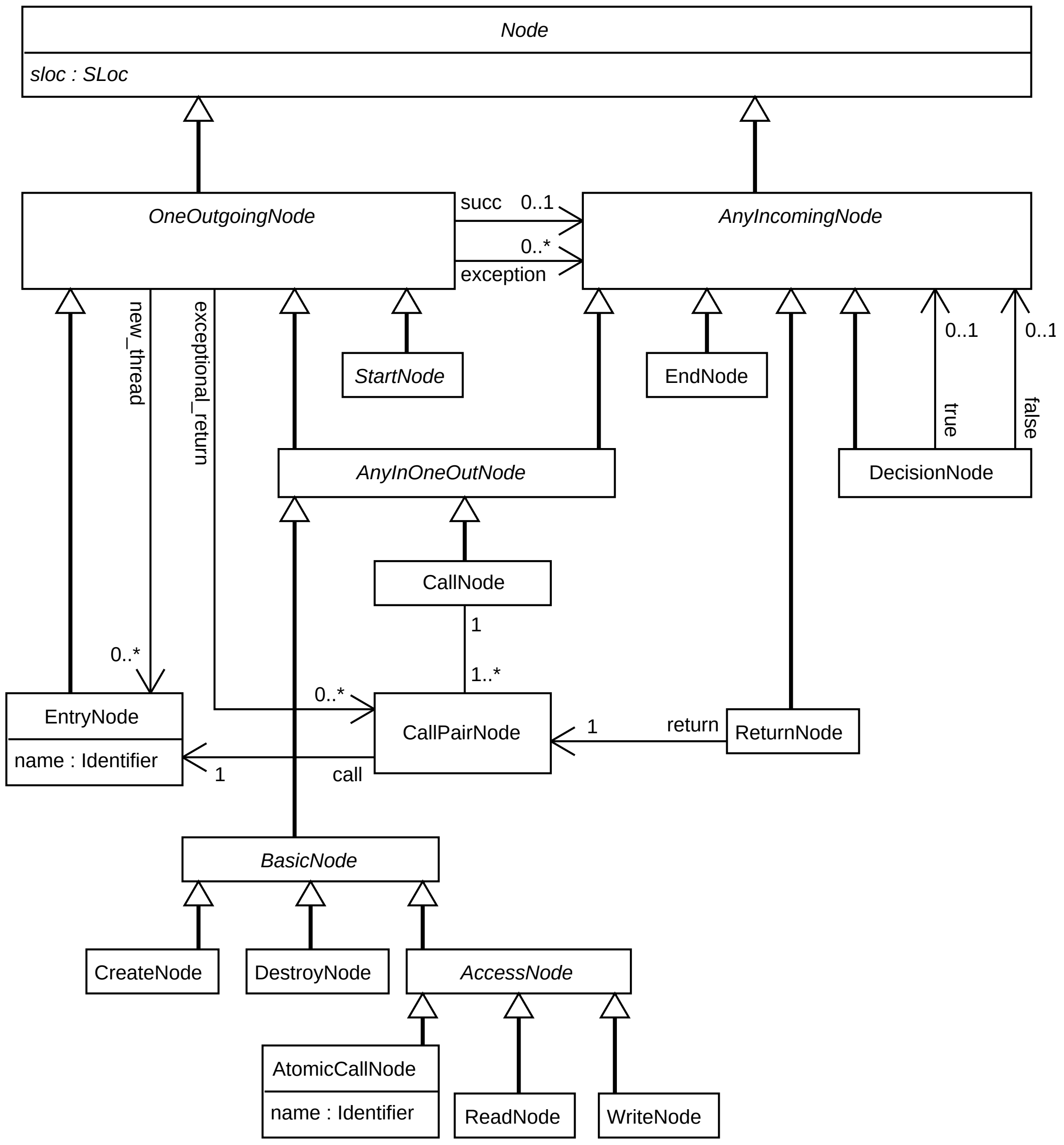


Figure 2.3: OPG meta-model.

Class	Symbol	Meaning
StartNode	●	Unique start point of the OPG.
CreateNode	create	Creation of the object. This can be a `new` or `malloc` statement or the declaration of a variable.
DestroyNode	destroy	Destruction of the object. This optional node can for example be a `free` statement.
ReadNode	read	Read access to the object's memory.
WriteNode	write	Write access to the object's memory.
AtomicCallNode	operation()	Call of one of the object's methods. These are called *atomic* methods.
DecisionNode	F ◇ T	A point where control flow can take two different ways, depending on the boolean value that was calculated by the previous operation or call.
CallNode	call	Routine call site. The information about the called routine is kept in CallPairNode.
CallPairNode	call	A helper node that keeps track of an EntryNode and the ReturnNode that belong together. A `call` edge leads to the corresponding method `entry` node, and a `return` edge leads from the `return` node back to this node.
EntryNode	entry	Method entry point.
ReturnNode	return	Returns from a method invocation to the call site.
EndNode	◉	End point of the OPG.

Table 2.1: Node classes overview. Each node corresponds to a statement in the source code, except for the Start-/EndNode and CallPairNode.

Name	Meaning
succ	Unconditional control flow.
true	Conditional edge, which is taken in case the previous predicate evaluates to *true*.
false	Conditional edge for the *false* case.
call	Routine call.
return	Return from a routine to the call site.
exception	Invocation of an exception handler.
exceptional_return	Exceptional return from a routine invocation, in case an exception has not been caught inside the routine. This is an alternative way of returning from a routine.
new_thread	Creation of a new thread.

Table 2.2: Edge types overview. Edges represent control flow.

set of nodes which are typed according to the meta-model classes, along with the connecting edges and the above mentioned additional constraints:

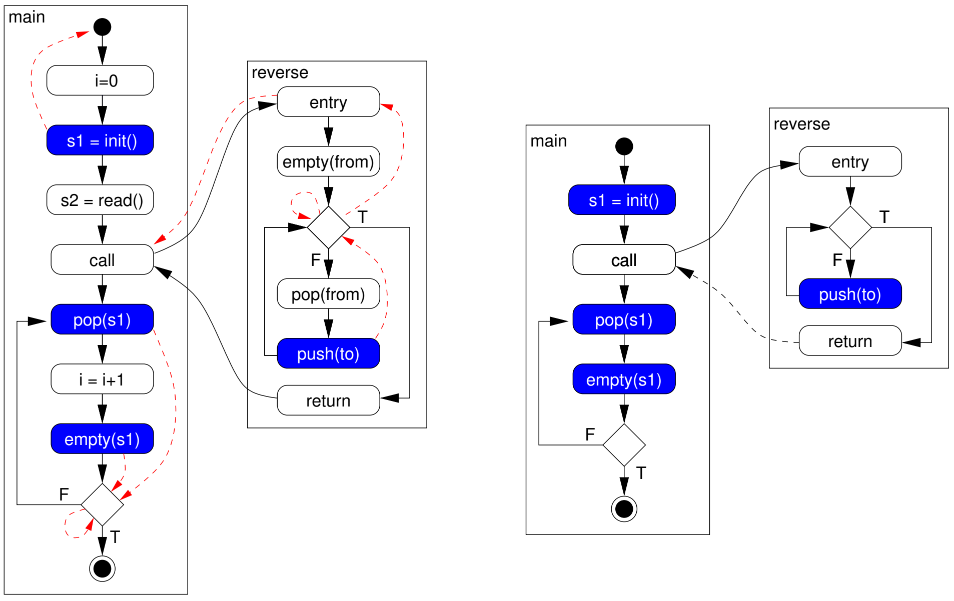
$$\begin{array}{rcl}
OPG & := & (V, E, s, t, l, a) \\
\text{with} & & s : E \to V,\ t : E \to V \\
 & & l : V \cup E \to Class \\
 & & a : (V \cup E) \times Attribute \to Value \\
 & & \exists\, st \in V :\ \ l(st) = \text{"}\texttt{StartNode}\text{"} \\
 & \wedge & |\{st \in V \mid l(st) = \text{"}\texttt{StartNode}\text{"}\}| = 1 \\
 & \wedge & \forall v \in V :\ v \text{ is reachable from } st \\
\text{where} & & V \text{ is the set of nodes,} \\
 & & E \text{ is the set of edges,} \\
 & & s \text{ and } t \text{ assign a source and target node to an edge,} \\
 & & l \text{ labels each node and edge with a class, and} \\
 & & a \text{ assigns additional attributes to nodes and edges.}
\end{array}$$

UML activity diagrams [165] are used to represent the work flow of a system and therefore can be used as a notation for Object Process Graphs. The mapping between OPG nodes and activity diagram nodes is straightforward: the Start-Node is mapped to UML's *initial node*, an EndNode maps to an *activity final node*. DecisionNodes are also called *decision*s in UML. CallPairNodes are depicted by connecting the call and return edges that belong together with an arc. All other node types are mapped to activities and labelled with their type. Table 2.1 shows the appearance of each node type.

2.3 Object Process Graph Construction Example

Let me illustrate by an example how an OPG is constructed from a program's CFG. The control flow graph for object `*s1` of Figure 2.1 is shown in Figure 2.4(a). The first step is to identify and mark atomic nodes. In the Figure, atomic function calls for `*s1` are marked in black. In the next step, the call of `reverse` and the decision nodes (diamonds) are added to the set of marked nodes, because there are atomic nodes which are control dependent on them. See the dashed arrows in the Figure which indicate selected control dependencies. After that, all other nodes are removed, resulting in the graph that is shown in Figure 2.4(b). Note that it contains only those operations in the program that may be applied to `*s1`; the other two objects `*s2` and `i` are not covered. The atomic methods in this example are those of the stack interface, namely, `init`, `pop`, `push`, and `empty`. That is, we treat the stack implementation as a black box.

Let us now take a control flow perspective of the process in which `*s1` is involved. The object is created and returned by `init` in line 3 and then passed to `reverse` in line 5. Within `reverse`, it is passed to the atomic method `push` in



(a) Control flow graph. (b) OPG for *s1.

Figure 2.4: Control flow graph and OPG for the sample program from Figure 2.1. Atomic method calls on object `*s1` are marked in black. The dashed arrows demark selected control dependencies. The right figure shows the OPG for `*s1` – a subgraph of the CFG.

line 15 as first parameter. Because that `push` call site is control dependent upon the condition in line 14, this condition is part of the OPG, too. Because `reverse` contains relevant nodes, its call site in line 5 is relevant as well. Upon return from `reverse`, `*s1` is accessed by `pop` in line 7 and `empty` in line 9, which are atomic methods. Again, because the loop body contains relevant nodes, the condition of the loop is kept, too.

Now, let us return to our original question about the fault in this program. The potential fault is easier to spot in the OPG: there is a path in the program in which `pop` may be applied to an empty stack. An empty stack may occur if the predicate in `reverse` is `false` the first time, which in turn is the case when the other stack object `*s2` is empty.

2.4 Static OPG Extraction

Object Process Graphs were introduced by Eisenbarth et al. [45, 46] to *"describe the set of [static] traces relative to a statically detectable object"*. A statically detectable object can be a local or global variable or a heap allocation site. It is identified by its allocation point. Dynamically, there can be many different incarnations of these statically detectable objects. The statically detectable objects are an equivalence class for dynamic objects that share the same potential behavior (that is, the sequence of operations applied to them).

Eisenbarth et al. [46] described a static OPG extraction algorithm. It is based on the CFG and runs in two phases. In the first phase, a corresponding object process node is added for each CFG node that is atomic for the given object. This is done through a context-sensitive traversal of the CFG, which means that multiple nodes may be added for the same CFG node if they occur in different contexts. In the second phase, these object process nodes are connected by control-flow edges. Additional merge and branch nodes are added in this phase if necessary: a predicate from the CFG is added as a branch node if any node from the OPG is control dependent on it. This construction directly follows from the definition of relevant nodes as given above.

To find out whether a node is relevant or not, static analysis depends on points-to information. Points-to analyses detect all potential aliases for objects. The problem that each points-to analysis faces is that many languages allow a programmer to circumvent the type system. Therefore, these analyses cannot rely on type information: each pointer may potentially point anywhere. The task of a points-to analysis is to find out where it can *really* point to, given the restrictions of the program.

Points-to analyses mainly differ in two aspects [77, 78]:

- Context-sensitivity: The analysis may either distinguish points-to relations at different call sites (context-sensitive), or it may unite all possible calling contexts (context-insensitive).
- Flow-sensitivity: The flow of control may be considered (flow-sensitive) or ignored (flow-insensitive).

Context- and flow-sensitive analyses deliver quite precise results, but are very expensive and do not scale for real world applications. On the other hand, when using less precise points-to information, the results of any analysis that is based on this information get less precise as well. (Additional information on concrete points-to analyses can be found in Chapter 5.)

This is also the case for static OPG extraction. Since points-to analyses usually are conservative, the extracted OPGs will always be an overestimation of the "real" OPG: they will most surely contain infeasible paths, and we have no clue what share of paths is in fact infeasible. The case study by Eisenbarth et al. [46] investigated how use of different points-to analyses affects OPG extraction results. Both the context- and flow-sensitive analysis by Wilson [213, 214] and a flow-insensitive, context-insensitive Steensgard-style analysis [177] were applied on different systems as the basis for static OPG extraction. The result was that the differences are immense. The exact Wilson analysis was only applicable for systems of up to 5 KSLOC, which makes it unusable in practice. The Steensgard analysis produced graphs that were between two and five times the size of the corresponding Wilson-based graphs on average (for heap objects). A system of 74 KSLOC resulted in OPGs of up to 90,000 nodes. In summary, the conservative assumptions that an efficient points-to analysis has to adhere to can make the results of static OPG extraction unusable.

2.5 Static vs. Dynamic Analysis

As discussed above, static analyses face several problems which may restrict their applicability. Besides the aliasing problem, also techniques such as polymorphism, dynamic binding, or distribution are hard to handle with static analyses. They are mostly based on conservative assumptions which lead to infeasible paths – paths that can never be executed by the program.

An alternative is to use dynamic analysis [183]. While static analysis is based on the source code only, dynamic analysis investigates a program's behavior during runtime. When the program is actually executed, we know for sure where a concrete pointer points to, or which concrete routine is called by a function pointer. Therefore, the problems of static analysis are not an issue for dynamic analysis.

Unfortunately, dynamic analyses face different problems. When a program is executed, its behavior depends on the input – and the possibilities for different input are usually infinite. Therefore, the results of any dynamic analysis will in most cases be incomplete. Another problem is that the behavior of the program is changed by any dynamic analysis: either the program itself has to be changed (*instrumentation*), or the program has to be executed in a virtual machine. It may become larger or be executed slower. Also, the resulting amount of data gets huge, depending on which information is to be monitored – specially in the presence of loops. Furthermore, execution of a program may be expensive, for example if special test equipment is required.

Apart from these potential drawbacks, dynamic analysis also offers additional advantages over static analysis. Static analysis has a quite local view on a program due to the necessary abstractions: the further it proceeds, the more information is lost. In contrast to that, dynamic analysis is capable of revealing more distant relationships. It may therefore be helpful for recovering *delocalized plans* – pieces of code that are conceptually related, but physically located in non-contiguous parts of a program [175]. Another advantage is that basic dynamic analyses are easier to implement than static analyses, which often require a strong infrastructure of control and data flow analyses as a basis.

Each approach has its pros and cons. In several ways, they complement each other: dynamic analysis gives us the lower bound of what could happen whereas

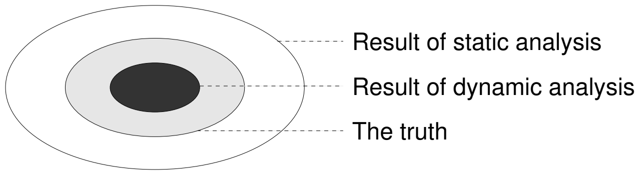


Figure 2.5: Results of static and dynamic analysis in comparison: static analysis usually delivers an overestimation, dynamic analysis an underestimation. The truth lies inbetween.

static analysis yields the upper bound. This is illustrated in Figure 2.5. Therefore, the combination of static and dynamic analyses is quite promising [51]. And, of course, it depends on the goal of the analysis which one is more appropriate. In Chapter 5, we will investigate how the results of static and dynamic OPG extraction compare to each other.

2.6 Summary

This chapter provided the basics about OPGs. It introduced a schema for this special class of graphs and discussed an existing static extraction technique. We learned that dynamic analysis is in many aspects complementary to static analysis. In particular, dynamic analysis avoids certain problems that static analysis faces. An OPG extraction technique that uses dynamic analysis is introduced in the next chapter. The Stack program will be used as a running example.

Part II

Extraction

Chapter 3

Dynamic OPG Extraction

We have seen how OPGs can be extracted by static analysis and what the drawbacks are. A corresponding dynamic analysis will circumvent the problems of static analysis and enable new applications for OPGs. On the other hand, it will only result in a subgraph of the "real" OPG.

This chapter introduces such a dynamic extraction technique. We start with the instrumentation of the subject system which makes it produce traces. Then I describe how to build a graph from these traces and how to transform that graph to an OPG. An example illustrates the approach.

3.1 Instrumentation

For collecting runtime information from a running program, which is the basis for any dynamic analysis, a *program monitor* is needed. This monitor can be located at different stages of compilation and execution. Figure 3.1 shows an overview of possible instrumentation locations. The monitor can either modify the code and insert tracing commands, or execute the program in a virtual machine [170]. Both of these possibilities have their pros and cons. When executing the program in a virtual machine, the connection to the source code may be missing, specially for programs that have been compiled to machine language (like C). When using a debugger for the same purpose, the line by line step width may not be fine-grained enough. For example, when several branches of a statement are located on the same line and the condition is complex (for example, containing shortcut assignments and short-circuit evaluation), one would have to emulate the entire evaluation in order to analyze each condition – this would end up in implementing a virtual machine. On the other hand, this approach may be capable of tracing library routines, which is not possible with code instrumentation in many cases.

Code instrumentation means that we insert additional statements into the program that output the state of the program or collect whatever information is necessary at a certain point. The advantage of this approach is that we do not depend on the existence of an appropriate debugger or virtual machine. On the other hand, we modify the code, so the program's behavior might change.

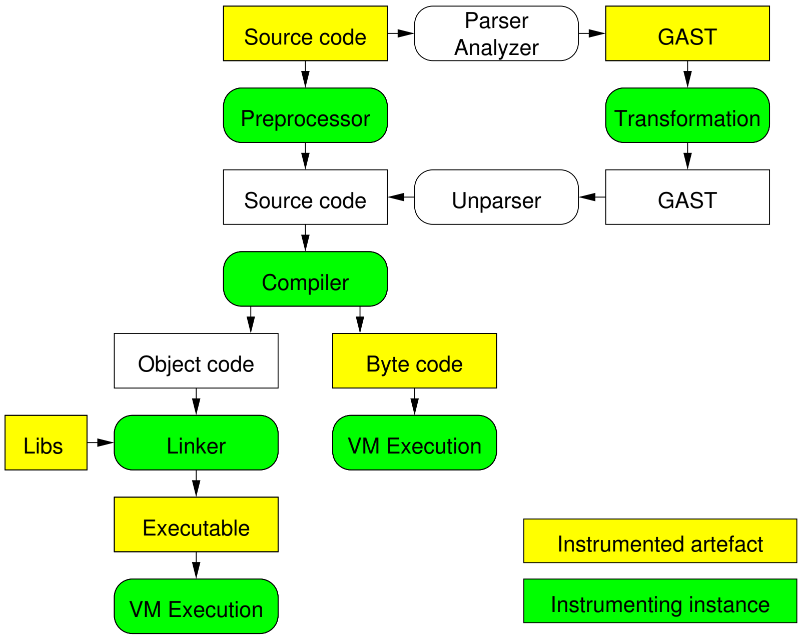


Figure 3.1: Overview of different possible instrumentation locations.

Additionally, increased program size may be a problem, particularly in embedded systems.

Different stages of code instrumentation are possible:

- *Source code*: Insert additional statements directly into the source code. This is only practicable for instrumentation of artifacts that are easily identifiable in the code. Problems could be caused by complex expressions. Also, this approach is highly dependent on the programming language.

- *Generalized Abstract Syntax Tree*: Insert additional nodes and edges into the generalized abstract syntax tree (GAST [98]), then generate source code from that. This has the advantage of working on a level that abstracts from the concrete programming language.

- *Compiler's intermediate representation / byte code*: Insert additional code into an intermediate representation. Such a representation is usually built of fewer atomic statements and can therefore be instrumented easier.

- *Binary code*: Insert additional code into the binary executable of the program. This can be done either statically or dynamically. The shortcoming of this method is that the instrumentation is different for every target machine. However, this is not an issue for virtual machine code (byte code).

For the purpose of DOPG extraction, it is important that the program monitor works on a fine-grained level. It is necessary to catch every branch in control flow for reconstructing conditional guards correctly. Every access to an object's attributes needs to be catched as well. Also, the connection to source code should be kept as good as possible, so that we can later get to the corresponding source location for every node of a Dynamic Object Process Graph.

Given these requirements, I chose to instrument C code on GAST level. A GAST is a language-independent abstract syntax tree which is annotated with additional semantic edges that describe the details of the original program [98]. A front end for a given programming language converts the source code to this uniform representation. The instrumentation step then inserts additional nodes and edges into the graph. It basically performs a sequence of graph transformations on this intermediate representation. Finally, from the modified graph, source code is generated, and this can be compiled and linked to an instrumented executable. The top right corner of Figure 3.1 illustrates the approach.

Instrumentation on this representation has the advantage that it is similar for different programming languages. The effort required to switch the language mainly depends on the design of the GAST and on the constructs to be instrumented. In my case, the GAST was already available: it is called *IML* in Bauhaus [154]. Only the code generation step (*unparser*) was missing and had to be implemented.

Although conceptually possible, instrumentation of Java code on this level did not work out in practice. It turned out that the IML produced by the different frontends was in fact not as similar as one would expect – probably mainly due to the early stage of development of Bauhaus' Java frontend. The incompleteness of the Java IML at the time my work was done raised the necessity for an alternative instrumentation technology for Java programs.

Java programs are compiled to byte code, which is then executed by a virtual machine (JVM). This byte code is based on a small instruction set which abstracts from different loop types, represents complex instructions as a sequence of basic instructions, and so on [118]. This makes instrumentation on this level very convenient: the code is already normalized, and there is no need for specially handling complex expressions. Also, a system can be analyzed even when the source code is not or not completely available. Furthermore, instrumenting byte code is very fast and easy to do. For these reasons, I additionally implemented a byte code based instrumentation for Java. The instrumentation is performed on Java byte code using the ASM framework[1]. However, this also has some disadvantages: instrumentation is only applicable for languages which are compiled to JVM byte code, and the connection to source code is only available on source line level.

The next paragraphs describe the transformations that are necessary to instrument code for DOPG extraction.

3.1.1 Normalizing Transformation

For the GAST based instrumentation, certain constructs are converted to a normalized form first. This is done in order to avoid handling all the different possibilities of specifying a loop separately. With this normalization, all those loops can later

[1] `http://asm.objectweb.org/`

```
void reverse (Stack *from, Stack *to) {
  while (!empty (from)) {
    push (to, pop (from));
  }
}
```

(a) Original code.

```
void reverse (Stack *from, Stack *to) {
  L1:
  if (!empty (from)) {
    push (to, pop (from));
    goto L1;
  }
  else {
  }
}
```

(b) Normalized code.

Figure 3.2: The `reverse` while loop before and after normalization.

be handled in a uniform way [98]. The information about the original type of control construct is lost, but this information is not important for the Object Process Graph representation, and it is still available through the source code location. A GAST like the IML already contains this normalization; it just has to be made explicit (convert nodes to more general ones) to get the right results in the code generation step.

All loops can be expressed in terms of `if`, `goto`, and `label`. Loops and also `switch`/`case` statements are replaced by `if`/`goto`/`label` constructs. Occurences of `break` and `continue` are replaced by `goto`s. Figure 3.2 shows the normalization of the while loop from the example program.

3.1.2 Instrumenting Transformation

Based on the normalized representation of the GAST or on the bytecode, the instrumenting transformation can be done. For the GAST, this is implemented by a graph transformation, and for the bytecode, it is done by inserting additional statements. Our goal is to gather the information necessary to reconstruct a Dynamic Object Process Graph – the control flow graph from the perspective of a given object. Therefore, we need to instrument all the constructs that influence control flow: conditional and unconditional jumps, function calls, and returns. Also, we need information about lifetimes of objects and about the points where they are accessed.

Event	Arguments	Description
`create`	obj_id	creation of new object
`destroy`	obj_id	object's lifetime ends
`read`	obj_id	read access to an object's memory
`write`	obj_id	write access to an object's memory
`branch_true`		true branch of a decision
`branch_false`		false branch of a decision
`label`	[name]	real or artificial label
`call`	func_id	call site, calls a routine
`entry`	func_id	routine entry
`return`	func_id	return from a routine
`exceptional_return`	func_id	uncaught exception is rethrown
`exception`		invocation of exception handler
`new_thread`		new thread is created

Table 3.1: Overview of all traced event types, along with their arguments and meaning.

Table 3.1 shows a summary of event types. Apart from one of these types, all the inserted tracing statements require a unique location identifier (a label that corresponds to a source code location). An additional argument can either be the identifier of the object (usually its address) or the name of the function or label. The following instrumentation locations are used to create trace events, which means that events of different types are generated:

- **Begin of object lifetime:** `create` marks the begin of object lifetime. This leads to a `create` node in the Object Process Graph. Different memory allocation functions – such as `malloc` and `calloc` – are instrumented with a create logging, as well as the points where local variables are declared. A global variable's lifetime begins when the program is started and ends with the termination of the program.

- **End of object lifetime:** `destroy` marks the corresponding end of lifetime of a heap variable, if it is explicitly destroyed in the code. It leads to a `destroy` node in the OPG. For local variables, this is when the variable is removed from the stack.

- **Object access:** `read`/`write` trace object memory or attribute access (see discussion below).

- **Routine calls:** `call`/`entry`/`return` trace routine calls at the call site, at the called routine's entry point, and at each return.

- **Atomic method call:** Calls of atomic methods (that is, the routines that belong to the object) are handled as normal calls with an access to their respective memory area first, because they may be used in different contexts: for objects of one class, the call may be atomic, but for others, it is probably not. Atomic method calls are reduced in a later step.

- **Loops:** `labels` are needed to identify locations in the program that are passed several times.
- **Gotos:** A `label` is inserted before the goto, and all `label`s are logged as well.
- **Conditional control flow:** `branch_true`/`branch_false` mark conditional branches. The information is logged when one of the two possible branches is entered. Branches which belong to the same decision are marked by a common base identifier.
- **Merge of control flow:** Additional `labels` are inserted at merge of control flow.
- **Exceptions:** The entry of every exception handler is marked with an `exception` event. Additionally, a try-catch block is added around each block that can potentially throw an exception. The handler logs an `exceptional_return` and rethrows the exception that was caught. Exception handling is discussed in more detail below.
- **Multithreading:** Creation of a new thread is logged as a `new_thread` event.

Calls are logged before and after each call, using the same location identifier. This simplifies subsequent graph construction. Additional location identifiers are inserted at every merge of control flow and before jumps. This is necessary for correct graph construction when different objects are accessed, which sometimes are relevant and sometimes are not.

Parameter passing does not need to be logged, because we are interested in them only when they are actually used, and usage is logged when read/write occurs. Explicit `goto` target logging is also not needed, since the statement that is executed after the `goto` is the logging of the target label.

Read/Write Instrumentation

An interesting part is the instrumentation of reads and writes. In C, we have to deal with pointers, pointers to pointers, and the like. Therefore, we need to check exactly which objects are being touched in an expression. As an example, take a look at the following expression (with `int **x`):

```
(**x)++;
```

This expression leads to a read of object `x` to get the address of the address, a read of object `*x` to get the address, a read of object `**x` to get the old value, the addition of 1, and a write of the resulting value to object `**x`. Therefore, three different objects are involved in this expression: two with one read operation each, and one with a read and a write operation. Each of these operations must be logged with the respective address. Therefore, the instrumented code must look like this:

```
log_read(&x), log_read(x), log_read(*x), log_write(*x), (**x)++;
```

In languages like C, pointers can be manipulated to point to any desired memory location. Any pointer may potentially point anywhere. Therefore, *all* read/write accesses through pointers are traced, no matter of what type the pointer is. This has the consequence that the trace covers object accesses for *all* objects that occur in a program run: all objects are traced at the same time.

Handling of Arrays and Records

An array or a record can be regarded as a single object, or each of its elements can be regarded as an object. The dynamic analysis can trace at both levels, that is, it is capable of tracing individual elements of an array or record as well as summarizing the individual accesses as a partial read or write to the composite structure as a whole. The simpler case is individual element tracking, and it requires additional effort to track composite objects. In contrast to that, static pointer analysis typically does not differentiate between individual elements of an array because the distinction can be made statically only in trivial cases. This is another advantage of dynamic analysis.

To reflect whole composite objects in the dynamic analysis, objects are identified by their base address. This means when some operation reads or writes to an address that is in the address range of a known object, the address that is logged by the instrumented program is the base address of the object. The instrumented program keeps a mapping from address ranges to base addresses in order to be able to resolve this. For heap variables, the size is given as a parameter to the `malloc` function, and for local and global variables, `sizeof` delivers the variable's size based on the information about the type. In the remainder of this thesis, we always regard composite objects, not individual elements, because these are more interesting for the investigated applications (see Part III).

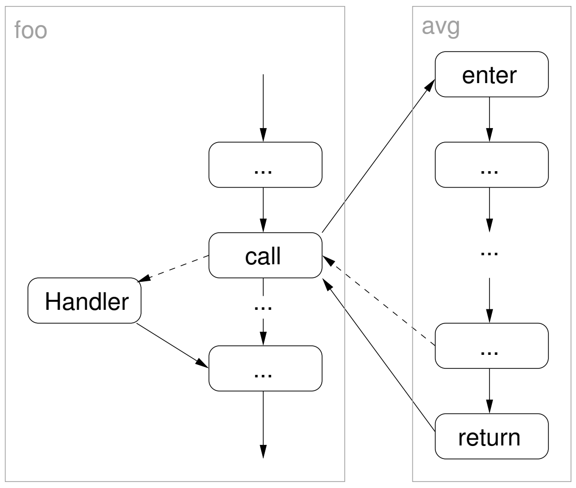
Exceptions

Languages like Java and C++ support **exception handling**, which leads to a much more complicated control flow graph, since exceptions may potentially be raised at many locations in the source code. Unchecked exceptions as in C++ or Java's `RuntimeException` may be raised without even being declared in the signature of a method. Related concepts that have a similar effect can also be found in C (`setjmp`). Exceptions lead to branches without an explicit condition and provide an alternative return mechanism for routines. Therefore, a routine in the control flow graph no longer has a single entry and single exit node, but may

```
void foo() {
  double[] arr = ...
  try {
    ...avg(arr)...
  } catch (Exception ex) {
    // Handler
    ...
  }
}

double avg(double[] val) {
  double sum = 0.0;
  ...
  double res = sum/val.length;
  ...
  return res;
}
```

(a) Code example.



(b) Corresponding DOPG.

Figure 3.3: Exception handling. Exceptional control flow is denoted by dashed lines. When `arr` has length zero, `avg` is left by an exceptional return edge, and the exception is caught by the exception handler within the calling method `foo`. Otherwise, `avg` returns normally.

additionally have any number of exceptional exit nodes. Since control flow must be reconstructed completely and correctly in the Dynamic Object Process Graphs, special care has to be taken for exceptions.

For this purpose, there are two special types of edges for representation in the Object Process Graph. These edges are taken when exceptions are raised. The edges of the first type `exception` lead to the corresponding exception handler. Edges of the other type `exceptional_return` are an alternative way of returning from a routine. Figure 3.3 shows an example where both edge types are involved.

To catch the complete control flow with exceptions, it is necessary to create an artificial exception handler that catches all exceptions around each method body. Also, existing exception handlers have to be instrumented. The occurrence of the exception is logged, and the exception is re-raised. This way, all exceptions and exceptional method exits can be traced. Overall, with this approach, exceptions can be dealt with in a straightforward way.

Instrumentation Example

Figure 3.4 shows the normalized and instrumented version of the `reverse` routine as it was introduced in Figure 2.1. Also, the instrumented version of a possible `push` implementation is shown. The `push` routine is important because it contains the relevant events (`reads` and `writes` on `*s1`). Although `push` is an atomic routine, it may as well be treated like a normal routine first; it will be replaced by an atomic

```
void reverse (Stack *from, Stack *to) {
  log_enter("reverse", "13");
  L1:  log_label("14.1");
  if (log_read(&from, "14.2"), !empty (from)) {
    log_branch_true("14.4T"), log_read(&to, "15.1"),
    log_read(&from, "15.2"), push (to, pop (from));
    log_label("16"), goto L1;
  }
  else log_branch_false("14.4F");
  log_label("17.1");
  log_return("reverse", "17.2");
}

void push(Stack *s, Item i) {
  log_enter("push", "90");
  log_read(&i, "91.1"), log_read(&s, "91.2"),
  log_read(&s->c, "91.3"), log_read(&s, "91.4"),
  log_read(&s->sp, "91.5"), log_read(&s, "91.6"),
  log_write(s->c, "91.7"), log_write(s, "91.8"),
  s->c[s->sp++] = i;
  log_return("push", "92");
}
```

Figure 3.4: Normalized and instrumented code example; see Figure 2.1 for the original source code of `reverse`. Begin-of-lifetime and call site logging have been omitted for better readability.

call later on. It is necessary to do so when objects of more than one class are traced in parallel (see discussion above).

Figure 3.6(a) shows an excerpt of a dynamic trace that has been generated by the instrumented program. Each row in the trace contains the type of event, the address or identifier of the concerned object, routine, or label, and the unique location identifier. Location identifiers in this example represent the line number of the original source code with an additional counter for disambiguation. (In the implementation, they are simply numbered serially.)

In this example, instrumentation makes the program much larger. More than 20 statements are inserted for tracing, while the original program consists of only very few statements. The exact amount depends on what we count as a statement. One can imagine that this overhead will also be noticable when the program is executed. The tracing and runtime overhead of this instrumentation is investigated in Chapter 4.

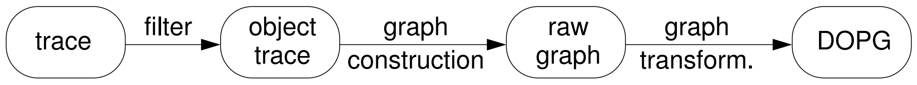


Figure 3.5: From trace to DOPG.

3.2 Trace to Dynamic Object Process Graph

After creating an instrumented version of the subject system, we can execute it and collect traces. These traces are then used as the basis for DOPG construction. Figure 3.5 shows an overview of the construction process. The trace is first filtered for a single object, which results in an object trace. From that, a "raw graph" is constructed, which is then transformed to the DOPG. In the following, the details of the individual steps are described.

3.2.1 Filtering

The traces that result from the described instrumentation contain information about *all* objects that occurred in a program run. Therefore, as a first step, a filter is applied that extracts the relevant information for only one object. The *object trace* is extracted from the complete trace. This is done by checking inside each routine invocation whether there is any read/write (or atomic method call) of the object. Reads and writes to other objects are removed. If there is no read or write left, the entire routine invocation is removed from the trace. This is done recursively. Figure 3.6(a) shows an example trace, where events that are filtered out are shown in gray.

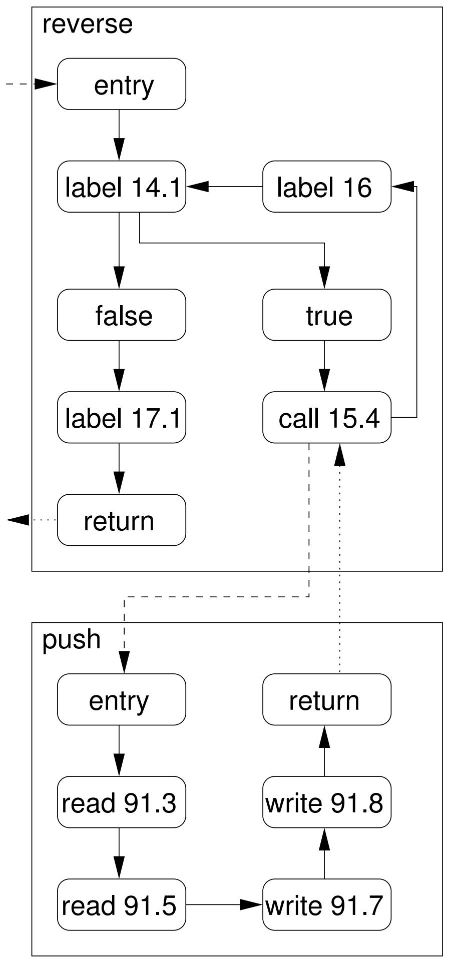
The filter could actually be moved to the instrumentation phase so that only those pieces of code are instrumented that deal with a certain type of expression, if objects of that type were to be investigated. Instrumenting more selectively would help to reduce the amount of data that is produced by the instrumented program and may avoid unacceptable performance degradation. However, this selective instrumentation could possibly reduce the accuracy and would question the advantages of dynamic analysis with respect to aliases and pointer arithmetics. The instrumentation would have to rely on type information, and there are many ways in C to circumvent the type system. The resulting insufficiencies of the necessary static analysis would then lead to incomplete dynamic traces. In contrast, it is safe to regard all parts of the program as potentially relevant for an object trace, so this is what we stick to.

3.2.2 Raw Graph Construction

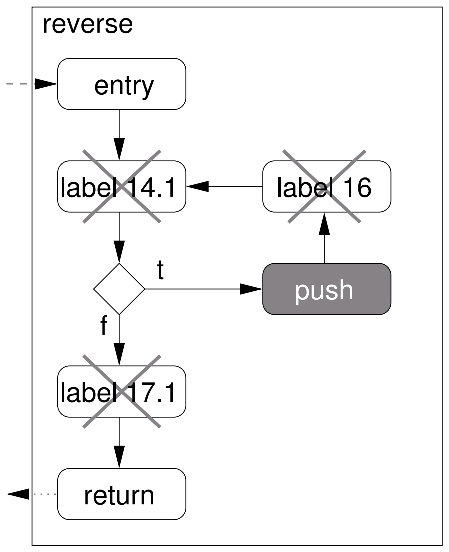
From the resulting object trace, a graph is generated. This graph is not yet an OPG: it is a simple untyped, attributed, directed graph. Its schema is shown in Figure 3.7. We call this intermediate graph a **raw graph**. Every location identifier that occurs in the trace leads to a `Node` in the raw graph. Consequently, every

```
enter  reverse  13
label           14.1
read   bfcbbb70 14.2
call            14.3
enter  empty
...
return empty
call            14.3
branch-true     14.4T
read   bfcbbb74 15.1
read   bfcbbb70 15.2
call            15.3
enter  pop
...
return pop
call            15.3
call            15.4
enter  push     90
read   bfcbbb64 91.1
read   bfcbbb60 91.2
read   0804c068 91.3
read   bfcbbb60 91.4
read   0804c068 91.5
read   bfcbbb60 91.6
write  0804c068 91.7
write  0804c068 91.8
return push     92
call            15.4
label           16
label           14.1
read   bfcbbb70 14.2
call            14.3
enter  empty
...
return empty
call            14.3
branch-false    14.4F
label           17.1
return reverse  17.2
```

(a) Trace excerpt. Irrelevant events are shown in gray.



(b) Raw object process subgraph.



(c) Close to completion.

Figure 3.6: Example trace excerpt for the call of routine `reverse` from Figure 2.1 and intermediate Object Process Graphs for object `*s1`.

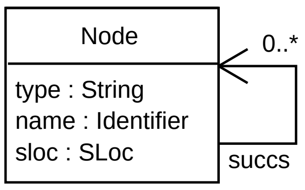


Figure 3.7: Raw graph meta-model.

`Node` corresponds to one location in the instrumented code. A `Node` has several attributes. The `type` attribute is the type of event that has been recorded, the optional `name` is the name of a routine. A `Node` also knows its source location (`sloc`). Edges are inserted as they are induced by the order of location identifiers. The algorithm in Figure 3.8 shows the basic idea. For the ease of understanding, the creation of CallPairNodes (which do not directly result from an event) is not shown in the algorithm; it is realized by keeping track of the call stack and identifying them by the combination of their Call- and EntryNode. The resulting graph contains all necessary information for the transformation to the DOPG, but also a lot of additional information which is removed in the next step.

```
Input  : an object trace
Output: the corresponding raw graph
        $G = (V, E, s, t, l, a)$
        (see Section 2.2)
create new node $st$
$a(st, \texttt{type}) := \texttt{Start};$
$V := \{st\};\ E := \emptyset;\ p := st$
foreach object trace event $t$ do
    $n :=$ get_or_create_node $(t)$
    $V := V \cup \{n\}$
    if $\nexists e \in E:\ s(e) = p \wedge t(e) = n$ then
        create new edge $e$
        $s(e) := p;\ t(e) := n$
        $E := E \cup \{e\}$
    $p := n$
```

Figure 3.8: Raw graph construction; `get_or_create_node`(t) yields a graph node for event t which is based on t's location identifier. The class l and attributes a are set accordingly. The corresponding node is returned if it exists, or a new node for t is created otherwise.

The tracing approach that has been described so far creates raw graphs offline. This means that during program execution, the events are just written to a trace file, and only after the program has terminated, further processing is performed. Yet, the graphs can as well be created online to avoid the generation of huge trace files. In particular, construction of the raw graph can be done online, and the graph transformations can be done after the program has terminated. This approach is described in detail in Chapter 4. This way, also different raw graphs from different program runs can be combined before applying the transformations.

Merging multiple graphs that share the same static allocation point is also possible with the offline approach: you simply start raw graph construction with an existing graph instead of the empty graph. This merging step may help to get a more complete Dynamic Object Process Graph and is necessary for comparison to static OPG extraction, which only delivers this kind of merged graphs.

3.2.3 Graph Transformation

A sequence of simplifying transformations is then applied to the raw graph. The nodes and edges are successively transformed to nodes and edges of the OPG meta-model. For this transformation, a combined meta-model is used: the `Node` class from the OPG meta-model is temporarily replaced by the raw graph `Node`, including its attributes and association.

In the following, I use the *single-pushout* graph transformation approach to describe the necessary transformations. A short introduction to the necessary graph transformation basics and definitions can be found in Appendix B. Basically, the single-pushout approach finds a subgraph with a structure as shown on the left hand side of a rule and replaces it with the right hand side. The replacement is done using the *gluing graph* that defines reference points (shown in gray in the rules). All dangling edges are simply removed.[2] Additionally, we use an extension for expressing negative application conditions as introduced by Habel et al. [66]: nodes and edges inside dotted borders demark application conditions. Such nodes and edges must exist for the transformation to be applicable, but are not relevant for the transformation step. If the dotted area is crossed out, such nodes and edges must *not* exist.

The following rules are applied:

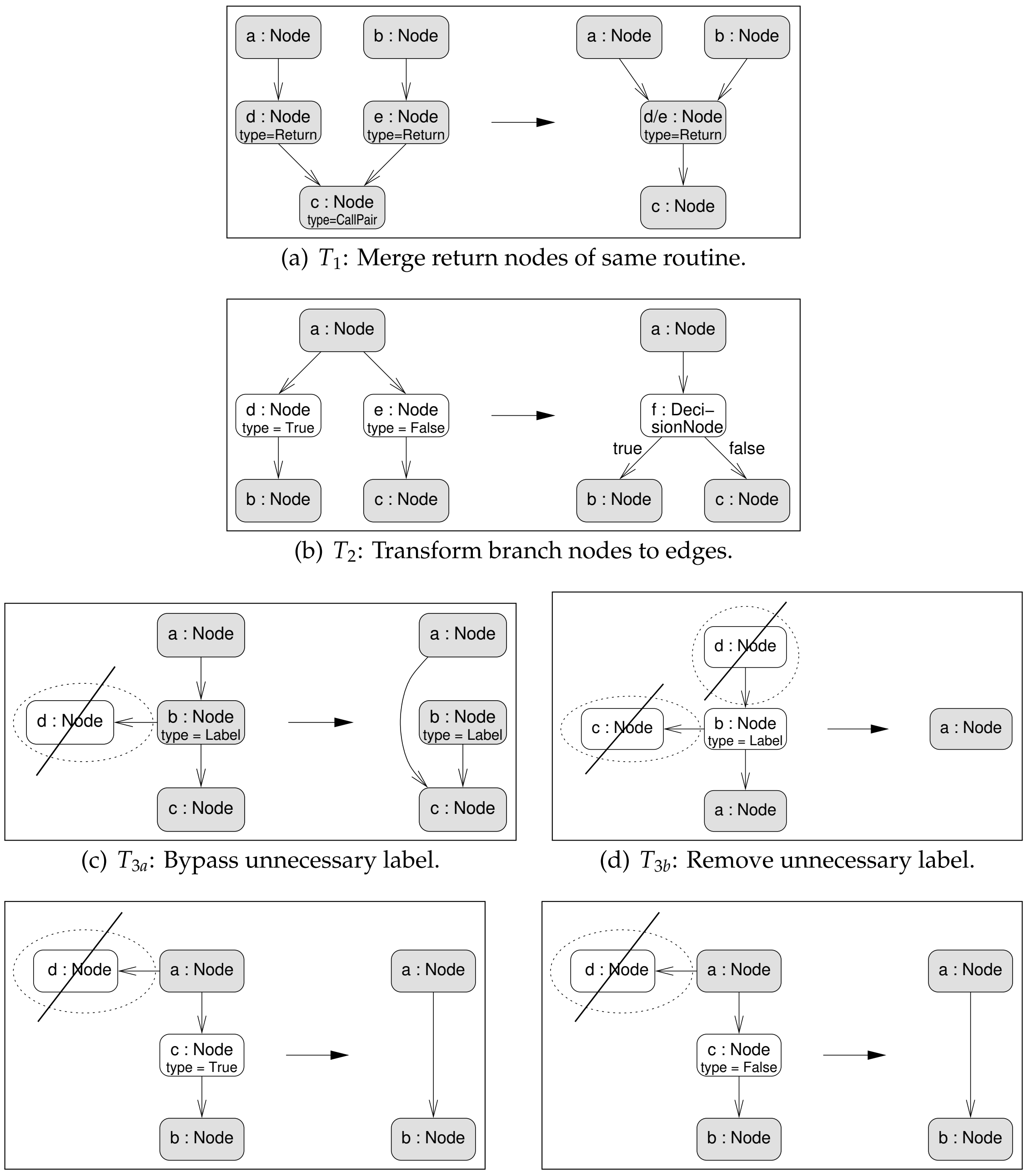
1. T_1 (Figure 3.9(a)): merge all `return` nodes for the same routine. (This transformation may be left out if returns from different source locations shall be distinguished.)

2. T_2 (Figure 3.9(b)): replace a pair of `branch_true` and `branch_false` *nodes* with a true and a false *edge* and insert a DecisionNode.

3. $T_{3a/b}$ (Figures 3.9(c) and 3.9(d)): remove unnecessary label nodes, that is, label nodes that have only one successor. The graph transformation is performed in two steps to cover the case that a label node has more than one incoming edge. This is similar for some others of these rules.

4. $T_{4a/b}$ (Figures 3.9(e) and 3.9(f)): remove branch nodes that are the only successor of a node. We cannot know in advance if both possible values

[2]This is different in the double-pushout approach, where dangling edges are not allowed to occur.



(a) T_1: Merge return nodes of same routine.

(b) T_2: Transform branch nodes to edges.

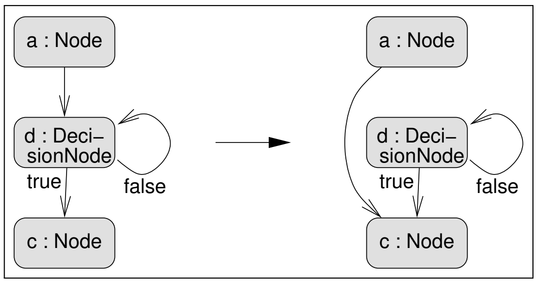
(c) T_{3a}: Bypass unnecessary label.

(d) T_{3b}: Remove unnecessary label.

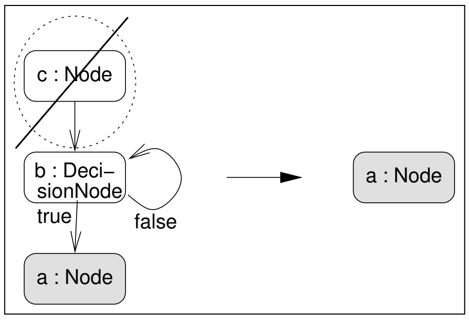
(e) T_{4a}: Remove single branch node.

(f) T_{4b}: Remove single branch node.

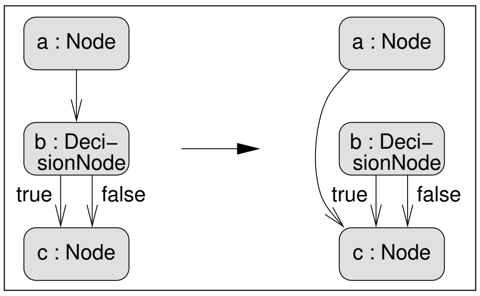
Figure 3.9: Transformations from Raw Graph to DOPG.



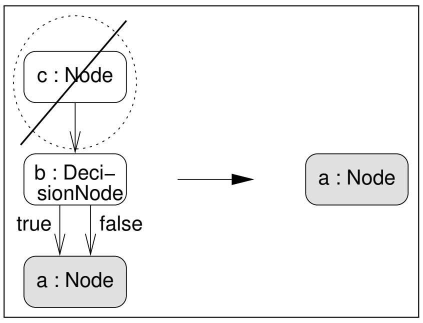
(a) T_{5a}: Bypass unnecessary decision node.



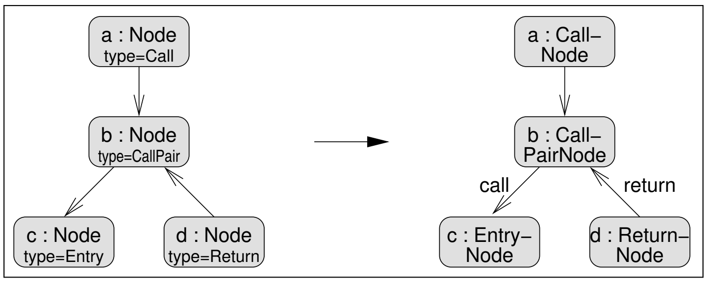
(b) T_{5b}: Remove unnecessary decision node.



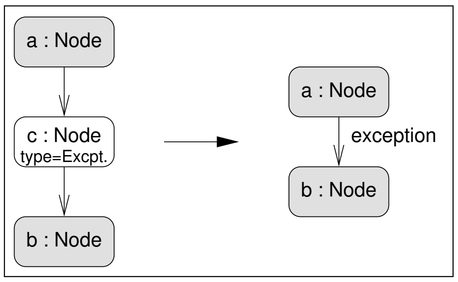
(c) T_{6a}: Bypass unnecessary decision node.



(d) T_{6b}: Remove unnecessary decision node.



(e) T_8: Transform call/entry/return. Note that only labels are changed – the nodes and edges remain the same.



(f) T_9: Transform exception node.

Figure 3.10: Transformations from Raw Graph to DOPG (continued).

of a decision will be encountered, therefore such constructs appear in the raw graph.

5. $T_{5a/b}$ (Figures 3.10(a) and 3.10(b)): remove local loops, that is, edges that lead from a decision node to itself. Such edges may appear as a result from other transformations. Note that the corresponding rules with exchanged *false*/*true* edge labels also have to be considered.

6. $T_{6a/b}$ (Figures 3.10(c) and 3.10(d)): remove decision nodes for which both successors are identical.

7. T_7 (Figure 3.11): remove subgraphs that are not relevant for control flow. Control dependency analysis is performed to find out which decision nodes are really relevant. The algorithm is provided in Figure 3.11 and described in detail below. Note that $T_{5a/b}$ and $T_{6a/b}$ are just special cases of this rule, which have been added as an optimization for frequently occuring constructs.

8. T_8 (Figure 3.10(e)): replace a combination of call/entry/return type raw nodes by the corresponding OPG node and edge types. This means in fact just relabelling the nodes and edges.

9. T_9 (Figure 3.10(f) shows a representative): transform exception nodes to exception edges. The corresponding transformation is done for raw graph nodes of type "new_thread". Each remaining raw graph node is transformed to the OPG node of the specified type.

10. T_{10}: Remove bodies of atomic methods. This transformation removes all nodes between EntryNode and ReturnNode of an atomic method. The remaining CallNode is then transformed to an AtomicCallNode.

The order of the rules is partly important. For example, return nodes have to be unified before the call/entry/return construct can be replaced, and the bypassing transformation T_{3a} has to be done before the removing transformation T_{3b} is applicable. Therefore, the order has to be obeyed.

The repeated application of these transformations terminates because in every transformation, either nodes or edges are removed, or edges are redirected in a way that allows elimination of nodes in the next step, or nodes or edges are relabelled (with reversal not possible). The completeness of the rules can be concluded from the structure of the raw graph. By construction, this structure is in fact more restricted than the raw graph's meta-model may suggest. For example, only "True" and "False" nodes may occur in a branch. Such restrictions eliminate a lot of cases that otherwise would have to be considered in the rules.

After termination, we remove those nodes and edges that are not reachable from the allocation point – in particular, the path from program start to the allocation point. Parts of the graph that do not have a path to an operation on the object are removed as well (this is called "Object Process Graph slicing" by Eisenbarth et al. [46]). This way, the graph is further reduced to the really relevant information. The result is a Dynamic Object Process Graph.

```
foreach routine $r$, represented by a subgraph $H \subseteq G$ do
    $C := \{n \in V_H \mid a(n, \texttt{type}) = \texttt{"Decision"}\}$
    while $\exists c \in C, n \in V_H \setminus C$: ($n$ is control dependent of $c$) do
        $C := C \setminus c$
    $P := \{n \in V_H \setminus C \mid \exists e \in E_H : \ s(e) = n, \ t(e) \in C\}$
    $Q := \{n \in V_H \setminus C \mid \exists e \in E_H : \ s(e) \in C, \ t(e) = n\}$
    foreach $(p, q) \in P \times Q$ do
        $V_R := C \cup \{p, q\}$
        while $\exists e \in E_H : \ (s(e) \notin V_R \wedge t(e) \in V_R \wedge t(e) \neq q) \vee$
                             $(s(e) \in V_R \wedge t(e) \notin V_R \wedge s(e) \neq p)$ do
            $V_R := V_R \setminus \{s(e), t(e)\}$
        $E_R := \{e \in E_H \mid s(e) \in V_R \wedge t(e) \in V_R\}$
        if $E_R \neq \emptyset$ then
            perform the graph transformation:
            $(V_R, E_R, \ldots) \supseteq (\{p, q\}, \emptyset, \ldots) \subseteq (\{p, q\}, \ \{e\}, \ \{(e \mapsto p)\}, \ \{(e \mapsto q), \ldots\})$
```

Figure 3.11: T_7: Elimination of nodes which are irrelevant for control flow.

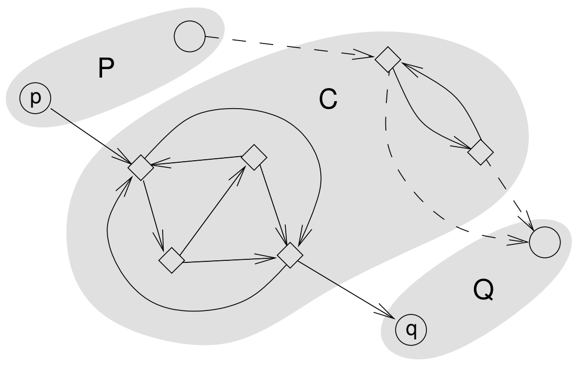


Figure 3.12: Illustration of the Algorithm from Figure 3.11. P, Q and C have been calculated, and (p, q) has been selected. The dashed arrows demark edges that satisfy the condition for e in line 9, because they cross the boundaries of C (i. e., V_R really).

Removing control-flow irrelevant subgraphs

The raw graph may contain nodes that are completely irrelevant for control flow. In particular, it may contain subgraphs that consist of decision nodes only and have only one entry and one exit node. Two examples for such subgraphs are shown in Figure 3.12. The left subgraph (within C) consists of four decision nodes, the right one of two such nodes. Each of these subgraphs can be removed from the overall graph and replaced by a single edge without loss of information.

The algorithm for identifying subgraphs of this kind is shown in Figure 3.11. It is based on control dependencies, which can be calculated based on post-dominance information [53]. The algorithm starts by identifying all decision nodes on which no other non-decision type node is control dependent. It puts those nodes into set C. It then collects all nodes that either lead into one of the nodes in C (in set P) or that are reachable from one of C's nodes (set Q). Then, for each pair $(p, q) \in P \times Q$, it collects all nodes and edges that connect p and q by successively removing nodes that are connected to other nodes outside $C \cup \{p, q\}$ (in V_R and E_R). If any edges are left, this means that a construct of the desired type has been found, and it is replaced by a single edge from p to q.

In the example from Figure 3.12, the sets P, Q, and C have been identified. Nodes p and q have been chosen, so the next step is to eliminate those nodes that are not connected to p or q. In this case, the nodes of the right subgraph in C will be removed from C. The 4 nodes and 9 edges of the left subgraph will then be removed and replaced by a single edge from p to q. The other subgraph of C will be replaced in a subsequent iteration.

Tracing example (continued)

Figure 3.6(b) shows the raw object process subgraph as reconstructed from the trace in Figure 3.6(a) using the algorithm in Figure 3.8. Figure 3.6(c) shows the next step: the `true` and `false` nodes have been transformed to edges, and the call of the atomic `push` routine has been replaced by an `atomic_call`. Then, unnecessary `label` nodes are removed. The result of application of this method on the complete example is the final Object Process Graph as shown in Figure 3.13(a). This graph is the same that would be derived by static analysis. However, we only get this result from dynamic analysis if `*s2` contains at least one element in our program runs.

Figure 3.13(b) shows a possible DOPG for the trace with empty `*s2`. The result depends on the behavior of routine `pop` when the stack is empty. When `pop` raises some kind of exception, the program may terminate immediately, leaving out the `empty` call. This small example illustrates how the completeness of the dynamically created Object Process Graph depends on the code coverage of the used test cases.

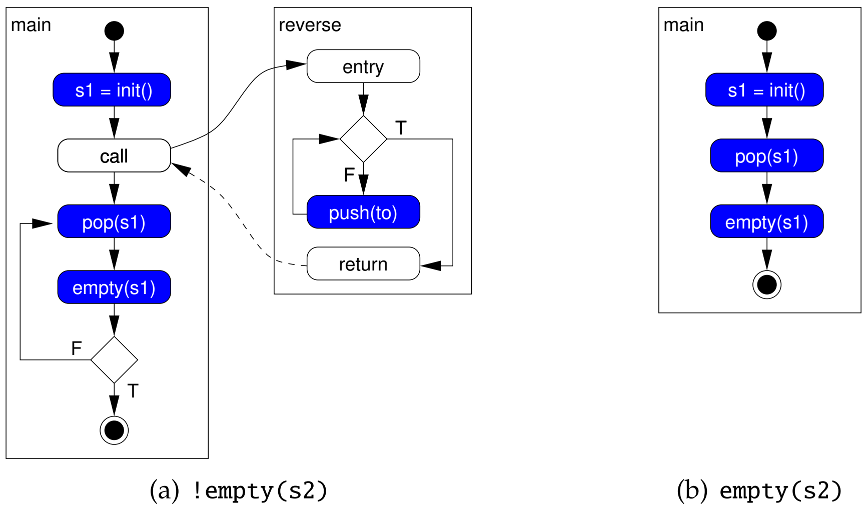


Figure 3.13: Final DOPG for *s1 for the example. Only when *s2 contains at least one element, the result is identical to the statically extracted Object Process Graph. Otherwise, the call of reverse is not relevant.

3.3 Additional Considerations

The description of dynamic creation of Object Process Graphs in the previous sections was mostly based on the C language, although the concept of exceptions (from C++ and Java) and Java bytecode instrumentation was covered as well. However, when analyzing other languages, special language features, or special classes of applications, additional issues have to be considered.

Object-oriented language features. Object-oriented languages add classes, objects, and methods. Method and constructor calls are already covered by the described instrumentation technique and do not make any extension necessary. With concepts such as overloading, we just have to take care about unique method signatures (*mangled names*). Since method entries are noticed, this tells us which method has really been entered with *dynamic binding*. When investigating objects of a certain class that are allocated at different points, the concept of *inheritance* gives us the additional choice of also taking instances of its subclasses into account or not. This should usually be done because subclasses are just specializations of the investigated class: they have to adhere to Liskov's substitution principle [119], which states that any instance of a given class (or type) can be replaced with an instance of one of its subclasses (subtypes) without changing the program's behavior.

Multithreading. In contrast to C, multithreading is very easy to do in Java. Also, even without explicitly starting new threads, any Java program has several threads running in parallel. Apart from the main thread, there is at least the garbage collector's thread, and when there is a GUI, there also is an AWT thread. Therefore, a dynamic analysis on Java applications *must* take care of multithreading. This is realized by writing one trace file per thread, which turns out to be faster than additionally writing a thread identifier into a single trace file. The inter-thread event order does not need to be considered, since the exact timing information gets lost in the OPG representation anyway. Tracing must be synchronized to work correctly, which additionally slows it down. In the OPG, the creation of a new thread is represented by *inter-thread edges* (`new_thread` event). These edges lead from the node that starts the new thread to the first node (*thread-root node*) that is executed by the new thread. This idea is based on the *interthread control flow graph* by Choi et al. [28].

C++ supports **templates**. This gives us the choice of instrumenting the template only (once) or instrumenting every *instance* of a template separately. Since control flow is statically identical in every instance, it would be reasonable to instrument only the template. Templates are an interesting subject for analysis, since reusable data structures should adhere a certain protocol as well. However, C++ programs will not be investigated in more detail in this thesis, because the C++ GAST was also not complete (specially with respect to templates) at the time these studies were performed.

System libraries/classes. Library functions cannot easily be instrumented for C code, and the same is true for Java API classes. Parts of the API appear to be protected from being modified. The Java Virtual Machine produces all kinds of error messages when trying to modify certain classes. Therefore, I did not instrument system libraries or classes belonging to the standard Java runtime environment at all. Consequences arising from this are discussed in the next paragraph ("Callbacks").

Callbacks. Callbacks are very important for applications that are based on a graphical user interface (GUI). For example, most event handling in Java's GUI framework (AWT/Swing) is done through callbacks. These are usually realized by providing an object of a class that implements a given interface. This leads to calls of application code from system routines – and since system routines cannot be completely traced, this causes incomplete and potentially misleading traces. Therefore, analysis must be robust against such effects. To achieve this, an artificial `call` node is created whenever there is an `entry` node without a `call` node. This correction can be integrated into the filtering step (see Section 3.2.1).

Special Java Issues

Instrumentation of Java byte code raises a few additional issues.

Object addresses. Addresses of objects are not accessible in Java. This raises the question of how to get a unique identifier of an object, which is necessary for a full textual trace. Fortunately, it turns out that using the `System.identityHashCode()` function delivers sufficiently disjunctive values.

Accessing new objects. New objects may only be accessed after the constructor of the root class `java.lang.Object` has been invoked. Before that, it is not possible to access the object. Therefore, we must distinguish between the point of creation of the object and the point where the object is accessible for the first time. New objects can be identified only at the latter point.

Reflection. In Java, classes and their methods can be accessed dynamically. *Reflection* allows to access a class based on its name, create instances of it, and call its methods. Since reflection is widely used, this technique also has to be considered. Byte code instrumentation can recognize calls to the reflection API and handle them as if the destination object was directly used.

3.4 Summary

This chapter introduced a graph transformation based technique for extracting OPGs by means of dynamic analysis. The instrumentation is done on GAST level by graph transformations. An alternative is to instrument on byte code level. The traces that result from executing the instrumented program are first transformed to a raw graph, which is then converted to a DOPG by a sequence of graph transformations. The approach is basically language-independent and was implemented for C and Java. It considers techniques such as multithreading and exception handling.

With this approach, we can generate Dynamic Object Process Graphs which describe a set of dynamic traces for a given program. Dynamic trace extraction is an enabling technique. Similar to program slicing [208], dynamic trace extraction slices the control flow graph so that only those statements are kept that are relevant for a particular aspect. In program slicing, relevance means control and data dependency. For dynamic trace extraction, the flow of operations and their conditions are of relevance.

However, the approach has one problem that every dynamic analysis faces: it generates huge trace files. Due to the necessary intense instrumentation, we also have to suspect a considerable runtime overhead. This issue is addressed in the next chapter.

Chapter 4

Online DOPG Extraction

The dynamic OPG extraction algorithm as described in the previous chapter showed promising results in a first prototype implementation (the corresponding case studies are presented in Chapter 6). However, it turned out to be too slow to be applicable in practice. Therefore, the runtime overhead and trace size problem has to be attacked in order to make the approach usable. In this chapter, I introduce optimizations to the technique and an extension which attacks this problem and even enables additional applications. A case study illustrates that this technique is applicable even for larger and interactive systems.

4.1 Problem Characterization

The basic dynamic OPG extraction approach from the last chapter involves several resource intensive steps:

- Traces become very large quickly, because a lot of information has to be recorded. For example, for each condition node of the control flow graph that is passed during execution, we have to remember which branch has been taken. Tracing produces hundreds of megabytes of data within seconds. This implies a high I/O overhead.

- Individual object traces have to be extracted, and only then, the Dynamic Object Process Graphs can be constructed from that. In this step, the trace file has to be read again and again (for each individual object). Due to the size of the trace, this also takes quite some time (see Section 4.3).

- During the construction of the DOPG for each object, the object traces have to be read again. They contain a lot of redundant information, caused by cycles and repeated function calls.

These points restrict the applicability of DOPG extraction in practice, because they slow down the entire application to a level which is not acceptable in many cases. Therefore, a way to avoid them is needed.

Optimizations

A possible solution to the amount-of-data problem is instrumenting only those locations that we know can possibly be relevant for a given object. However, as discussed in Chapter 3, this would have to be decided statically, which would partly eliminate the advantages of dynamic analysis.

Another solution is to compress the trace data. Reiss et al. [157] list a lot of possibilities to compress a dynamic trace. The following practices for compressing DOPG traces are used:

- Use the shortest possible identifiers for locations,
- use numbers to identify method signatures, since those can be quite long for Java programs (including package and parameters),
- use single characters to distinguish the node type.

These optimizations reduce the average amount of data required to store an event to 12 bytes. Compared to the original trace file format (see Figure 3.6(a), the runtime reduction is only about 14% for the case studies from Section 4.3, while trace files could be compressed to 15% of their original size. The amount of data could be reduced even further by using a binary format or applying online compression to the data. However, this would increase the runtime overhead and reduce readability of the trace file.

4.2 Online Construction

A better approach for tackling the trace size problem is to limit the trace to objects of the relevant class dynamically. The **relevant class** is the class whose instances the analyst is interested in, that is, for whose instances he wants to extract DOPGs. The relevant class is a characterization of all the objects of interest.

After returning from a called routine, it is known if this invocation of the routine was relevant for the regarded object or not. If the call was relevant, the trace of the called routine has to be remembered, else it can be discarded. The problem with this approach is that large amounts of trace data – in the extreme, the entire trace – have to be remembered before it can be decided if that data is needed or not, which can easily fill up all memory.

Therefore, this approach has to be complemented by a different representation of trace data. Loops have to be represented in an efficient way. Since we are interested in the *raw graphs* as described in Section 3.2.2 anyway, why not directly construct those graphs online instead of remembering all the trace events? The number of nodes in this graph is limited by the number of nodes in the CFG, while the number of trace events is not limited. Therefore, it is possible to remember executed parts of the graph temporarily. Only when the program terminates, the resulting raw graphs need to be written to file, which eliminates the I/O overhead

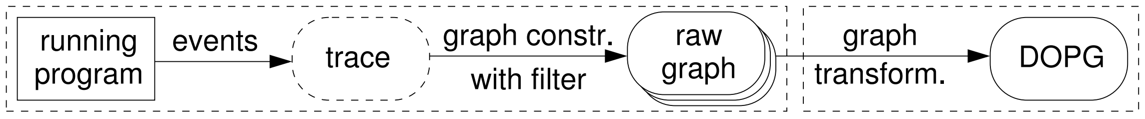


Figure 4.1: Online construction of a Dynamic Object Process Graph. In this approach, the trace is not explicitly represented. Dashed rectangles represent a process.

during execution. By allowing to just record the graphs for objects of certain classes, the number of graphs that must be constructed simultaneously is limited as well, which prevents explosion of additionally required memory. However, this approach is probably not adequate when lots of instances of a class are to be traced.

The algorithm basically works as follows:

- As events occur, construct the corresponding graph, as described in Figure 3.8. When a routine is left, eliminate that call from the graph again. This way, always keep only all routines of the current call stack in this graph. It represents the path from the main routine to the current node. Let us call this graph the "current stack graph". It will be used to indicate the current program location when a new object is created.

- Whenever a new object of a relevant class is instantiated, create a copy of the current stack graph for this object.

- Apply each other event (that is, each event other than instantiation) to the current stack graph and to all copies. This means that an edge is inserted between the previously visited node and the node that corresponds to the event's unique source location. Events that relate to one particular relevant object are only added to this object's graph.

  For the copies, remove routine calls (the same way as for the construction of the current stack graph) only if the routine invocation is not relevant for the object of this copy. This means there must be at least one relevant node within the called routine in order to keep it. As construction goes on, this will discriminate the different graphs from each other.

The overall process is sketched in Figure 4.1. The trace itself does not have an explicit representation. Events are directly integrated into the raw graphs (one for each object of interest). The basic instrumentation and the transformation from raw graph to Dynamic Object Process Graph remains the same as in the basic approach from Chapter 3.

Data structures. To allow an efficient implementation of this idea, I decided to have just one graph that contains all nodes and edges that were visited in the entire program run. The raw graphs for our objects and the current stack graph are then just views (that is, subgraphs, called *GraphViews*) of the complete

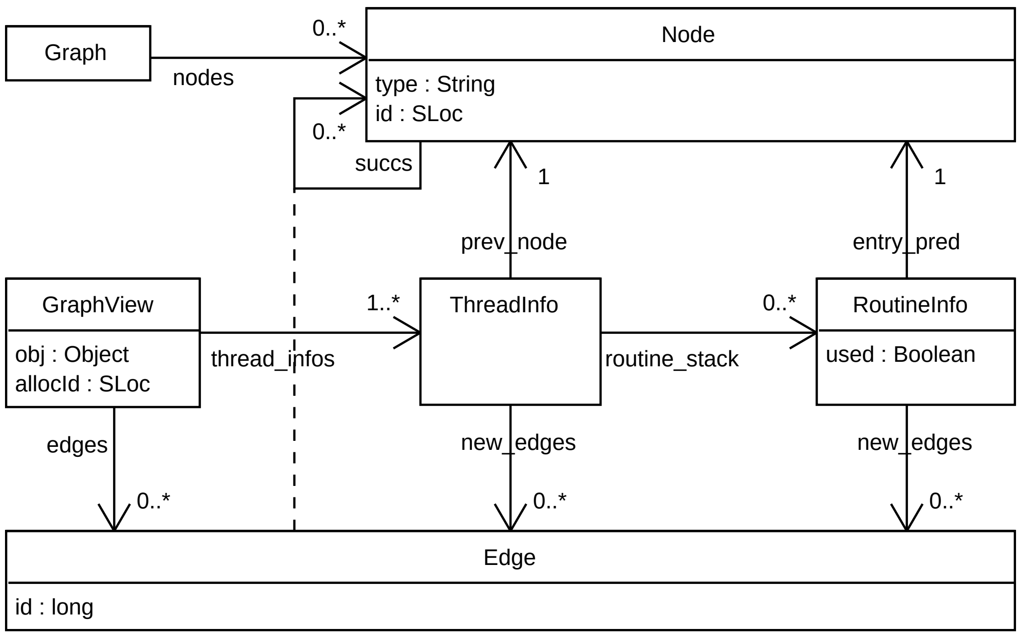


Figure 4.2: Raw graph online construction meta-model.

graph. There will not be any nodes without edges, so it is sufficient to keep track of the edges only. This can be done very efficiently by using running edge numbers and a bit set implementation. Multithreading must be considered in the data structures because multiple threads may be constructing the same graph at different locations at the same time. The overall data structures used are shown in Figure 4.2 and explained in Table 4.1. Additionally, a mapping from objects to GraphViews has to be kept.

Algorithm. With these data structures, the algorithm in Figures 4.3 and 4.4 can be applied for raw graph construction. Figure 4.3 shows the main loop for event processing. New GraphViews, based on the current stack GraphView, are created as new relevant objects (that is, instances of the relevant class) are created. For each unique instrumentation location `id` that occurs in an event, a node is created, and edges corresponding to their sequence are added to the different GraphViews. Special care must be taken of operations on relevant objects, which are only applied to the GraphView that belongs to that object.

Figure 4.4 describes in detail how processing a node updates a GraphView. Note that this processing may lead to preliminary or final addition of nodes and edges, as well as removal of preliminarily added elements. The latter is the case when a return event has been received, and the routine invocation turns out to be irrelevant. To make sure that the necessary information is available, a new RoutineInfo is created on routine entry. This new object collects the necessary information for the entered routine. Edges that are not yet visible in this GraphView are preliminarily added, until it is known whether the routine invocation is relevant or not. Only when it turns out to be relevant, those nodes are permanently added to the GraphView. While RoutineInfo.`new_edges` keeps track of edges that

Class	Meaning
Graph	Raw graph containing one node for each instrumented location that has been visited in this program run. Edges are directly kept within the nodes.
Node	A raw graph node, along with a type (see 3.1.2), a unique source location identifier (id), and edges to the node's successors (succs).
Edge	A conceptual association class. In our implementation, an edge is not represented by an instance, but by a unique id. This means that objects do not hold references to instances of this class, but rather the corresponding id.
GraphView	This class represents a subgraph of the Graph. A subgraph is defined by the set of contained edges – the attached nodes implicitly belong to the subgraph as well. There is one GraphView instance for each object which is being traced. Additional information about the current state of affairs is kept in thread_info for each thread separately.
ThreadInfo	There is one instance of this class for each thread. It holds information about the current call stack (routine_stack), along with information about all edges which have so far only been preliminarily added by this thread (new_edges). It also contains the node which was last visited by this thread (prev_node).
RoutineInfo	This class holds information about the currently executed routine. One such instance exists for each routine on the call stack (per thread). It remembers the call site (entry_pred, copied from prev_node) and the set of edges which have been preliminarily added to the graph within this routine (new_edges). The used flag becomes true when the function has been recognized to be relevant. When the routine is left and the flag still not set, the routine is not relevant for this invocation.

Table 4.1: Meta-model classes and attributes explained.

are not contained in the GraphView and that have not been preliminarily added in one of the calling routines, ThreadInfo.`new_edges` contains all edges that have been preliminarily added in any routine on the call stack. This allows a very fast check and update of these sets.

Figure 4.5 shows an example in the process of constructing a GraphView. On the left, edges have been preliminarily added to both `new_edges` sets for each routine (gray). Then, an operation on the GraphView's object is encountered (black circle). This leads to setting the `used` flag, which in turn has the effect that the RoutineInfo.`new_edges` are permanently (black) added to the GraphView when the routine is left (return node). Because the `used` flag is propagated up the call stack, all callers' edges will also be permanently added to the GraphView's `edges` set later. If there had not been a relevant node within the routine, all edges from RoutineInfo.`new_edges` would have been removed from the `edges` set again.

```
Start with an empty GraphView ("current stack GraphView") and an empty graph.
foreach incoming event do
    if event is the creation of an object of the relevant class then
        create a deep copy of the current stack GraphView for this object.
    n = get_or_create_node(event)
    if event is an operation on a relevant object (including creation of the object) then
        processNode(n, GraphView for this object)
    else
        foreach GraphView g do
            processNode(n, g)
```

Figure 4.3: Main algorithm.

4.3 Case Study: Tracing Overhead Online/Offline

In the following case study, the overhead that is imposed by DOPG instrumentation is measured, and this overhead is compared for online and offline construction of raw graphs. The raw graph is the common data structure that is produced by both approaches, so further processing is identical. The case study examines the instrumentation overhead in terms of running time and the number of trace events that occur within each program run. Also, the size of the resulting trace files is measured for the offline approach. The goal is to find out whether construction of Dynamic Object Process Graphs is feasible using these methods.

4.3.1 Subject Systems and Procedure

As subject systems, several Java programs of different size are investigated, tracing for potentially representative objects within typical use cases. Table 4.2 shows some size measures of the investigated systems.

ArgoUML[1] is a widely-used open-source UML modeling tool which supports all standard UML 1.4 diagrams. Graph models for the different diagrams must be a central concern for this tool. Therefore, as a first use case, the construction of a class diagram with `ClassDiagramGraphModel` as the relevant class is used. A class diagram for the observer pattern is drawn, consisting of four classes, one note, one aggregation, two inheritance relations, and six methods. The result is saved, and the application is quit. The second use case is the construction of a sequence diagram (relevant class: `SequenceDiagramGraphModel`). The constructed diagram consists of three actors and three synchronous interactions. The result is also

[1] `http://argouml.tigris.org/`

```
Input: node $n$ that shall be processed, GraphView $gv$ to be updated
if ThreadInfos contains a ThreadInfo for the current thread then
    use this ThreadInfo $ti$
else
    $ti$ = new ThreadInfo()
    add $ti$ to ThreadInfos
if $n$.type = "entry" then
    $ri$ := new RoutineInfo()
    $ri$.entry_pred := $ti$.prev_node
    $ti$.routine_stack.push ($ri$)
    $ri$.used := false
else
    $ri$ := $ti$.routine_stack.top()
Edge $e$ := ($ti$.prev_node, $n$)
if $e \notin gv$.edges and $e \notin ti$.new_edges then
    $ti$.new_edges := $ti$.new_edges $\cup\{e\}$
    $ri$.new_edges := $ri$.new_edges $\cup\{e\}$
if $n$.type = "return" or $n$.type = "exceptional_return" then
    $ri_{ex}$ := $ti$.routine_stack.pop()
    $ri$ := $ti$.routine_stack.top()
    $ti$.new_edges := $ti$.new_edges $\setminus$ $ri_{ex}$.new_edges
    if $ri_{ex}$.used then
        $gv$.edges := $gv$.edges $\cup$ $ri_{ex}$.new_edges
    else
        $n$ := $ri_{ex}$.entry_pred
    $ri$.used := $ri$.used or $ri_{ex}$.used
else if $n$.type = "operation" and operation relates to $gv$.obj then
    $ri$.used := true
$ti$.prev_node := $n$
```

Figure 4.4: processNode: Updating a GraphView.

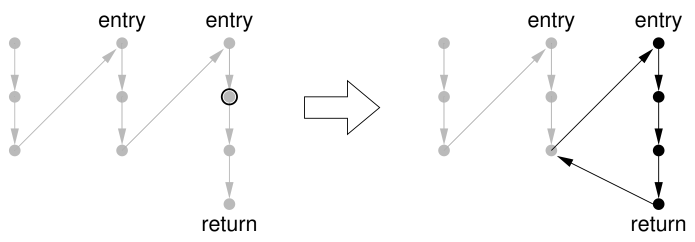


Figure 4.5: Example: A relevant routine invocation leads to permanent addition of edges to the GraphView.

	ArgoUML	J	JHotDraw	ANTLR
LOC [K]	264	158	71	53
SLOC [K]	131	130	28	38
#Classes	4,285	1,277	398	145
#Methods	32,263	9,081	3,422	1,696
Bytecode [KB]	17,943	5,782	1,650	1,216
Instrumented [KB]	25,133	9,670	2,258	2,392
Ratio [%]	+40.0	+67.3	+36.9	+96.7
Instr. Time [s]	13.9	7.1	2.5	9.1
Inserted Calls	436,766	197,516	41,037	41,652

Table 4.2: Subject system properties and static instrumentation overhead. Instrumented byte code size, Ratio of instrumented byte code size to original bytecode size, and CPU Time needed for instrumentation.

saved. In the third use test case, we are interested in objects of the `Project` class. The actions performed include loading existing projects and creating new projects.

J[2] is a text editor with different editing modes, XML support, compiler/debugger, mail client and lisp interpreter. Classes related to the mail client feature were left out from instrumentation for this experiment[3]. The first use case concerns the `Editor` class, which appears to be the central class of J. We start J, load files, create a new file, copy and paste, save a file, close files, and exit. In the second use case, we look at the `JavaMode` class, which is responsible for Java specific behavior. We load a Java file, use the Java tree display to jump to different methods, and fold and unfold methods (that is, hide/show the body).

JHotDraw[4] is a Java GUI framework for graphical applications. It was originally written as an example for the application of design patterns, but is now used in many applications as well. The tests are performed using the sample JavaDraw application that comes with JHotDraw. As the first use case, the `ZoomDrawingView` class – the view that contains all the visible objects that are drawn – is investigated. The use case consists of creating a new drawing, drawing eight rectangles, starting the animation for 5 seconds, stopping it again, then zooming into the image, and exiting. The second use case traces for instances of `QuadTree`, which is used to partition a drawing. A new drawing is created, eight rectangles are drawn, all eight rectangles are deleted again one after the other, and the application is quit.

ANTLR[5] is a parser generator which is also written in Java. In this study, it serves as a representative for batch tools – all the previously introduced tools are interactive. The test case is the creation of a parser according to the Java grammar as supplied as an example with ANTLR. Two different classes are regarded: one

[2]`http://armedbear-j.sourceforge.net/`

[3]This was done to circumvent problems with the ASM framework that occurred for some classes of the mail part which contained abnormal byte code. However, sending emails is not a basic functionality for a text editor, so this should not be a severe restriction.

[4]`http://www.jhotdraw.org/`

[5]`http://www.antlr.org/`

Test case	number of threads	obj.	mio events offline	online	offl. trace size [MB]
ArgoUML:					
– ClassDiag.	12	1	23.2	25.8	248.3
– SequenceDiag.	9	1	12.2	13.7	130.7
– Project	10	2	12.7	11.3	119.0
J:					
– Editor	12	1	18.7	21.2	191.4
– JavaMode	20	1	9.3	8.1	82.0
JHotDraw:					
– ZoomDrawView	4	1	1.1	1.4	11.5
– QuadTree	3	10	0.9	0.9	9.9
ANTLR:					
– Grammar	1	2	85.7	91.5	809.9
– DFA	1	75	85.7	91.5	809.1

Table 4.3: Measurement of collected data size. Note that after-call is not an event for offline tracing.

is the `Grammar` class, which represents a grammar in memory, the other one the `DFA` class, which implements a deterministic finite state automaton.

General Procedure. For the interactive use cases, it is important to always execute user commands in the same order with similar timing. In order to limit the influence of such timing differences, each use case is executed at least three times for each of the different versions. The three versions consist of the unmodified code (original), the instrumented code with trace logging (offline), and the instrumented code with raw graph construction (online). Then, the average of these runs is taken. Also, execution times are measured in terms of CPU time instead of elapsed real time to further minimize those effects. All use cases and time measurements were performed on a 3 GHz Pentium IV machine.

4.3.2 Results

Static instrumentation overhead. The bottom part of Table 4.2 provides an overview of the cost for doing the instrumentation. Byte code manipulation is the same for both the online and offline approach, so this does not need to be distinguished. The implementations only differ in what happens in the functions that are called by the instrumented code.

Instrumentation is done very fast. Up to 1.3 MB of byte code are instrumented per second, or up to 2,300 methods per second. The number of inserted calls to tracing routines averages between 100 and 300 per class. This increases byte code size by up to 97%, which is mainly due to the location identification strings that have to be stored for each event location.

	CPU time [s]				
Test case	original	offline	+filter	=sum	online
ArgoUML:					
– ClassDiag.	19.0	130.4	29.8	160.2	155.0
– SequenceDiag.	16.2	59.4	15.8	75.2	62.8
– Project	12.6	46.0	29.2	75.2	43.0
J:					
– Editor	3.7	81.3	28.5	109.8	58.5
– JavaMode	3.5	29.4	9.6	39.0	24.5
JHotDraw:					
– ZoomDrawView	3.4	7.6	1.8	9.4	6.4
– QuadTree	3.7	7.3	12.3	19.6	10.1
ANTLR:					
– Grammar	15.3	104.6	80.7	185.3	126.5
– DFA	15.3	108.0	4,805.8	4,913.8	2,501.8

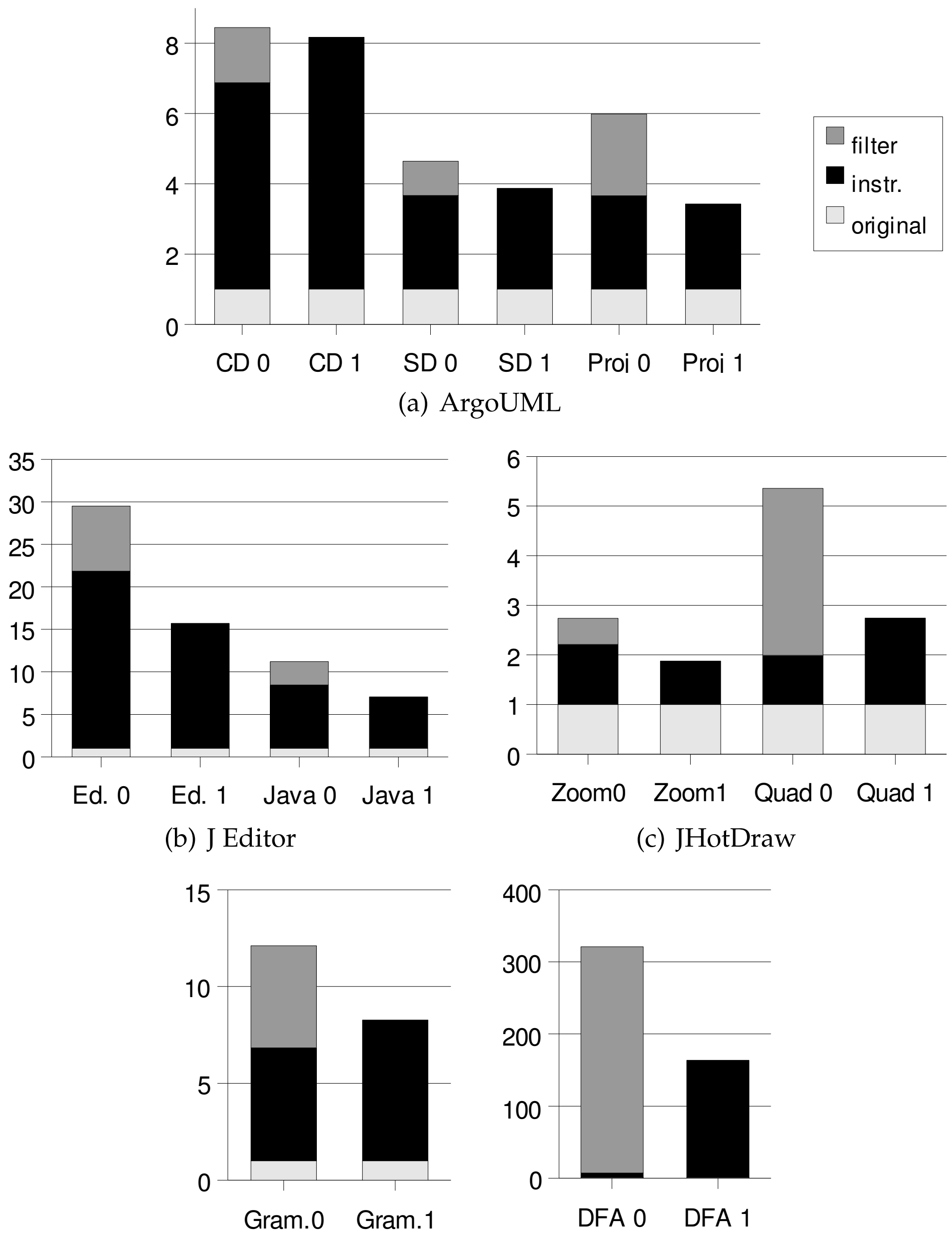
Table 4.4: Measurement of data collection performance.

ANTLR's figures are quite different from the GUI tools': there are much fewer classes, but a very high instrumentation density. This is probably due to the fact that batch tools contain pure application logic, while GUI based tools naturally contain a lot of GUI code, which has a low control flow complexity, but is lengthy.

Data collection overhead. The overhead produced by tracing for Dynamic Object Process Graphs with the online and offline method is shown in Tables 4.3 and 4.4. Offline construction requires the additional object trace filtering step and the following raw graph construction, so these three steps have to be combined to be comparable to online tracing. These two offline steps will be called "filter".

The use cases differ in the number of objects that are created of the relevant class and in the number of concurrently running threads. This also has a great impact on tracing overhead. In the online approach, for each additional object, another GraphView has to be created and maintained, so runtime will increase. For the offline approach, additional objects only become important in the filter phase, because then, an own object trace must be extracted for each of the objects. When many threads are involved, they must all be synchronized, so there will be more waiting time than with a few threads. This affects both the online and the offline approach. The online approach additionally has to create and update the ThreadInfo data structures for each thread.

The number of events that occur during use case execution differs between online and offline approach, although the same actions have been performed. This is due to the fact that arrival at the call site (on return from a subroutine) is not an event for offline tracing, but is one for online tracing. Another reason is that timing changes, specially in the presence of multithreading and interaction. In this way, instrumentation modifies application behavior. For ArgoUML, a longer running time corresponds to more events, which is not true for the other applications.



(a) ArgoUML

(b) J Editor

(c) JHotDraw

(d) ANTLR

Figure 4.6: CPU time [s] consumed by online and offline tracing in comparison to normal program execution. 0=offline, 1=online. The light gray area indicates normal program execution time, the black area denotes the additional tracing overhead. The dark gray offline area indicates time consumed by filtering for object traces. CD=ClassDiagramGraphModel, SD=SequenceDiagramGraphModel, Proj.=Project, Ed.=Editor, Java=JavaMode, Zoom=ZoomDrawView, Quad=QuadTree, Gram.=Grammar

The trace files produced by the offline approach consumed 809 MB for the largest use case. This trace contains the events of only 15 seconds CPU time. When running even larger use cases, trace file size will increase accordingly. This illustrates the need for alternatives that do not need to store those large amounts of data.

Concerning execution times, in some cases the online variant requires more CPU time, in other cases this is true for the offline variant. When taking into consideration the additional filtering effort required for the offline approach, the online variant is always faster. Figure 4.6 visualizes CPU time ratios for the different use cases. The y-axis denotes the factor to which the consumed CPU time increases. The use cases as executed with the original program are normalized to 1. The black parts of the bars depict the overhead during execution, while the gray area corresponds to the additionally needed filtering for the offline approach. Execution time overheads differ a lot, from 88% for the `ZoomDrawingView` (online) up to factor 22 for the `Editor` (offline). J even has a higher overhead than ArgoUML, although ArgoUML supposedly is the more complex application. Apparently, J does a lot of background work (such as autosaving) which produces lots of events.

In one case, the online approach is remarkably slower: in the ANTLR use case with 75 simultaneously followed objects. This can be explained by the increased overhead of constructing multiple GraphViews. As expected, in cases when many instances of a class are created, the offline approach is more efficient concerning application runtime. On the other hand, the effort required for offline filtering also increases and here even exceeds the application runtime by far. In sum, the total CPU time required for the offline approach is twice as high as for the online approach in this case.

Another observation is that use cases with many threads (J, ClassDiagram) generally have a higher runtime overhead when instrumented. This strengthens the assumption that performance loss is higher in the presence of many threads.

From the user's view, the interactive applications were usable nearly as normal in all cases. The tracing overhead did not slow down the applications to an unacceptable degree. A maximum CPU usage increase by factor 22 for the J editor may seem very high but was not disturbing in practice.

For the batch application ANTLR, the tracing overhead of a factor of up to 8 for a single object is acceptable as well. However, when there are many objects to be traced in parallel, the online approach slows down the application significantly. For the `DFA` case, this results in a factor of 163, which is unacceptable in many cases. The offline approach has a runtime overhead of only factor 7. In summary, each approach has its preferred application scenarios.

4.4 Summary

This chapter has shown that Dynamic Object Process Graphs are applicable in practice, even for larger and interactive systems. The optimizations for the offline approach and the newly introduced online approach both allow tracing applications with an acceptable overhead. The online approach turned out to be always faster when investigating a single object, while the offline approach clearly leads to a shorter application runtime when following many objects simultaneously. However, the overall runtime including filtering for all objects is always longer than immediate online extraction.

Online construction of Dynamic Object Process Graphs also opens new application potentials. For example, the evolving graph can be shown while the application is running, providing information about which relevant parts of the application have already been visited. See Chapter 10 for a discussion of some ideas for taking advantage of these possibilities.

Chapter 5

Case Study: Comparison to Statically Extracted OPGs

As introduced in Chapter 2, there already exists a static OPG extraction technique in the Bauhaus project. So far, it has been an open question how much the results of static and dynamic OPG extraction for the same object differ. To clarify this question, I conducted another case study. It investigates the application of OPG extraction to three different programs for quantitatively comparing the results of static and dynamic analysis.

5.1 Procedure

The comparison of static and dynamic OPGs must be based on a comparable representation. Both approaches deliver slightly different graphs that have to be unified before they can be compared. One difference is that dynamic tracing can produce OPGs for every single instance, whereas static tracing is only capable of delivering one combined OPG for all objects that have the same allocation point. The results of dynamic analysis therefore have to be combined for all objects that are declared or allocated at the same source code location. Another difference is context-sensitivity: the static analysis is context-sensitive, whereas the dynamic extraction algorithms from Chapters 3 and 4 are not. Therefore, all nodes of the static OPG that originate from the same source location have to be unified first. The implementation of the static analysis does not differentiate between different call traits to the same allocation point (as opposed to the information in the corresponding paper [46]), therefore both approaches handle static allocation points context-insensitively.

After these unifications, OPGs for all statically detectable objects are extracted by both analyses. For the dynamic analysis this is done for all test cases. As mentioned in Section 2.5, a program requires input parameters to execute – and, in particular, to perform a dynamic analysis. We call the set of input parameters that are used as the basis for dynamic analysis the *test suite*, which consists of individual *test cases*. For the test suite, a high *test coverage* is anticipated to get as close as possible to the complete dynamic trace. Due to the large amount of data produced by offline dynamic tracing, the tests were run only for small to medium size programs and short test cases. This allowed the extraction OPGs for *all* objects – which is not feasible with the online approach (see Chapter 4).

In this case study, we distinguish between two different kinds of objects: local variables and heap objects. Local variables are mostly basic types, such as `int` or pointers, but also `structs` and arrays. Heap objects are created by calling memory allocation functions such as `malloc`. Every access of the allocated memory area is then treated as an access to the object. Objects of the third type – global variables – are not considered; they are similar to heap objects in that they may be used everywhere in the application and during most of the application runtime.

The choice of this general kind of object leads to a large number of objects even for small programs. Local variables are usually only used in a very limited fraction of a program, but can still be interesting for protocol validation. The Object Process Graphs for local variables are not necessarily just intraprocedural, because they can be passed as reference parameters to other functions, and then, automatic validation becomes of interest.

As an indication for the similarity of two corresponding graphs, the number of nodes and edges in each graph can be compared. When the number of nodes and edges is identical, the two graphs should be identical in most cases as well, because a DOPG is always a subgraph of the corresponding statically derived OPG. (This was also checked and confirmed for random samples.) The only exception occurs when the static trace is incomplete due to insufficiencies of points-to analysis.

5.2 Underlying Points-To Analysis

In Chapter 2, we learned that points-to analysis has a strong influence on any static analysis. This makes a short discussion of the used underlying analyses necessary. In this case study, two different points-to analyses are used to obtain the static traces, namely, the one by Wilson [213, 214] and the one by Das [39]. This is done because Wilson's precise technique is only applicable to very small systems, and even some of the small analyzed systems also only allow the use of less precise analyses.

Wilson's points-to analysis is a context-sensitive and flow-sensitive analysis that models the effects of pointer expressions (including arithmetics) by way of an *abstract storage*. The abstract storage consists of a set of *abstract blocks* that represent contiguous pieces of memory. This way, programming practices that circumvent the type system – as in particular possible in C – are safely handled. Wilson's analysis is very precise but also very costly. It could only be applied to one of the analyzed systems, namely, `concepts`. In contrast to that, Das's analysis is flow-insensitive and context-insensitive. This analysis scales well, but produces less precise results.

Both pointer analyses cannot distinguish accesses to different array elements, except for rare cases. Hence, such accesses are treated as partial accesses to the whole array. They do not trace the individual objects in the array.

System	LOC	SLOC	Functions	Coverage
`concepts`	8,469	3,647	171	75.8%
`ftp`	6,077	4,984	239	59.5%
`grep`	10,749	7,563	149	54.7%

Table 5.1: Size measures and statement coverage of test suite for subject systems.

Moreover, the current implementations of the pointer analyses in Bauhaus optimistically ignore effects through library calls. This optimistic approach may produce incomplete information in certain cases. For instance, a pointer to a function may be transfered to a library function whose code is not available and then called from within the library. This control flow goes unnoticed if the effect of the library function is neglected. To be conservative, one would need to assume the worst for every library function – an approach that lessens the precision of the static Object Process Graphs to the point of uselessness. Ideally, one should model the behavior of library calls through stubs linked to the application to be analyzed. However, that is a lot of work and requires to know the effects of all library functions.

As a consequence of the optimistic approach, the dynamic analysis, which tracks such effects, may yield a Dynamic Object Process Graph that is not a complete subgraph of the static one.

5.3 Subject Systems

The static analysis by Eisenbarth et al. was only available for C at the time this case study was performed (2005). Therefore, I was only able to compare object traces of C programs. Meanwhile, the static analysis is also available for C++. However, the basic static analyses have not changed for C, so the results would be the same today. Table 5.1 shows various size measures for the chosen systems. LOC denotes the lines of code including blank lines and comments as measured by the Unix tool `wc`. SLOC denotes the physical lines of codes, that is, counts only lines of code with at least one C token. It was computed with the tool `sloccount`[1]. Column "Functions" gives the number of function definitions, and the "Coverage" column shows the test coverage (on line level) that was reached when running the test cases for dynamic analysis.

The first analyzed system is **concepts**[2]. Concepts is a tool for performing formal concept analysis. The `test.in` file that is included in the distribution was used for the tests, and `concepts` was run with different command line arguments. This way, a test coverage of 75.8% (as measured by `gcov`) was reached. When the generated scanner and parser in `concepts` is not considered, coverage increases to 91.2%.

[1]`http://www.dwheeler.com/sloccount/`

[2]`http://www.st.cs.uni-saarland.de/~lindig/src/concepts.html`

		#local variables		#heap objects	
System	points-to	static	dynamic	static	dynamic
`concepts`	Wilson	284	268	25	24
`ftp`	Das	340	260	7	3
`grep`	Das	456	275	67	43

Table 5.2: Static/dynamic object counts.

The remaining code that is not executed is mostly for error handling. These branches would only be executed when `malloc` cannot allocate any more memory, when a function is used in a wrong way or in a way different from `concepts`, or when something like file I/O fails. Therefore, most of these cases will never occur under normal conditions. However, they are of course contained in the static traces, but are not contained in the dynamic traces. Also, some of the data structures are written to be generally useful, but are used in a quite constrained way within `concepts`. They therefore contain code that can never be reached in the context of this application.

NetKit ftp[3] is the standard internet file transfer program for Unix. For this study, version 0.10 of this tool was analyzed. A set of test cases that use all the different ftp commands and modes in different combinations was used. However, it was hard to reach a high test coverage, since a large portion of the code deals with error handling and the different behavior of different servers. Therefore, tests covered only about 60% of the code.

Grep[4] is a GNU tool to search for regular expressions in text files. For this study, version 2.5 of `grep` was analyzed. Grep comes with a test suite with about 380 test cases which was used for the dynamic analysis. Hence, as opposed to the other two case studies, I did not write my own test cases, but used existing ones, which makes the experiment even more realistic. Interestingly enough, however, I found that the statement coverage of this test suite is only 55 percent, far lower than in `concepts` and also lower than in the `ftp` case study.

5.4 Results

Table 5.2 shows how many local and heap objects were detected by static and dynamic analysis. Dynamic analysis did not detect all statically detected objects, since some objects are declared and used within dead code or are not covered by the test cases. However, for `concepts`, nearly all statically detectable objects are also used in the test cases.

The medium, average, and maximum node and edge counts for the static and dynamic analyses of local and heap variables are shown in Table 5.3. Obviously, the number of nodes and edges is much higher in the statically extracted graphs

[3]`ftp://ftp.uk.linux.org/pub/linux/Networking/netkit/`

[4]`http://www.gnu.org/software/grep/`

			#nodes			#edges		
System	Class	S/D	med	avg	max	med	avg	max
`concepts`	Local	Stat.	12	20	317	14	26	444
		Dyn.	6	12	82	6	15	107
	Heap	Stat.	380	517	1,594	589	864	3,004
		Dyn.	86	99	311	129	153	584
`ftp`	Local	Stat.	21	27	152	28	41	251
		Dyn.	7	10	42	7	11	56
	Heap	Stat.	25	87	417	24	141	717
		Dyn.	21	28	53	28	42	83
`grep`	Local	Stat.	19	29	453	25	44	791
		Dyn.	10	18	138	10	24	208
	Heap	Stat.	152	184	1,242	262	313	2,086
		Dyn.	51	59	186	78	102	349

Table 5.3: Object Process Graph node and edge counts per system, object class, and extraction method: Median, average, and maximum values.

than it is in the dynamic graphs. Also, there is a big difference between local variables and heap objects. The dynamic graphs for local variables cover much more of the static graphs than the heap object graphs do.

Interestingly, the number of nodes resulting from static analysis with respect to local variables is comparable across systems even though the sizes of the systems differ by a factor of 2 (see Table 5.1 in terms of SLOC). On the other hand, the respective numbers for heap variables and static analysis differ by an order of magnitude. Even though the more precise pointer analysis was used for `concepts`, the static Object Process Graphs of `concepts` are the largest. For this system, also the Dynamic Object Process Graphs are much larger. This indicates that heap variables are used more globally in `concepts` than in the other two systems.

Figure 5.1 shows the percentage of nodes and edges that are contained in the Dynamic Object Process Graph for local variables, compared to the static graph for the same object. The objects are uniformly distributed along the X axis in ascending order of percentage (empirical distribution function), and the Y axis shows the percentages. The curve for edges is always very close to the node percentage curve it belongs to.

For local variables, there is always a set of graphs for which the node and edge counts are identical, which indicates that the entire graphs are most probably identical. For `concepts`, this is true for about 40% of the graphs, but for the other test cases, it is true for only 15% or less. It is also noticeable that `concepts`' curve rises faster than the other candidates' curves do, which probably has to do with the higher test coverage that was reached in `concepts`, compared to the other two systems.

As can be seen from Figure 5.1, it is apparently possible to completely construct local variables' Object Process Graphs by dynamic trace extraction in many cases.

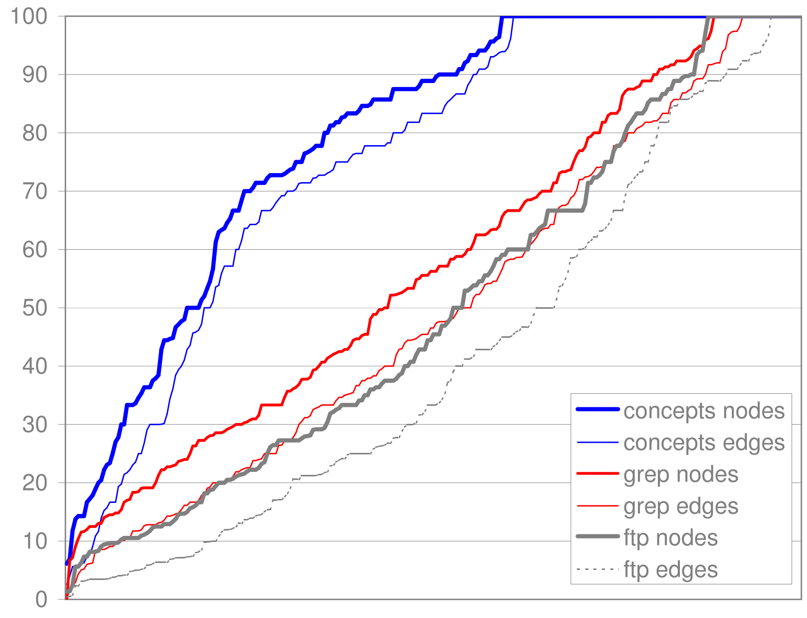


Figure 5.1: Local Variables: Percentage of nodes in Dynamic Object Process Graph compared to the corresponding static one. The X axis is an equidistant enumeration of all OPGs in ascending order of percentages (empirical distribution function).

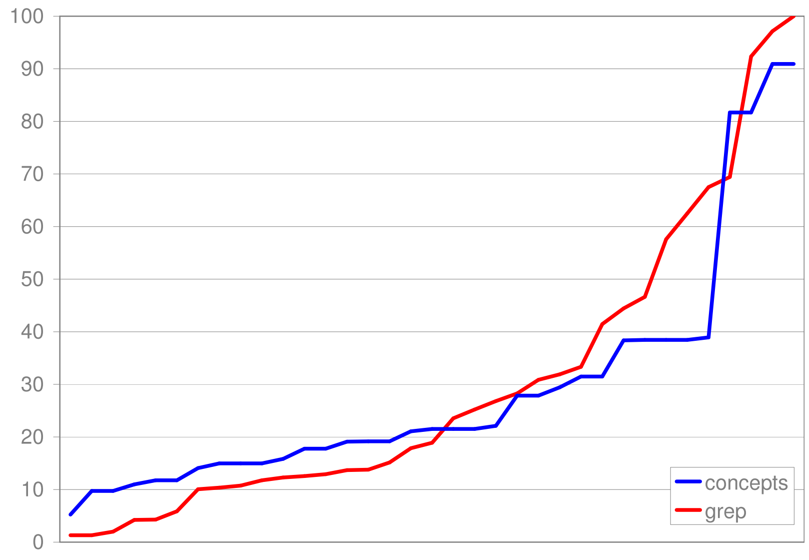


Figure 5.2: Heap objects: Percentage of nodes present in DOPG. `ftp` is missing because it only contains three heap objects.

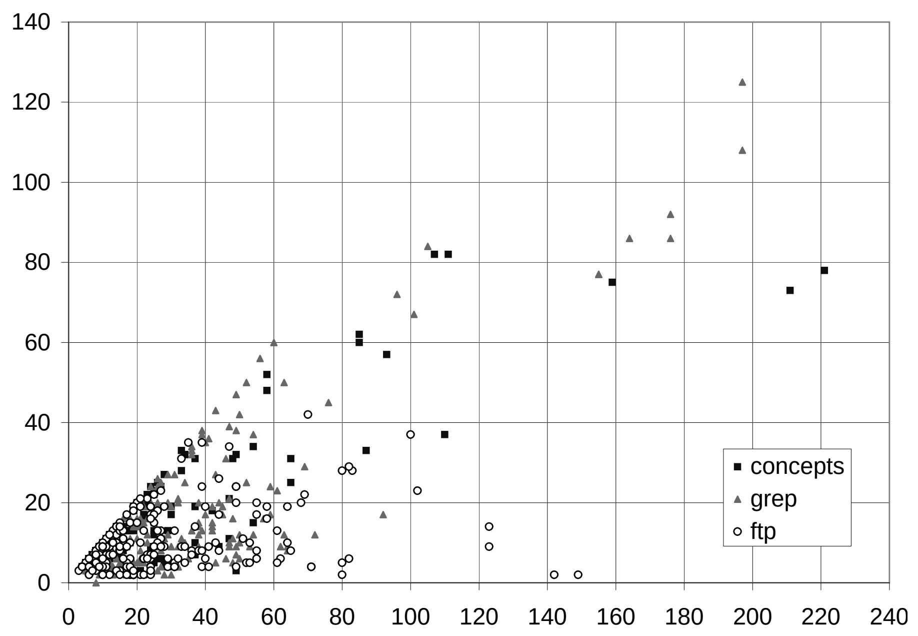


Figure 5.3: Local variables: Number of nodes in statically extracted object process graphs (X) compared to the number of nodes in their dynamic counterparts (Y).

Yet, the graph is still incomplete in many other cases because the test cases do not lead to the execution of every possible sequence of operations with regard to each object. Also, static traces may contain infeasible paths.

In contrast to that, when regarding Figure 5.2, which shows the same information for heap objects, we see that those curves stay much lower for most of the objects and hardly ever reach the 100% line. For `concepts`, 85% of the objects even stay below the 40% line. The high test coverage does not have a great effect in this case. The `grep` test case has a much lower test coverage, but the two curves are very close to each other.

The results for heap objects differ much more between static and dynamic analysis. Manual evaluation of the graphs for these cases revealed that in fact the static traces contain a lot of irrelevant nodes and edges. Those paths often have nothing to do with the investigated object, but are included in the static trace. Anyway, as described above, this is usually due to the imprecision of points-to analysis which in many cases cannot decide for sure whether a given pointer variable points to the investigated object or not. On the other hand, due to missing test cases, also missing parts can be found in the dynamically extracted pendants. Since heap variables usually have a much longer lifetime than local variables, a lot more operations may be applied to them, and these operations can be distributed over many places in the code.

Figures 5.3 and 5.4 show the number of nodes in the static graph (X axis) compared to the number of nodes in the corresponding dynamic graph (Y axis). The number of DOPG nodes is a lower bound because test cases are usually incomplete. Those cases where the static trace was incomplete due to the impre-

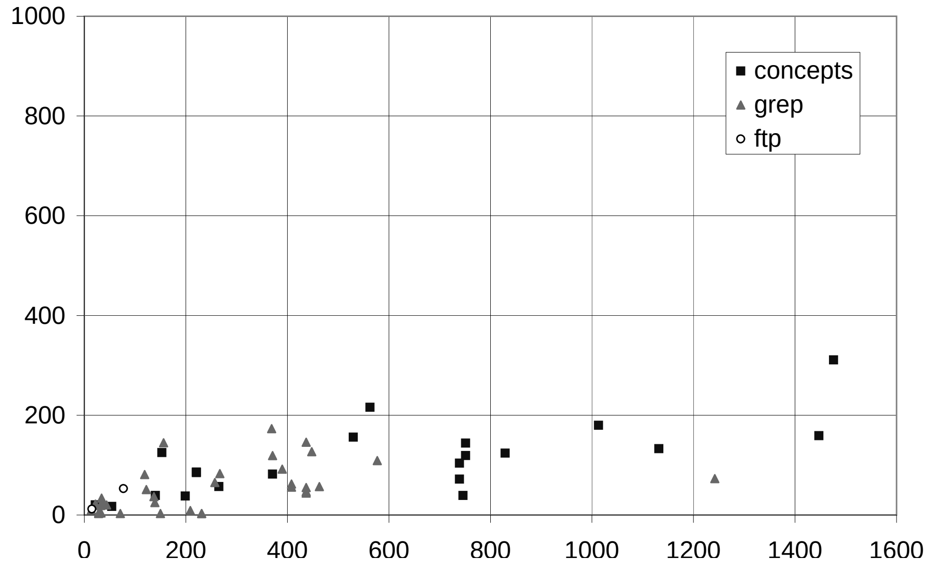


Figure 5.4: Heap objects: Number of nodes in statically extracted object process graphs (X) compared to the number of nodes in their dynamic counterparts (Y).

cisions mentioned above (see Section 5.2) were elminated. They were identified by comparing the static to the dynamic OPG: when the dynamic OPG is not a subgraph of the static OPG, the static OPG must be incomplete. Therefore, in this graph, the number of nodes and edges in the dynamic trace is always smaller or equal to the corresponding number in the static trace; that is, the points in the charts are at or below the diagonal. From Figure 5.3, one can see that most of the local variables' graphs are quite small (less than 60 nodes), and that dynamic analysis delivers good results for these graphs in many cases. As the static graphs get larger, the dynamic results do not contain all those nodes anymore, but still deliver a relatively high percentage. When regarding the corresponding diagram for heap objects in Figure 5.4, we can see that their static graphs are much larger, and that the dynamic graphs mostly contain only a fraction of the static nodes.

In summary, for local variables, static and dynamic tracing deliver results that are often close to each other or even identical. Results were even better for the system that had a higher test coverage. However, this is only true for quite small graphs of up to 30 nodes. In contrast to that, for heap objects, the results differ a lot. Those static graphs mostly contain five to ten times as many nodes as the dynamic graphs. However, more complex usage patterns as they occur with heap objects – their lifetime may only end when the application terminates, and they may be used from many places – are more interesting for analysis: local variables' usage is often obvious from looking at a very limited amount of code.

This study regarded the completeness of dynamic OPGs in comparison to static ones. However, the relevance of completeness depends on the application. If we are interested in understanding how a certain set of use cases is implemented, it is usually enough to perform the dynamic analysis based on only those use cases. It is not necessary to reach a high test coverage in this case, since this would only make the result much more complex without adding any relevant information. On the other hand, if we want to recover the protocol, all possible cases should be regarded. This is particularly true if we want to find potential protocol violations. So for this application of Dynamic Object Process Graphs, completeness is more important.

5.5 Summary

In this chapter, we compared Dynamic Object Process Graphs to Object Process Graphs gathered through static analysis. The comparison showed that the Object Process Graphs for local variables are largely similar in size for dynamic and static analysis, but those for heap objects differ by an order of magnitude. The difference may be partly attributed to an incomplete test suite, but the largest part is likely to be explained by the imprecision of global static analyses. Nevertheless, although on average the Object Process Graphs for local variables were similar in size, their maximal values were quite different. This observation suggests a combined approach in which dynamic analysis is used for objects whose static Object Process Graph is too big due to the imprecise underlying points-to analysis.

Part III

Applications

Chapter 6

Case Study: What DOPGs can tell

The previous chapters introduced a novel dynamic OPG extraction technique and compared it to an existing static one. The comparison showed that dynamically derived OPGs are mostly much smaller than the static ones. The question now is whether these graphs are useful despite that reduction, that is, whether the right parts of the static OPGs are left out.

This chapter reports from several case studies in which experiences in applying the dynamic OPG extraction technique were gathered. DOPGs were extracted and visualized for a number of systems. Whereas static OPGs are usually too large for immediate visualization, this is often not the case for dynamic OPGs. We therefore can see and discuss what the extracted graphs look like and what their potential applications are.

6.1 Subject Systems

All the subject systems of this case study are written in C. This is because this case study was performed with a first prototype implementation of DOPG extraction that only supported this language. Table 6.1 summarizes various size measures for the subject systems. The first data column (LOC) contains the source lines of code including blank and commented lines as counted by the Unix program `wc`. The second column shows the number of non-empty non-comment lines as counted by `sloccount`[1]. The third column lists the number of C files the implementation consists of. The fourth column contains the number of function definitions, which equals the number of nodes in the call graph without library calls. The fifth column shows the number of nodes in the static control flow graphs for these systems. The last two columns provide data on the result of the dynamic analysis: they count the number of nodes and edges of the Dynamic Object Process Graphs for the particular component that was investigated for the respective system. As one can see, the reduction in terms of number of nodes is enormous. On average, the size of the Dynamic Object Process Graph is 0.65%

[1]`http://www.dwheeler.com/sloccount/`

					CFG	DOPG	DOPG
System	LOC	SLOC	Files	Functions	nodes	nodes	edges
sdcc	177,749	120,218	146	3,207	115,036	166	307
ircII	49,734	39,825	59	1,168	22,568	115	161
Rhapsody	18,827	14,693	37	499	9,561	67	111
SQLite	60,776	38,575	65	914	22,774	300	463

Table 6.1: Size measures of the subject systems.

of the size of the static control flow graph. Compared to that, the average size of a *slice* is still about 33% of the original program for static slicing and about 20% for dynamic slicing [19]. Even in comparison to the call graph, the reduction of a DOPG is remarkable. However, the question is if these graphs are still useful. The case study in this chapter demonstrates that this is in fact the case.

6.2 Symbol Table of a Large Compiler

This part of the case study demonstrates that the results of DOPG extraction are useful even for important components of larger systems. It shows how the result may help in program understanding.

sdcc[2] is an open source C compiler for small devices. Version 2.4.0 was used for this study. When analyzing a compiler, it is interesting how a central data structure like the symbol table is used. Therefore, the Dynamic Object Process Graph for the symbol table of sdcc was extracted. This component could easily be located based on its filename (`SDCCsymt`). Figure 6.1 shows the result for the compilation of a small program with two functions, a for loop and some integer calculations. The size of the Dynamic Object Process Graph (measured as number of nodes) is only 0.14% of the complete static control flow graph of the whole system (see Table 6.1).

In the graph in Figure 6.1 and in the following, italic routine calls denote application routines, while regular routine names with brackets denote routines of the investigated interface (atomic routines). Two additional long distance arrows (in red) were manually inserted to give an idea of the flow of control. From this graph, we can see that first, the symbol table is initialized (`initCSupport` ①), then the parser is called (`yyparse` ②). Inside the parser, the lexer ③ is called, which in turn uses the symbol table to check the type of a symbol (`check_type` ④). Also, for creating functions, the symbol table is accessed multiple times. The following routines all use the symbol table directly or indirectly:

- `checkFunction`: has the function already been defined?
- `findSym()`: is the name already in use?

[2]`http://sdcc.sourceforge.net/`

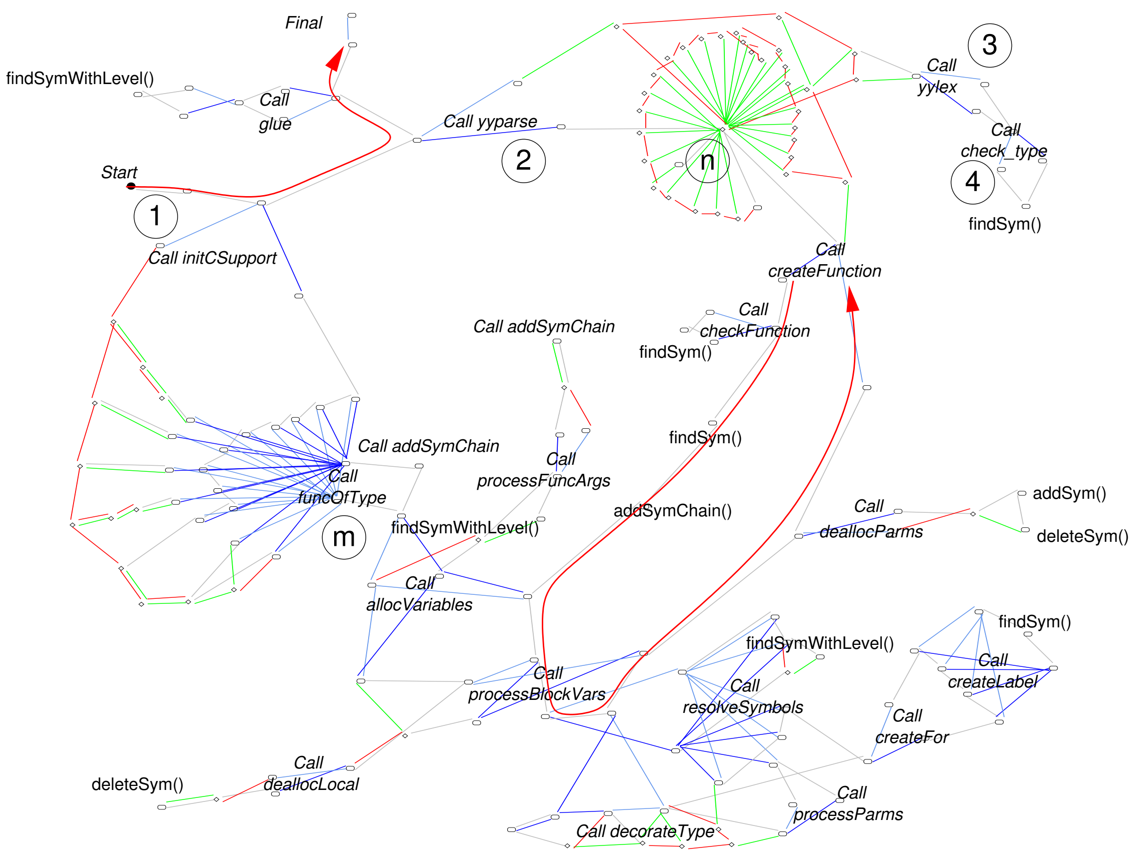


Figure 6.1: DOPG for sdcc.

- `addSymChain()`: resolve forward declarations
- `allocVariables`: create function declaration
- `processBlockVars`: allocate symbols for block
- `resolveSymbols`: resolve text to symbol table symbols
- `decorateType`: type resolving and checking
- `processBlockVars`: deallocate symbols for block
- `deallocParms`: deallocate parameters that have been allocated in `allocVariables` / `processFuncArgs`

It is an interesting observation of this case study that the above list shows a drawback of the DOPG representation. Routine `processBlockVars` is called twice. The first time, it is called to allocate symbols, the second time, it is called for deallocation. The concrete action is passed to the routine as a parameter. This distinction gets lost in the DOPG representation because the branch occurs only inside the routine. The representation does not show which call takes which branch – although the first call always takes one branch, the second call the other. The same problem occurs for program slicing, too, when we enter the same routine by way of two different data dependencies in two different calling contexts. To make this distinction, we would need to add context-sensitivity to the representation. To keep the representation as sparse as possible, we should refrain from completely unfolding the graph to make the context-sensitivity explicit. We would rather add context-sensitivity as annotations and unfold them on demand of the analyst in an interactive browser.

The graph contains two interesting constructs. The first one consists of the multiple calls of `funcOfType` ⓜ in `initCSupport`. The compiler creates the names for support routines like `__fsadd`. Many of those names are created in loops for different numeric types. So this routine consists of a sequence of calls of `funcOfType`. Since we always have an edge to the entry and one from the return node, we get this interesting network with the calling nodes located around those two nodes and connected to each other. The second interesting construct is the "spoked wheel" inside `yyparse` ⓝ. This "spoked wheel" is caused by the generated parser, which consists of a big switch in the reduce step that depends on the reduction rule to be applied. Only when the reduction rule for a function is applied, `createFunction` is called. Due to the transformation of the switch statement to a sequence of decisions in the normalization step (see Section 3.1.1), a set of other state-indicating values has to be checked before this point is reached. In case of other rules, each one returns via the true edge to the point where the next token is read – the central node of the wheel. This leads to this special structure.

In summary, the extracted Dynamic Object Process Graph gives a good overview of how the symbol table is used and in which context. We are able to recognize the individual steps related to the symbol table even though they are globally distributed in a very large control flow graph.

6.3 Sockets in IRC Clients

This section demonstrates support in program understanding by applying DOPG extraction to a standard component used in many programs. I will first show the protocol for the component and then show Object Process Graphs obtained through dynamic analysis for two different systems.

The use of standard Unix I/O operations is investigated in this study. These operations are applied to the `FILE` data type. This data structure and the operations taking it as an argument type form an abstract data type. Alternatively, handles can be used, which are represented by a regular `int` value. In order to be able to process such handles in the same way as FILE objects, the operations on handles are modeled as read/write operations; that is, executing an operation on a handle is regarded as an access to the memory address that the handle "points to". System routines are replaced by decorators[3] that additionally record that memory

[3]A *Decorator* is a standard design pattern that allows us to attach additional functionality to an

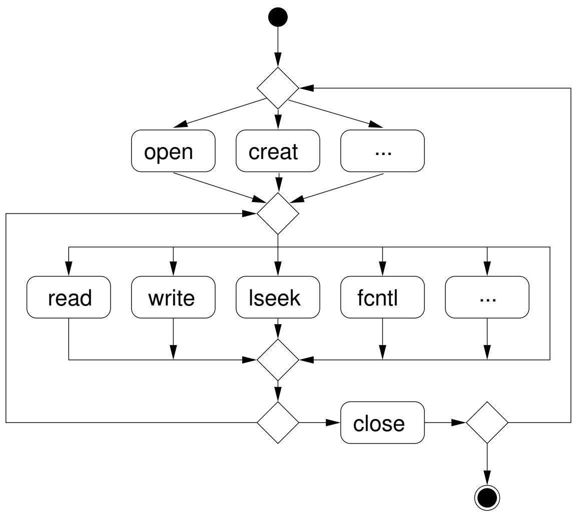


Figure 6.2: Example protocol for files in C.

access. In C, all calls of such routines can be easily replaced by their decorators via macros.

This data type is used for this study because it is frequently used in programs, and because it has an interesting protocol. The expected protocol, that is, the allowable sequence of operations, is described in Figure 6.2 (the dots indicate the many other operations that can be applied to a `FILE` or a handle). It is interesting to note that usually only excerpts of this protocol occur for an object. This excerpt characterizes the object. The excerpt can be thought of as a role as, for instance, a sequential writer or reader or random accessor, etc. The Object Process Graph may help to indicate the role of an object.

6.3.1 ircII

ircII is a console internet relay chat application[4]. The central concern of a chat application is communication, which is done using sockets (a special kind of file handle). Therefore, the control flow with respect to the operations that are performed on the socket handle should give a good overview about the organization of the program's basic functionality.

Dynamic tracing was performed on the `ircII` client. The test session included connecting to an IRC server and executing a few typical commands. The resulting raw trace contained 6.8 million events, which were then converted to an Object Process Graph as described in Chapter 3. Figure 6.3 shows the complete Object Process Graph. According to Table 6.1, the Dynamic Object Process Graph contains only 0.45% of the nodes of the complete static control flow graph of the whole system. Even compared to the call graph, which is often used for program

object [57].

[4] `http://www.eterna.com.au/ircii/`

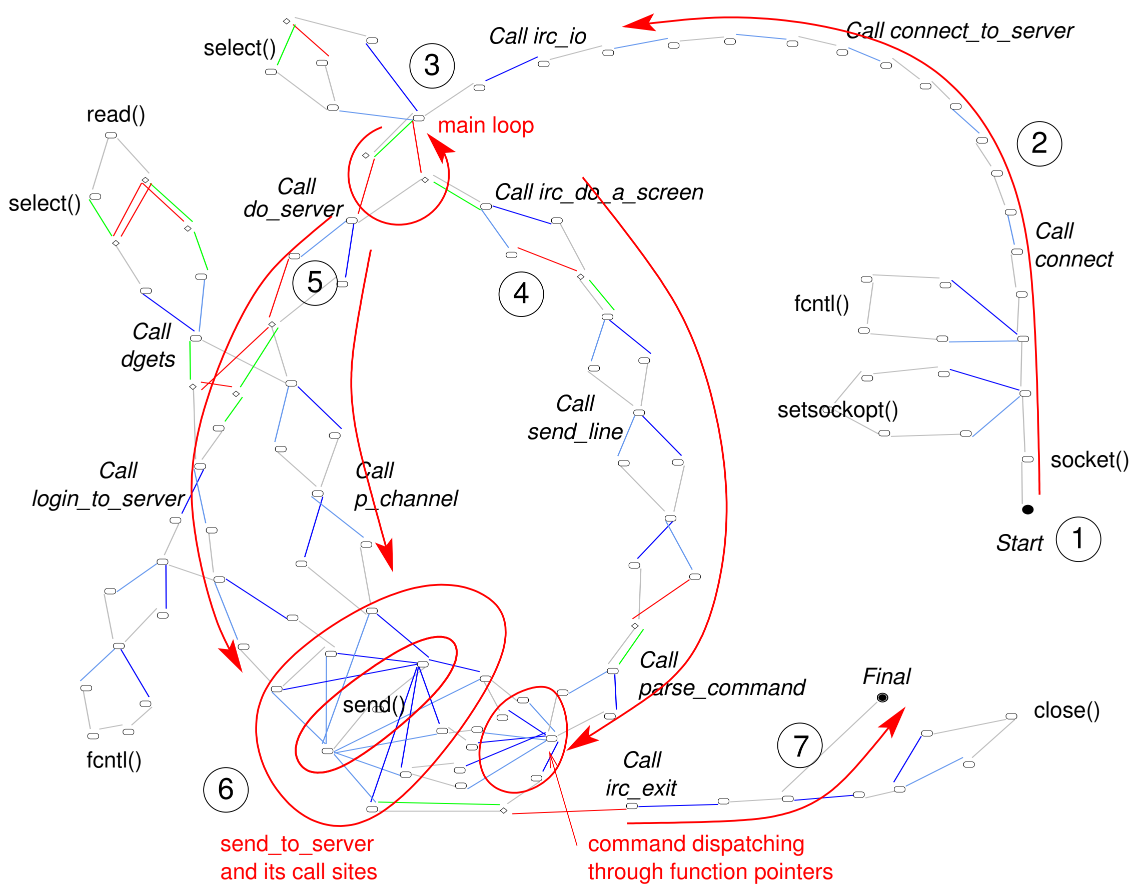


Figure 6.3: DOPG for ircII.

understanding through visualization, there is still a reduction in the number of nodes to 10%.

From the graph in Figure 6.3, it is quite easy to see what is going on in the `ircII` application. Things start with the *Start* node on the right ①. After the socket is created and initialized in `connect` ②, the I/O main loop is entered (`irc_io` ③). From there, user commands are executed (`irc_do_a_screen` ④), and information from a server is received and processed (`do_server` ⑤). Most calls then lead to the `send_to_server` routine ⑥. Finally, if an exit command has been issued, the socket is closed, which leads to the *Final* node in the graph ⑦.

The information in the Object Process Graph can help to gain an initial understanding of the program:

- Identify parts that are relevant for communication. In this case, also the main loop has been located by simply following the flow of control.
- Understand call paths and control dependencies.
- Identify central routines (such as `send_to_server`).

Even when the irc_io routine has been identified as the main loop by other means of feature location, it is much harder to see the basic interactions by investigating the 246 lines of C code than it is to just look at the 5 nodes in the graph. A feature location technique that works on basic block level [99] could deliver the right basic blocks, but one would still have to identify the relevant statements within a potentially large set. Such a set difference based technique also has the problem that the main loop is executed in every program run and is therefore hard to isolate (also see Section 9.1.5).

Apart from this basic information, also implementation details can be seen from this graph. For example, user commands are dispatched through function pointers (see indication in the figure). This is the case when a single call node has several outgoing call edges to different entry nodes – for direct calls, it has at most one outgoing call edge. If dispatching was, for example, implemented through a switch, we would see a cascade of decision nodes at that place.

In the graph, most of the different types of nodes and edges that were defined in Chapter 2 are being used. Edge types and directions are not visible in this figure. They are not necessary to get a basic understanding. But when going into details, the types can provide additional information.

Most of the edges from *Start* to *Call irc_io* are return edges. There are no call edges there because the object lifetime of the socket handle starts only with the call of the socket function, and this call takes place deep in the call stack. On the other side, control flow goes down into different routines (Call edges), and later goes back up again (Return edges). This produces the typical call chains that split and join again and again.

A simple standard spring embedder layout was used for this graph, along with some manual postprocessing. The red long-distance arrows have been inserted manually. Also, some of the node labels (certain function calls within call chains) have been manually removed to make the graph more readable. Readability of this graph could probably be improved even further by using a specialized layout algorithm.

6.3.2 Rhapsody

Rhapsody IRC[5] is another text console IRC client. According to the authors, it has been written entirely from scratch and does not have common roots with ircII. Anyway, the use of sockets should be similar, and it is interesting to see how the Object Process Graphs of the two IRC clients differ. I performed the same dynamic analysis on Rhapsody as I did on ircII. The result is shown in Figure 6.4.

The general structure looks similar to the ircII graph (Figure 6.3): first, the socket is created ①, and then we get into the main loop (dashed arrow to ②). From there, we have different emerging paths for reading a line ③ and processing channel and server events ④ that all eventually lead to a central send routine ⑤.

[5]http://rhapsody.sourceforge.net/

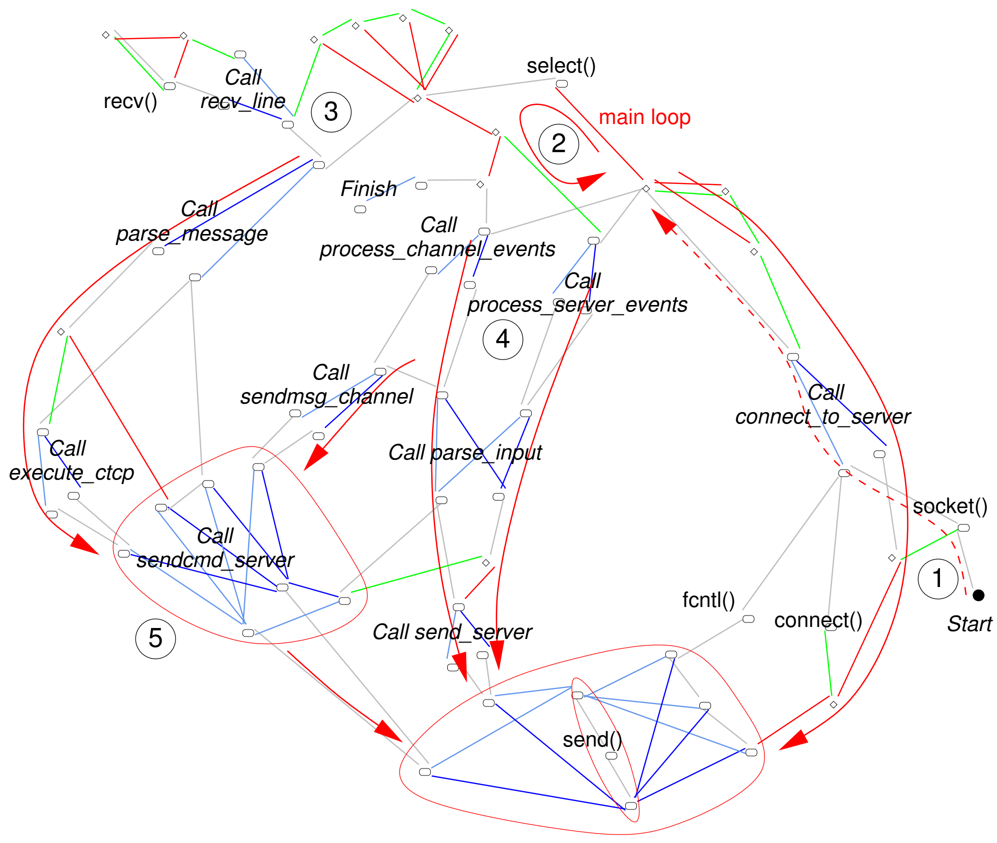


Figure 6.4: DOPG for Rhapsody IRC.

In contrast to ircII, we also get back into `connect_to_server`. Apparently, we first need to go into the main loop and then get back to connect to the server. Another difference is the routine that is used to receive text from the server: Rhapsody uses `recv`, while ircII calls `read`. In contrast to ircII, Rhapsody has an additional intermediate function for sending server commands (`sendcmd_server`). Apart from these differences, most routine names that appear in these graphs can be easily mapped to each other, which is possible because of the similar and limited set of operations that have been applied in the use cases and the meaningful names that routines have in both applications. Of course, when names are not meaningful, this becomes much more difficult. However, commonalities in the structure of the graphs could still give hints for a mapping.

What is also interesting here is that `close` is never called on the socket. The question is whether that is caused by wrong usage of the application (disconnect was not explicitly called), or if it is an application error. Although the socket will be implicitly closed when the application terminates, this might lead to inconsistent behaviour when reusing that part of the program in a different context. In fact, examination of the code revealed that the socket is only closed when *disconnect* is explicitly performed by the user before exit.

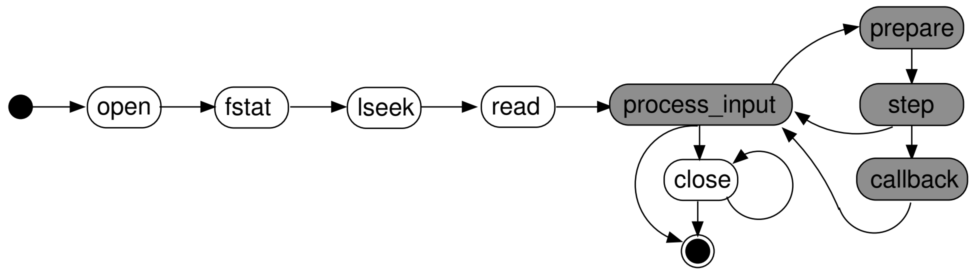


Figure 6.5: Simplified DOPG for SQLite.

In summary, each of the two Dynamic Object Process Graphs provide a good overview of the respective IRC client, and they can also be used to map functionality to each other. In the next section, we investigate a different usage pattern of the file handle.

6.4 SQLite Database

As a last case, the technique to obtain Dynamic Object Process Graphs is applied for files in a database system. This part of the case study reveals that Object Process Graphs can become large and difficult to cope with. To handle such large graphs, further processing or assistance is necessary.

`SQLite` (version 3.2.6) is a C library that implements a self-contained, embeddable, zero-configuration SQL database engine.[6] I chose this application because a database engine can naturally be expected to contain interesting file system interactions. The library includes a command line utility for accessing databases and executing SQL statements, which is used as a driver for dynamic analysis. The call graph of `SQLite` contains 1,649 nodes and 3,098 edges (see Table 6.1).

The application of dynamic tracing on the `SQLite` tool results in a complex graph. It consists of 300 nodes and 463 edges, which is only 1.32% of the number of nodes of the static global control flow graph, but still too much to depict it here. The test cases that were performed include standard database operations, such as `insert`, `delete`, `update`, `select` statements, as well as metacommands like dumping all tables. Also for this graph, the operations performed can be identified in the graph and help gaining an initial understanding of the system. But, due to the size of the graph, this is much harder than in the previous examples. Further assistance from a tool would probably be needed for using this large DOPG efficiently.

However, the graph can still be used as a basis for further simplifying transformations. One possible approach is to reduce the graph to the atomic operations. Figure 6.5 shows a simplified Object Process Graph for `SQLite`. Whereas the original Object Process Graph contains 300 nodes, the simplified one contains only

[6]`http://www.sqlite.org/`

about 60 nodes. In Figure 6.5, the gray nodes represent more complex graphs containing additional calls of `read`, `write`, `lseek` and other operations. This simplification was done manually.

Such a simplified representation is better suited to validate and reconstruct protocols, although it introduces some imprecision. It can be processed by standard algorithms for finite state automata to unify Object Process Graphs and validate protocols. This approach is described, discussed, and evaluated in Chapter 7.

6.5 Summary

The case studies that were presented in this chapter illustrate that the results from DOPG extraction can provide useful information about an application: they may help in locating features, recovering a component's protocol, and even getting an idea about the general structure of an application. The graphs turned out to be quite useful despite their relatively small size.

However, the SQLite case study also showed the limitations of pure DOPGs. For large applications and heavily used objects, the graph may be so large that its immediate visualization is no longer comprehensible. Further processing is needed in this case. Such an approach is investigated in the next chapter: the reduction to atomic methods for protocol recovery. The use of visualized DOPGs for program understanding is further examined in Chapter 8.

Chapter 7

Dynamic Protocol Recovery

The motivation of this thesis started with the need to recover protocols as one aspect of a system's architecture, and in the last chapters, we got to know extraction techniques that can provide the basis for that. Also, the SQLite case study in Section 6.4 showed that it could be a good idea to reduce a DOPG to the primitive operations. In this chapter, I elaborate on this idea and show how DOPGs can be used as a basis for protocol recovery. The necessary transformations to protocol automata are described in detail. In a case study, the new technique is quantitatively compared to other existing dynamic protocol recovery approaches. For this comparison, I introduce and use a novel similarity measure for finite state automata. [1]

7.1 Introduction

Modern static bug finders and security vulnerability detectors perform sophisticated checks. These tools are based on advanced analyses, such as global control and data flow analyses, model checking, or abstract interpretation. There are both successful open-source tools, such as FindBugs, PMD, Lint and its derivatives as well as commercial tools, such as Intelli/J, Grammatec/CodeSonar, Coverity/Prevent, and Polyspace, using such technologies.

Many of these tools find generic defects, such as potential null pointer dereferences or buffer-overflow problems. They may be used to limit the effects of weak programming languages (for example, no index range checks at runtime) or frequent programming errors. Although useful, such tools do not help in finding application-specific defects, where a component is not used according to its specification. More advanced tools may detect such problems by allowing an analyst to create customized checks. Engler et al. [49], for instance, developed a technique where checkers can be specified as finite state automata (FSA). These FSA specify

[1]The contents of this chapter have been previously published [149]. Due to an error in the conference paper, the automaton difference calculation algorithm and the resulting distance metrics that are presented in this thesis differ slightly from the originally published version.

the allowable sequences of operations of a component – its protocol – in terms of a regular language.

Engler's technique checks code by traversing the control flow graph and symbolically executing the operations. The effect of the symbolic execution is a transition in the FSA and is applied for each operation in the code that is part of the alphabet of the FSA. An error state in the FSA indicates a potential defect in the code. The technique has been successfully transferred to industry by way of Engler's spin-off Coverity[2], implemented in their tools *Prevent* and *Extend*. Coverity runs its checkers regularly on large open-source projects, such as GNU/Linux. These checks have discovered thousands of defects in GNU/Linux, including several security alerts. Similar techniques are used by GrammaTech's CodeSonar.

To write application-specific checkers, an analyst needs to know the protocol of a component in the first place. However, as discussed in Chapter 1, the protocol is not specified in many cases. Sometimes, it is informally included in the documentation, but this makes it impossible to check compliance of applications with these protocols automatically. If the protocol was available in a machine-readable way, adherence could be checked, and this could help to make software more reliable and less erroneous. Also, the protocol could be used to automatically generate state-based tests for the component.

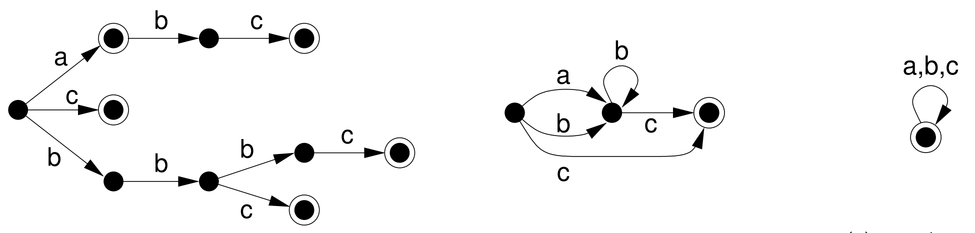
In summary, the usefulness of having the protocol of a component is undoubted. It is therefore desirable to be able to reconstruct the protocol of a component. In the remainder of this chapter, I will therefore only show how the protocol can be recovered based on DOPGs and what the resulting protocols look like. The fact that these protocols can be used to detect errors in applications has already been proven by others, as the Coverity example illustrates.

7.2 Protocol Representation

As introduced in Chapter 1, a *protocol* in the sense of this thesis describes the sequencing constraints that are imposed on a component's methods: it tells us in which order these methods may be called. The protocol is part of the *interface* of a component. So far, we have not discussed which formalism should be used to express these sequencing constraints, and in particular, which expressiveness is required or possible.

A recovered protocol will be used for checking whether given applications adhere to the protocol. It therefore makes sense to investigate what the protocol should look like for this purpose. *Protocol validation* aims at checking whether all actual sequences of operations conform to the protocol. All actual sequences of operations form a language; likewise, a protocol can be considered a language. Hence, protocol validation needs to check whether one language is a subset of another language. This test is in general only possible for regular languages. Consequently, regular languages or finite state automata (FSA) are usually the

[2]`http://www.coverity.com/`



(a) Prefix tree acceptor. (b) One possible generalization. (c) Another generalization.

Figure 7.1: Prefix tree acceptor for the language sample {a,abc,c,bbbc,bbc} and two possible generalized automata that may be derived from it.

notion of choice for protocol validation (see Section 9.5.8 for details). Therefore, and to be comparable to other protocol recovery approaches, we also choose FSA to represent a protocol. In such protocol automata, automaton states represent program states, and transitions correspond to operations on a component.

Regular languages are restricted in their expressiveness in that they cannot deal with recursion: they cannot count. For some cases, such as for the stack example from Chapter 1, it would be good to be able to specify conditions that involve counting. For example, you may want to specify that the number of calls to one method relates to the number of calls to another method in a certain way. This is not possible with a regular language. On the other hand, regular languages cover a lot of the sequencing constraints that occur in practice, and the possibility of doing automatic adherence checks compensates for this disadvantage.

7.3 Related Research

Related research in general is discussed in Chapter 9. However, I need to introduce the alternative existing techniques to which I will compare my approach. Therefore, let us start with an introduction of existing dynamic protocol recovery techniques.

Dynamic protocol recovery has been subject to prior research. Researchers have mostly focussed on automaton learning in this area: program traces, that is, sequences of invocations of a component's methods, are fed into a learner which produces an automaton that accepts the given traces – and more [5, 152, 169]. Constructing an automaton that accepts exactly the given set of traces (*prefix tree acceptor*, PTA) is simple: the common prefixes for each sequence lead to the same branch node, and every unique suffix leads to a leaf node. Figure 7.1(a) shows an example. However, such an automaton is not very useful. It only represents the concrete sequencing information of the regarded applications that use the component, although other usages might be allowed as well. Therefore, generalizations are necessary. One possible result of generalization for the example PTA is shown in Figure 7.1(b). Most automaton learning techniques apply different heuristics to transform the prefix tree acceptor to a more general form, thus reducing the number of states and transitions. This reduction is usually performed by applying different state-merging strategies. However, when generalizing too much, the automaton becomes useless as well (*overgeneralization*). In the extreme, the protocol automaton could be reduced to a single state with transitions on all possible events that lead back to this state, which allows any sequence of operations. This is illustrated in Figure 7.1(c). The challenge is to apply the right amount of generalization to get a protocol automaton that is most useful and meaningful for a particular purpose. For example, one anticipated goal may be to get a protocol that is as close to the real specification as possible.

Let us now look at some previously published protocol recovery approaches that are based on automaton learning. These are the ones to which the DOPG based approach will be compared later on.

Successor method. A straight-forward approach is to represent each method by one state. Transitions between states indicate legal sequences of method calls. Richetin et al. call this the *successor method* [160].

Whaley. As pointed out by Whaley et al. [209], the successor method has some drawbacks: Firstly, only knowing the last method cannot capture the proper behavior of an object in many cases, because its sequencing constraints are often more complex. Secondly, there may be methods that are state preserving; including these in the model destroys its accuracy, since state-preserving methods may be called in any state. They do not affect the further behavior because they do not change the state. As a consequence, Whaley et al. propose to use multiple FSA per component. Each FSA describes the protocol of only a subset of methods (*model slicing*), for example one that implements an interface or accesses a particular field. Whaley et al. also propose to ignore state-preserving methods during the construction of the automaton and to just annotate them to the modifier states. However, potentially important information is lost this way.

k-tails method. Biermann et al. [17] introduced the k-tails method. It starts with the prefix tree acceptor and then merges states that are indistinguishable in the set of accepted output strings up to a given length k. Steven P. Reiss [157] uses an extended k-tails algorithm for learning an FSA that represents a set of traces. The extensions deal with the creation of self loops (that is, edges for which source and target node are identical) for sequences of length $\leq k$ of the same symbol, and an automaton minimization step is performed at the end (see [157] for details). Reiss' approach is not originally intended for protocol recovery, but for compressing a trace. Nevertheless, the resulting FSA can as well be regarded as an interface's protocol when the trace consists of interface interactions only.

sk-strings approach. Raman and Patrick [152] modified the k-tails method for stochastic automata. They merge states that are indistinguishable for their top *s* percent of the most probable k-strings. A k-string does not need to end in an accepting state when it has length *k*. The result is a *probabilistic* FSA (PFSA) that is annotated with transition frequencies. Glenn Ammons [5] uses this approach to infer an automaton that represents the protocol. In a postprocessing step, a *corer* removes infrequently traversed edges from the PFSA. Ammons applied this technique to XWindows programs and found several errors in the use of XWindows components.

These are the approaches that are most closely related to the DOPG based approach. Other, more distantly related approaches to protocol recovery are not covered by the comparison, but are discussed in Chapter 9.

Differences to the OPG based approach. Since the presented approaches are solely based on the sequence of method invocations of the investigated component, they cannot distinguish between loops and repeated invocations of a method. Also, automaton learning requires (that is, delivers better results in the presence of) negative examples to prevent the resulting automaton from overgeneralizing [60], but these are never generated by real program runs of correct programs. On the other hand, generalization is necessary when recovering a protocol, because traces are only samples of all possible method invocation sequences. The problem here is to find the right compromise between generalization and specialization.

Another thing to be considered is that you usually do not know if a program is correct or not; it is therefore unknown whether an example is a positive or negative one. Protocol recovery approaches can only assume that the program is correct. The identification of errors is then transferred to the phase when the user manually inspects the recovered protocol.

The protocol recovery approach that is described in the next section is based on Object Process Graphs. Compared to the automaton learning approaches' input, which is just the call sequence information of a component, a Dynamic Object Process Graph contains more information: it describes the overall control flow of an application with respect to a single instance of a component (that is, one object). Object Process Graphs contain information about the sequence in which operations of the regarded object are being called or may be called. They also contain information about loops, which are quite important for a protocol. This makes OPGs a potentially good basis for protocol recovery.

7.4 OPG Based Protocol Recovery

The idea for OPG based protocol recovery is to transform a set of OPGs to a single protocol automaton. The different input OPGs represent different usages of instances of the same component. Figure 7.2 sketches the general idea for this transformation. The transformation uses well-known algorithms from automata

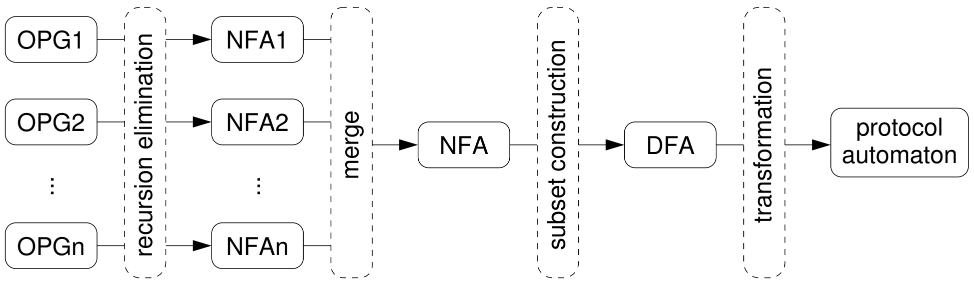


Figure 7.2: OPG to protocol automaton transformation overview.

theory. A short introduction to finite state automata, the used notation, and the necessary algorithms can be found in Appendix A. The foundations for the OPG based protocol recovery approach have been laid for static OPGs by Heiber [74] and Haak [65].

7.4.1 Algorithm

The general OPG based approach involves the following steps, which are independent of whether the OPGs are extracted dynamically or statically:

1. **Eliminate recursion** for each OPG. The transformed graphs contain only `create`, `atomic_call`, and `access` nodes. A transformation of labels from nodes to incoming edges is performed on each graph. The resulting graphs can then be regarded as non-deterministic finite-state automata (NFA). This step is explained in more detail in Section 7.4.2.

2. **Merge** all these NFA, accomplished by merging their start nodes. The result will usually be a highly redundant automaton because certain steps in using a component are always the same. This redundancy is reduced in the next steps.

3. **NFA to DFA.** This step uses the *subset construction* (see Appendix A or [81]) to get a deterministic finite-state automaton (*DFA*) that is equivalent to the given non-deterministic one, which means that it accepts exactly the same language. In our case, the algorithm has the effect that commonalities close to the `start` node are unified.

4. **Minimization.** Next, the automaton is minimized with respect to the number of states. The minimization algorithm (see Appendix A or [81]) finds all groups of states that can be distinguished by some input string. If two states cannot be distinguished, they are merged to a single state. This step tends to unify parts of the automaton that are close to the accepting nodes, since these naturally form one group.

5. **Additional transformations.** Optionally, further simplifying transformations may be applied, depending on the degree of generalization that is

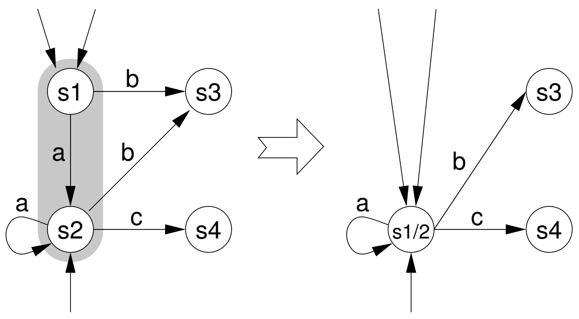


Figure 7.3: Simplifying transformation (a) example. States s1 and s2 will be merged because all transitions from s1 (a and b) lead to the same states as those from s2.

desired. For example, if the exact number of calls to the same method in a sequence is not considered interesting, this can be reduced to a simple loop.

Two different but related transformations are used in the case study (Section 7.6) as the 5th step:

(a) Merge two states *s*1 and *s*2 if there is a transition from *s*1 to *s*2 and the transitions emerging from *s*1 are a subset of *s*2's, considering transition event and target state. This means that *s*2 must have a transition to itself, labeled with at least the same events that lead from *s*1 to *s*2. Figure 7.3 shows an example for such a case. Practically, this means that we do not care about the minimum number of invocations of a method *a* before looping calls to *a*. For more complex components, this transformation may be extended to work on sequences of different method invocations as well (for example by common substring detection).

(b) The second transformation that is applied was introduced by Angluin [7] for inferring *zero-reversible languages*: if a state has two or more incoming transitions with the same label, the source states of these transitions are merged into one state. For the resulting automaton, it is always clear without ambiguity which state was the previous one, given the last input symbol. This is a further generalization of the previous transformation. In the case study in Section 7.6, the effect of both of these transformations on the protocol is investigated.

Depending on the type of transformation applied in step 5, previous transformations might have to be repeated. Transformations in this step are intended to be generalizing and thus change the language described by the FSA.

Steps 3 and 4 use well known techniques. They are explained in detail elsewhere [81]. The recursion elimination step is a bit more special and is discussed in the next section.

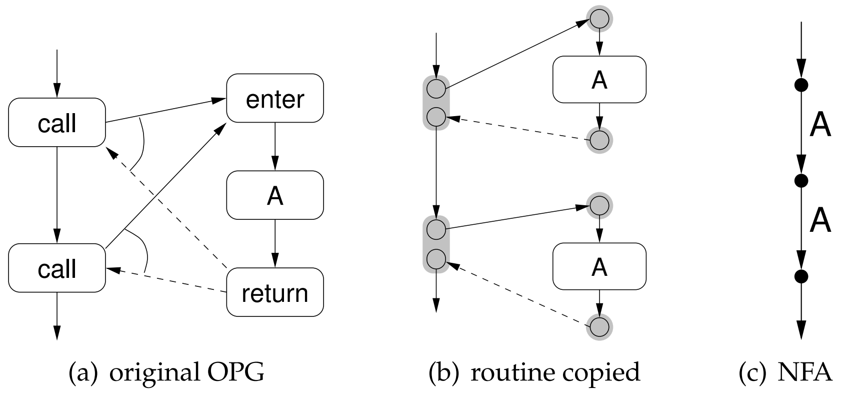


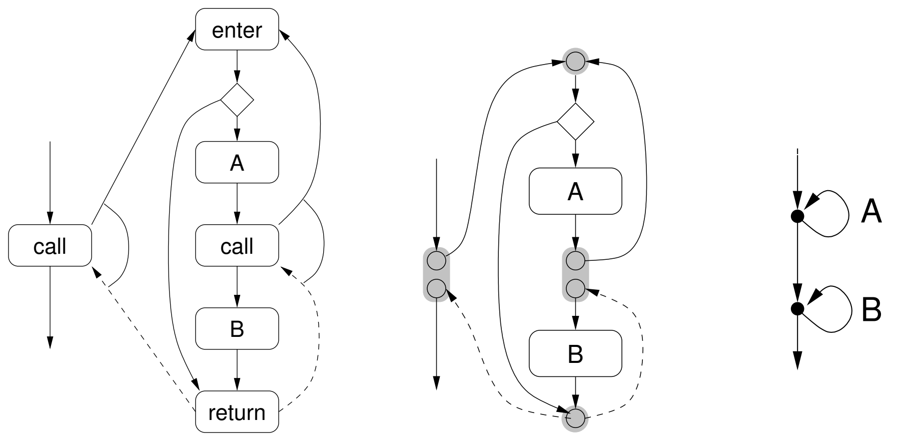
Figure 7.4: OPG to NFA transformation: Multiple invocations of routine A lead to copies (inlining).

7.4.2 Recursion Elimination

As discussed in Section 7.2, regular expressions or, equivalently, finite state automata are the notion of choice for most protocol validation approaches. In contrast to that, OPGs are capable of describing a context-free grammar. They contain routine calls and therefore allow recursion, which cannot be represented by regular expressions. Hence, to get from an OPG to an FSA, the first step is to eliminate recursion.

An intuitive way for eliminating recursion from an OPG is described by Haak [65] in his diploma thesis. Call nodes are split into two separate nodes, one connecting the incoming edge with the call edge, and the other connecting the return edge with the outgoing edge. In case of function pointers or dynamic binding, multiple call and return edges occur. For each invocation of a route, a copy of that routine's subgraph is inserted into the graph (*inlining*). The only exception is the recursion case, when the existing copy is used. This way, the mapping between call and return edges is maintained – otherwise, one would not know which return edge to take, resulting in unnecessary overgeneralization. Only in the recursion case, the call/return mapping is lost; this is the price we have to pay for using regular expressions.

Figure 7.4 shows how the repeated invocation of the same routine leads to copies of the routine in the resulting graph. If the invoked routine was not copied, this would result in a loop, and then any number of invocations would be allowed. In Figure 7.5, a recursive routine is resolved. Applying this transformation, the correspondence between call and return and the guarantee that there are as many calls as returns is lost. Note that the former return node has two outgoing edges. In the OPG, the choice of which return edge to take was unambiguously given by the corresponding call edge. In the resulting graph, the choice has become non-deterministic. This has the consequence that instead of A^nB^n, the resulting automaton accepts A^mB^n which is more general.



(a) original OPG (b) no recursion (c) NFA

Figure 7.5: Recursion elimination example. The original OPG creates sequences A^nB^n, the resulting NFA accepts A^mB^n $(m, n \geq 0)$.

The new intermediate nodes (resulting from `call`, `entry`, and `return` nodes) are then removed from the graph, along with all other nodes that are *not* atomic method calls, attribute accesses, or the `start` or `final` node. Finally, the graph is further transformed such that edges (instead of nodes) are labeled with the atomic method names or read/write accesses. This can be accomplished by moving each node's label to all incoming edges. The result can be regarded as a non-deterministic finite-state automaton (NFA): nodes become states, and edges become transitions. End nodes are transformed to accepting states.

7.4.3 Generalization

As discussed in Section 7.3, protocol recovery needs generalization. A certain amount of generalization is intrinsic for the DOPG based protocol recovery approach. Sources of generalization include:

- Loops for which the loop body always contains only one atomic method call or attribute access. This is not represented in the DOPG. Figure 7.6 shows a `for` loop that could cause this problem. The `read` method is called once for each object in the array. If `x` contains the regarded object only once, `read` is always called only once, but the loop still remains in the DOPG. In order to better handle this case, the number of invocations can be counted – but then, you do not know if the resulting set of counts is just due to the chosen test cases, or if it is complete. Therefore, this generalization should be accepted.

- Multiple invocations of the same method, when different operations are relevant for the object. This may happen for example for invocations with different parameters or a different environment. The different operations are then simply combined, although they may never occur together in one

```
for (int i = 0;
     i < x.length;
     i ++) {
  x[i].read();
}
```

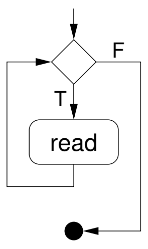


Figure 7.6: Overgeneralization caused by loops (when `x` contains the object exactly once).

call. This generalization could be prevented by adding context sensitivity in creating distinct copies for different method invocations, but this could lead to an explosion of the graph's size.

- Recursion elimination, as discussed in Section 7.4.2.
- Additional simplifying transformations (see Section 7.4.1). The first transformation (a) has two effects: for multiple method invocation, we lose the information whether a certain minimum number of invocations is required, and by the subset criterion, we allow additional transitions that were not possible before. Transformation (b) loses even more information, allowing further additional transitions.

All in all, there are many sources of generalization in the proposed protocol recovery process. The amount can be influenced by choosing more or less additional transformations in step 5, whereas the other transformations (steps 1–4) are mandatory. This makes the approach adjustable to the amount of generalization that is desired, depending on the intended use of the resulting protocol.

7.5 Comparing Protocol Automata

In order to compare results of the different protocol recovery techniques, it is necessary to compare the languages that are accepted by the recovered protocol automata. An exact comparison of languages is often not possible because they are infinite when loops are present. Therefore, alternative comparison measures that are based on the finite state automaton or regular expression representation have to be applied.

Lo et al. [122] compare two protocol automata A and B with a heuristic approach. The similarity between A and B is measured in terms of their generated sentences. Automaton A is used to generate random sentences, which automaton B tries to accept. The share of accepted words is then regarded as the precision of A with respect to B, or as the recall of B with respect to A. Since this approach is random based, it does not allow an exact comparison. The frequency at which the different words occur is unknown in an FSA and cannot be considered. However, this approach delivers a first approximation of similarity.

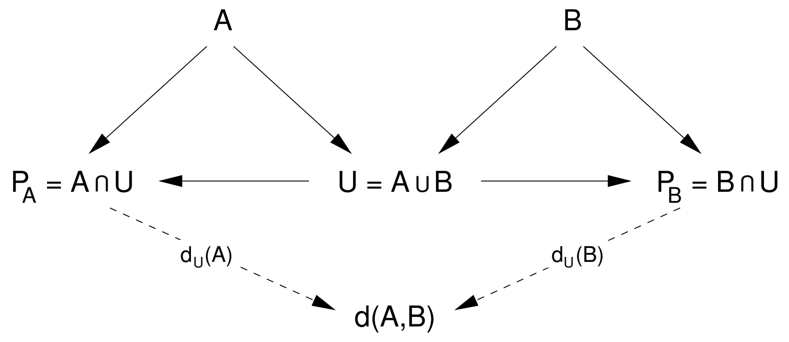


Figure 7.7: Illustration of the five involved automata in automaton difference calculation. Arrows denote data flow; solid ones are automaton usage, dashed ones are numbers.

I propose a new measure for automaton similarity that is inspired by Levenshtein's measure for string distance [110]. The new measure can be regarded as a kind of edit distance between two automata. The idea is to count the number of edge additions and deletions that are necessary to get from one automaton to the other. To do this counting in a defined (but not necessarily minimal) way, we first need a unified representation of both, which naturally is the union of the two automata. Each of the original automata can then be found in the union automaton by following the transitions in the original and the union automaton in parallel, which corresponds to the product automaton construction [81]. The product automaton describes the intersection of both automata, that is, their common language. The distance between each of the two automata and their union is calculated based on the information from product automaton calculation. Finally, the two distances are combined, resulting in the difference between the two original automata. Figure 7.7 illustrates the data flow and the involved automata of the metric.

The measure is calculated as follows. Let $T(X)$ denote the number of transitions in automaton X.

1. Input: two deterministic and minimal automata A and B.
2. Calculate the deterministic and minimal union U of A and B (see steps 2–4 of the recovery algorithm in Section 7.4.1).
3. While calculating the product automata P_X of X and U (for $X \in \{A, B\}$), that is, following the transitions in both automata in parallel, count
    - the number of transitions n_X in U that are never taken. Such unused edges correspond to deletions.
    - the number of transitions t_X in P_X which are the result of taking a transition in U more than once. This corresponds to insertions.

   The algorithm for this step is defined in Figure 7.8, and it is explained below.

```
**Input**: two minimized DFA $A_L = (Q_L, \Sigma, \delta_L, q_L, F_L)$ and
        $A_U = (Q_U, \Sigma, \delta_U, q_U, F_U)$ with $L(A_L) \subseteq L(A_U)$
$R := \emptyset$;  $T := \emptyset$;  $t_x := 0$;  $Q := \{(q_L, q_U)\}$
while $Q \setminus R \neq \emptyset$ do
    select $r = (p, q) \in Q \setminus R$
    $R := R \cup \{r\}$
    foreach $a \in \Sigma$ do
        if $\delta_L(p, a)$ is defined and $\delta_U(q, a)$ is defined then
            if $r \in T$ then
                $t_x := t_x + 1$
            $s := (\delta_L(p, a), \delta_U(q, a))$
            $Q := Q \cup \{s\}$
            $T := T \cup \{(p, a)\}$
$n_x := |\text{domain}(\delta_U)| - |T|$
```

Figure 7.8: Calculation of t_x and n_x, based on lazy product automaton calculation.

4. The distance between $X \in \{A, B\}$ and U is

$$d_U(X) := \frac{n_X + t_X}{T(P_X) + T(U)} \tag{7.1}$$

5. The difference between A and B is

$$d(A, B) := \frac{d_U(A) + d_U(B)}{2} \tag{7.2}$$

The algorithm for calculating t_x and t_n is shown in Figure 7.8. It is based on lazy product automaton construction. The states of the product automaton consist of pairs of states from the two original automata. Starting from the pair of start states, the algorithm checks for each input symbol if it is accepted by both automata in the current state, and if so, finds out to which states of the two automata they lead. The pair of target states is then the target state of the transition in the product automaton. T keeps track of the edges of U that have already been used, and this information is used to calculate t_x. The value of n_x is then the number of edges in U that have not been used.

Figure 7.9 shows an example for distance calculation between an automaton A (left) and the union U (top) of A and another automaton B (B is not shown). The product automaton P_A is located in the center. Construction of P_A starts at the start nodes of A and U. From here, `create` is accepted by both automata, so this is part of the product automaton. From the next state, `push` is accepted by

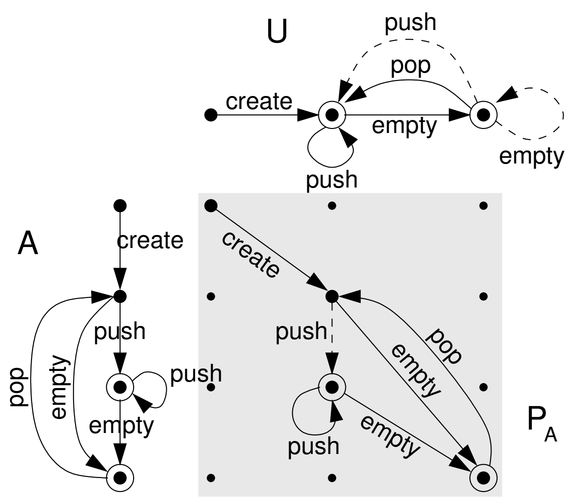


Figure 7.9: Example for automaton difference calculation. The top automaton is U (dashed transitions are never taken), the left one is A, and the central one is P_A (dashed transition inserted by taking the push transition a second time).

both automata. For U, this leads back to the same state (self-loop), which is not the case for A, giving $t_A = 1$. Two edges of U are never used, which means that $n_A = 2$. This results in a difference between A and U of $d_U(A) = \frac{2+1}{6+6} = 0.25$.

This metric is non-negative and symmetric. For a distance metric, a third property is required: triangle inequality. Unfortunately, this property is difficult to prove due to the complexity of the underlying algorithms. However, I did not find any counterexamples. The result of the metric is a value between zero and one, indicating the degree of differences between the two automata: zero indicates identity, and one indicates that they are completely disjunctive. The maximum value is reached when only one transition of U is taken repeatedly, and all other transitions are not taken. This gives $n_X = T(U) - 1$ and $t_X = T(P_X) - 1$, which makes $d_U(X)$ converge to 1 (given an increasing number of transitions).

7.6 Case Study

In this case study, the different protocol recovery techniques that were described in Sections 7.3 and 7.4 are applied with the same input data, and the resulting protocol automata are compared to each other. In particular, the following questions are addressed:

- How close do the different approaches come to the real specification?
- How do the DOPG based protocols compare to other approaches' results? Are they more general, equivalent, or more specialized?
- Which of the two introduced simplification steps are useful for DOPG based protocol recovery?

No.	System	Lang.	KLOC	Component	AP	Inst.	AMC
1	ArgoUML	Java	264	`ArrayList`	66	346	13,817
2	ANTLR	Java	38	`ArrayList`	2	29	1,291
3	J	Java	158	`ArrayList`	13	37	36,426
4	ArgoUML	Java	264	`Stack`	2	7	352
5	J	Java	158	`Stack`	2	13	943
6	sqlite3	C	60	file handle	1	2	253
7	rhapsody+ircII	C	18+49	socket	2	4	8,412

Table 7.1: Characteristics of the different components as used in the respective subject system: number of distinct allocation points (AP), number of investigated instances, and number of atomic method calls (AMC).

- How does the automaton difference metric compare to Lo's metric?

7.6.1 Setup and Subject Systems

For this experiment, several Java and C applications and different components they use are investigated. The case study concentrates on standard components: `java.util.ArrayList` and `java.util.Stack` for Java, and the file or socket handle for C. The following subject systems are used:

- ArgoUML[3], a UML modeling tool (Java)
- ANTLR[4], a parser generator (Java)
- J[5], a powerful text editor (Java)
- sqlite3[6], an SQL database engine (C)
- rhapsody[7] and ircII[8], two console IRC clients (C)

These systems are first instrumented to produce the necessary traces. Then the instrumented systems are executed with typical usage scenarios. From the resulting trace files, one object trace is extracted for each instance of the regarded component. These object traces are then used to construct the corresponding DOPGs, which are the basis for the new protocol recovery approach. The automaton learning approaches require a simpler trace that contains just the sequence of atomic method calls. This can be easily extracted from the object trace. Based on these two representations, protocol recovery is performed with the different approaches. Figure 7.10 illustrates the general setup.

[3]`http://argouml.tigris.org/`
[4]`http://www.antlr.org/`
[5]`http://armedbear-j.sourceforge.net`
[6]`http://www.sqlite.org/`
[7]`http://rhapsody.sourceforge.net/`
[8]`http://www.eterna.com.au/ircii/`

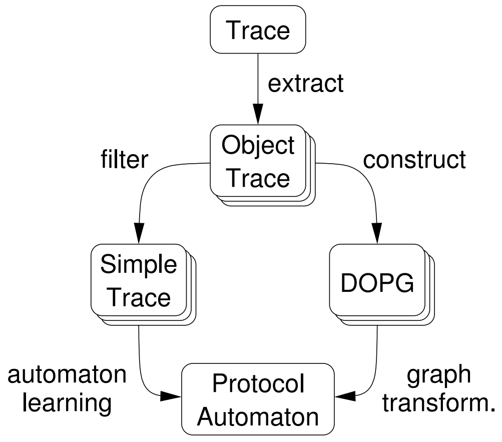


Figure 7.10: Case study setup.

Table 7.1 shows which components of which applications are investigated, how many instances occur in each case, and how often the component's methods are called in total. It also shows the number of distinct allocation points. For the socket component, rhapsody's and ircII's traces are combined to get more information about the general usage of a socket.

The following protocol recovery approaches and tools are applied (see Section 7.3 for details):

- Tree: The minimized prefix tree acceptor that represents exactly the set of original traces. This is used to evaluate the other approaches' degree of generalization.
- Successor: The straightforward approach of using one state per method. For this method and for the "Tree", I use an own implementation that additionally performs automaton minimization at the end.
- k-tails: Reiss' tool is used (with k=3), which is available as part of the Bloom system[9].
- sk-strings: The tool by Anand Raman is used (the same that Ammons uses).
- DOPG: Pure DOPG based protocol recovery with no additional simplifications. This leaves out step 5 of the algorithm from Section 7.4.1.
- DOPG-A: Based on DOPG, this one additionally applies the simplifying transformation (a) as described in step 5.
- DOPG-B: Additionally applies transformation (b) in step 5.

This results in 7 automata per component/system that are to be compared. Unfortunately, there is no official specification for the regarded components available. Therefore, I created the real specification of the protocol as I would expect

[9]`http://www.cs.brown.edu/~spr/`

it. The specification is based on protocol recovery results, which were manually inspected and manipulated towards the real protocol. This real specification is represented as an 8th automaton and used as a reference. For all these automata, the number of states and transitions was measured to get an impression of their size.

Each of these automata is then compared to the specification. For this comparison, the specification is reduced to those methods that are really used in the respective traces. It is reasonable to do so because methods that are not called are not visible for any dynamic analysis. The comparison is performed by applying both introduced similarity measures: random-based precision/recall approximation, and automaton difference. Also, the correlation between these two measures is calculated.

7.6.2 Results

Tables 7.2, 7.3 and 7.4 show the results of the study, and Figures 7.11 and 7.12 present them graphically. This section discusses the results in detail.

Automaton sizes. In Table 7.2, the sizes of the different resulting protocol automata in terms of number of states and transitions are displayed. The sizes of the specification and prefix tree automata are given for comparison purposes. The table shows that all approaches reach a good compression ratio compared to the prefix tree acceptor. However, sizes still differ in a wide range. In five out of seven cases, the DOPG based automaton is the largest. With additional simplifications, especially with transformation 5(b) (see Section 7.4.1), the DOPG based automata can be reduced to a size comparable to the other approaches. Only for case 3, the DOPG-B automaton is still very large. In summary, DOPG based protocols appear to remain more specialized than the other approaches' protocols.

Automaton/language similarity. Tables 7.3 and 7.4 show the results for language and automaton comparison of each recovered protocol automaton with the specification. For each case, the first and second columns contain the precision and recall as calculated with Lo's method (100,000 samples). These numbers are visualized in Figure 7.11. The third column displays the automaton's difference to the specification automaton; Figure 7.12 is the corresponding chart. The last line shows the Pearson product-moment correlation coefficient between precision and distance (prec.) and recall and distance (rec.).

The chart in Figure 7.11 gives a good overview on the different precision and recall measures. It is desirable to achive both high precision and a high recall. k-tails comes closest to this for some cases, but in other cases, precision or recall may also drop below 20%. DOPG-B delivers similar results. sk-strings, Succ, and Tree always have a recall below 30%, whereas precision is mostly quite high. DOPG-B seems to reach a higher recall at the price of a lower precision, compared to DOPG and DOPG-A. Let us now look at the results in detail.

- For system/component no. 1, k-tails is closest to the specification and also delivers a 100% recall automaton. The successor method results in a higher

No.	Spec.	Tree	Succ.	k-tails	PFSA	DOPG	DOPG-A	DOPG-B
1	4:31	6k:6k*	19:64	3:19	42:126	233:687	232:684	20:50
2	4:17	259:273	12:18	8:12	20:40	43:61	36:46	20:28
3	3:10	34k:34k*	6:14	2:6	13:42	475:760	469:754	307:465
4	4:8	334:334	7:9	4:6	8:10	27:38	26:37	6:8
5	4:7	871:872	5:7	3:5	21:25	4:7	3:5	3:5
6	4:8	241:241	8:14	4:8	8:14	24:54	22:49	3:8
7	4:12	8k:8k*	13:33	9:19	49:73	34:80	23:50	14:30

Table 7.2: Protocol automaton sizes (states:transitions) for the different approaches. Specification and prefix tree sizes are given for comparison. The automata marked with a * have not been minimized due to memory restrictions.

No.	1			2			3		
Tree	100	32	n/a	100	15	48	100	28	n/a
Succ.	87	33	30	88	15	28	83	34	30
k-tails	42	100	17	75	17	25	75	100	17
PFSA	3	29	66	93	16	33	4	25	56
DOPG	37	33	42	95	15	36	79	36	62
DOPG-A	37	32	42	80	20	34	79	36	62
DOPG-B	17	38	30	83	22	29	85	65	58
prec./rec.	-75	-82		-4	-87		-40	-79	

Table 7.3: Automaton/language similarities when compared to the real protocol automaton: precision, recall, and difference. Some values could not be computed due to memory limitations (n/a). The last line shows the correlation between precision/recall and automaton difference (all values in percent).

No.	4			5			6			7		
Tree	100	43	54	100	21	51	100	0	50	100	32	n/a
Succ.	100	44	21	75	23	10	100	1	18	89	33	26
k-tails	50	48	25	67	100	21	100	100	6	16	100	28
PFSA	100	44	22	100	21	35	100	1	18	3	29	31
DOPG	100	43	43	89	57	25	75	1	17	63	33	34
DOPG-A	100	43	42	67	100	16	75	1	17	63	33	36
DOPG-B	40	47	32	67	100	16	58	100	6	29	35	34
prec./rec.	3	-74		37	-61		24	-69		-82	-72	

Table 7.4: Continuation of Table 7.3.

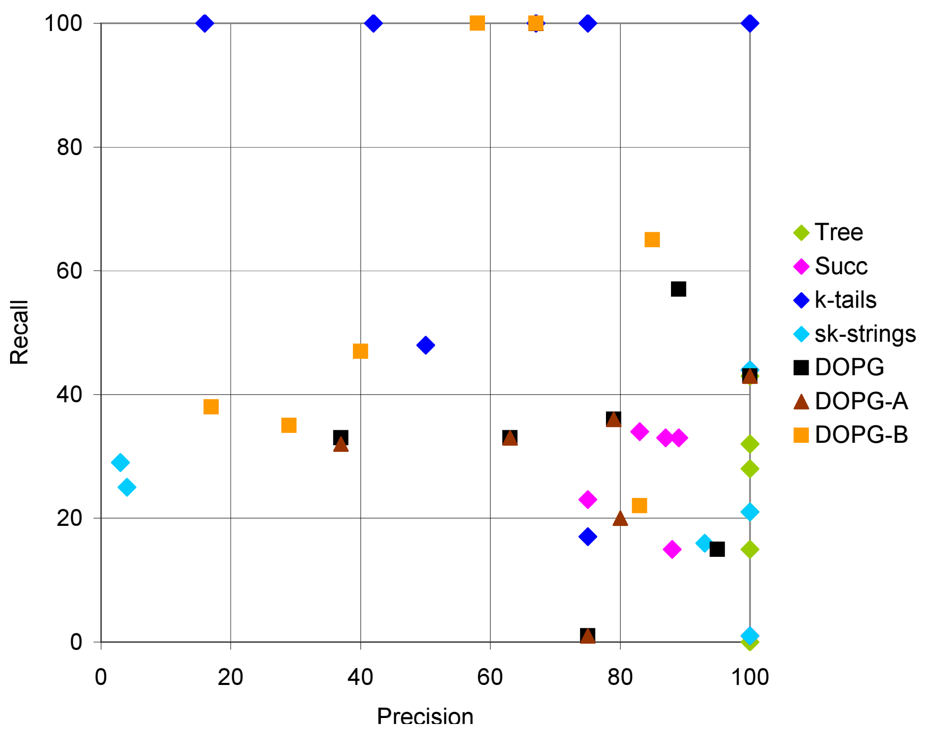


Figure 7.11: Precision and recall of different techniques.

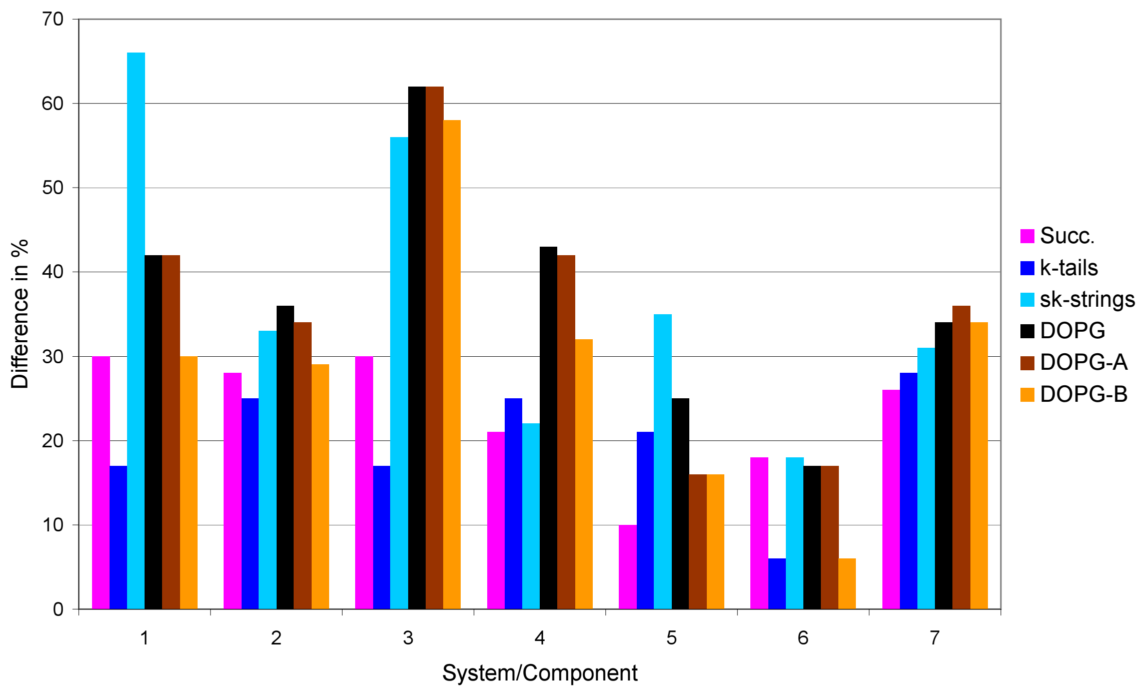


Figure 7.12: Automaton differences between specification and recovered protocol for different recovery techniques.

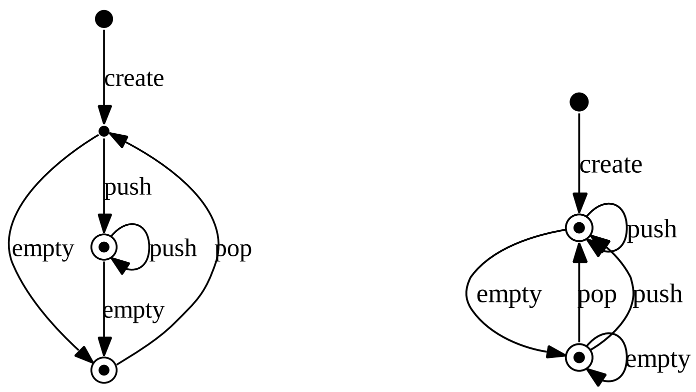
precision, but its recall is not higher than the prefix tree acceptor's. Both are better in quality than the DOPG based protocols, but those are still better than PFSA.

- For no. 2, DOPG has the highest precision, while DOPG-B has the highest recall and the lowest difference to the specification.
- In no. 3, precision and recall for DOPG-B are both quite good. k-tails has a higher recall at the price of a lower precision. However, the DOPG based automaton is about factor 100 larger than the k-tails result.
- For no. 4, results do not vary much between the different approaches.
- In no. 5, PFSA reaches the highest precision but the lowest recall (same as prefix tree acceptor). k-tails and DOPG-A/B have the same precision and recall; they result in different automata of the same size. This is indicated by the measured difference to the specification.
- In no. 6, k-tails results exactly in the specification. Most other approaches have a very high precision and very low recall, except DOPG-B which has a lower precision but 100% recall.
- For no. 7, the successor method has the highest precision, while k-tails delivers the highest recall. DOPG is still better than PFSA in this case.

All in all, the results are diverse. In some cases, DOPG provides the best quality protocol automata, but in other cases, other approaches have better results. k-tails is often close to the specification, but also often generalizes too much. An interesting observation is that in many cases, the approaches reduce precision but do not increase recall. This means that generalization in these cases does not extend the accepted language towards the specification. Instead, the automaton's size is reduced at the price of decreased quality.

There is no correlation between automaton difference and precision, but between measured automaton difference and recall. The correlation is in the range from -0.61 to -0.87. This means that the more the difference to the specification increases, the fewer allowable sequences of operations are accepted by the automaton.

DOPG variants. Regarding the differences between the different DOPG variants, there are a few more things to observe. With an increasing degree of generalization, recall increases and precision drops. However, this is not true in all cases. The difference between DOPG and DOPG-A is small in most cases, and the languages accepted by the respective automata do not differ much. Only the transformation to DOPG-B reduces the automaton's size significantly in all cases, also affecting precision and recall (increased recall, reduced precision). In summary, DOPG-A does not have a great effect and can therefore be omitted in practice.



(a) Successor (b) DOPG

Figure 7.13: Two protocol automata for system/component no. 5 (constructor calls omitted). The successor automaton accepts a subset of the DOPG automaton's language.

Example. Figure 7.13 shows the protocol automata for scenario 5 that result from the successor and DOPG algorithms. The DOPG automaton allows `push` and `empty` after an `empty`, which are both not allowed in the successor one. The language of the successor automaton is completely accepted by the DOPG automaton. Both automata show that there can never be a `pop` without a preceeding `empty`. This example demonstrates that, although similar in size, the accepted languages of the automata might differ.

7.7 Summary

This chapter has described a novel dynamic protocol recovery approach. Using Dynamic Object Process Graphs as the basis, a protocol automaton is derived by a sequence of transformations. Compared to using the traditional prefix tree acceptor, DOPGs have the advantage of containing information about loops and about context.

The case study showed that the approach is applicable in practice. It also showed that the resulting automata are usually more detailed than the other approaches' results. They are closer to the application, which may be helpful for program understanding. On the other hand, this is not necessarily an advantage when the goal is recovery of the specification. However, when incorporating simplifying transformations, the resulting protocols were often better than the other protocol recovery approaches' results.

Also, a new metric for comparing automata was introduced. In the case study, this metric indicated differences between automata where other metrics did not find any difference. On the other hand, a correlation between this metric and the measured recall as defined by Lo and Khoo could be detected.

All in all, DOPG based protocol recovery is a promising approach. It provides a concrete detailed protocol, which may be adjusted to the desired level of generalization by choosing appropriate simplifying transformations.

Chapter 8

Supporting Program Understanding by Visualized DOPGs

Dynamic Object Process Graphs have been introduced as a view on a program's control flow graph from the perspective of a single object. This representation is intended to provide an understanding of how a component is being used in an application. As the case studies in Chapter 6 illustrated, such graphs may even give a first impression of the entire application's structure – if the investigated object is of central concern for the application. In particular, we investigated two IRC chat clients and found that DOPGs for the socket object give a good impression of the application's overall structure. A similar result was obtained for the file handle object of a database system.

This raises the question of whether such graphs are helpful for program understanding in general, or which tasks can be efficiently supported by using them. To shed some light onto these issues, I conducted a controlled experiment. The design, procedure, and results of this experiment are described in the following.

8.1 Description of the Experiment

8.1.1 Hypotheses

I pose the following **research questions**:

- Does the availability of visualized DOPGs lead to faster answers to program understanding questions that are in some way related to a given component?
- Are these questions answered less erroneously?

My **hypothesis** is that both questions can be answered positively:

$\boxed{\text{H1}}$ When DOPGs are available, program understanding questions that are related to a given component are answered faster.

Rationale: DOPGs represent an application's control flow from a single component's perspective. This information should be helpful for understanding how the component is being used. The maintainer can focus on those spots in the code that are relevant and sees their relation immediately. He does not have to browse through a lot of code in search for uses of the component or for the connection between these uses. DOPGs also contain call paths to the points where the component is used, which should give a good impression about the basic structure of relevant parts of the application. This should significantly reduce the time needed for finding out about component-related issues.

$\boxed{\text{H2}}$ When DOPGs are used, such questions are answered less erroneously.

Rationale: When the DOPGs are suited to the given question, that is, represent an adequate use case, they provide detailed information about the component's relevant use. Since they concentrate all the neccessary information, the maintainer should get a very good understanding of the dependencies of the component. Consequently, he should be able to gain a deeper understanding of the component's use and to answer corresponding questions less erroneously.

The corresponding null hypotheses are:

$\boxed{\text{H0}_1}$ It does not make a difference for the response time to the questions whether DOPGs are available or not.

$\boxed{\text{H0}_2}$ There is no difference between the number of errors in the answers, no matter if DOPGs are available or not.

8.1.2 Experiment Design

In order to check these hypotheses, I let two groups of subjects solve the same tasks. One of these groups (*control group*) worked with standard techniques only, while the other group (*experimental group*) additionally used DOPGs. The **independent variable** – the variable that is subject to controlled variation – in this controlled experiment was the availability of visualized DOPGs. The experiment basically has a *between-participants after-only design* [29]. This means that each participant was either member of the experimental group *or* member of the control group.

Due to the relatively low number of participants that was to be expected, a *within-participants* design – where each participant is member of each experimental group – would have been preferable, but was difficult to implement in this case. Giving the same program understanding task twice would obviously lead to strong sequencing effects. It is difficult to find two different subject systems that are equivalent with respect to the tasks, or to find two different but still comparable tasks. Therefore, I decided to let each participant investigate two different systems with different tasks: for one system, the participant was assigned to the experimental group, and for the other system, he was assigned to the control group. The order of the two systems and which of the systems had DOPGs

Group	Size	System 1	DOPG	System 2	DOPG
1a	6	Gantt	yes	Argo	no
1b	7	Argo	no	Gantt	yes
2a	6	Gantt	no	Argo	yes
2b	6	Argo	yes	Gantt	no

Table 8.1: Experimental (sub)groups and sizes: Order of systems and DOPG availability.

available was randomly assigned for each participant. This setup results in four combinations and therefore four subgroups as shown in Table 8.1. It can be considered as performing two interleaved between-participants experiments.

The **dependent variables** measure the effect of controlled variation of the independent variable and should help answering the research questions. In this case, dependent variables were:

- the time needed to answer each question,
- the correctness of these answers, and
- the subjective user satisfaction/confidence/productivity.

The first variable was measured while the tasks were being performed, the second one was determined after the experiment had finished, and the third one was acquired through a post-study questionnaire.

Relevant **extraneous variables** were:

- the participant's experience in programming and software maintenance, in particular in Java;
- participant's familiarity with the subject systems;
- participant's familiarity with the Eclipse environment;
- experimenter effects: how the participants are instructed, what the experimenter expects from the experiment, and so on;
- instrumentation: how the dependent variables are measured.

Differences between most of these extraneous variables were cancelled out by randomized assignment of participants to groups. Experimenter and instrumentation effects and other threats to validity are discussed in Section 8.1.5.

8.1.3 Subjects

All participants were computer science students from the University of Bremen who had already received the intermediate diploma ("Vordiplom"), which means that they had all participated in a one-year project with 5 to 7 people and developed a Java system of several thousand lines of code. Another precondition they had to meet was basic knowledge in using Eclipse for Java development. 35 students replied to a call for participation that was sent out to all computer science students at the University of Bremen. 27 of these participated in the experiment. Two of them took part in a pilot experiment which was conducted to adjust tasks and timing of the experiment. This left 25 participants for the main experiment. Participation was voluntary. The participants did not receive any valuable consideration, with the exception of their participation in a lottery. This lottery awarded two cash prizes to two of the participants who were choosen by lot after the experiment was finished.

As the pre-study questionnaire (see Appendix E.1) showed, the subjects had between 1 and 10 years (one outlier had 25 years) of programming experience (median 5, std. dev. 2.7) and between 1 and 8 years of Java experience (median 2.5, std. dev. 1.4). 60% of the participants rated their own programming capabilities as being "above average". The largest system the students had worked on before was between 1 and 200 KLOC in size (median: 17 KLOC). 21 of the students reported maximum sizes above 10 KLOC. Half of the participants did not have much experience in working on systems written by others, but the other half did. The distributions of these properties were similar in the two experimental groups (experimental/control group per system). This was the result of randomization; it was not actively enforced (no *merging*).

8.1.4 Experiment Tasks

To make DOPGs available to programmers in practice, I created a DOPG viewer Eclipse plugin. This enables the subjects to view DOPGs integrated in an established environment. This has several advantages: the usual functions of an integrated development environment are readily available, in particular cross-reference capabilities, search functions, and the code browser. Also, most users will already be familiar with these Eclipse features or a similar environment and can work on a project as they usually do.

The plugin has the following features: load DOPG, perform a spring layout algorithm, zoom in and out, stretch and tighten layout, pan (drag the viewport), move nodes, find *start* node (start of application) and find *create* node (object creation). Another very important function is the possibility to jump to the corresponding source code location of a node by double-clicking on it. A screenshot of the plugin is shown in Figure 8.2 on the right-hand side. Due to the used graph visualization framework, the graphs look slightly different from they way they were shown in the previous Chapters.

The choice of subject systems was performed based on the following considerations:

- Complex/large enough to be realistic.

System	Files	LOC	SLOC
Jetris	12	1,885	1,527
GanttProject	475	61,892	43,100
ArgoUML	1,725	319,797	160,509

Table 8.2: Source code measures of the subject systems. Jetris was only used for training. SLOC excludes empty and comment lines.

System	Class	APs	nodes DOPG	edges
Jetris	Figure (subclasses)	14	16/21/94	21/31/144
GanttProject	GanttTask	3	66/293/661	84/409/963
ArgoUML	ClassDiagramGraphModel	1	167	237

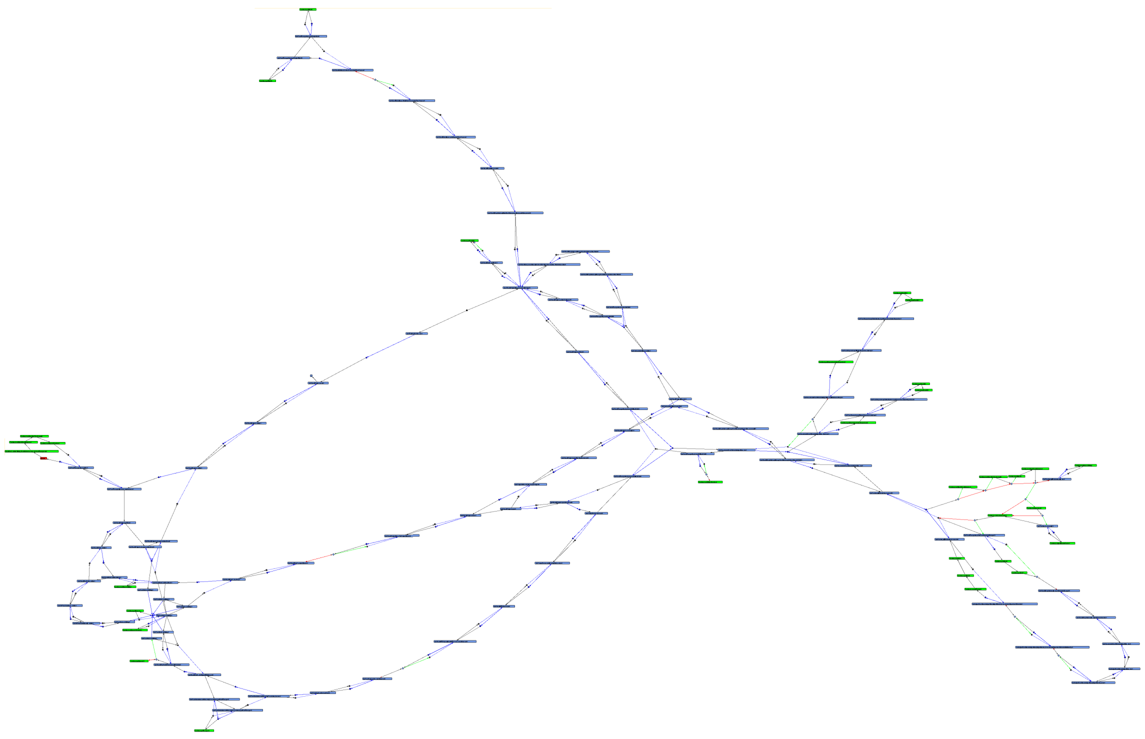
Table 8.3: DOPG measures of the subject systems. The left part of the table shows the preselected relevant class and the number of used allocation points, and the right part contains the corresponding DOPG size measures (minimum/median/maximum counts).

- Application is roughly known to the participants from a user's perspective.
- Code is unknown to the participants.
- Written in Java, since this is the language that all students are familiar with.
- Must be executable (for dynamic analysis).

I chose two subject systems which meet all these requirements: GanttProject[1], a project planning tool, and ArgoUML[2], a tool for drawing UML diagrams. The Tetris game Jetris[3] was used for training. Table 8.2 shows the characteristics of these systems. LOC is the physical lines of code, while SLOC is the number of source lines of code as counted by `sloccount`[4] (that is, excluding comments and empty lines). ArgoUML is four to five times the size of GanttProject, but it also seems to have a better comment ratio.

For each subject system, I identified one class that could be considered as being of central concern to this type of application. For GanttProject, I chose the `GanttTask`, which represents a task in a project. For ArgoUML, I chose the `ClassDiagramGraphModel`, which defines a binding between the underlying graph model and the graph editor. Finally, for Jetris, the `Figure` class was selected. These selections were purely based on investigating the class names and choosing classes that sounded promising. Table 8.3 shows the number of corresponding allocation points ("APs"), that is, the number of locations where instances of these classes are created.

[1]`http://www.ganttproject.biz/`
[2]`http://argouml.tigris.org/`
[3]`http://jetris.sourceforge.net`
[4]`http://www.dwheeler.com/sloccount/`


(a) GanttProject

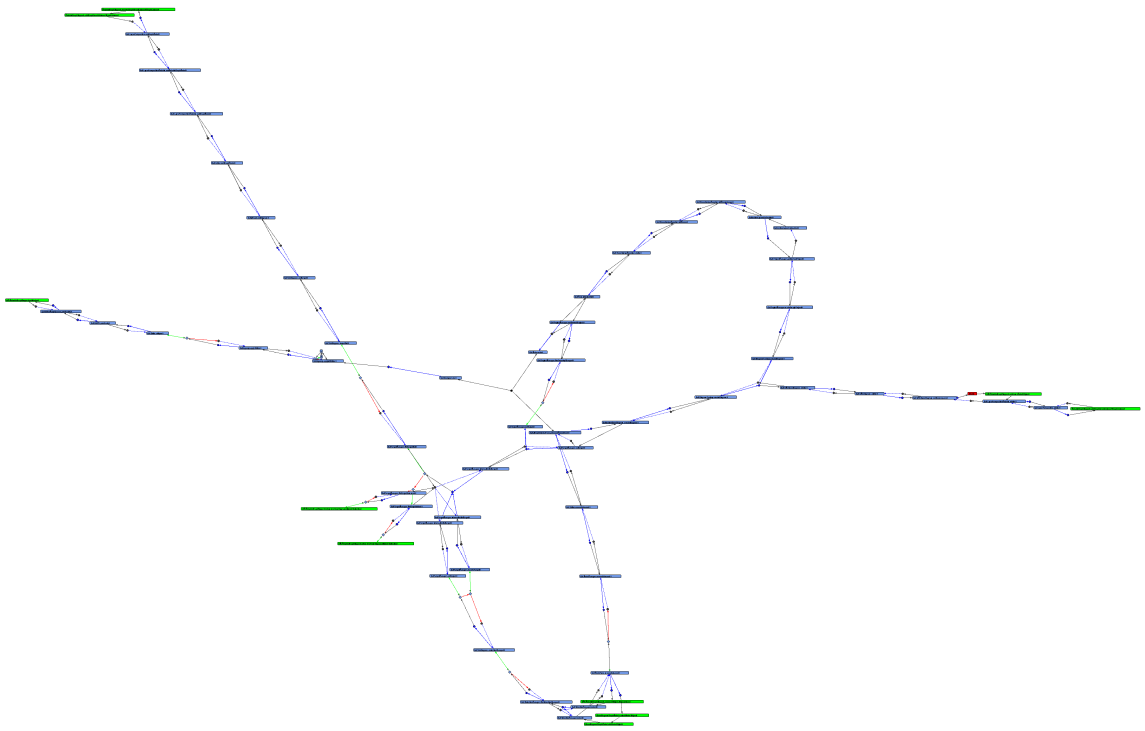

(b) ArgoUML

Figure 8.1: DOPGs from the experiment as shown by the Plugin.

To extract DOPGs for the selected classes, use cases were executed for each of the systems. For GanttProject, a project file was loaded and a task's duration was changed. For ArgoUML, two classes were created using the (empty) class diagram editor. Then, they were connected by an association. Jetris was played for about a minute. The sizes of the resulting DOPGs are also given in Table 8.3. The extracted DOPGs are shown in Figure 8.1 to give an impression of their structure.

In order to be able to maintain an unknown system, an engineer first has to understand it. If he is asked to extend or fix a certain feature, he first has to locate it. Feature location is a program understanding task that involves more than just telling a code location: the maintainer has to gain a basic understanding of certain parts of the system as well. Therefore, this kind of task can be considered a good representative for program understanding, and it is employed in the experiment.

The concrete tasks that had to be performed on GanttProject were the following:[5]

1. GanttProject supports hierarchical tasks. Which code is responsible for updating the length of a parent task when a child task's duration is changed?
2. Which component is responsible for drawing the task (box) in the Gantt diagram?
3. Which class keeps the information about dependencies between tasks?

All three tasks are related to the GanttTask class: the first and third one are about the relation between tasks, while the second one is about drawing a task. Tasks 1 and 2 have to do with the GUI: the change of a task duration is triggered by user action, and the task box is drawn for display to the user. The third question is about the internal data model.

For ArgoUML, the following questions were posed:

1. Which code has to be changed to make ArgoUML show an empty sequence diagram instead of an empty class diagram after startup?
2. How is the user's addition of an element to a diagram (for example, adding a class to a class diagram) implemented?
3. Which class is responsible for recording and keeping a history of selections?

These three tasks have a relation to the ClassDiagramGraphModel: The model must be present when a class diagram is displayed or edited. Therefore, it is involved when an empty class diagram is created (task 1). Adding an element to the diagram must also be reflected in the model (task 2). The third task is about selection management, which is more loosely related. All tasks have to do with the GUI.

It should be noted that the first task is of a kind that cannot be solved by the standard feature location technique of building the difference between two sets of

[5] Appendix D contains translations that are closer to the original (German) task descriptions.

Introduction	10 min
Training (Talk)	15 min
Training tasks	25 min
Experimental task 1	25 min
Questionnaire	5 min
Experimental task 2	25 min
Questionnaire	5 min
Debriefing	10 min

Table 8.4: Session overview.

executed units [44, 99]: the feature is *always* executed and will therefore be present in all sets.

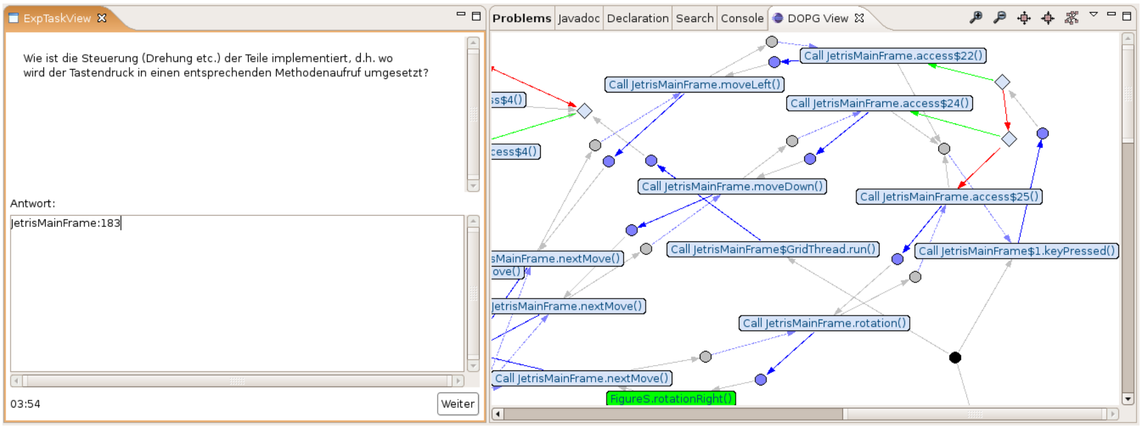
In summary, I tried to define the tasks for the two systems in a way that makes them similar in nature and therefore at least approximately comparable.

8.1.5 Experimental Procedure

The participants were scheduled for one of a set of seven sessions. This was neccessary because only four identical workstations were available for the experiment. Each session participant was randomly assigned to one of the four different subgroups. This gave every participant the same 25% chance of being member of any of these groups. Each session lasted for two hours and had a structure as shown in Table 8.4. In each session, the participants were first given a short introduction, followed by a training of relevant methods. This covered the general purpose of program understanding (why it is necessary), general concepts of static and dynamic analysis, concrete approaches such as top-down and bottom-up program understanding or searching for key words in the code, using basic capabilities of Eclipse, such as code browsing, and demonstrated the *File* (textual) and *Java* (cross reference) search functions in more detail. The participants were also trained about DOPGs, that is, their meaning, how to read and use them, and how to use the DOPG Eclipse plugin (see Figure 8.2). This training was conducted in approximately the same way for each participant group. This was achieved by using identical slides and an experimenter's handbook (see Appendix C) that described in detail what had to be done and said. However, questions of the participants did of course differ. Also, the experimenter was always the same person – me.

After that, each participant executed a set of six training tasks to get familiar with the environment and views (see Appendix D). The tasks were similar to those that were given in the main part, but were much easier to accomplish due to the small size of the training system. The first three tasks were in parallel demonstrated by the experimenter. He showed how to solve each task in two different ways: by using Eclipse search functions, and by using the DOPG plugin.

For the main part, participants were told that they should use as much time as they needed for each task. They were not told in advance how many tasks


ExpTaskView
Problems
Javadoc
Declaration
Search
Console
DOPG View
Weiter
Call JetrisMainFrame.access$22()
Call JetrisMainFrame.moveLeft()
Call JetrisMainFrame.access$24()
Call JetrisMainFrame.moveDown()
Call JetrisMainFrame.access$25()
Call JetrisMainFrame$GridThread.run()
Call JetrisMainFrame$1.keyPressed()
Call JetrisMainFrame.rotation()
Call JetrisMainFrame.nextMove()
JetrisMainFrame.nextMove()
sMainFrame.nextMove()
FigureS.rotationRight()

Wie ist die Steuerung (Drehung etc.) der Teile implementiert, d.h. wo
wird der Tastendruck in einen entsprechenden Methodenaufruf umgesetzt?

Antwort:

JetrisMainFrame:183

03:54

Figure 8.2: Eclipse plugins: The left view guides the participants through the experiments, giving information and tasks and accepting their answers. The right view is the DOPG viewer.

would be given, and they were not allowed to go back to a previous task once they had finished it. Before the first experimental task started, the experimenter demonstrated the two use cases of the two subject systems which had been used to generate the DOPGs. All participants were also told which was the chosen relevant class for each system.

During execution of the experimental tasks, measures of the dependent variables were recorded. These measures had to be taken in exactly the same way for each participant. I created and used an additional Eclipse plugin[6] that automatically led through the experimental tasks, presented the questions, and recorded the subject's answers and reaction times. Figure 8.2 shows how the corresponding Eclipse view looks. After each series of experimental tasks (one system), there was a post-study interview (see Appendix E). The session closed with a debriefing. Among other things, participants were told at the end that they are requested not to talk about the experiment to future participants until the series of experiments is finished.

During each session, the screen was recorded in order to be able to clarify any effects in the evaluation phase. Also, Eclipse events like switching views were recorded. This information was used to assess the usage intensity of the different views, and in particular to check whether the DOPG had really been used to answer the questions.

8.1.6 Threats to Internal Validity

Given the described experimental setup and procedure, can we be sure that any observed effect is only caused by the variation of the independent variable, or are there any other extraneous variables that may have influenced the result? The following factors should be considered:

[6]`http://sourceforge.net/projects/exclipse/`

- **Individual participant differences:** Each subject has a different background regarding programming and software maintenance experience. These differences should be canceled out by the randomized assignment of participants to groups.

- **Instrumentation:** Measurement of the dependent variables may differ between participants. This threat was considered by automating the measurement process (see Section 8.1.5).

- **Session differences**: The training that the different groups received may have differed in details between sessions, although it was largely defined by the experimenter's handbook. There were also differences in the questions that were raised by different groups. However, since each session had one representative from each of the four subgroups, this effect should be canceled out.

- **Sequencing effects:** The effects caused by analyzing two different systems one after the other were equalized by counterbalancing the order of these systems.

- **Subjects' perception:** It could not be concealed that the subject of investigation were DOPGs, since this was the only unusual element in the experiment. The participants knew which technique was being investigated, and they could guess that this technique was invented by the experimenter. This may have influenced their behavior or answers.

8.1.7 Threats to External Validity

There are quite a number of factors that affect the generalizability of any results. Storey [180] discusses some of these threats in more detail.

- **Program representativeness:** It is unknown whether the programs chosen for this experiment are representative for real maintenance situations or not. However, they should be large enough to be realistic.

- **Task representativeness:** The experimental tasks are not necessarily representative for real maintenance situations. Also, they were selected by the experimenter and may be of a kind that is particularly well suited for DOPG based analysis. Generalization to different tasks is restricted.

- **Experience:** The participants were all computer science students, not professional programmers. Results may be different for experienced, professional Java programmers.

- **Familiarity with the system:** Being confronted with a completely unknown system is not the common situation in software maintenance. Usually, the system to be maintained is already well known to the maintainer. Results can not be generalized to such a sitation.

System / Question		visibility min	max	avg	std	focus min	max	avg	std
ArgoUML	Q1	36	78	55	15	11	58	31	15
	Q2	0	81	45	22	0	81	34	21
	Q3	0	55	34	21	0	51	26	21
GanttProject	Q1	15	100	56	23	14	44	33	11
	Q2	3	96	64	36	2	73	31	26
	Q3	34	100	60	30	22	57	41	16

Table 8.5: Relative visibility and focus time of DOPG view. The numbers (in percent) relate to the overall time that was spent on each task. min=minimum, max=maximum, avg=average, std=standard deviation.

- **DOPG experience:** DOPGs were a new concept to all participants. They had to learn and understand this concept within a short period of time before using them. Subjects with more experience in using DOPGs will probably perform differently.

- **Experimenter effect:** The experimenter is the same person who invented DOPGs. This may have influenced any aspect of the experiment.

- **Choice of relevant classes:** The decision about the relevant class was not performed by the subjects. Would they have chosen the same relevant class, or some other class? The results are therefore not generalizable to the case that relevant classes are not preselected. Also, the effort for constructing DOPGs was not taken into consideration in the experiment. DOPGs were calculated outside the experiment.

8.2 Results and Discussion

This section presents and discusses the results of the experiment. The two subject systems and their associated tasks are not directly comparable and are therefore examined separately.

Let us first examine whether DOPGs have really been used. For this purpose, the times when the DOPG view was visible or in focus were recorded during the experiment. Table 8.5 shows the measurements. The percentages tell us that the graphs have been used with different intensity by different subjects, but they have been used in all except two cases. For ArgoUML Q3, one participant had already found the answer to Q3 while working on Q2; for ArgoUML Q2, DOPGs were not used by an experiment group member in one case – which resulted in a wrong answer. On average, DOPGs were actively used for about 33% of the time.

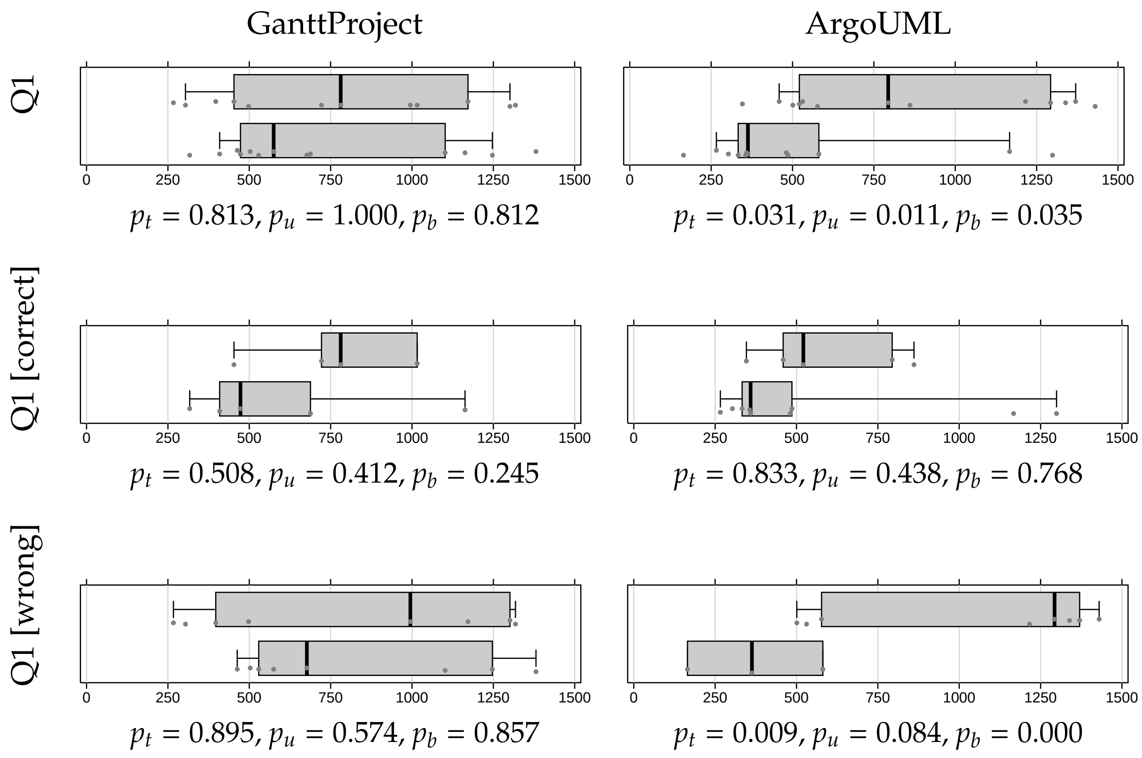


Figure 8.3: Statistical evaluation of dependent variable measures (response time for question 1). Within each box, the upper barchart represents the control group, and the lower barchart shows the experimental group's measures. The X axis represents the response time to question Q1 in seconds. Additionally, the p-values for different tests are given under each barchart: t-test (p_t), Mann-Whitney U test (p_u), and the bootstrap test (p_b).

8.2.1 Response Time

Now, we look at response times to questions, which were measured automatically during conduction of the experiment. The response time for each question was not limited, and several participants needed all available time only for the first task. Because of that, there is only complete data for the first task of each subject system. The first task was solved by all participants, whereas response times for the other tasks are only available for a subset of participants (see Table 8.5). Also, the answer to the third question was in some cases already found while working on the second question, which further distorts response times for question 3. Therefore, a meaningful evaluation of response times is only possible for the first task of each system.

Results are summarized as barcharts in Figure 8.3. This figure is arranged as a matrix: The first column contains data from the GanttProject system, the second column from ArgoUML. The first row contains barcharts for response times to question one (*Q1*). This includes all answers, no matter if they were right or wrong. The second row displays response times for correct answers only (*Q1* *[correct]*), and row three contains response times for wrong answers (*Q1 [wrong]*). Each box contains two barcharts: the upper one shows the control group's data, the lower one the experimental group's data. Each barchart contains the following information: the box indicates the 25% and 75% quantiles (*interquartile range*), the whiskers indicate the 10% and 90% quantiles, the thick line indicates the median, and each dot represents one data point.

On first sight, the charts show that the median response time to question 1 was always shorter when DOPGs were present. On the other hand, the interquartile ranges for GanttProject do not show a big difference. To find out whether the means or medians are really statistically different from each other and not result of pure chance, I performed several tests on the different data sets:

- Student's t-test: This test assumes that the data is normally distributed, but it is quite robust against violations of this precondition. We therefore apply this test as an additional indicator even though the distribution is unknown.
- Mann-Whitney U test: A non-parametric test which only considers the ranking of data, not absolute differences. It assesses whether two samples come from the same distribution.
- Bootstrapping: A resampling method [43]. It takes several thousand random samples from the original data, calculates the mean differences from those samples, and analyzes the resulting distribution. The relative location of the zero-crossing of the mean difference values results in the p-value.

Details about these tests can be found in Appendix F. The t-test and bootstrapping are based on the mean, while the U test is based on the median. The results of applying the different statistical tests are displayed below each box in Figure 8.3 as p-values. A p-value is the probability of obtaining the observed effect when the null hypothesis is true. All p-values result from two-sided tests. The numbers show that only differences between means for *Q1* times for ArgoUML are statistically significant (p=3.5%) for all three tests. Also, this is true for *Q1 [wrong]* times (ArgoUML), according to the t-test and the Bootstrap test (p=0.9%). All other differences between means have a high probability to occur under the null hypothesis.

These results suggest that answer times may be faster when DOPGs are available, but it depends on the system under investigation or the task. For the first GanttProject task, there is no significant indication that DOPGs helped, but for the first ArgoUML task, such indication can be detected. Therefore, hypothesis H1 is too general and must be rejected.

8.2.2 Correctness of Answers

The second dependent variable to be evaluated is the correctness of answers. Since participants finished different numbers of tasks, we look at the share of correct answers instead of absolute counts. Comparing absolute counts would additionally

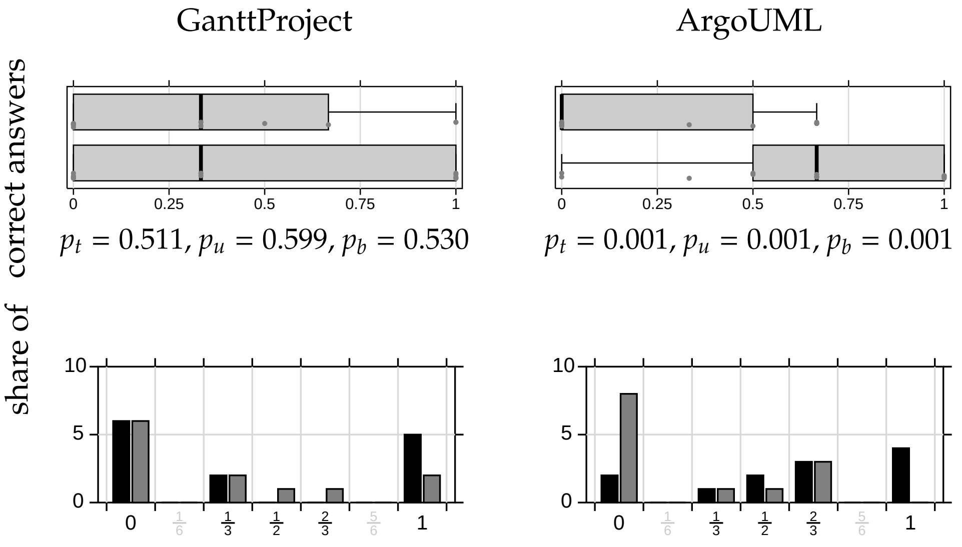


Figure 8.4: Statistical evaluation of dependent variable measures (correctness of answers). The upper barcharts represent the control group, the lower barcharts contain the experimental group's data. In the histograms, black bars represent the experimental group, and gray bars show the control group's counters. The X axis denotes the share of correct answers, the Y axis the number of occurences of each value.

take into consideration response times, which we investigated separately in the previous section. The correctness of each answer was determined by first checking whether the answer (usually a source code location) complied to the sample solution. If it did not, the code was checked manually. If it was closely related to the sample solution, the answer was judged as correct, otherwise wrong.

Figure 8.4 shows the histograms for the share of correct answers for both subject systems. The bars for the experimental group are black, and those of the control group are gray. The charts reveal that there are no great differences in share of correct answers between the two groups for GanttProject. There were only a few more correct answers when DOPGs were available. This difference is not significant. For ArgoUML, the difference is obvious: without DOPGs, about 60% of the participants did not give any correct answer – with DOPGs, two out of three answers were correct (mean). These mean differences are clearly statistically significant (p=0.1%).

Table 8.6 additionally shows the absolute number of correct and wrong answers for each question and experiment group. The numbers show that the DOPG group always performed better than the control group, except for Q3 of ArgoUML. However, the counts for Q2 and Q3 are too low to be meaningful. The most significant difference between experiment and control group is for Q1 of ArgoUML: while 75% of the DOPG group gave the correct answer, only 38% of the comparison group had their answer correct. Also, for GanttProject Q3, *all* answers were correct for the DOPG group, whereas for ArgoUML Q2, there was *no* correct answer from the control group.

System/Question		with DOPG correct	with DOPG wrong	w/o DOPG correct	w/o DOPG wrong
GanttProject	Q1	5	8	4	8
	Q2	3	4	3	5
	Q3	4	0	2	4
ArgoUML	Q1	9	3	5	8
	Q2	4	6	0	8
	Q3	2	3	3	3

Table 8.6: Counts of correct and wrong answers.

In summary, the results are again twofold: for ArgoUML, there is a strong tendency towards more correct answers when DOPGs were available, while no such difference at all can be detected for the GanttProject system and tasks. When only looking at the ArgoUML figures, the null hypothesis can be rejected at the 0.1% significance level. For GanttProject, it definitely cannot be rejected. It is further interesting to note that no correlation between programmer's experience and response times or correctness of answers could be detected.

8.2.3 Questionnaire

After performing the tasks for one subject system, each participant filled out a questionnaire. This was requested to evaluate their subjective satisfaction, confidence, and productivity. Among others, the following questions had to be answered:

- "Were you able to solve the tasks efficiently?" The average answer was "rather not", no matter if DOPGs were available or not.
- "Are your results correct?" The average answer was "rather yes". Again, the availability of DOPGs had no significant influence.

The participants were also asked to rate how helpful each of the available and relevant Eclipse features was for solving the tasks. Figure 8.5 shows the distribution of answers to this question. Answers may be missing when the feature has not been used. For GanttProject, ratings for the different features are similar, no matter if DOPGs were available or not. With DOPGs, they get the best rating of all features on average, but the *Java search* rating is quite close to it. Without DOPGs, the search functions have to be used more intensively; their rating varies. For ArgoUML, DOPGs were considered to be "very useful" when present. In their absence, the *Code browser* and *Java search* functions were used and found to be quite helpful, but the ratings are not as good as for DOPGs.

The findings from the questionnaire confirm our earlier findings: DOPGs supported the ArgoUML tasks much better than the GanttProject tasks.

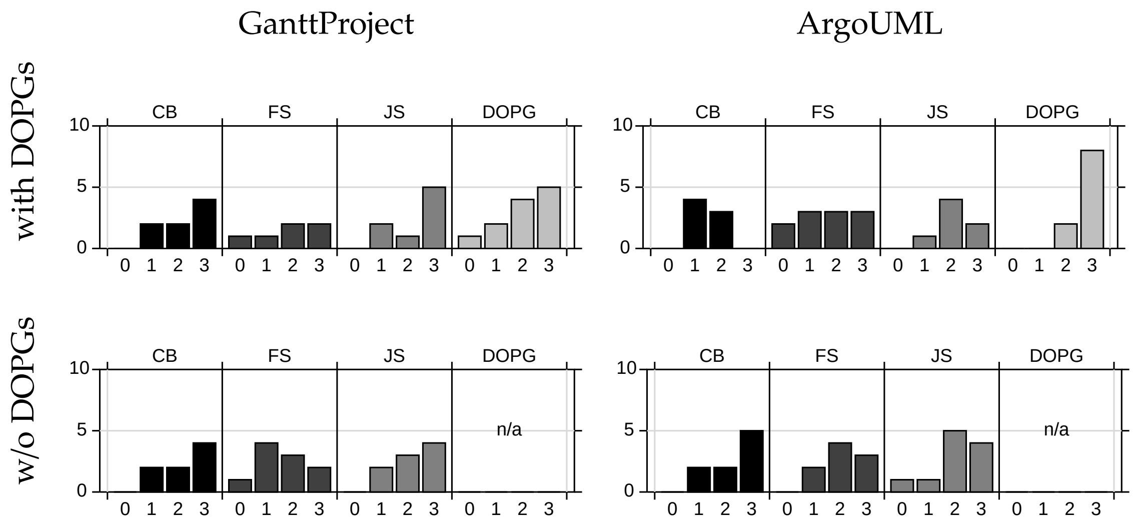


Figure 8.5: Rating of different tools. CB = Code browser, FS = File search, JS = Java search. The numbers on the X axis denote the rating: 0 = not helpful, 3 = very helpful. The Y axis shows the number of occurences of each rating.

8.2.4 Discussion

The results are different from what I expected. When the experiment was designed, tasks for the two subject systems were considered to be of comparable difficulty. However, it turned out that participants had much more problems with GanttProject and its tasks than with ArgoUML. This is particularly astonishing when considering the fact that ArgoUML is more than four times the size of GanttProject. Some participants mentioned that ArgoUML's JavaDoc documentation was quite helpful, so this may have been one reason. Also, each of the six tasks was solved correctly by at least two students of each group, which means that the tasks were not infeasible.

There may be several reasons for the different DOPG performance with GanttProject and ArgoUML. The most potential reason, however, is the difference in number, size, and complexity of the DOPGs. For GanttProject, there were three DOPGs available, and two of them were quite large – much larger than the ArgoUML graphs. For ArgoUML, there was only one DOPG of medium size. Maybe the effort for choosing among several DOPGs is higher than expected, although the best-suited DOPG was presented as first choice. Most probably, the graph size has an important impact: if a graph is very large, it is just too hard to work with it.

It is currently not know if this is really the reason for the different results. It is nevertheless an interesting finding that DOPGs improve program comprehension performance for certain tasks and systems – but for others, they do not. If I had chosen only one of those two systems for the experiment, the outcome would have depended only on this choice: for ArgoUML, the null hypothesis would have been rejected, but for GanttProject, I would not have been able to reject it.

8.3 Summary

I designed and performed a controlled experiment to find out whether DOPGs support program understanding. As a result, I could not reject the null hypothesis for my research question, which means that DOPGs do *not* support program comprehension *in general*. However, the experiment showed that it depends on the subject system, tasks, choice of DOPG objects, and most probably on the number, size, and structure of the DOPGs whether they are helpful or not. For one of the investigated systems, DOPGs were clearly beneficial, while for the other system, this was as clearly not the case.

These findings illustrate the general problems one has in designing or appraising empirical studies of this kind: even if an adequate subject system and representative tasks have supposedly been chosen, you cannot be sure what the results would have been with a different system or different tasks. This should be kept in mind when dealing with this kind of study.

Part IV

Finale

Chapter 9

Related Work

The work presented in this thesis is concerned with several areas in which others have performed research before or in parallel. This includes tracing and trace compaction techniques, dynamic program visualization, experimentation, and protocol recovery. Also, a few approaches are comparable in nature. Those are presented in more detail in this chapter.

9.1 Related Techniques

The first section discusses techniques that are quite closely related to Dynamic Object Process Graphs. They either have a similar representation or extraction technique, use related algorithms, or have similar applications.

9.1.1 Static Trace Extraction

Static trace extraction and the representation of static traces as Object Process Graphs were first introduced by Rainer Koschke and colleagues at the University of Stuttgart. The ideas have been investigated in depth by several students. Hanssen [73] and Vogel [196] developed the static trace extraction technique. This technique was later published at a conference and in a journal [45, 46].

The basic ideas from this work have heavily influenced my own work on DOPGs. In particular, the representation of a trace as an Object Process Graph is largely identical. I extended it by advanced concepts such as exception handling and multithreading and provided a meta-model for it. Also, my thesis is entirely based on dynamic analysis, whereas the analyses from Stuttgart are all static. Applications like visualization for program understanding are usually not possible with statically extracted OPGs of real-world programs due to their sheer size (see Chapter 5).

The main application of static tracing is protocol recovery. This is discussed in Section 9.5.1.

9.1.2 Slicing

Program slicing [207, 208] uses control and data dependencies to identify all expressions that influence a given variable at a certain source location (backward slicing) or are influenced by this variable (forward slicing). The variable and its occurrence form the *slicing criterion*. Weiser has shown that slices correspond to the mental model of a programmer during debugging [207].

OPG extraction is similar to program slicing in that it reduces a graph that represents program aspects, but the information represented by the graphs differs – and also the notion of "relevance". For program slicing, every node reachable via control and data dependencies is relevant. For object tracing, a node is relevant only if it relates to one particular object.

Another big difference is the size of the resulting graphs. Empirical studies on static and dynamic slice sizes by Harman et al. [19] report that static slicing reduces a program to about 33% of its original size and to about 20% for dynamic slicing. Compared to that, DOPGs reduce the CFG of a program to a much smaller fraction – usually below 1% (see Chapter 6).

Static Slicing

As OPG extraction, slicing can be performed by means of either static or dynamic analysis. Overviews on program slicing were published by Tip [190] and Binkley and Gallagher [18]. Modern static program slicing techniques represent a program as a system dependency graph [82]. A program slice is then a reduced system dependency graph that contains only nodes relevant to the slicing criterion. When using this representation, slicing is reduced to reachability in a graph.

Slicing has been extended to object-oriented programs by explicitly modeling the parameter passing of the object's attributes to its methods and by using static pointer analyses or dynamic information to handle dynamic binding [25, 26, 108, 112, 135, 178, 179]. For example, Mock et al. [130] use dynamic points-to data to improve static slicing. They found that slices are reduced significantly in size only for programs that make intense use of function pointers.

Liang [112] introduced so-called *object slicing*, where the slices are further reduced to the statements contained in an object's methods. Yet, these variants for object-oriented programs are still following the original ideas of program slicing for procedural programs.

Dynamic Slicing

Static slicing has been complemented by dynamic slicing. A dynamic analysis knows the concrete values of variables that have occurred. This significantly reduces the size of a slice, which otherwise often consists of the entire program. The idea of dynamic slicing was first presented by Korel and Laski [93, 94]. The first approaches had very high memory requirements, so dynamic slicing could only be applied to small programs [2, 67]. Korel et al. [95] give an overview of the first ten years of dynamic slicing.

Gyimóthy et al. [64] present an algorithm that is efficient in terms of memory requirements and can therefore be applied to real-world C programs [15]. Wang et al. [203] use compressed bytecode traces for slicing Java programs, which results in major space savings. Apiwattanapong et al. [9] introduce a very simple dynamic slicing technique: they regard everything that is executed after the assignment of the slicing variable as belonging to the slice. Their studies show that this simplistic approach is both efficient and precise, compared to other (more complex) dynamic slicing approaches.

Dynamic slicing and DOPG extraction both have the advantage that they reduce the size of the program to be investigated, compared to their static counterpart.

Union Slices

Beszédes et al. [14, 187] introduce *union slices* as a compromise between static and dynamic slicing. Static slices lack precision, and dynamic methods require the execution of many test cases. Union slices result from the union of dynamic slices for a set of test cases. The union of dynamic slices for all possible executions is called the *realizable slice*. According to the experiments that are described in this paper, the size of union slices is usually below 20% of the program size for small programs (4–21 KLOC of C code), which is comparable to Harman's results [19]. The test cases reached a coverage of 45–68%. As expected, when adding more test cases, slice size growth quickly decreases. This is another example of comparison of static and dynamic analyses. When using DOPGs for protocol recovery, different traces are also combined to get a more complete result.

9.1.3 Call Graph Restriction to a Use Case

Walkinshaw et al. [202] present an approach to reduce a call graph to those methods that are potentially relevant to the execution of a given use case. This approach requires a set of *landmark methods* as input which are elicited from the scenario specification, probably by means of a feature location technique. Then, all direct paths through the call graph that contain all landmark methods are identified by inducing *hammock graphs* between each pair of such methods. In the last step, paths that can influence and can be influenced by the paths in the hammock graphs are identified. This step uses intra-procedural slicing with call statements as slicing criteria.

The reduction of the call graph using this approach is quite high. A call graph from their case study could be reduced from 251 nodes and 719 edges to 16 nodes and 20 edges, leaving most of the relevant information for the investigated use case. On the other hand, precision and recall was not as good for many other cases. The quality of the results depends on the selected landmark methods, on the scenarios in which they are analyzed, and on the system being inspected. This is similar to DOPGs where the usefulness of the results may depend on the choice of the investigated object. Another commonality is the reduction of a given input dependency graph to a potentially useful subgraph.

9.1.4 Object Flow Analysis

Lienhard et al. introduce an object-centric approach called "Object Flow Analysis" [113–115]. Their dynamic analysis captures how object references are passed through the system at runtime. The creation of object aliases is explicitly recorded, as well as the history of state evolution of these aliases. Similar to DOPGs, an "Object Flow" represents the life cycle of an object at runtime, that is, where it is instantiated and how it is then passed through the system. The analysis is applied for program understanding [114], detecting feature dependencies [117] and for unit test construction support [116].

Program understanding: The information is visualized in "inter-unit flow views" or "transit flow views". An inter-unit flow view shows how many objects are transferred between two "units". A mapping from classes to conceptual units has to be defined manually. When the mapping is adequately chosen, this view gives a good impression about how objects are passed between the different units. The transit flow view shows detailed information about all objects that show up in a given class.

Unit test construction: A "Test Blueprint", similar to a UML object diagram, shows which objects are used, which references between objects are accessed, what objects are instantiated, and what side effects (modifications of attributes) are produced in a given program run. This information can be used to create a minimal test case that uses the same elements. A trace view is used for execution unit selection.

Detecting feature dependencies: By tracking back aliases of an object that is used in one feature, this analysis detects when the object originates from a different feature and concludes that there must be a feature dependency between the two features. Dependencies are visualized in "object dependency graphs", where nodes correspond to objects and edges denote object dependencies.

Whereas Object Flow Analysis focusses on the points where aliases for an object are created, DOPG extraction follows an object through the flow of control – no matter by which alias it is represented. Also, a DOPG always relates to one object or to a group of objects that were allocated at the same location in the code. Object Flow Analysis examines the passing of *all* objects, but could be limited to objects of a given class only. DOPG extraction requires the selection of the objects of interest, whereas Object Flow Analysis requires a mapping to units. However, using such a mapping for DOPGs would also be a possibility to make large graphs better readable. Finally, it is interesting to note that they also use a compiler and an IRC client for their case studies, which is another commonality.

A related static approach is presented by Tonella et al. [191]. They statically extract an object diagram based on an *object flow graph* construction. This graph contains as nodes the program locations that may hold a reference to an object, while its edges connect two locations when there is a program statement through which an object referenced by the first location can be assigned to the second. This graph is calculcated incrementally using a flow propagation algorithm. The result can be transformed to an object diagram. The results of this static analysis are then complemented by the corresponding dynamic analysis.

9.1.5 Feature Location

Feature location is the process of locating the set of program units that implement a given feature. There are a number of approaches that use dynamic analysis for this task. When a feature is exhibited in a program run, the code that is responsible for the feature must have been executed and must therefore be contained in the trace. Dynamic analysis reduces the search space from the entire program to only a part of it. DOPGs also have feature location capabilities. Therefore, related feature location techniques are discussed here.

One basic idea of dynamic feature location approaches is to execute a program twice, where one run exhibits the desired feature and the other one does not. The difference between the traces of the two runs then contains the code that implements that feature, or at least part of it [212, 215]. Eisenbarth et al. [44, 99, 171] extended this idea to locate more than one feature at once by using formal concept analysis. The result is a classification of each executed computational unit as being relevant, conditionally specific, irrelevant, or split with respect to a given feature. A related technique by Dallmeier et al. [37] uses differences in sequences of method calls between passing and failing runs of a program to identify defect classes. Eisenberg et al. [47] also perform feature location based on traces. They use different heuristics for ranking a code element's relevance to a feature. Similar to Eisenbarth's approach, they require a mapping of test cases to exhibited features.

Greevy et al. [61] use a two-sided approach for the characterization of features and computational units and introduce measurements for both perspectives. Feature fingerprints and computational unit classes are then used to correlate features to code. As opposed to the previous approaches, this one uses isolated feature traces: tracing is only started immediately before the desired feature is executed, and stopped afterwards.

Another class of dynamic feature location approaches is based on only one trace and uses some additional technique to find the pieces of code that correspond to a given feature. Tracing is used as a filter to reduce the search space. Liu et al. [121] use information retrieval techniques to collect information from identifiers and comments. The features are then located by natural language queries, which return a ranked list of source code elements. Rohatgi et al. [163] apply dependency graph based metrics on the trace that measure the impact of class changes to the rest of the system. The authors' assumption is that the smaller the impact set of a component modification, the more likely it is that the component is specific to a feature.

A visual approach to feature location is presented by Lukoit et al. [127]. Their "TraceGraph" visualizes online which components have been used during which time intervals. When a user executes a certain feature, he can immediately see which components are involved in the TraceGraph. DOPGs also support feature location by visualization.

An approach by Safyallah et al. [166] uses a sequential pattern mining algorithm (as used in specification mining, see Section 9.5.6) to identify behavioral patterns that occur frequently in a given set of traces. From these traces, it must be known that they all exhibit the desired feature. General functionality that is used in every execution of any feature is identified by using the algorithm on a larger set of different use cases.

In Chapter 6, we have seen that DOPGs also cover feature location aspects. This is partly due to the filtering that is performed in the DOPG extraction process – similar to the single-trace approaches. However, DOPGs are further filtered by relevance for an object. The remaining code pieces (identified by nodes in the DOPG) may be particularly useful for locating features that have something to do with that object.

9.2 Tracing

Dynamic analysis is based on observations of a running system. The system must be instrumented with code that records information about the running program – its evolving state and/or the executed artifacts. This implies several questions: how to instrument the code, what to instrument, and how to handle the potentially large size of the resulting traces. This section shows how other people have solved these challenges.

9.2.1 Instrumentation

The general alternatives for instrumentation have been discussed in Chapter 3. The whole range of instrumentation techniques is applied in published approaches. These techniques include source code annotation scripts [141], also in conjunction with static analysis [89], using a preprocessor to insert tracing delegates [5], Java bytecode instrumentation [99, 155, 210], using the Java debug interface [63, 157], or other ways of interacting with a debugger [186]. When only method invocation count information is needed, even a standard profiler can be used [44]. Recently, aspect-oriented programming (AOP) is used to insert tracing aspects into a program [35, 62, 224]. And sometimes, several of the techniques are used in combination [22].

The approaches also differ in the instrumentation density, or in the amount and level of information required. The majority of approaches traces at method level. They either only trace method entry or also method exit, and some of them additionally trace the call site. In many cases, object allocation and deallocation is also traced [141]. Only few approaches take method parameters [62] or return values [5, 30] into account. Thread creation, termination, and synchronization is monitored by a number of approaches as well [155]. The program state, that is, the current call stack [169] or program location [186], or the internal object state [141, 218] is sometimes recorded. Very few dynamic approaches look at the intra-procedural control flow, such as conditions [69, 186] or loops [22].

Apparently, everyone creates his own instrumentation infrastructure. AOP is one technique that can be used for specifying instrumentation locations at a higher level, but it is not (yet) detailed enough for every purpose. For example, in AspectJ[1], there is no way to select those locations in the code where control flow branches. Reiss [158] sketched the requirements for a general instrumentation framework. However, if it was realized, it could still only cover a certain class of dynamic tracing applications. Tools like ATOM [176] allow specification of an instrumentation, but work on a very low level, and they are restricted to a certain hardware platform.

DOPG extraction requires a quite dense instrumentation, including attribute accesses, control flow branches, and the like. This level of detail is hardly reached by any of the mentioned approaches. Systä [186] and Briand [22] come closest to this density.

9.2.2 Trace Compaction and Representation

The large size of trace files is one big issue for any dynamic analysis. For tracing with a high density, as it is necessary for DOPG extraction, it is even more important. Therefore, this is an important topic for DOPG extraction. A lot of research has been conducted to find compact representations of traces. Hamou-Lhadj provides a good introduction to the topic [68, 72].

Reiss et al. [157] describe different possibilities of reducing a trace's size. They present different encodings on two different levels. In a first phase, filtering and compaction techniques are applied, and in a second phase, the data is encoded in an attempt to infer its structure. These encodings include string compaction (represent recurring strings by a shorter id), class selection, N-level call compaction (as used in gprof's output), interval compaction (summarizing data that lies within a given interval), construction of a directed acyclic graph for representing call trees, run-length encoding of sequences, grammar-based encoding, and finite-state automata induction (k-tails algorithm). They also present a case study comparing the compression achieved by different encodings. The presented techniques are quite general. They cover most of the common trace compaction approaches. Some of them have also been applied to DOPG extraction (see Section 4.1).

The presented techniques can be very effective. For example, Reiss [156] reports that a combination of context-free grammar encoding, DAG representation of the dynamic call tree, common subsequence detection, and run-length encoding resulted in a lossless trace size reduction of factor 800, where gzip only reached

[1]`http://www.eclipse.org/aspectj/`

factor 5. This appears to be a pretty good ratio, compared to the factor 7 that was reached in the DOPG trace compression effort (see Chapter 4).

Hamou-Lhadj et al. [68, 71] are also concerned with trace compression. They introduce a framework for lossless trace compression which reduces redundancy. Their approach is based on the common subexpression algorithm. In a preprocessing step, repeated sequences are run-length encoded. Then, parts of the call tree that are identical are identified and reduced to a single representation.

Zaidman et al. [225] manage trace data volume by dividing trace data into recurring event clusters. They use a heuristic based on the relative frequency of events.

An alternative way to reduce the trace size is to take samples from it. Chan et al. [24] investigate if this approach is any useful for their dynamic architecture reconstruction tool AVID [199]. The approach further reduces the completeness of dynamic analysis results. However, Chan et al. found that the trace samples may be useful anyway, specially in combination with animation. For DOPGs, we want to catch the entire lifecycle of an object, therefore sampling is not an option.

Yet another approach is to take dynamic measurements, like counts of method invocations instead of the method invocations itself [41, 99]. However, this is only possible if it provides enough information for the anticipated analysis. It is not enough for DOPG extraction.

9.3 Dynamic Software Visualization

Software visualization is an active research area of its own. In this section, we can therefore only glance at a few influential approaches to dynamic software visualization.

One of the first approaches to dynamic visualization of object oriented systems is presented by De Pauw et al. [141]. They introduce a general instrumentation scheme, a protocol for communication between instrumented program and visualization component, and different visualizations. Their visualizations focus on getting an overview of relations and instantiated objects.

Lange et al. [106] describe a dynamic analysis that collects method invocation data along with the associated objects. They use merging, pruning, and slicing to reduce the search space. Their results are visualized in three different views: Object view (method invocation between concrete objects), class graph (objects of the same class are merged), and bar chart (similar to UML sequence diagrams, shows the order of invocations).

Jerding et al. [89] create a compacted dynamic call tree to reduce the trace size. They visualize the entire trace graphically, which gives evidence of *interaction patterns*. These can be found by various pattern matching algorithms. Their visualizations consist of an *Execution Mural* (similar to sequence diagrams, but on class level) and an *Information Mural* (displays the entire trace for navigation). Both of these visualize the entire trace, but are scalable to details. They also show that their approach may be helpful for understanding the architecture of a system and locate certain components [88]. Similar to DOPGs, this approach achieves feature location through visualization.

De Pauw et al. [142] propose to visualize a trace as an *execution pattern view* instead of a sequence diagram. It is basically a graphical call tree, where objects are represented by vertical bars. They additionally propose several interactive techniques that let the user expand, elide, and extract execution information, as well as a number of generalizations that basically detect recurring execution patterns.

Richner and Ducasse [161] present an approach that is based on a logic programming language. It combines information gained from static (inheritance, attributes, methods, attribute access, invocation) and dynamic analysis (method invocation). The results of a logical query are visualized as dependency graphs.

Mancoridis et al. [58] present a tool called "Gadget" that extracts the dynamic call graph: it shows the method invocations that occured between classes during runtime. They use clustering on this graph to present the dynamic structures in a modular fashion that is easier to understand.

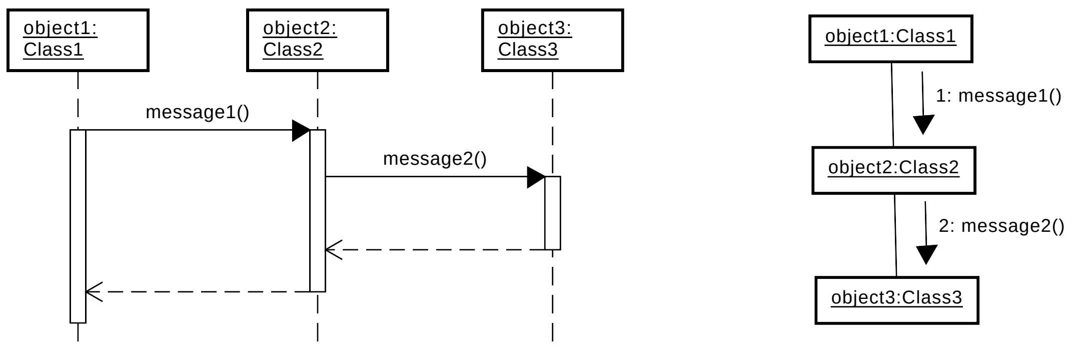
Ducasse and Lanza pioneered the use of polymetric views in Software Reengineering [107]. Polymetric views visualize measurements as different properties of geometric objects, like the dimensions of a rectangle, its color, shape, or position. They also apply this approach to dynamic measurements ("lightweight trace", [41]). The authors propose several assignments of measurements to properties, depending on the intended use of the visualization. Examples include the "instance usage overview", the "communication interaction view", or the "creation interaction view".

9.4 Diagram Extraction

The visualizations from the previous section were proprietary. Meanwhile, the Unified Modelling Language (UML) provides a widely-used standard for visualizing many aspects of software. UML diagrams can not only be used for designing software, but also for visualizing aspects of existing programs. This section presents approaches to reverse engineering UML diagrams.

9.4.1 Interaction Diagrams

Interaction diagrams describe how groups of objects collaborate in some behavior. The UML distinguishes between sequence diagrams and collaboration diagrams [165]. A **collaboration diagram** is an object diagram that additionally contains the exchanged messages and their order. A **sequence diagram** contains the same information, but puts more emphasis on the order of messages. Sequence diagrams visualize the interaction by showing each participant with a lifeline that runs vertically down the page and the ordering of messages by reading down the page. When showing an exemplary instance of a use case, they are also called **scenario diagrams**. Only since UML 2.0, sequence diagrams support



(a) Sequence diagram. (b) Collaboration diagram.

Figure 9.1: Basic UML interaction diagram types.

loops, conditions, and the like, which extends their range of application. **Message sequence charts** (an ITU standard) are interaction diagrams that are largely identical to sequence diagrams.

Interaction diagrams are quite easy to extract dynamically by tracing object creation and method invocation. The problem is intelligent filtering of the information to display – if this is not done, the diagrams become large and unmanageable. There are just too many objects and too much communication between them for immediate visualization. Also, in the presence of loops, large portions of such diagrams consist of repetitions. It is desirable to abstract from such sequences.

DOPGs are not concerned with interactions between different objects, but with dependencies between locations in the code where one concrete object is used. However, DOPG extraction also aims at reducing the vast amount of information contained in a trace to a useful subset. So there are some commonalities that suggest a discussion of interaction diagram extraction techniques – in particular, dynamic ones.

A number of approaches use pattern matching [35, 162] or a trace similarity metric [167] to group similar sequences of method invocations. Other automatic techniques for the simplification of interaction diagrams include constructor hiding, minimum/maximum stack depth filtering, object clustering, and getter/setter hiding [35]. Additionally, interactive techniques such as projection to certain objects, call compression (showing call details on demand only) [102], or zooming [35] may be used. Some of the published approaches allow examination of object interactions while the program is running [62, 162]. Let us now look at some interesting aspects of these approaches in more detail.

Gschwind and Oberleitner [62] provide different perspectives of object interactions: use of an object, calls from an object, and how and when an object is passed as a parameter. Cornelissen et al. [35] investigate the visualization of test suites as scenario diagrams. Using the techniques mentioned above, they state that all test cases of their subject system were comprehensible. However, this system was very small (3 KLOC, 20 classes), so it is unclear whether these observations also hold for larger systems. Jiang et al. [90] derive sequence diagrams from state machines. They use the latter for merging several traces. Merging state machines to get a more representative picture is also used in the DOPG based protocol recovery approach. The approach by Salah et al. [167] tries to identify typical class usage scenarios.

Briand et al. [21, 22] also aim at reverse engineering UML sequence diagrams. The interesting thing about their work is that it uses a quite formal approach, and that it traces control-flow within method bodies. The authors define two meta-models, one for the trace and one for the sequence diagram. The transformation from trace to sequence diagram is then performed by algorithms which are directly derived from consistency rules that are defined between the two meta-models. The rules are described in the Object Constraint Language (OCL). This could be an alternative to the automaton transformation technique that is applied for DOPG extraction.

There are also approaches to static object interaction diagram extraction. For example, Tonella et al. [192] establish the static objects at call sites through a context-insensitive and flow-insensitive propagation of objects created at an allocator call site. The caller and callee relationship at the call site is then used to create collaboration and sequence diagrams. Kollmann et al. [92] present an approach to extract collaboration diagrams based on a transformation between meta-models. Rountev et al. [164] also extract UML sequence diagrams statically. Their analysis is based on the calculation of *branch and loop successors* on the control flow graph. Wu et al. [216] address the problem of identifying certain interactions by using graph patterns. Those are then located by a relational calculator and presented as scenario diagrams.

A recent article by Bennett et al. [13] investigates the usefulness of different features that are commonly used for interactive exploration of reverse-engineered sequence diagrams. The result is that most of these features are in fact considered useful by the users.

9.4.2 State Diagrams

Another UML diagram type that is subject to reverse engineering is the state diagram. A **state diagram** graphically represents a finite state machine – a common technique to describe the behavior of a system. It consists of states and transitions, where states correspond to program states, and event-triggered transitions lead from one state to another. In OOP, a state diagram is usually used for a single class to show its lifetime behavior. DOPGs are represented by activity diagrams, but those are similar to state diagrams: activities are the "state of doing something".

The following approaches either try to reconstruct state diagrams (or finite state machines, FSM) for program comprehension or documentation purposes, or they aim at supporting the software design phase. Protocols in the sense of sequencing constraints are also mostly represented as state machines; state machine inference approaches that aim at protocol recovery are separately discussed in Section 9.5, but are also related to state diagram extraction.

Modelling Support

Koskimies and Mäkinen [101] build on Biermann's algorithm [17] and extend it to synthesize state diagrams from sequence charts. Sent messages become actions in states, and received messages are mapped to transitions. The algorithm increases the number of states step by step until all trace items can be associated to the states. It therefore constructs a minimal state diagram.

Mäkinen and Systä [128] extend this approach by an interactive component. It aims at preventing overgeneralization by querying the user for additional information. This approach uses an *observation table* that is filled based on sequence diagram information and completed by interactive membership queries. When it is closed and consistent, an FSM can be constructed from it. Then, the user has to decide whether the resulting FSM is acceptable – otherwise, he must give a counterexample, and the process starts over again.

Whittle et al. [210] propose a different technique for generating UML statecharts from sequence diagrams. Their technique requires annotation of scenario interactions by pre- and post-conditions on global state variables for merging multiple scenarios. These are expressed in the Object Constraint Language (OCL). Each state in the resulting statechart corresponds to one concrete state vector (resulting from the global state variables). This means that states basically have to be defined manually.

Uchitel et al. [194] present a technique for generating labeled transition systems (LTS; basically an FSM where all states are accepting) from message sequence chart specifications. Their approach requires a high-level message sequence chart (hMSC) and explicit component state labeling as additional input. The hMSC shows the transitions from one scenario to the next, but does not cover interleavings between sequence charts.

Damas et al. [38] propose an interactive and incremental approach that uses positive and negative scenarios as input. It uses the RPNI algorithm [137] for grammar inference and extends it by interactive state merging: the system presents scenarios to the user that have to be classified as being counter-examples or desired behavior. The result is an LTS.

Program Understanding and Documentation

Systä [184–186] builds on Koskimies approach [101] to create state diagrams for objects from traces. Trace information is collected using breakpoints in a debugger, and trace size is reduced by string-matching based detection of repetitions and subscenarios. State diagrams for single objects are then inferred from these scenarios, using program location information (*state boxes*) to avoid overgeneralization. However, only selected classes can be traced with this approach, because the tracing overhead is very high. The commonality with DOPGs is the use of static program location information: Systä also uses this information to identify repeated locations in the trace, and it is used to display the location of conditions. In contrast to DOPGs, the relation to the overall control flow of the investigated application is not revealed – the result is the internal behavior of a selected object or method.

9.4.3 Other Diagrams

An approach by Hamou-Lhadj et al. [69] uses fan-in analysis to detect *utilities*. Those are regarded as being irrelevant for the trace and removed from it, leaving only high-level components. In continuation of this work [70], the authors add trace summaries, which are the result of iteratively removing the routines with the highest level of *utilityhood*. Additionally, manually specified *known implementation details* are removed. The result is presented as a *use case map (UCM)*, which consists of paths, components, and responsibilities. A UCM abstracts from inter-component communication. Therefore, UCMs do not provide a global overview of the system. However, as for DOPG extraction, conditions have to be traced in this approach.

Smit et al. [173] use GUI event traces for redocumenting use cases. These traces are clustered according to the similarity of the user interface events. The result is a kind of activity diagram ("alignment") that shows the different steps and alternatives during use case execution.

9.5 Protocol Recovery

The meaning and importance of protocol recovery has been discussed in Chapter 7. Also, some of the most common approaches to regular grammar based inference of protocols have been introduced there. This section gives a broader overview of the related work in this field. It discusses static and dynamic techniques, grammar inference and object state based techniques, and covers the related areas of process and specification mining.

9.5.1 Static Trace Extraction

An extensive body of work has been performed on protocol recovery based on static trace extraction at the University of Stuttgart. The protocol recovery approach as described in Chapter 7 largely originates from there. Koschke and Zhang [100] give a good overview of the general idea. The details have been investigated by several students' theses. Heiber [74] evaluates different notations for protocols and describes a set of transformations for protocol recovery based on those static traces. He distinguishes between automatic (safe) transformations, such as common prefix and suffix reduction, and semi-automatic (unsafe) transformations. Among the latter is merging conditional branches with differently labeled edges and simplification of conditional branches. His work is of rather conceptual nature. Haak's thesis [65] is closer to practice and describes the method for recursion elimination that we have already seen in Chapter 7. He also presents a few ideas for validating protocol graphs against OPGs.

More recent work by Jung [91] extracts a *protocol structure graph* from a static OPG. This graph contains routines and relations between them (call relation, dominance relation, cycles, delegation, sequence of calls, loops, control dependencies). The approach is based on *structural analysis*. It reduces the graph size to 20-50% of the original OPG for the investigated (very small) programs. The approach transforms an OPG to a representation on a higher level, but it does not extract a verifyable protocol.

Vogel [197] also describes the transformation from OPGs to an FSA that we have met in Chapter 7, but leaves out the simplification step. He measures and compares the automaton sizes of the different steps. In contrast to my thesis, Vogel regards `read` and `write` as the only atomic operations; this means that the resulting automata only have these two symbols as their alphabet. He then investigates *subprotocol* occurences, that is, occurences of automata that accept a subset of the language of an automaton for a different object.

In his dissertation, Vogel [198] covers the topics structural analysis, transformation from OPG to FSA, reports and metrics on that, and protocol recovery. The focus is on static trace extraction and its application for protocol recovery.

In my thesis, I extend the protocol recovery approach by additional simplifying steps, evaluate how it performs for dynamically extracted OPGs, and quantitatively compare these results to other dynamic protocol recovery approaches.

9.5.2 Regular Grammar Inference

A commonly used technique for dynamic protocol recovery is regular grammar inference. The problem of regular grammar inference is to induce a regular language or its associated acceptor (FSA) from examples. The number of possible solutions to this problem is infinite, so the challenge is to select a proper grammar among them. Most approaches start by constructing a prefix tree acceptor – an automaton that accepts exactly the provided set of words. This automaton is then successively generalized by merging similar states. A classical example for an automaton learning technique is the k-tails algorithm [17]: it merges states that are indistinguishable in the set of accepted output strings up to a given length k. Pure automaton learning techniques tend to overgeneralize. An approach to preventing this is to defer the decision of whether to merge two states or not to an expert by posing *membership queries* [8]. This may be done interactively by asking the user, or automatically generated and executed test cases can help to answer these questions.

Automaton learning is a research area of its own. The following overview concentrates on automaton learning for protocol recovery. For this application, the alphabet consists of a component's methods, and the language examples are legal sequences of method invocations. These are usually extracted by means of dynamic analysis, that is, the calls of client applications to the investigated component's methods are recorded.

Whaley et al. [209] use the *successor method* [160] for directly constructing a protocol automaton: each state corresponds to a state-modifying method, and a transition between two states indicates a legal sequence of method calls. This simple construction has the drawback that only primitive sequencing constraints can be expressed – real protocols will usually be more complex.

Ammons et al. [5] use the sk-strings automaton learning technique [152], which merges states that are indistinguishable for their top *s* percent of the most probable *k*-string. The result is a *probabilistic* FSA that is annotated with transition frequencies. Infrequently traversed edges can then be removed from the automaton. Another extension to the k-tails algorithm is presented by Lorenzoli et al. [126]. Their technique called GK-tail generates *extended* finite state machines (EFSMs). EFSMs model the interplay between data values and component interactions by annotating FSM edges with conditions on data values. In other approaches, transitions are only labelled with methods.

Walkinshaw et al. [201] apply the QSM grammar inference technique: pairs of states that are candidates for merging are selected by the "Blue Fringe" algorithm [105], and the decision of whether to merge or not is deferred to the user. This approach results in an FSM with a good accuracy, but comes at the price of heavy user interaction. The number of questions that need to be answered seems to rise exponentionally with the number of states.

To improve the accuracy, robustness, and scalability of automaton learning based approaches, Lo and Khoo [123] propose a protocol recovery architecture called SMArTIC, which consists of four consecutive steps: First, erroneous traces are filtered out by detecting common behavior. Next, traces are divided into groups of "similar" traces (clusters). Then, protocol automata are generated for each group. Using a PFSA specification miner is proposed for this step, but other miners may be used as well. Last, the automata are merged into a single one..

9.5.3 Object State Based Approaches

A different class of dynamic protocol recovery approaches is based on object state in the sense of the concrete values of an object's attributes. These approaches derive states from the different configurations of an object's attributes and connect them according to the way in which methods (that is, modifiers) change that state. Since an attribute may have an arbitrary number of possible values, the automaton states usually have to abstract from the concrete value (*observer abstraction*). This can either be done automatically [36, 217] or manually by providing special methods that explicitly indicate the state [200]. Another approach to state reduction is to regard each attribute separately (*state slicing*, [209, 217, 218]). Xie and Notkin combine this approach with automatic test case generation for extensively exercising object states [218]. Walkinshaw et al. [200] apply the same idea to static analysis: the transitions between states are automatically extracted from the source code by means of symbolic execution. Their technique also identifies the paths through the code that govern a detected transition.

Publication	s/d	add. input	obs.	perm.	technique	state meaning
Alur [4]	s	safety property	–	+	iGI: L*	obj. state
Ammons [5]	d	def./users	+	–	GI: PFSA	–
Beyer [16]	s	–	–	+	game	obj. state
Henzinger [76]	s	state-tracking predicates	+	+	abstraction refinement (CEGAR)	obj. state
Lo [123]	d	–	–	–	clustering	–
Lorenzoli [126]	d	–	–	–	GI: GK-tail	–
Quante [149]	d	–	–	–	OPG transf.	SLoc
Walkinshaw [200]	s	–	+	–	symbolic execution	user defined
Walkinshaw [201]	d	–	–	–	iGI: QSM	–
Whaley [209]	s/d	–	+	–	successor	method
Xie [217]	d	–	+	–	observer abstraction	obj. state
Xie [218]	d	–	–	–	structural abstraction	obj. state

Table 9.1: Protocol recovery approaches. s/d = static/dynamic, add. input = additional input required, obs. = uses observers, perm. = delivers permissive interface, GI = grammar inference, iGI = interactive GI.

These approaches have the advantage that states can be meaningfully labelled, since they directly correspond to the underlying object states. DOPG states can also be meaningfully labelled, but represent source locations, not object states.

9.5.4 Other Static Approaches: Avoiding Safety Violations

Apart from dynamic analyses and static OPG extraction, there are a number of other static approaches to protocol recovery.

Whaley et al. [209] propose to use static analysis on components to detect their sequencing constraints when they have been designed to guard against misuse (*defensive programming*). They locate predicates in the API's code that control whether exceptions are thrown, and they use constant propagation analysis to find out where the fields (that are compared to a constant in the predicate) are set. This simple analysis delivered a number of forbidden sequences of operations when run on the Java standard class library. The same idea is mentioned by Koschke and Zhang [100]: they call it *glass-box understanding*.

A similar, but refined approach is followed by Alur et al. [4]. Given a Java class and a safety property (for example, an exception that shall not be thrown), their tool answers the following question: "How is the interface to be used such that the safety property is not violated?" They use predicate abstraction to construct a model and then apply a game theoretic approach for constructing the interface. The approximation is based on automaton learning (L* algorithm) and symbolic model checking. The learning algorithm repeatedly queries the user with membership questions.

Henzinger et al. [76] use predicate abstraction and refinement [31]. Their approach requires initial predicates for abstracting (safety abstraction), that is, the user has to manually specify the set of predicates needed to track the state of an object (similar to the object state based approaches). The goal of this approach is to construct an interface that is not only safe, but also permissive. The interface is *safe* if no call sequence violates the library's internal invariants; the interface is *permissive* if it describes all sequences that cannot lead to an error state.

Beyer et al. [16] conduct a comparison and evaluation of three different static interface synthesis algorithms: a direct game algorithm, and two improvements of it, namely the approaches by Alur et al. [4] and Henzinger et al. [76]. The direct game algorithm works as follows. It first constructs the errorless automaton (that is, the automaton that contains only those transitions that do not lead to an error state), then eliminates all states from which all successors lead to error states (pruning), and then minimizes this automaton. All three algorithms deliver the same result and have the same worst-case complexity, but each one has a class of programs for which it is fastest.

9.5.5 Recovering Algebraic Specifications

A few approaches do not recover the protocol as state machines, but as algebraic specifications. Pre- and postconditions are another important aspect of a component's interface besides sequencing constraints. They can as well be used for comprehension and automatic testing.

Henkel and Diwan [75] use method signatures to automatically generate a large number of terms, guided by heuristics, where each term corresponds to a legal sequence of method invocations on an instance of the class. The legal sequences are found out by a backtracking algorithm. Their outcomes are compared, yielding equations between terms, which are then generalized to axioms. Tillmann et al. [189] use symbolic execution to discover axiomatic class specifications. They identify modifier and observer methods; a modifier method's behavior is then summarized and expressed in terms of observer methods.

Example:
```
requires key != null otherwise ArgumentNullException;
ensures containsKey(key);
ensures count == old(count) + 1;
```

A related dynamic technique is the invariant discovery by Ernst et al. [52]. It locates program invariants by monitoring the runtime state of a program and attempting to match invariant templates to expressions. Nimmer et al. [133] performed a comparison of this technique to the corresponding static one. The

Publication	s/d	input	pattern compl.	mining method
Acharya [1]	s	–	partial order	FCPO mining
El-Ramly [48]	d	–	sequence	IPM2
Engler [50]	s	–	fix templates	internal consistency / statistical analysis
Gabel [56]	d	FSA	regular expr.	BDD based
Li [111]	s	–	multi	freq. itemset mining
Liu [120]	s	–	fix templates	pattern matching
Lo [124]	s/d	QRE	sequence	CLIPER
Lo [125]	d	–	sequence	LS-Set
Ramanathan [153]	s	–	sequence	Apriori-all
Wasylkowski [204]	s	–	pair	freq. itemset mining
Weimer [206]	s/d	–	pair	pattern matching
Yang [219, 220]	d	QRE	seq., |L|=2	pattern matching

Table 9.2: Specification mining approaches.

result was that the dynamically derived specifications were very close to the static ones, even for small test suites.

All these approaches are only capable of inferring simple properties, not temporal ones.

9.5.6 Specification Mining

Specification mining is the process of detecting usage or programming patterns from the code or from execution traces. As opposed to protocol recovery, which aims at identifying the full set of sequencing constraints for a single component, specification mining just identifies partial constraints – for example, sequences like (open, close). It does not necessarily cover the whole interface, and it may reveal relations between methods of different components. Of course, the techniques are similar and related to protocol recovery and are therefore discussed in this section.

Sequential pattern mining was pioneered by Agrawal and Srikant [3]. Their approach called *frequent itemset mining* discovers temporal patterns that are supported by a significant number of sequences. The sequences were mined on a database of customer sales transactions, and it took a few years until the same technique was applied to software.

The approaches differ in the complexity of patterns that they are able to detect. They can detect sequences of length two [50, 204, 206], sequences over an alphabet of size two [219, 220], or sequences of arbitrary length [48, 124, 125, 153]. Other approaches recognize a set of predefined templates that are particularly useful for detecting programming errors [50, 111, 120]. Only one approach accepts arbitrary finite automata as the pattern [56]. An other approach delivers specifications as partial orders [1]. Table 9.2 shows an overview of published approaches.

The main application area of these approaches is finding bugs. The advantage is that these bug finders can work automatically, without requiring any additional input. They are based on the assumption that the majority of the code is correct. If there are variations from the normal behavior, this is potentially a bug. However, there are also applications in other areas of reverse engineering: El-Ramly et al. [48] use a sequence mining approach for recovering user-usage scenarios of GUI based programs by mining series of screen identifiers.

The approach by Wasylkowski et al. [204] focusses on single objects and is closely related to static trace extraction. Therefore, I will discuss it in more detail. The technique detects usage patterns from code examples and then finds violations of these patterns, which may indicate potential errors. It starts with a state-based model similar to the control flow graph for a method (*method model*). Then, an *object usage model* is created by replacing all transitions that do not call a method on the given object nor use the object as a parameter to some method by epsilon transitions. The result is an automaton that describes how the object is used within one method. In the next step, frequent itemset mining is used for mining programming patterns. These patterns consist of sequences of two methods. Then, confidence calculation and formal concept analysis is applied for selecting violations by identifying imperfect (that is, quite similar) blocks: they are always formed by two neighboring blocks in the lattice. The violations are finally ranked using a *uniqueness factor*. This static approach has several commonalities with OPG extraction. It starts with the control flow graph, but it is just the intraprocedural one. It concentrates on single objects, but also on single methods. And it is aimed at extracting specifications, but just partial ones.

Mining Specifications of Malicious Behavior

Another related field is finding a special class of specifications: those that identify malicious behavior. This is useful for identifying programs that are infected by a certain malware or virus.

Sekar et al. [169] use a technique similar to DOPG extraction to learn state automata. They take into consideration the static *program state* (basically the program counter) which can be extracted from the call stack. The resulting automata represent sequences of system calls, and their states correspond to the static calling location. The automata are used for detecting anomalous program behavior. This is closely related to violations of a given protocol, and the idea could be regarded as a lightweight DOPG variant. However, it only models system call sites and their dependencies.

An approach by Christodorescu et al. [30] mines specifications of malicious behavior by comparing traces of infected software to traces of the original software. It analyzes def-use dependencies on parameters and return values, based on their types and values. This results in a partial data dependence graph, which is then constrasted with dependence graphs of benign programs. The result is a graph that characterizes the particularities of infected programs.

Schuler et al. [168] introduce a way to calculate a *dynamic birthmark* for Java programs. They observe how a program uses Java Standard API objects. If another program uses the same objects in exactly the same way, this is a sign that the corresponding code has been copied or stolen. This is also a kind of protocol that is used for a different purpose.

9.5.7 Process Mining

A topic closely related to protocol recovery and specification mining is process mining. It attempts to reconstruct models of business processes by analyzing concrete sequences of events. However, it faces a number of different challenges, such as alternative and parallel routing or human errors in the event logs.

The seminal work on this topic is presented by Cook and Wolf [32] who focus on discovering models of software processes. They infer finite state machines from event sequences. Different inference techniques are compared: Markov methods, neural networks, and grammar inference (k-tails). The markov and the grammar inference approach showed promising results. Weijters et al. [205] propose a heuristic approach which uses rather simple metrics to induce a dependency graph. The advantage of the heuristic approach is that it can deal with noisy data.

For an extensive discussion and survey of issues and approaches in workflow mining, the interested reader is referred to the article by van der Aalst et al. [195].

9.5.8 Grammar based Protocol Specification

Protocols can be specified based on a grammar. Some of the relevant approaches are discussed in this section. However, protocol specification is not in the focus of this thesis, so alternative specification techniques are not covered.

Regular Languages / Finite State Automata

The use of regular languages to model the dynamic behavior of objects was first suggested by Nierstrasz's "Regular types for active objects" [132]. Shortly after, Yellin et al. [221, 222] proposed to specify sequencing constraints by means of finite-state grammar. Their protocols are bidirectional: they distinguish between send, receive, and mixed states. This allows them to define and check protocol compatibility between two components. Another related approach to protocol specification is the Trace Assertion Method [138] which can also be modelled with finite automata as well [87]. Plasil et al. [144] define an architectural description language for behavior protocols that is similar to regular expressions. It is extended by operators for specifying parallelity. De Alfaro and Henzinger introduced the notion of "Interface automata" [40]. They argue that an interface's protocol should not only be safe, but also permissive, which means that they describe all sequences that do not lead to an error. In summary, the use of regular expressions for protocol specification is widely accepted.

Context-Free Grammars

A few recent publications combine FSA with context-free grammars for checking protocols. This has the advantage that the protocol can deal with recursion (see Section 7.4.2). Hughes et al. [83] introduce *interface grammars* that specify component protocols as context free grammars. They automatically generate a component stub that implements a parser which in turn checks conformance to the grammar. Thus, such protocol specifications can only be checked dynamically. Zimmermann et al. [228] follow a different approach. They model the protocol with FSA, but component use with a context-free grammar. Then they present an algorithm for statically checking $L(G) \subseteq L(A)$ for a CFG G and an FSA A.

9.5.9 Protocol Validation

When a protocol of a given component has been extracted, it may be used for checking if a client application uses the component correctly. Doing this check dynamically is easy: the state of the FSA is simply updated as the component's methods are called. When the FSA gets into an error state or a transition is not supported, an error has been detected. Checking correct use of the protocol statically is more difficult. We present some approaches to static protocol validation in the following.

Olender and Osterweil [136] use a data-flow framework in which state transitions are propagated through the control flow graph. This allows checking for existentially quantified constraints. Their approach can only be applied for parameterless routines. Therefore, if operations relate to objects, this technique cannot be applied directly. OPGs remove parameters from operations, since all operations relate to one particular object – so they may be used as input to this approach.

Butkevich et al. [23] introduce a Java language extension for specification of sequencing constraints. The order in which the methods of a class may be called is specified by a *labeled transition system* which describes an NFA by regular expressions and states. These constraints can be checked statically. Additionally, *state predicates* allow extended checks during runtime by choosing among branches of the NFA.

Engler et al. [49] use programmer-written compiler extensions that check a given rule. This mechanism is called "meta-level compilation". The rules are formulated in a state-machine language called *metal*. The authors claim that their approach has detected hundreds of errors in real-world systems.

Holger Bär [10, 11] proposes another static approach to verification of component protocols. He extracts the usage protocol from the code (this is related to static trace extraction) and then checks whether the language of the resulting automaton is a subset of the specified protocol. He uses extended automata for which the allowance of transitions may depend on the return value of a boolean function of the component. This somewhat increases the expressiveness of the regular language.

All these approaches can be used to check if a given program uses a component correctly, if the protocol of that component is available.

9.6 Experimentation in Software Engineering

Experimentation as a means for testing or disproving theories is essential for disciplines such as physics or medicine. However, as noted by Basili [12], it has long been widely ignored in Software Engineering. Only during the last ten years, there has been an increased interest in this topic. This section summarizes some of the published work. Since the field is quite large meanwhile, it concentrates on a few representative overview articles and books.

An elaborate book about experimentation in Software Engineering has been written by Lutz Prechelt [145]. It contains many examples, discussion of published experiments, and useful hints. Unfortunately, it is written in German, and it is out of print. Another German book on the topic is Andreas Zendler's professional dissertation [227], which contains an overview of historical experimental Software Engineering results and a methodology for conduction of such experiments.

There are a number of surveys on experimentation in Software Engineering. Sjøberg et al. [172] identify 113 controlled experiments in 5,453 articles and conference papers on Software Engineering. 87% of the experiments' subjects were students, and only 9% were professionals. Commercial applications were used in 14% of the experiments. Höfer et al. [79] investigate the articles of one journal only (Journal of Empirical Software Engineering). They find that professionals are used as subjects in 78.9% of the case studies, but controlled experiments are mostly conducted with students (60%). Another survey by Tonella et al. [193] et al. concentrates on empirical studies in reverse engineering. It reports that of 260 papers and articles on the topic, 24.6% did not provide any empirical evidence at all. Another 53.4% contained only case studies or experience reports. 21.8% had an evaluation which controlled the setting (quasi-experiments, controlled experiments, and observational studies). Only seven papers or articles reported from a controlled experiment. In summary, controlled experiments are still widely neglected in Software Engineering research.

Rajlich et al. [151] state that program comprehension is a research area where it is feasible to validate one's claims with relatively inexpensive experiments. They propose to measure accuracy, accurate response time, and inaccurate response time, which is pretty close to the dependent variables in the DOPG experiment (Chapter 8). Ten years later, Di Penta et al. [143] give a good overview of empirical studies on program comprehension and the problems associated with it. They also cite a lot of examples where empirical studies in the context of program comprehension have been conducted.

The experiment as described in Chapter 8 is also inspired by Storey's publications [181, 182]. Although her work focusses on evaluating interfaces of reverse engineering tools, many of the ideas can be applied to related areas. For the DOPG experiment, the focus was not on the interface, but on the usefulness for program comprehension and maintenance.

Chapter 10

Conclusions

This chapter summarizes the contributions and conclusions of this thesis and proposes further research directions.

10.1 Summary and Conclusions

This thesis introduced Dynamic Object Process Graphs, along with techniques for their extraction and their applications. Dynamic Object Process Graphs are a projection of the control flow graph to those nodes and edges that are relevant for one particular object. They summarize dynamic operations sequences for this object, representing loops and control dependencies. This representation of an object trace overcomes the space problem of traditional tracing, which is caused by the creation of huge traces for longer program runs. Dynamic Object Process Graphs are limited in size, and we have seen that it is possible to construct them on-the-fly while the program is running.

In several case studies, the behavior of different components was extracted as Dynamic Object Process Graphs. These graphs can help to understand how a particular component is used in an application. If the component is a key component of the application, its Object Process Graphs may give an overall picture of the application and help to understand the system as a whole. On average, the Dynamic Object Process Graphs for these components were less than 1% of the size of the global static control flow graph. This reduction shows the potential for extracting the relevant information from an otherwise huge information space spanned by all possible behavior.

Dynamic Object Process Graph extraction is an enabling technique similar to program slicing with applications in program comprehension, testing, and protocol recovery. Its feasibility and use was demonstrated in several case studies with non-trivial programs. The application of DOPGs for protocol recovery was described in detail, and the comparison to other dynamic protocol recovery techniques showed that DOPGs are a good basis for that: it often delivers better results than the other techniques.

Also, the use of DOPGs for program understanding was investigated. A controlled experiment was conducted to find out whether DOPGs are useful for program understanding or not. The results were not absolutely clear: for one system, it was clearly useful, but for another program, it was not. The conclusion is that this is most probably due to the different sizes of DOPGs: they only seem to be usable when the graph is not too large.

Now let us look back at the hypotheses that were stated at the beginning of this thesis in Section 1.2.

Hypothesis 1: OPGs can be extracted dynamically (feasibility). Based on the case studies in Chapters 4 and 6, this first hypothesis can clearly be confirmed: dynamic OPG extraction is possible, and it is applicable in practice. This was shown even for large and interactive systems. The choice of extraction technique (online versus offline) depends on the number of objects that is to be traced. For single or few objects, the online technique is faster, but for many or all objects, the offline technique should be used. However, there may be cases where dynamic OPG extraction is not possible, for example when timing is critical, which is affected by instrumentation. Also, execution may be difficult in the embedded domain. This is where using the corresponding static analysis may still be an option.

Hypothesis 2: DOPGs are a good basis for protocol recovery. Chapter 7 explained in detail how DOPGs can be transformed to protocol automata. The approach was implemented and compared to a number of other dynamic protocol recovery techniques. In this comparison, DOPG based protocol recovery delivered good results that often were better than other approaches' results. Therefore, this hypothesis was as well confirmed.

Hypothesis 3: Visualized DOPGs can be helpful for program understanding. The fact that DOPGs *can* be helpful for program understanding was also confirmed by the case studies in Chapter 6 and the controlled experiment in Chapter 8. However, it could not be shown that DOPGs are *always* useful for program understanding. Also, the cases *when* they are useful for *which* tasks could not be conclusively identified. We just collected evidence that DOPGs are useful in certain cases, and we got some hints about the conditions for that.

10.2 Opportunities for Future Research

During the research on this topic, a number of continuative items and ideas that could be worth investigating have been identified. They are presented and discussed in this section.

10.2.1 Identifying Appropriate Objects

An open question is how to find objects that deliver the information that is desired. Of course, this depends on the goal of the analyst: if the goal is protocol recovery, this choice is straight-forward. But when the goal is to get an initial overview of the whole system, the choice may not be obvious. In my case studies, I was quite successful in identifying adequate objects based on their name and some trial and error. However, an automatic technique that identifies good candidates would be helpful.

One approach that could be worth trying is using a webmining metric. Zaidman et al. [223, 224, 226] applied such a metric on runtime coupling information and reported good results for identifying key classes of a system. The advantage of using a webmining metric is that it takes transitive dependencies into account. The question is if Zaidman's key classes are of a kind that is also relevant for DOPG extraction.

10.2.2 Combination with Feature Location

Another kind of dynamic analysis is the feature location approach as described by Eisenbarth, Koschke, and Simon [44]. They use concept analysis to classify routines as specific, conditionally specific, relevant or irrelevant for a given feature. The same approach also works on basic block level [99]. However, the resulting concepts are quite fine-grained and often distributed throughout the code.

Dynamic tracing can be combined with that approach. Instead of looking at routines or basic blocks, one could use nodes and edges of the Dynamic Object Process Graph as executed units. Concept analysis then tells us which nodes and edges are specific, relevant, and so on for each feature. The advantage would be that the results are much more readable than sets of basic blocks, since we will usually get contiguous sequences of operations (that is, connected nodes) for a test case. In contrast to that, basic blocks may be distributed throughout the program, and their relationships may not be easily comprehensible. Another advantage is that DOPG level result are more detailed than routine level information, because DOPG nodes correspond to individual statements. Also, concept analysis could help to make large DOPGs more readable by restricting them to certain features. This could lead to a good compromise between completeness and readability.

10.2.3 Layout Algorithms for DOPGs

In the context of this thesis, only rudimentary layout algorithms (*spring embedder*) and some manual postprocessing were applied for visualization of DOPGs. However, it could be beneficial to create advanced layout algorithms that are particularly suited to DOPGs. A better layout may even further improve the usability of DOPGs for program comprehension. The effect of the layout on helpfulness for program comprehension could as well be investigated by a controlled experiment.

There are several possibilities for improving the layout for DOPGs. For example, nodes that belong to the same function could be grouped together and marked by a common background color. Functions that are called from many places could be moved to a central location, or maybe also duplicated. The direction of control flow could be visualized by a top-down or left-to-right order of the nodes. The layout algorithm could also handle the typical structures of DOPGs (such as call chains) in a specialized way.

10.2.4 Improving Protocol Recovery

Transition Order and Counts

Information about transition order and transition counts is currently not evaluated in the analysis. This data could help to further improve protocol recovery results. Cases where a loop body always contains only one relevant method call or attribute access for an object (as shown in Figure 7.6) could be handled better this way. On the other hand, since dynamic analysis is hardly ever complete, this could lead to different errors: maybe in some special cases, there can be more than one relevant operation.

Context Sensitivity

Another source of generalization is the missing context sensitivity. Adding context sensitivity would improve the precision of protocol recovery results. However, for program understanding purposes, it is probably better to use the smaller context-insensitive graphs. This is because larger graphs are harder to understand, and because it may be confusing to have multiple nodes for the same source location. See Section 7.4.3 for a discussion of the effect of these extensions.

10.2.5 Using Concurrency Information

It may also be useful to include information about threads, locks, and the like in DOPGs. For example, Java's synchronization behavior could be additionally traced and transformed into the DOPG. Visualizing the information which thread executed which statements and which thread locked which objects could help to identify concurrency problems. When several objects are involved, DOPGs could also be extended to display more than one object at once by overlaying their graphs. The common synchronization points may indicate potential conflicts. If the involved objects (or their classes) are known, this may help in the investigation of concurrency problems.

10.2.6 Online Visualization

Online tracing was introduced to avoid writing large trace files. However, it has more advantages than that: it also enables online visualization of intermediate results. In particular, the current raw graphs can be transformed to DOPGs from time to time (say, every second) which can then be visualized while the program is running. Such an online visualization has been prototypically implemented and delivers interesting results. It shows how a DOPG evolves when features of the subject system are executed. However, this technique was not investigated any further.

One idea for improving the usefulness of this visualization is to remember the time of last execution for each node or edge. Recently visited nodes could then be shown bright and slowly fade out until they are executed again. This could give an even better impression about which parts of the DOPG are currently active. Similarly, node execution *counts* could be visualized.

Another related idea is to allow offline visualization of the evolving graph. Such a tool could be used as a kind of graphical debugger to follow the flow of control through the graph. This could be particularly useful in conjunction with concurrency information, as proposed above.

10.2.7 Product Line Consolidation

When several variants of a software that originate from the same code base are to be consolidated, the question is where they differ. To create a product line from these variants, the differences have to be unified using *variation points*.

DOPGs or protocol automata may be useful for finding these points. By comparing the DOPGs for the same object from two software variants, the point where behaviour differs can be identified and proposed as a potential variation point. A similar idea was proposed by Cornelissen et al. [34], who work directly on the traces.

The topic of DOPG based identification of variation points is currently being investigated by Bernhard Scholz in his diploma thesis at the University of Bremen.

10.2.8 Criteria for Usefulness

The question when DOPGs are useful for which program understanding tasks could not be answered conclusively by the experiment from Chapter 8. However, the experiment showed up some directions for further investigation of this question. The results indicated that the usefulness probably depends on the graph size and on the number of extracted graphs. Another experiment could examine a corresponding hypothesis. Yet, there may also be other factors that affect the results. For example, it may depend on the structure or architecture of the subject system if DOPGs are useful for its analysis. Therefore, clarification of this question is difficult; probably, numerous experiments would be required to solve it.

10.3 Closing Words

Work on this thesis started with the need to create a dynamic counterpart for "Static Trace Extraction" [46]. However, as work on the dynamic technique progressed, Dynamic Object Process Graphs turned out to be much more than a counterpart. Completely new application potentials were discovered and investigated, such as online construction and visualization for program understanding. And, as the previous section showed, DOPGs pave the way for many other uses. Dynamic Object Process Graphs and their extraction is therefore an enabling technique. I hope that the technique and results will be useful as a basis for further research, which may come up with additional exciting applications of DOPGs. The potential is immense. Hopefully it will be employed.

Appendix A

Finite State Automata

This section formally defines finite state automata and contains some algorithms on automata that are needed in the thesis, in particular in the protocol recovery chapter. Details can be found in the book *"Introduction to Automata Theory, Languages, and Computation"* by John Hopcroft et al. [81].

A.1 Basic Definitions

A **deterministic finite state automaton (DFA)** is defined as the quintupel

$$A = (Q, \Sigma, \delta, q_0, F)$$

where Q is a finite set of states, Σ is a finite set of *input symbols*, $\delta : Q \times \Sigma \rightarrow Q$ is the transition function, $q_0 \in Q$ is the *start state*, and $F \subseteq Q$ is the set of *final* or *accepting states*. The transition function δ may be only partially defined, that is, it may be undefined for certain combinations of Q and Σ.

A **non-deterministic finite state automaton (NFA)** is defined as a DFA where the transition function maps to sets of states instead of a single state: $\delta : Q \times \Sigma \rightarrow 2^Q$

A transition on the empty string is called an **epsilon-transition**. The empty string is denoted ϵ. For a DFA, $\delta(q, \epsilon) = q \;\; \forall q \in Q$.

The **extended transition function** describes what happens when we start in any state and follow any sequence of inputs. For a DFA, it is defined like this:

$$\begin{aligned} \hat{\delta}(q, \epsilon) &= q \\ \hat{\delta}(q, w) &= \delta(\hat{\delta}(q, x), a) \;\; \forall x \in \Sigma^+,\; a \in \Sigma,\; w = xa \end{aligned}$$

The corresponding NFA definition is:

$$\begin{aligned} \hat{\delta}(q, \epsilon) &= \{q\} \\ \hat{\delta}(q, w) &= \bigcup_{i=1}^{k} \delta(p_i, a) \;\; \forall x \in \Sigma^+,\; a \in \Sigma,\; w = xa \end{aligned}$$

$$\text{with} \qquad \hat{\delta}(q,x) = \{p_1, p_2, \ldots, p_k\}$$

The accepted **language** $L(A)$ of an automaton A is defined as the set of continuous transition sequences that start in the start state q_0 and end in an accepting state:

$$L(A) \quad = \quad \{w \in \Sigma^* \mid \hat{\delta}(q_0, w) \in F\}$$

A.2 Algorithms

A.2.1 NFA to DFA: The Subset Construction

Given an NFA $N = (Q_N, \Sigma, \delta, q_N, F_N)$, the **subset construction** constructs a DFA D that accepts the same language as N:

$$\begin{aligned} D \;&:=\; (Q_D, \Sigma, \delta_D, \{q_N\}, F_D) \\ \text{with} \quad & \forall S \subseteq Q_N,\ a \in \Sigma :\ \delta_D(S,a) := \bigcup_{p \in S} \delta_N(p,a) \\ & Q_D := 2^{Q_N},\ \ F_D := \{S \subseteq Q_N | S \cap F_N \neq \emptyset\} \end{aligned}$$

Usually, the number of reachable states of D is much smaller than $2^{|Q_N|}$. The subset construction is therefore calculated using *lazy evaluation* – calculating only those transitions that are really needed. Figure A.1 shows the basic algorithm for calculating Q_D and δ_D.

```
$Q_D := \{\{q_N\}\};\ T := \emptyset$
while $Q_D \setminus T \neq \emptyset$ do
    select $S \in Q_D \setminus T;\ \ T := T \cup \{S\}$
    foreach $a \in \Sigma$ do
        $R := \bigcup_{q \in S} \delta_N(q,a);\ \ Q_D := Q_D \cup \{R\};\ \ \delta_D(S,a) := R$
```

Figure A.1: Subset construction with lazy evaluation.

A.2.2 The Union of two Automata

Given two automata $A_L = (Q_L, \Sigma_L, \delta_L, q_L, F_L)$ and $A_M = (Q_M, \Sigma_M, \delta_M, q_M, F_M)$, the automaton that accepts the union of both automata languages – the **union automaton** – can be constructed in the following way:

$$\begin{aligned} A_U \;&:=\; (Q_U, \Sigma_L \cup \Sigma_M, \delta_U, q_0, F_L \cup F_M) \\ \text{with} \quad & Q_U := Q_L \cup Q_M \cup \{q_0\} \\ & \delta_U := \delta_L \cup \delta_M \cup \{(q_0, \epsilon) \mapsto \{q_L, q_M\}\} \end{aligned}$$

```
Input  : a DFA $A_L = (Q_L, \Sigma, \delta_L, q_L, F_L)$
Output: a DFA $A_M = (Q_M, \Sigma, \delta_M, q_M, F_M)$ with $L(A_L) = L(A_M)$
        and $|Q_M|$ is minimal
// find out all distinguishable pairs of states
$R := \{\{p,q\} \mid p \in F_L,\ q \in Q_L \setminus F_L\}$
while $\exists p, q \in Q_L,\ a \in \Sigma:\ \{\delta_L(p,a), \delta_L(q,a)\} \in R$ do
    $R := R \cup \{\{p,q\}\}$
// find out all sets of equivalent states
$S := \{\{p,q\} \mid p, q \in Q_L\} \setminus R$
while $\exists P, Q \in S:\ P \neq Q$ and $P \cap Q \neq \emptyset$ do
    $S := ((S \setminus P) \setminus Q) \cup (P \cup Q)$
// merge the states of each set $S$
$Q_M := S \cup \{\{q\} \mid q \in Q_L \setminus \bigcup_{T \in S} T\}$
$F_M := \{T \in Q_M \mid T \cap F_L \neq \emptyset\}$
$q_M :=$ the $T \in Q_M$ with $q_L \in T$
$\delta_M(T, a) := V \in Q_M$ such that $\exists p \in V:\ \delta_L(q,a) = p$   for $T \in Q_M, a \in \Sigma$
```

Figure A.2: Table-filling algorithm for calculating a DFA with a minimum number of states. R contains pairs of distinguishable states (and could be represented by a table), S contains pairs of indistinguishable states.

A.2.3 Automaton Minimization

Two states p and q of a DFA are **equivalent** if:

$$\forall w \in \Sigma^* : \quad \hat{\delta}(p,w) \in F \quad \Leftrightarrow \quad \hat{\delta}(q,w) \in F$$

Two states that are not equivalent are called *distinguishable*. The minimization of an automaton with respect to its number of states is equivalent to removing all states that are equivalent to some other state. The algorithm that achieves this is called the *table-filling algorithm* by Hopcroft et al. and presented in Figure A.2.

A.2.4 Product Automaton

Given two automata $A_L = (Q_L, \Sigma, \delta_L, q_L, F_L)$ and $A_M = (Q_M, \Sigma, \delta_M, q_M, F_M)$, the **product automaton** A_P is the automaton that accepts $L(A_L) \cap L(A_M)$. It is constructed as follows:

$$\begin{array}{rcl} A_P & := & (Q_L \times Q_M, \Sigma, \delta, (q_L, q_M), F_L \times F_M) \\ \text{with} & & \delta((p,q),a) := (\delta_L(p,a), \delta_M(q,a)) \ \forall p \in Q_L,\ q \in Q_M,\ a \in \Sigma \end{array}$$

Usually, not all of the $|Q_L| \cdot |Q_M|$ states are reachable. As in the subset construction, lazy evaluation can be used to reduce the calculation to only reachable states. The algorithm is shown in Figure A.3.

```
$Q_D := \{(q_L, q_M)\};\ T := \emptyset$
**while** $Q_D \setminus T \neq \emptyset$ **do**
    select $s \in Q_D \setminus T;\quad T := T \cup \{s\}$
    **foreach** $a \in \Sigma$ **do**
        **if** $\delta_L(p, a)$ is defined **and** $\delta_M(q, a)$ is defined **then**
            $r := (\delta_L(p, a), \delta_M(q, a))$
            $Q_D := Q_D \cup \{r\};\quad \delta_D(s, a) := r$
```

Figure A.3: Product automaton calculation with lazy evaluation.

Appendix B

Graph Transformation Basics

This section gives a quick introduction to graph transformations. More details can be found elsewhere [6, 103].

Let Σ be a set of labels. A multiple directed labelled **graph** over Σ is a system $G = (V, E, s, t, l)$ where V is a finite set of **nodes**, E is a finite set of **edges**, $s, t : E \to V$ are mappings assigning a **source** $s(e)$ and a **target** $t(e)$ to every edge in E, and $l : E \to \Sigma$ is a mapping assigning a label to every edge in E. An edge e in G goes from the source $s(e)$ to the target $t(e)$ and is **incident** to $s(e)$ and $t(e)$. The components V, E, s, t, and l of G are also denoted by V_G, E_G, s_G, t_G, and l_G, respectively. The set of all graphs over Σ is denoted by $\mathcal{G}_\Sigma$.

A graph $G \in \mathcal{G}_\Sigma$ is a **subgraph** of a graph $H \in \mathcal{G}_\Sigma$, denoted by $G \subseteq H$, if $V_G \subseteq V_H$, $E_G \subseteq E_H$, $s_G(e) = s_H(e)$, $t_G(e) = t_H(e)$, and $l_G(e) = l_H(e)$ for all $e \in E_G$.

For graphs $G, H \in \mathcal{G}_\Sigma$, a **graph morphism** $g : G \to H$ is a pair of mappings $g_V : V_G \to V_H$ and $g_E : E_G \to E_H$ that are structure-preserving, i. e. $g_V(s_G(e)) = s_H(g_E(e))$, $g_V(t_G(e)) = t_H(g_E(e))$, and $l_H(g_E(e)) = l_G(e)$ for all $e \in E_G$.

For a graph morphism $g : G \to H$, the image of G in H is called a *match* of G in H, i. e. the match of G with respect to the morphism g is the subgraph $g(G) \subseteq H$ which is induced by $(g(V), g(E))$.

A **graph transformation rule** $r = (L \supseteq K \subseteq R)$ consists of three graphs $L, K, R \in \mathcal{G}_\Sigma$ such that K is a subgraph of L and R. The components L, K, and R of r are called **left-hand side**, **gluing graph**, and **right-hand side**, respectively. When the gluing graph is a set of nodes, the graphical representation of r may omit the gluing graph K by depicting only the graphs L and R, with numbers or symbols uniquely identifying the nodes in K. In this thesis, gluing graph nodes are filled with gray.

The **application of a graph transformation rule** to a graph G consists of replacing a match of the left-hand side in G by the right-hand side such that the match of the gluing graph is kept. Hence, the application of $r = (L \supseteq K \subseteq R)$ to a graph $G = (V, E, s, t, l)$ comprises the following three steps (in the single-pushout approach):

1. **Choose** an occurence of the left-hand side L in G by choosing a graph morphism $g : L \to G$.

2. **Check** the application conditions.

3. **Remove** the occurence of L up to the occurence of K $g(K)$ from G as well as all *dangling edges*, i. e. all edges incident to a removed node. This yields the *context graph* Z of L which still contains an occurence of K. $Z = G \subseteq (g(L) \subseteq g(K))$

4. **Glue** the context graph D and the right-hand side R according to the occurences of K in Z and R. That is, construct the disjoint union of Z and R and, for every item in K, identify the corresponding item in Z with the corresponding item in R. This yields the *gluing graph* $H = Z + (R - K, g)$ where $(R - K, g) = (V_R \setminus V_K, E_R \setminus E_K, s', t', l')$ with
$s'(e') = s_R(e')$ if $s_R(e') \in V_R \setminus V_K$ and $s'(e') = g(s_R(e'))$ otherwise,
$t'(e') = t_R(e')$ if $t_R(e') \in V_R \setminus V_K$ and $t'(e') = g(t_R(e'))$ otherwise, and
$l'(e') = l_R(e')$ for all $e' \in E_R \setminus E_K$.

$$\begin{tikzcd}
L \arrow[d, "g"] & \supseteq & K \arrow[d, "d"] & \subseteq & R \arrow[d, "h"] \\
G & \supseteq & Z & \subseteq & H
\end{tikzcd}$$

The application of a rule r to a graph G is denoted by $G \Rightarrow_r H$ where H is a graph resulting from an application of r to G. A rule application is called a **direct derivation**, and the iteration of direct derivations $G_0 \Rightarrow_{r_1} G_1 \Rightarrow_{r_2} \ldots \Rightarrow_{r_n} G_n$ ($n \in \mathbb{N}$) is called a **derivation** from G_0 to G_n. The derivation from G_0 to G_n can also be denoted by $G_0 \Rightarrow_P^n G_n$ where $\{r1, \ldots, r_n\} \subseteq P$, or by $G_0 \Rightarrow_P^* G_n$ if the number of direct derivations is not of interest. The string $r_1 \ldots r_n$ is called an **application sequence** of the derivation $G_0 \Rightarrow_{r_1} G_1 \Rightarrow_{r_2} \ldots \Rightarrow_{r_n} G_n$.

Appendix C

Experimenter's Handbook

The Experimenter's Handbook was written to ensure that all sessions are performed identically, or at least as similar as possible. In particular, each participant has to receive the same information in the same way in order to ensure that any session difference does not influence the results. The handbook provides some consistency and control over the running of each experimental session.

The handbook was originally written in German and was translated to English for reproduction in this thesis.

C.1 Introduction

This Experimenter's Handbook describes in detail how the experimenter has to proceed during an instance of the experiment. The different phases of the experiment are to be followed precisely as described in this handbook. Use it as a checklist to ensure you followed each step.

Material

The following documents and things have to be prepared for each participant. They have to be handed over to the participant at the very beginning of the experiment.

- 1 Pre-test questionnaire
- 2 Post-study questionnaires
- DOPG legend printout
- A pencil

The following software has to be installed and configured on every workstation:

- A digital clock that displays seconds

- Eclipse 3.2.2
  - Larger editor font, activate line number display
- Plugins for DOPGs and experimental procedure (exclipse)
- Three Eclipse workspaces, each containing one of the subject systems, according to the (randomly assigned) group
- Each workspace must contain the correct tasks for the group
- recordMyDesktop for recording screen contents

On the experimenter's laptop, the same Eclipse environment has to be installed. It must also contain the training slides. You further need four cards with the numbers 1 to 4 on it (for random group assignment).

Phases

The experiment consists of the following phases:

1. Welcome, orientation (10 min.)
2. Training talk (15 min.) and practice (25 min.)
3. Experimental tasks and questionnaires (2×[25+5] min)
4. Finishing (Debriefing, 10 min.)

Each phase has a certain time limit which must not be exceeded. First, all participants are welcomed and introduced to the experiment. The training phase gives the users the required skills in program understanding, Eclipse, how to use the relevant functions, the meaning of DOPGs, and how to use the two plugins. The practice part allows the participants to practice the techniques and tasks using a small software system and a set of practice tasks. In the following phase, the users solve a set of tasks on a given subject system and complete a post-study questionnaire. This phase is repeated with a different subject system and tasks. The finishing phase involves a short informal discussion.

Rules

The following general rules are to be followed:

- The experimenter is not allowed to help the participants in how to solve a task.
- The experimenter may answer simple questions about the tools.

C.2 Phase 1: Welcome & Introduction

- Randomly assign the participants to the workstations.

- Let each participant randomly draw one card (out of four) that determines which group he/she will be in.

- Configure the workstation for the respective group: `./configure [1-4]` This will configure the workspaces and tasks for this group and start screen recording.

- Start Eclipse. Workspace `ws0` is automatically selected.

- Let each participant fill out a pre-study questionnaire. Make sure that all questions are answered.

- Introduce the experimenter (yourself).

- Tell the participants what the experiment is about: "Evaluate the suitability of different tools for program understanding."

- Present general conditions and ask participants if these are okay with them (otherwise they cannot participate):

  - All data is collected anonymously and kept in confidence.
  - The results of the experiment will be published.
  - All user interactions and screen contents are being recorded.

- Tell them that a technique is being evaluated, not the participants.

- Tell them that it is not necessary to solve all tasks that will be presented. Each participant shall invest as much time as needed for each task and only then proceed to the next task.

- Tell them to turn off their mobile phones.

- Tell them that their performance in the experiment is independent from the lottery. The lottery is performed among all participants in a purely random way, and each participant has the same chance of winning a price – independent from their experimental results.

- Give an overview of the different phases and their duration (first slide of presentation).

C.3 Phase 2: Training and Practice

In this phase, you give an introduction to program understanding in general and how it can be accomplished with Eclipse standard features. Apart from that, there is an introduction of how to use the two Eclipse plugins and the meaning of DOPGs (how to read and use them).

The first part is of theoretical nature and provides the participants with knowledge they may require for the experiment. There is a presentation (slides) available that is to be used for this training. It covers the following topics:

- Why program understanding is important, and why we need tool support.
- Basic techniques: Static and dynamic analysis; concrete examples like textual search, cross reference tools, debugging.
- Introduction to Dynamic Object Process Graphs – their meaning, how they are constructed, and what kinds of the different nodes and edges there are.
- Description of the maintenance scenario.

Then, the participants have some time for practicing. The Jetris system and tasks are used for this part. Invite the participants to use their practice environment to try out things while you demonstrate the following:

- Introduce the subject system: Start Jetris, move and drop a few figures.
- Show source code navigation.
- Show textual search.
- Show Java search (cross reference).
- Show DOPG plugin:
  - Selecting a graph; the meaning of the names and numbers in the menu; where the graphs come from (different static allocation points); the relation to the source code (location of allocation point).
  - Navigation through the graph: zoom in/out, stretch/tighten, spring layout, jump to source, panning, go to Start/Create, find function.
  - Working with the graph.
- Show Experimental plugin (exclipse).

The practice tasks should be solved after this short demonstration. After about 10 minutes, the experimenter starts demonstrating the solutions to the first three tasks. He demonstrates how to find the answers a) with Eclipse standard features, b) with the help of DOPGs.

1. “Where is the main method?” Solutions: Manual search in project/browser, textual search by name, Java search for the signature, and find it using DOPGs (via find start node).

2. “Which is the location in the code where instances of the different Figure classes are created?” Solutions: Search for constructor, and search using DOPGs (DOPG list and create nodes).

3. “How is control (rotation etc.) of the Figures implemented, i. e., where is the key press event transformed to a corresponding application logic method call?” Solution: find “rotation” in Figures, then search for that; browse through DOPG (from start node) to find the dispatching location.

C.4 Phase 3: Experimental Tasks

At this point, the participants should have become somewhat familiar with the environment. Now demonstrate the two use cases that underly the DOPGs of the two subject systems. First start GanttProject, load the sample project, and enlarge a nested Gantt task to a length that exceeds the length of the parent’s task. Point the attention to the parent task which is automatically enlarged as well. Then start ArgoUML, create two classes, and connect them by an association.

Now ask the participants to switch their Eclipse workspace to `ws1` (`File/Switch Workspace`). The right project and tasks are then loaded automatically. Make sure that the necessary views are visible for each participant. The participants are now automatically guided through the differen tasks. After 25 minutes, everyone is notified by the experimental plugin that the time is over. Now ask everyone to fill out the post-study questionnaire. Finally, let everyone switch to `ws2` and start over again.

C.5 Phase 4: Finishing

The experiment ends with a short debriefing:

- Collect questionnaires.
- Answer any questions the participants might have about the experiment.
- Tell the participants about when the lottery is to take place.
- Tell them when they can expect to hear about the results of this experiment.
- Tell them that they are not allowed to talk about the experiment to future participants.
- Thank for participating!

Appendix D

Experimental Tasks and Instructions

The instructions and tasks that were presented to the experiment participants were presented by the Eclipse plugin "exclipse" that was specially developed for this experiment. It supports messages, timers, and textual aquisition of answers. The messages that were displayed on the different pages are presented in the following sections. They were originally written in German.

D.1 Practice System: Jetris

Page 1: General Remarks

The following tasks are intended to help you to get used to the required Eclipse features:

- Code browser
- Search function (text search)
- Cross reference function (Java search)
- DOPG plugin

First, you should try to get a general idea about the static structure of the Jetris project. Then you will continue with the practice tasks. Please start working on these tasks as soon as you are finished with getting basically comfortable with the Eclipse environment. The first tasks will be resolved by the experimenter, and he will demonstrate several possible approaches. Similar techniques should be used to solve the other tasks as well. It is probably best to try out different approaches. This way, you will learn which technique is best suited for which task.

Attention: Questions and tasks can only be switched forward. You cannot get back to a previous question. Please stick to one question, until you have found the answer. Only when you are absolutely sure that you cannot solve the task, write a comment into the answer textfield and proceed to the next task.

Page 2: Introduction to Jetris

The parts that are falling down in Jetris are called “Figures”. They are of central concern for a Tetris game. The graphs for the different Figure classes have been created during a typical Jetris session. They are available in the DOPG view.

Now change to the DOPG view and select the different graphs from the pull-down menu. The selected graph is then shown in the DOPG view.

You will find a similar information about a central class of the respective subject system when you start working on a new system.

Pages 3-8: Tasks

1. Where is the main method? I. e., in which class and in which line?
2. At which locations in the code are instances of the different Figure classes being created?
3. How is control of the figures (rotation etc.) implemented, i. e., where is the key event translated to an application logic method call?
4. Who is responsible for moving the current part downwards automatically?
5. When a figure has arrived at the bottom, a new figure appears at the top.
   (a) Which conditions in the code decide when a new figure has to be started from top?
   (b) Where are these conditions in the graph (indicate the node number, as displayed in the tool tip)?
6. How is it recognized that a row is completely filled (and must be eliminated)?

Page 9: End Page

The practice tasks are finished. You may continue to investigate the system or work with Eclipse until we continue.

D.2 System 1: ArgoUML

Page 1: Introduction to ArgoUML

In this part of the experiment, you will work on some tasks regarding ArgoUML. ArgoUML was already demonstrated briefly. You can also start it yourself (Run/Run Last Launched). After startup, a class diagram editor opens up. We defined two classes and connected them by an association, and then we created a new class diagram.

Experimental group: This use case was also performed to extract a DOPG for the ClassDiagramGraphModel instances. Choose the graph in the DOPG view

(UMLClassDiagram); it is then displayed in the DOPG view. You should try to use this graph for solving the upcoming tasks.

Control group: The class ClassDiagramGraphModel is of central concern for class diagrams in ArgoUML. It may be helpful as a starting point of your analyses.

Pages 2-4: Tasks

1. Which code has to be changed to make ArgoUML open up an empty sequence diagram instead of an empty class diagram on startup?
2. How is the addition of objects to a diagram (e. g., adding a class to a class diagram) implemented?
   More precisely, which class implements that? Give a short description how it works.
3. In ArgoUML, it is possible to browser through the selection history. Which class is responsible for recording selections?

Page 5: End Page

The tasks for ArgoUML are finished. You may continue to investigate the system until the experiment continues. You can use the text input box if you want to submit any comments.

D.3 System 2: GanttProject

Page 1: Introduction to GanttProject

In this part of the experiment, you will work on some tasks regarding GanttProject. GanttProject was already demonstrated briefly. You can also start it yourself (Run/Run Last Launched). After startup, the Gantt diagram editor opens up. We loaded the file HouseBuildingSample.gan (Project/Recent Projects). We then changed the duration of a subtask by enlarging the task's bar over the limits of its parent task.

Experimental group: This use case was also performed to extract DOPGs for the GanttTask instances. The class "GanttTask" represents the work packages. Choose a graph in the DOPG view (TaskManagerImpl); it is then displayed in the DOPG view. You should try to use these graphs for solving the upcoming tasks.

Control group: The class GanttTask represents a working package. It is of central concern for Gantt diagrams in GanttProject. It may be helpful as a starting point of your analyses.

Pages 2-4: Tasks

1. The tasks are organized as a hierarchy: parent tasks are displayed as a bracket and enclose the period in which all child tasks are contained. How

is the length of a parent task adjusted when the length of a child task is extended over the limits of the parent's period (i. e., which code is responsible for that)?

2. Which component is responsible for drawing a task, i. e., where is the rectangle of the task bar drawn?
3. Gantt diagrams support dependencies: a task can only be started when another one is finished. Which class is responsible for holding this dependency information?

Page 5: End Page

The tasks for GanttProject are finished. You may continue to investigate the system until the experiment continues. You can use the text input box if you want to submit any comments.

Appendix E

Experimental Questionnaires

E.1 Pre-study Questionnaire

1. How many years of programming experience do you have?
   ☐ years
2. How many years of Java programming experience?
   ☐ years
3. How would you rate your programming experience?
   ☐ none, ☐ weak, ☐ average, ☐ good, ☐ very good
4. What is the largest system you have worked on?
   ☐ lines of code, or ☐ files Language(s): ☐
5. How much experience do you have with maintaining code written by someone else?
   ☐ none, ☐ very little, ☐ some, ☐ quite some, ☐ a lot
6. Which programming environments and tools do you normally use?
   ☐ Eclipse, ☐ NetBeans, ☐ others – which ones? ____________________
7. How familiar are you with using the Eclipse workbench for Java development?
   ☐ none, ☐ very little, ☐ some, ☐ quite some, ☐ a lot
8. How much experience do you have with Java GUI development?
   ☐ none, ☐ very little, ☐ some, ☐ quite some, ☐ a lot
9. Did you attend the Software Reengineering Lecture?
   ☐ no, ☐ yes, ☐ currently attending
10. Do you know the program GanttProject?
    ☐ no, ☐ no, but I know Gantt diagrams/tools, ☐ yes, already used it,
    ☐ yes, I use it regularly, ☐ yes, I know the code

11. Do you know the program ArgoUML?
☐ no, ☐ no, but I know UML diagram tools, ☐ yes, already used it,
☐ yes, I use it regularly, ☐ yes, I know the code

E.2 Post-study Questionnaire

1. I could effectively complete the tasks using the provided tools.
☐ no, ☐ rather no, ☐ rather yes, ☐ yes, ☐ no reply
2. What was missing for completing the tasks effectively?
3. I am sure that my results are correct.
☐ no, ☐ rather no, ☐ rather yes, ☐ yes, ☐ no reply
4. Which tools or features of Eclipse did you use (0 = not used, 4 = intensively used), and how helpful did they turn out to be for solving the tasks (0 = not helpful at all, 4 = very helpful)?

	usage intensity					helpfulness				
	0	1	2	3	4	0	1	2	3	4
Codebrowser	○	○	○	○	○	○	○	○	○	○
Text search	○	○	○	○	○	○	○	○	○	○
Java search	○	○	○	○	○	○	○	○	○	○
Debugger	○	○	○	○	○	○	○	○	○	○
Graphs	○	○	○	○	○	○	○	○	○	○
others:										
____________		○	○	○	○	○	○	○	○	○
____________		○	○	○	○	○	○	○	○	○

5. Further remarks:

Appendix F

Statistical Tests

A number of statistical tests were used in the experiment to test the null hypothesis that the means of two populations are equal. Each test results in a p-value which can be regarded as the probability of getting the observed result by pure chance. These tests are briefly introduced in the following.

F.1 Student's t-test

A t-test is a statistical hypothesis test in which the test statistic has a Student's distribution if the null hypothesis is true. It assumes that the population is normally distributed. We use the variant by Welch for unequal sample sizes and potentially unequal variance. Let X and Y be the samples from the two populations.

$$t = \frac{\overline{X} - \overline{Y}}{\sqrt{\frac{s_X^2}{|X|} + \frac{s_Y^2}{|Y|}}}$$

where X is the sample mean and s_X is the sample variance. The degrees of freedom d is approximated as follows:

$$d = \frac{\left(\frac{s_X^2}{|X|} + \frac{s_Y^2}{|Y|}\right)^2}{\frac{s_X^4}{|X|^2\cdot(|X|-1)} + \frac{s_Y^4}{|Y|^2\cdot(|Y|-1)}}$$

The t and d values can then be used with the t-distribution to calculate the p-value.

F.2 Mann-Whitney U test

The Mann-Whitney U test is a non-parametric test for assessing whether two samples of observations come from the same distribution. Whereas Student's t-test requires an interval scale, the U test only requires an ordinal scale, and it does not assume a certain distribution.

Given two samples X and Y, the U value is calculated as follows:

1. Arrange all the observations into a single ranked series, no matter which sample they are in. Sort the values and then number them from 1 to $N = |X| + |Y|$, that is, assign each one its rank.

2. Add up the ranks for the observations which came from sample X, giving R_X. The sum of ranks in sample Y is then $R_Y = \frac{N \cdot (N+1)}{2} - R_X$.

3. $U_X = R_X - \frac{|X| \cdot (|X|+1)}{2}$, $U_Y = R_Y - \frac{|Y| \cdot (|Y|+1)}{2}$, $U = \min(U_X, U_Y)$

The value for p can then be read from a significance table.

F.3 Bootstrapping

Bootstrapping is a statistical method that is based on resampling [43]. It produces an arbitrarily large sample based on a given (small) sample. The distribution of the sample is considered to be the distribution of the underlying universe. The method is non-parametric and independent from the size of the available sample.

In the DOPG experiment, bootstrapping is used to calculate the p-value. Given two samples X and Y, the p-value is calculated as follows:

1. Sample n data points with replacement from the original data and calculate their average $\overline{x}$.

2. Repeat step (1) a large number of times N for both samples X and Y, resulting in bootstrap estimates $\overline{x}_1, \overline{x}_2, \ldots, \overline{x}_N$ and $\overline{y}_1, \overline{y}_2, \ldots, \overline{y}_N$.

3. Calculate the difference between each pair $d_i = \overline{x}_i - \overline{y}_i$ for $i \in \{1, \ldots, N\}$.

4. Sort the differences as $\overline{d}_{(1)} \leq \overline{d}_{(2)} \leq \ldots \leq \overline{d}_{(N-1)} \leq \overline{d}_{(N)}$.

5. Find the zero crossing z of this sequence: $\overline{d}_z < 0$ and $\overline{d}_{z+1} \geq 0$. Its position indicates the one-sided p-value: $p = \min(\frac{z}{N}, \frac{N-z}{N})$.

Glossary

Allocation Point The point in a program where the regarded object is created; for example, the location of a `malloc` or `new`.

Atomic Method In an OPG for a given object, the methods that implement the object's interface are called *atomic*.

Attribute A logical data value of an object.

Black Box (1) A system or component whose inputs, outputs, and general function are known but whose contents or implementation are unknown or irrelevant. (2) Pertaining to an approach that treats a system or component as in (1). *Contrast with:* Glass box [85]

Class In object-oriented programming, a *class* is a blueprint to create objects. It defines the attributes and methods that the created objects all share.

Component A group of related elements with a unifying common goal or concept relevant at the architectural level. An *atomic component* is a non-hierarchical component that consists of related global constants, variables, subprograms, and/or user-defined types. A *subsystem* is a hierarchical component consisting of related atomic components and/or lower-level subsystems. [96] In this thesis, the terms *component* and *atomic component* are used synonymously.

Conceptual View The *conceptual view* is a view of the software architecture. In this view, the functionality of the system is mapped to architecture elements called *conceptual components*, with coordination and data exchange handled by elements called *connectors*. [80]

Dynamic Analysis The process of evaluating a system or component based on its behavior during execution. *Contrast with:* Static Analysis [85]

Event A notable occurence at a particular point in time. In dynamic analysis, an *event* occurs when an instrumentation point is passed. The event is then either written to file or processed on-the-fly. *Also see:* Online, Offline

Execution Time The amount of elapsed time used in executing a computer program. [85]

Execution Trace A record of the sequence of instructions executed during the execution of a computer program. Often takes the form of a list of code labels encountered as the program executes. [85]

Finite State Machine/Automaton A computational model consisting of a finite number of states and transitions between those states, possibly with accompanying actions. [85]

Function A software module that performs a specific action, is invoked by the appearance of its name in an expression, may receive input values, and returns a single value. [85] A function does not modify the state.

Glass Box (1) A system or component whose internal contents or implementation are known. (2) Pertaining to an approach that threats a system or component as in (1). *Contrast with:* Black Box [85]

Global Variable A variable that can be accessed by two or more non-nested modules of a computer program without being explicitly passed as a parameter between the modules. *Also see:* Local Variable [85]

Heap Object/Variable An object or variable that is created during runtime and allocated on the heap. It is for example created by calling `malloc` in C or `new` in Java. Such an object has to be explicitly destroyed again (`free` in C), or it may be removed by a garbage collector (Java).

Instance One concrete exemplar of an abstraction that supports multiple copies of the state space, for example abstract data types or classes. The instances of classes are called *objects*. [188]

Instrumentation Instructions inserted into software to monitor the operation of a system or component. [85]

Interface The services of a module are defined by the *interfaces* it provides. A module may also need the services of another module to perform its function. These services are defined by the *interfaces* it requires. [80]

Label A name or identifier assigned to a computer program statement to enable other statements to refer to that statement. [85]

Local Variable A variable that can be accessed by only one module or set of nested modules in a computer program. In contrast to heap objects, a *local variable* is allocated on the stack. *Also see:* Global Variable [85]

Method In object-oriented programming, a *method* is a subroutine that is exclusively associated either with a class or with an object.

Multithreading Execution of multiple threads for one process in parallel.

Object-oriented Language A programming language that allows the user to express a program in terms of objects and messages between those objects. [85]

Object-oriented Programming (OOP) Programming that focusses on the design and implementation of objects. In particular, OOP builds on the concepts of encapsulation, polymorphism, and implementation inheritance. [188]

Object An entity that combines state (*fields*) and behavior (*methods*) and has some unique identity. [188]

Offline In dynamic analysis, doing trace analysis after the subject system has terminated. While it is executed, trace information is just written to a file. *Contrast with:* Online

Online In dynamic analysis, doing some kind of analytical event processing while the subject system is running. *Contrast with:* Offline

Procedure A portion of a computer program that is named and that performs a specific action. [85] It may modify state. *Contrast with:* Function

Program Monitor A software tool that executes concurrently with another program and provides detailed information about the execution of the other program. [85]

Protocol The set of allowed sequences of routine calls that may be invoked on a component or on an instance of it.

Routine A subprogram that is called by other programs and subprograms. [85]

Runtime (1) The period of time during which a computer program is executing. (2) see *Execution Time*. [85]

Statement In a programming language, a meaningful expression that defines data, specifies program actions, or directs the assembler or compiler. [85]

Static Analysis The process of evaluating a system or component based on its form, structure, content, or documentation. *Contrast with:* Dynamic Analysis [85]

Subprogram A separately compilable, executable component of a computer program. [85]

Subroutine A routine that returns control to the program or subprogram that called it. [85]

Test (1) An activity in which a system or component is executed under specified conditions, the results are observed or recorded, and an evaluation is made of some aspect of the system or component. (2) To conduct an activity as in (1). [85]

Thread A single sequential flow of control within a process.

Trace (1) A record of the execution of a computer program, showing the sequence of instructions executed, the names and values of variables, or both. (2) To produce a record as in (1). [85]

Validation The process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements. [85]

Variable A quantity or data item whose value can change. [85] *Also see:* Local variable, global variable, heap variable

View A representation of a whole system from the perspective of a related set of concerns. [86]

Viewpoint A specification of the conventions for constructing and using a view. A pattern or template from which to develop individual views by establishing the purposes and audience for a view and the techniques for its creation and analysis. [86]

Bibliography

[1] Mithun Acharya, Tao Xie, Jian Pei, and Jun Xu. Mining api patterns as partial orders from source code: From usage scenarios to specifications. In *Proc. of ESEC/FSE '07*, 2007.

[2] Hiralal Agrawal and Joseph R. Horgan. Dynamic program slicing. In *Proc. of Conf. on Programming language design and implementation (PLDI)*, pages 246–256, New York, NY, USA, 1990. ACM Press.

[3] Rakesh Agrawal and Ramakrishnan Srikant. Mining sequential patterns. In *Proc. of 11th Int'l Conf. on Data Engineering (ICDE)*, pages 3–14, 1995.

[4] Rajeev Alur, Pavol Černý, P. Madhusudan, and Wonhong Nam. Synthesis of interface specifications for Java classes. In *Proc. of 32nd Symp. on Principles of Programming Languages (POPL '05)*, pages 98–109. ACM, 2005.

[5] Glenn Ammons, Rastislav Bodík, and James R. Larus. Mining specifications. In *Proc. 29th Symp. on Principles of Prog. Languages (POPL)*, pages 4–16, 2002.

[6] Marc Andries, Gregor Engels, Annegret Habel, Berthold Hoffmann, Hans-Jörg Kreowski, Sabine Kuske, Detlef Plump, Andy Schürr, and Gabriele Taentzer. Graph transformation for specification and programming. *Science of Computer Programing*, 34(1):1–54, 1999.

[7] Dana Angluin. Inference of reversible languages. *Journal of the ACM*, 29(3):741–765, 1982.

[8] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, 1987.

[9] Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. Efficient and precise dynamic impact analysis using execute-after sequences. In *Proc. of 27th ICSE*, pages 432–441. ACM, 2005.

[10] Holger Bär. *Protokollprüfung in komponentenorientierten Systemen*, pages 179–221. Fraunhofer IESE / FZI, 2003. `http://app2web.fzi.de/themen/ap4/cbse_handbuch.pdf`.

[11] Holger Bär. *Statische Verifikation von Softwareprotokollen*. PhD thesis, University of Karlsruhe, Germany, 2005.

[12] Victor R. Basili. The role of experimentation in software engineering: past, current, and future. In *Proc. of 18th ICSE*, pages 442–449, 1996.

[13] C. Bennett, D. Myers, Margaret-Anne Storey, D. M. German, D. Ouellet, M. Salois, and P. Charland. A survey and evaluation of tool features for understanding reverse-engineered sequence diagrams. *Journal of Software Maintenance and Evolution: Research and Practice*, 20(4):291–315, July 2008.

[14] Árpád Beszédes, Csaba Faragó, Zsolt Mihály Szabó, János Csirik, and Tibor Gyimóthy. Union slices for program maintenance. In *Proc. of 18th ICSM*, pages 12–21, 2002.

[15] Árpád Beszédes, Tamás Gergely, Zsolt Mihály Szabó, János Csirik, and Tibor Gyimóthy. Dynamic slicing method for maintenance of large C programs. In *Proc. of 5th CSMR*, pages 105–113. IEEE Computer Society, March 2001.

[16] Dirk Beyer, Thomas A. Henzinger, and Vasu Singh. Algorithms for interface synthesis. In *Proc. of 19th Int'l Conf. on Computer Aided Verification (CAV 2007)*, volume 4590 of *Lecture Notes in Computer Science*, pages 4–19. Springer, 2007.

[17] A. W. Biermann and J. A. Feldman. On the synthesis of finite-state machines from samples of their behaviour. *IEEE Transactions on Computers*, 21:591–597, 1972.

[18] David Binkley and Keith B. Gallagher. Program slicing. *Advances in Computers*, 43:1–50, 1996.

[19] David Binkley and Mark Harman. A survey of empirical results on program slicing. *Advances in Computers*, 62, 2004.

[20] Grady Booch. Software archeology, 2004. A presentation given at the Rational Users Conference. `http://www.booch.com/architecture/blog/artifacts/Software%20Archeology.ppt`.

[21] Lionel C. Briand, Yvan Labiche, and Johanne Leduc. Toward the reverse engineering of UML sequence diagrams for distributed Java software. *IEEE Transactions on Software Engineering*, 32(9):642–663, 2006.

[22] Lionel C. Briand, Yvan Labiche, and Y. Miao. Towards the reverse engineering of UML sequence diagrams. In *Proc. of 10th WCRE*, pages 57–66, 2003.

[23] Sergey Butkevich, Marco Renedo, Gerald Baumgartner, and Michal Young. Compiler and tool support for debugging object protocols. In *Proc. of 8th FSE*, pages 50–59, New York, NY, USA, 2000. ACM Press.

[24] Andrew Chan, Reid Holmes, Gail C. Murphy, and Annie T. T. Ying. Scaling an object-oriented system execution visualizer through sampling. In *Proc. of 11th IWPC*, pages 237–244, 2003.

[25] Jiun-Liang Chen, Feng-Jian Wang, and Yung-Lin Chen. Slicing object-oriented programs. In *Proc. of 4th Asia-Pacific Software Engineering and Int'l Computer Science Conf. (APSEC)*, pages 395–404, 1997.

[26] Zhenqiang Chen and Baowen Xu. Slicing object-oriented java programs. *ACM SIGPLAN Notices*, 36(4):33–40, 2001.

[27] Elliot J. Chikofsky and James H. Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–18, 1990.

[28] Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O'Callahan, Vivek Sarkar, and Manu Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proc. of Conf. on Programming language design and implementation (PLDI)*, pages 258–269. ACM Press, 2002.

[29] Larry B. Christensen. *Experimental Methodology*. Allyn & Bacon, Boston, USA, 8th edition, 2001.

[30] Mihai Christodorescu, Somesh Jha, and Christopher Kruegel. Mining specifications of malicious behavior. In *Proc. of 15th FSE*, pages 5–14, 2007.

[31] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Proc. of 12th Int'l Conf. on Computer Aided Verification*, volume 1855 of *LNCS*, pages 154–169. Springer, 2000.

[32] Jonathan E. Cook and Alexander L. Wolf. Discovering models of software processes from event-based data. *ACM Transactions on Software Engineering and Methodology*, 7(3):215–249, 1998.

[33] Thomas A. Corbi. Program understanding: Challenge for the 1990's. *IBM Systems Journal*, 28(2):294–306, 1989.

[34] Bas Cornelissen. Identification of variation points using dynamic analysis. In *Proc. of 1st International Workshop on Reengineering towards Product Lines (R2PL)*, 2005.

[35] Bas Cornelissen, Arie van Deursen, Leon Moonen, and Andy Zaidman. Visualizing testsuites to aid in software understanding. In *Proc. of 11th CSMR*, pages 213–222, 2007.

[36] Valentin Dallmeier, Christian Lindig, Andrzej Wasylkowski, and Andreas Zeller. Mining object behavior with ADABU. In *Proc. of 4th Int'l Workshop on Dynamic Analysis (WODA)*, 2006.

[37] Valentin Dallmeier, Christian Lindig, and Andreas Zeller. Lightweight defect localization for Java. In *Proc. of 19th ECOOP*, pages 528–550, 2005.

[38] Christophe Damas, Bernard Lambeau, Pierre Dupont, and Axel van Lamsweerde. Generating annotated behavior models from end-user scenarios. *IEEE Transactions on Software Engineering*, 31(12):1056–1073, 2005.

[39] Manuvir Das. Unification-based pointer analysis with directional assignments. In *Proc. of Conf. on Programming language design and implementation*, pages 35–46. ACM Press, 2000.

[40] Luca de Alfaro and Thomas A. Henzinger. Interface automata. In *Proc. of 8th European software engineering conference*, pages 109–120, 2001.

[41] Stéphane Ducasse, Michele Lanza, and Roland Bertuli. High-level polymetric views of condensed run-time information. In *Proc. of 8th CSMR*, pages 309–318, 2004.

[42] Jürgen Ebert, Volker Riediger, and Andreas Winter. Graph technology in reverse engineering, the tgraph approach. In Rainer Gimnich, Uwe Kaiser, Jochen Quante, and Andreas Winter, editors, *10th Workshop Software Reengineering (WSR 2008)*, volume 126 of *GI Lecture Notes in Informatics (LNI)*, pages 67–81. GI, 2008.

[43] Bradley Efron and Robert J. Tibshirani. *An introduction to the bootstrap*. Chapman & Hall/CRC, 1998.

[44] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. Locating features in source code. *IEEE Transactions on Software Engineering*, 29(3), 2003.

[45] Thomas Eisenbarth, Rainer Koschke, and Gunther Vogel. Static trace extraction. In *Proc. of 9th WCRE*, pages 128–137, 2002.

[46] Thomas Eisenbarth, Rainer Koschke, and Gunther Vogel. Static object trace extraction for programs with pointers. *Journal of Systems and Software*, 77(3):263–284, Sep 2005.

[47] Andrew David Eisenberg and Kris De Volder. Dynamic feature traces: Finding features in unfamiliar code. In *Proc. of 21st ICSM*, pages 337–346, 2005.

[48] Mohammad El-Ramly, Eleni Stroulia, and Paul Sorenson. From run-time behavior to usage scenarios: an interaction-pattern mining approach. In *Proc. of 8th Int'l Conf. on Knowledge Discovery and Data Mining*, pages 315–324. ACM Press, 2002.

[49] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proc. of 4th Symp. on Operating System Design and Implementation*, pages 1–16. USENIX Association, 2000.

[50] Dawson R. Engler, David Yu Chen, and Andy Chou. Bugs as inconsistent behavior: A general approach to inferring errors in systems code. In *Proc. of Symp. on Operating System Principles*, pages 57–72, 2001.

[51] Michael D. Ernst. Static and dynamic analysis: synergy and duality. In *PASTE*, pages 35–38. ACM, June 2004. invited talk.

[52] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, 2001.

[53] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM TOPLAS*, 9(3):319–349, 1987.

[54] Richard K. Fjeldstad and William T. Hamlen. Application program maintenance study – report to our respondents. In *GUIDE 48 Proceedings*, 1983.

[55] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA, 1999.

[56] Mark Gabel and Zhendong Su. Symbolic mining of temporal specifications. In *Proc. of 30th ICSE*, pages 51–60, 2008.

[57] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns*. Professional Computing Series. Addison-Wesley, 1995.

[58] Juan Gargiulo and Spiros Mancoridis. Gadget: A tool for extracting the dynamic structure of Java programs. In *Proc. of 13th Int'l Conf. on Software Engineering & Knowledge Engineering (SEKE'2001)*, pages 244–251, 2001.

[59] Nahum D. Gershon. From perception to visualization. In L. Rosenblum, R. A. Earnshaw, J. Encarnacao, H. Hagen, A. Kaufman, S. Klimenko, G. Nielsen, F. Post, and D. Thalmann, editors, *Scientific Visualization: Advances and Challenges*. Academic Press, 1994.

[60] E. Mark Gold. Language identification in the limit. *Information and Control*, 10(5):447–474, 1967.

[61] Orla Greevy and Stéphane Ducasse. Correlating features and code using a compact two-sided trace analysis approach. In *Proc. of 9th CSMR*, pages 314–323, 2005.

[62] Thomas Gschwind and Johann Oberleitner. Improving dynamic data analysis with aspect-oriented programming. In *Proc. of 7th CSMR*, pages 259–268, 2003.

[63] Yann-Gaël Guéhéneuc, Rémi Douence, and Narendra Jussien. No Java without caffeine: A tool for dynamic analysis of Java programs. In *Proc. of 17th ASE*, pages 117–126, 2002.

[64] Tibor Gyimóthy, Árpád Beszédes, and Istán Forgács. An efficient relevant slicing method for debugging. In *Proc. of 7th ESEC/FSE*, pages 303–321, London, UK, 1999. Springer-Verlag.

[65] Dietrich Haak. Werkzeuggestützte Herleitung von Protokollen. Diploma thesis, University of Stuttgart, Computer Science, February 2004. DA-2135.

[66] Annegret Habel, Reiko Heckel, and Gabriele Taentzer. Graph grammars with negative application conditions. *Fundamenta Informaticae*, 26(3/4):287–313, 1996.

[67] Robert J. Hall. Automatic extraction of executable program subsets by simultaneous dynamic program slicing. *Journal Automated Software Engineering*, 2:33–53, 1995.

[68] Abdelwahab Hamou-Lhadj. *Techniques to Simplify the Analysis of Execution Traces for Program Comprehension*. PhD thesis, Ottawa-Carleton Institute for Computer Science, School of Information Technology an Engineering, University of Ottawa, Ontario, Ottawa, Canada, 2005.

[69] Abdelwahab Hamou-Lhadj, Edna Braun, Daniel Amyot, and Timothy Lethbridge. Recovering behavioral design models from execution traces. In *Proc. of 9th CSMR*, pages 112–121, 2005.

[70] Abdelwahab Hamou-Lhadj and Timothy Lethbridge. Summarizing the content of large traces to facilitate the understanding of the behaviour of a software system. In *Proc. of 14th ICPC*, pages 181–190, 2006.

[71] Abdelwahab Hamou-Lhadj and Timothy C. Lethbridge. Compression techniques to simplify the analysis of large execution traces. In *Proc. of 10th IWPC*, pages 159–168, 2002.

[72] Abdelwahab Hamou-Lhadj and Timothy C. Lethbridge. A survey of trace exploration tools and techniques. In *Proc. of Conf. of the Centre for Advanced Studies on Collaborative Research (CASCON)*, pages 42–55. IBM Press, 2004.

[73] Sven Hanssen. Extraktion statischer Traces zur Wiedergewinnung von Protokollen. Master's thesis, University of Stuttgart, Computer Science, May 2000. SA-1768.

[74] Timo Heiber. Semi-automatic protocol recovery. Diploma thesis, University of Stuttgart, Computer Science, 2000. DA-1822.

[75] Johannes Henkel and Amer Diwan. Discovering algebraic specifications from java classes. In *Proc. of 17th ECOOP*, volume 2743 of *LNCS*, pages 431–456. Springer, 2003.

[76] Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Permissive interfaces. In *Proc. of 13th FSE*, pages 31–40, 2005.

[77] Michael Hind. Pointer analysis: haven't we solved this problem yet? In *Proc. of Workshop on Program Analysis For Software Tools and Engineering (PASTE)*, pages 54–61, 2001.

[78] Michael Hind and Anthony Pioli. Which pointer analysis should I use? In *Int'l Symposium on Software Testing and Analysis (ISSTA)*, pages 113–123, 2000.

[79] Andreas Höfer and Walter F. Tichy. Status of empirical research in software engineering. In *Empirical Software Engineering Issues*, volume 4336 of *LNCS*, pages 10–19, 2007.

[80] Christine Hofmeister, Robert Nord, and Dilip Soni. *Applied Software Architecture*. Object Technology Series. Addison Wesley, 2000.

[81] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, second edition, 2001.

[82] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Program. Lang. Syst.*, 12(1):26–60, 1990.

[83] Graham Hughes and Tevfik Bultan. Interface grammars for modular software model checking. In *Proc. of Int'l Symp. on Software Testing and Analysis (ISSTA)*, pages 39–49, 2007.

[84] IEEE Standards Board. IEEE software engineering standard 729-1993: Glossary of software engineering terminology, 1983.

[85] IEEE Standards Board. IEEE standard glossary of software engineering terminology—std. 610.12-1990, 1990.

[86] IEEE Standards Board. IEEE recommended practice for architectural description of software-intensive systems—std. 1471-2000, 2000.

[87] Ryszard Janicki and Emil Sekerinski. Foundations of the trace assertion method of module interface specification. *IEEE Transactions on Software Engineering*, 27(7):577–598, 2001.

[88] Dean F. Jerding and Spencer Rugaber. Using visualization for architectural localization and extraction. In *Proc. of 4th WCRE*, pages 56–65, 1997.

[89] Dean F. Jerding, John T. Stasko, and Thomas Ball. Visualizing interactions in program executions. In *Proc. of 19th ICSE*, pages 360–370, 1997.

[90] Juanjuan Jiang, Johannes Koskinen, Anna Ruokonen, and Tarja Systä. Constructing usage scenarios for API redocumentation. In *Proc. of 15th ICPC*, pages 259–264, 2007.

[91] Benjamin Jung. Analyse der Struktur von Software-Protokollen. Master's thesis, University of Stuttgart, Germany, 2007. DA-2591.

[92] Ralf Kollmann and Martin Gogolla. Capturing dynamic program behaviour with uml collaboration diagrams. In *Proc. of 5th CSMR*, pages 58–67, 2001.

[93] Bogdan Korel and Janusz W. Laski. Dynamic program slicing. *Inf. Process. Lett.*, 29(3):155–163, 1988.

[94] Bogdan Korel and Janusz W. Laski. Dynamic slicing of computer programs. *Journal of Systems and Software*, 13(3):187–195, 1990.

[95] Bogdan Korel and Jürgen Rilling. Dynamic program slicing methods. *Information and Software Technology*, 40(11-12):647–660, November 1998. Special issue on program slicing.

[96] Rainer Koschke. *Atomic Architectural Component Recovery for Program Understanding and Evolution*. PhD thesis, Institute for Computer Science, University of Stuttgart, http://www.iste.uni-stuttgart.de/ps/rainer/thesis, 2000.

[97] Rainer Koschke. Zehn Jahre WSR – Zwölf Jahre Bauhaus. In *10th Workshop Software Reengineering (WSR 2008)*, volume 126 of *GI Lecture Notes in Informatics (LNI)*, pages 51–65. GI, 2008.

[98] Rainer Koschke, Jean-Francois Girard, and Martin Würthner. An intermediate representation for reverse engineering analyses. In *Proc. of 5th WCRE*, pages 241–250, Oct 1998.

[99] Rainer Koschke and Jochen Quante. On dynamic feature location. In *Proc. of 20th ASE*, pages 86–95. ACM Press, November 2005.

[100] Rainer Koschke and Yan Zhang. Component recovery, protocol recovery and validation in bauhaus. In *3. Workshop Software-Reengineering, Bad Honnef, Germany*, Fachberichte Informatik, May 2001.

[101] Kai Koskimies and Erkki Mäkinen. Automatic synthesis of state machines from trace diagrams. *Software - Practice and Experience*, 24(7):643–658, 1994.

[102] Kai Koskimies and Hanspeter Mössenböck. Scene: Using scenario diagrams and active text for illustrating object-oriented programs. In *Proc. of 18th ICSE*, pages 366–375, 1996.

[103] Hans-Jörg Kreowski, Renate Klempien-Hinrichs, and Sabine Kuske. Some essentials of graph transformation. In *Recent Advances in Formal Languages and Applications*, volume 25 of *Studies in Computational Intelligence*, pages 229–254. Springer, 2006.

[104] Philippe Kruchten. Architectural blueprints – the 4+1 view model of software architecture. *IEEE Software*, 12(6):42–50, November 1995.

[105] Kevin J. Lang, Barak A. Pearlmutter, and Rodney A. Price. Results of the Abbadingo One DFA learning competition and a new evidence-driven state merging algorithm. In V. Honavar and G. Slutzki, editors, *Grammatical Inference; 4th Int'l Colloquium (ICGI-98)*, volume 1433 of *LNCS/LNAI*, pages 1–12. Springer, 1998.

[106] Danny B. Lange and Yuichi Nakamura. Object-oriented program tracing and visualization. *IEEE Computer*, 30(5):63–70, 1997.

[107] Michele Lanza and Stéphane Ducasse. Polymetric views-a lightweight visual approach to reverse engineering. *IEEE Transactions on Software Engineering*, 29(9):782–795, 2003.

[108] Loren Larsen and Mary Jean Harrold. Slicing object-oriented software. In *Proc. of 18th ICSE*, pages 495–505. ACM Press, 1996.

[109] Meir M. Lehman and Laszlo A. Belady. *Program evolution: processes of software change*. Academic Press Professional, Inc., San Diego, CA, USA, 1985.

[110] Vladimir. I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady*, 10(8):707–710, February 1966.

[111] Zhenmin Li and Yuanyuan Zhou. PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. In *Proc. of 10th ESEC/13th FSE*, pages 306–315, 2005.

[112] Donglin Liang and Mary Jean Harrold. Slicing objects using system dependence graphs. In *Proc. of ICSM*, pages 358–367. IEEE Press, 1998.

[113] Adrian Lienhard, Stéphane Ducasse, and Tudor Gîrba. Object flow analysis: taking an object-centric view on dynamic analysis. In *Proc. of Int'l Conf. on Dynamic Languages (ICDL)*, pages 121–140, 2007.

[114] Adrian Lienhard, Stéphane Ducasse, and Tudor Gîrba. Object flow analysis: Taking an object-centric view on dynamic analysis. *Journal of Computer Languages, Systems and Structures*, 2008. to appear.

[115] Adrian Lienhard, Stéphane Ducasse, Tudor Gîrba, and Oscar Nierstrasz. Capturing how objects flow at runtime. In *Proc. of Int'l Workshop on Program Comprehension through Dynamic Analysis (PCODA)*, pages 39–43, 2006.

[116] Adrian Lienhard, Tudor Gîrba, Orla Greevy, and Oscar Nierstrasz. Test blueprint - exposing side effects in execution traces to support writing unit tests. In *Proc. of 12th CSMR*, pages 83–92, 2008.

[117] Adrian Lienhard, Orla Greevy, and Oscar Nierstrasz. Tracking objects to detect feature dependencies. In *Proc. of 15th ICPC*, pages 59–68, 2007.

[118] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Prentice Hall PTR, 2nd edition, 1999. `http://java.sun.com/docs/books/jvms/`.

[119] Barbara Liskov. Data abstraction and hierarchy. In *OOPSLA '87: Addendum to the proceedings on Object-oriented programming systems, languages and applications*, pages 17–34. ACM, 1987.

[120] Chang Liu, En Ye, and Debra J. Richardson. Software library usage pattern extraction using a software model checker. In *Proc. of 21st ASE*, pages 301–304, 2006.

[121] Dapeng Liu, Andrian Marcus, Denys Poshyvanyk, and Vaclav Rajlich. Feature location via information retrieval based filtering of a single scenario execution trace. In *Proc. of 22nd ASE*, pages 234–243, 2007.

[122] David Lo and Siau-Cheng Khoo. Quark: Empirical assessment of automaton-based specification miners. In *Proc. of 13th WCRE*, pages 51–60, Washington, DC, USA, 2006. IEEE Computer Society.

[123] David Lo and Siau-Cheng Khoo. Smartic: towards building an accurate, robust and scalable specification miner. In *Proc. of 14th FSE*, pages 265–275, New York, NY, USA, 2006. ACM Press.

[124] David Lo, Siau-Cheng Khoo, and Chao Liu. Efficient mining of iterative patterns for software specification discovery. In *Proc. of 13th Int'l Conf. on Knowledge Discovery and Data Mining*, pages 460–469, 2007.

[125] David Lo, Siau-Cheng Khoo, and Chao Liu. Mining temporal rules for software mainentance. *Journal of Software Maintenance and Evolution: Research and Practice*, 20(4):227–247, July 2008.

[126] Davide Lorenzoli, Leonardo Mariani, and Mauro Pezze. Automatic generation of software behavioral models. In *Proc. of 30th ICSE*, pages 501–510, 2008.

[127] Kazimiras Lukoit, Norman Wilde, Scott Stowell, and Tim Hennessey. Tracegraph: Immediate visual location of software features. In *Proc. of ICSM*, pages 33–39, 2000.

[128] Erkki Mäkinen and Tarja Systä. MAS - an interactive synthesizer to support behavioral modeling in UML. In *Proc. of 23rd ICSE*, pages 15–24, 2001.

[129] Onaiza Maqbool and Haroon Babri. Hierarchical clustering for software architecture recovery. *IEEE Transactions on Software Engineering*, 33(11):759–780, 2007.

[130] Markus Mock, Darren C. Atkinson, Craig Chambers, and Susan J. Eggers. Improving program slicing with dynamic points-to data. In *Proc. of 10th FSE*, pages 71–80, 2002.

[131] Gail C. Murphy, David Notkin, and Kevin J. Sullivan. Software reflexion models: Bridging the gap between source and high-level models. In *Proc. of Symposium on the Foundations of Software Engineering*, pages 18–28, 1995.

[132] Oscar Nierstrasz. Regular types for active objects. In *Proc. of 8th OOPSLA*, pages 1–15, New York, NY, USA, 1993. ACM Press.

[133] Jeremy W. Nimmer and Michael D. Ernst. Automatic generation of program specifications. In *Proc. of Int'l Symp. on Software testing and analysis*, pages 229–239, New York, NY, USA, 2002. ACM Press.

[134] John T. Nosek and Prashant Palvia. Software maintenance management: Changes in the last decade. *Software Maintenance: Research and Practice*, 2:157–174, 1990.

[135] Fumiaki Ohata, Kouya Hirose, Masato Fujii, and Katsuro Inoue. A slicing method for object-oriented programs using lightweight dynamic information. In *Proc. of 8th Asia-Pacific Software Engineering Conference (APSEC)*, pages 273–280, 2001.

[136] Kurt M. Olender and Leon J. Osterweil. Interprocedural static analysis of sequencing constraints. *ACM Transactions on Software Engineering and Methodology*, 1(1):21–52, January 1992.

[137] Carratalá Oncina and P. García. *Inferring regular languages in polynomial update time*, volume 1, pages 49–61. World Scientific Publishing, 1991.

[138] David L. Parnas. A technique for software module specification with examples. *Communications of the ACM*, 15(5), May 1972.

[139] David L. Parnas. Software aging. In *Proc. of 16th ICSE*, pages 279–287, 1994.

[140] David Lorge Parnas, Paul C. Clements, and David M. Weiss. The modular structure of complex systems. In *Proc. of 7th ICSE*, pages 408–417. IEEE Press, 1984.

[141] Wim De Pauw, Richard Helm, Doug Kimelman, and John M. Vlissides. Visualizing the behavior of object-oriented systems. In *Proc. of OOPSLA*, pages 326–337, 1993.

[142] Wim De Pauw, David H. Lorenz, John M. Vlissides, and Mark N. Wegman. Execution patterns in object-oriented visualization. In *Proc. of 4th USENIX Conf. on Object-Oriented Technologies and Systems (COOTS)*, pages 219–234, 1998.

[143] Massimiliano Di Penta, R. E. Kurt Stirewalt, and Eileen Kraemer. Designing your next empirical study on program comprehension. In *Proc. of 15th ICPC*, pages 281–285, 2007.

[144] Frantisek Plasil and Stanislav Visnovsky. Behavior protocols for software components. *IEEE Transactions on Software Engineering*, 28(11):1056–1076, 2002.

[145] Lutz Prechelt. *Kontrollierte Experimente in der Softwaretechnik*. Springer Verlag, Berlin, 2001.

[146] Jochen Quante. Online construction of dynamic object process graphs. In *Proc. of 11th CSMR*, pages 113–122, March 2007.

[147] Jochen Quante. Do dynamic object process graphs support program understanding? – A controlled experiment. In *Proc. of 16th ICPC*, pages 73–82, 2008.

[148] Jochen Quante and Rainer Koschke. Dynamic object process graphs. In *Proc. of 10th CSMR*, pages 81–90, 2006.

[149] Jochen Quante and Rainer Koschke. Dynamic protocol recovery. In *Proc. of 14th WCRE*, pages 219–228, 2007.

[150] Jochen Quante and Rainer Koschke. Dynamic object process graphs. *Journal of Systems and Software*, 81(4):481–501, April 2008.

[151] Vaclav Rajlich and George S. Cowan. Towards standard for experiments in program comprehension. In *Proc. of 5th IWPC*, pages 160–161, 1997.

[152] Anand V. Raman and Jon D. Patrick. The sk-strings method for inferring PFSA. In *Proc. of Workshop on Automata Induction, Grammatical Inference and Language Acquisition*, 1997.

[153] Murali Krishna Ramanathan, Ananth Grama, and Suresh Jagannathan. Path-sensitive inference of function precedence protocols. In *Proc. of 29th ICSE*, pages 240–250. IEEE Computer Society, 2007.

[154] Aoun Raza, Gunther Vogel, and Erhard Plödereder. Bauhaus – a tool suite for program analysis and reverse engineering. In *Reliable Software Technologies – Proc. of 11th Ada-Europe Int'l Conf. on Reliable Software Technologies (Ada-Europe 2006)*, pages 71–82. Springer, 2006. LNCS 4006.

[155] Steven P. Reiss. Visualizing Java in action. In *Proc. of Symposium on Software Visualization (SoftVis)*, pages 57–66. ACM, 2003.

[156] Steven P. Reiss and Manos Renieris. Generating Java trace data. In *Java Grande*, pages 71–77, 2000.

[157] Steven P. Reiss and Manos Renieris. Encoding program executions. In *Proc. of 23rd ICSE*, pages 221–230, 2001.

[158] Steven P. Reiss and Manos Renieris. Languages for dynamic instrumentation. In *Proc. of Workshop on Dynamic Analysis*, May 2003.

[159] David Rice. *Geekonomics: The Real Cost of Insecure Software*. Addison-Wesley Longman, 2007.

[160] Marc Richetin and François Vernadat. Regular inference for syntactic pattern recognition: A case study. In *Proc. of 7th Intl. Conf. on Pattern Recognition*, pages 1370–1372, 1984.

[161] Tamar Richner and Stéphane Ducasse. Recovering high-level views of object-oriented applications from static and dynamic information. In *Proc. of ICSM*, pages 13–22, 1999.

[162] Tamar Richner and Stéphane Ducasse. Using dynamic information for the iterative recovery of collaborations and roles. In *Proc. of 18th ICSM*, pages 34–43, October 2002.

[163] Abhishek Rohatgi, Abdelwahab Hamou-Lhadj, and Juergen Rilling. An approach for mapping features to code based on static and dynamic analysis. In *Proc. of 16th ICPC*, pages 234–239, 2008.

[164] Atanas Rountev, Olga Volgin, and Miriam Reddoch. Static control-flow analysis for reverse engineering of uml sequence diagrams. In *Proc. of PASTE*, pages 96–102, 2005.

[165] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language – Reference Manual*. Addison-Wesley, 1998.

[166] Hossein Safyallah and Kamran Sartipi. Dynamic analysis of software systems using execution pattern mining. In *Proc. of 14th ICPC*, pages 84–88, 2006.

[167] Maher Salah, Trip Denton, Spiros Mancoridis, Ali Shokoufandeh, and Filippos I. Vokolos. Scenariographer: A tool for reverse engineering class usage scenarios from method invocation sequences. In *Proc. of 21st ICSM*, pages 155–164, 2005.

[168] David Schuler, Valentin Dallmeier, and Christian Lindig. A dynamic birthmark for Java. In *Proc. of 22nd ASE*, pages 274–283, 2007.

[169] R. Sekar, Mugdha Bendre, Dinakar Dhurjati, and Pradeep Bollineni. A fast automaton-based method for detecting anomalous program behaviors. In *Proc. of Symposium on Security and Privacy (SP '01)*, pages 144–155, 2001.

[170] Sameer Shende. Profiling and tracing in linux. In *Proc. Extreme Linux Workshop #2, USENIX*, June 1999.

[171] Daniel Simon. *Lokalisierung von Merkmalen in Softwaresystemen*. Ph.d. dissertation, University of Stuttgart, Germany, 2005.

[172] Dag I. K. Sjøberg, Jo Erskine Hannay, Ove Hansen, Vigdis By Kampenes, Amela Karahasanovic, Nils-Kristian Liborg, and Anette C. Rekdal. A survey of controlled experiments in software engineering. *IEEE Transactions on Software Engineering*, 31(9):733–753, 2005.

[173] Michael Smit, Eleni Stroulia, and Kenny Wong. Use case redocumentation from gui event traces. In *Proc. of 12th CSMR*, pages 263–268, 2008.

[174] Harry M. Sneed and Stefan Opferkuch. Training and certifying software maintainers. In *Proc. of 12th CSMR*, pages 113–122, 2008.

[175] Elliot Soloway, Robin Lampert, Stan Letovsky, David Littman, and Jeannine Pinto. Designing documentation to compensate for delocalized plans. *Commun. ACM*, 31(11):1259–1267, 1988.

[176] Amitabh Srivastava and Alan Eustace. ATOM: a system for building customized program analysis tools. In *Proc. of Conference on Programming Language Design and Implementation*, pages 196–205, 1994.

[177] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Symposium on Principles of Programming Languages*, pages 32–41, 1996.

[178] Christoph Steindl. Intermodular slicing of object-oriented programs. In *International Conference on Compiler Construction*, volume 1383, pages 264–278. Lecture Notes in Computer Science, Springer, 1998.

[179] Christoph Steindl. *Program Slicing for Object-Oriented Programming Languages*. Dissertation, Johannes Kepler University Linz, 1999.

[180] Margaret-Anne Storey. Theories, methods and tools in program comprehension: Past, present and future. In *Proc. of 13th IWPC*, pages 181–191, 2005.

[181] Margaret-Anne D. Storey. *A Cognitive Framework for Describing and Evaluating Software Exploration Tools*. PhD thesis, School of Computing Science, Simon Fraser University, December 1998.

[182] Margaret-Anne D. Storey, Kenny Wong, Hausi A. Müller, P. Fong, D. Hooper, and K. Hopkins. On designing an experiment to evaluate a reverse engineering tool. In *Proc. of 3rd WCRE*, page 31, Washington, DC, USA, 1996. IEEE Computer Society.

[183] Eleni Stroulia and Tarja Systä. Dynamic analysis for reverse engineering and program understanding. *Applied Computing Review*, 10(1), 2002.

[184] Tarja Systä. On the relationships between static and dynamic models in reverse engineering Java software. In *Proc. of 6th WCRE*, pages 304–313, 1999.

[185] Tarja Systä. *Static and Dynamic Reverse Engineering Techniques for Java Software Systems*. PhD thesis, University of Tampere, Department of Computer and Information Sciences, Tampere, Finland, May 2000.

[186] Tarja Systä. Understanding the behavior of Java programs. In *Proc. of 7th WCRE*, pages 214–223, 2000.

[187] Attila Szegedi, Tamás Gergely, Árpád Beszédes, Tibor Gyimóthy, and Gabriella Toth. Verifying the concept of union slices on java programs. In *Proc. of 11th CSMR*, pages 233–242. IEEE Computer Society, March 2007.

[188] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 2002.

[189] Nikolai Tillmann, Feng Chen, and Wolfram Schulte. Discovering likely method specifications. In *Proc. of 8th Int'l Conf. on Formal Engineering Methods (ICFEM'06)*, volume 4260 of *LNCS*, pages 717–736. Springer, 2006.

[190] Frank Tip. A survey of program slicing techniques. *Journal of programming languages*, 3(3):121–189, September 1995.

[191] Paolo Tonella and Alessandra Potrich. Static and dynamic C++ code analysis for the recovery of the object diagram. In *Proc. of 18th ICSM*, pages 54–63, 2002.

[192] Paolo Tonella and Alessandra Potrich. Reverse engineering of the interaction diagrams from C++ code. In *Proc. of 19th ICSM*, pages 159–168, 2003.

[193] Paolo Tonella, Marco Torchiano, Bart Du Bois, and Tarja Systä. Empirical studies in reverse engineering: state of the art and future trends. *Empirical Software Engineering*, 12(5):551–571, 2007.

[194] Sebastián Uchitel, Jeff Kramer, and Jeff Magee. Synthesis of behavioral models from scenarios. *IEEE Transactions on Software Engineering*, 29(2):99–115, 2003.

[195] Wil M. P. van der Aalst, B. F. van Dongen, J. Herbst, L. Maruster, G. Schimm, and A. J. M. M. Weijters. Workflow mining: a survey of issues and approaches. *Data & Knowledge Engineering*, 47(2):237–267, 2003.

[196] Gunther Vogel. Extraktion statischer Objekt-Traces zur Erkennung und Beschreibung von Konnektoren. Diploma thesis, University of Stuttgart, Computer Science, 2001. DA-1940.

[197] Gunther Vogel. Transformation und Vergleich von endlichen Automaten zur Analyse von Software-Protokollen. In *Proceedings der INFORMATIK 2007, Band 2*, volume 110 of *GI Lecture Notes in Informatics (LNI)*, pages 268–274. GI, 2007.

[198] Gunther Vogel. *Statische Herleitung und Analyse von Software-Protokollen*. PhD thesis, Institute for Computer Science, University of Stuttgart, 2008/09. Work in progress.

[199] Robert J. Walker, Gail C. Murphy, Bjørn N. Freeman-Benson, Darin Wright, Darin Swanson, and Jeremy Isaak. Visualizing dynamic software system information through high-level models. In *OOPSLA*, pages 271–283, 1998.

[200] Neil Walkinshaw, Kirill Bogdanov, Shaukat Ali, and Mike Holcombe. Automated discovery of state transitions and their functions in source code. *Software Testing, Verification and Reliability*, 18(2), 2008.

[201] Neil Walkinshaw, Kirill Bogdanov, Mike Holcombe, and Sarah Salahuddin. Reverse engineering state machines by interactive grammar inference. In *Proc. of 14th WCRE*, pages 209–218, 2007.

[202] Neil Walkinshaw, Marc Roper, and Murray Wood. Understanding object-oriented source code from the behavioural perspective. In *Proc. of 13th IWPC*, pages 215–224, 2005.

[203] Tao Wang and Abhik Roychoudhury. Using compressed bytecode traces for slicing Java programs. In *Proc. of 26th ICSE*, pages 512–521, 2004.

[204] Andrzej Wasylkowski, Andreas Zeller, and Christian Lindig. Detecting object usage anomalies. In *Proc. of 15th FSE*, pages 35–44, 2007.

[205] T. Weijters and Wil van der Aalst. Rediscovering workflow models from event-based data. In V. Hoste and G. de Pauw, editors, *Proc. of 11th Dutch-Belgian Conference on Machine Learning (Benelearn 2001)*, pages 93–100, 2001.

[206] Westley Weimer and George C. Necula. Mining temporal specifications for error detection. In *Proc. of 11th Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 3440 of *LNCS*, pages 461–476. Springer, 2005.

[207] Mark D. Weiser. *Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method*. PhD thesis, The University of Michigan, 1979.

[208] Mark D. Weiser. Program slicing. In *Proc. of 5th ICSE*, pages 439–449. IEEE Computer Society Press, 1981.

[209] John Whaley, Michael C. Martin, and Monica S. Lam. Automatic extraction of object-oriented component interfaces. In *Proc. of Symposium on Software Testing and Analysis (ISSTA)*, pages 218–228, July 2002.

[210] Jon Whittle and Johann Schumann. Generating statechart designs from scenarios. In *Proc. of 22nd ICSE*, pages 314–323, 2000.

[211] Norman Wilde, Paul Matthews, and Ross Huitt. Maintaining object-oriented software. *IEEE Software*, 10(1):75–80, 1993.

[212] Norman Wilde and Michael C. Scully. Software reconnaissance: mapping program features to code. *Journal of Software Maintenance*, 7(1):49–62, 1995.

[213] Robert P. Wilson. *Efficient, Context-Sensitive Pointer Analysis*. PhD thesis, Department of Electrical Engineering, Stanford University, December 1997.

[214] Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proc. of Conf. on Programming Language Design and Implementation*, pages 1–12, 1995.

[215] W. Eric Wong, Joseph R. Horgan, Swapna S. Gokhale, and Kishor S. Trivedi. Locating program features using execution slices. In *Proc. of Symposium on Application-Specific Systems and Software Engineering and Technology (ASSET)*, pages 194–203, 1999.

[216] Jingwei Wu, Ahmed E. Hassan, and Richard C. Holt. Using graph patterns to extract scenarios. In *Proc. 10th IWPC*, pages 239–250, 2002.

[217] Tao Xie and David Notkin. Automatic extraction of object-oriented observer abstractions from unit-test executions. In *Proc. of 6th Int'l Conf. on Formal Engineering Methods (ICFEM)*, volume 3308 of *LNCS*, pages 290–305. Springer, 2004.

[218] Tao Xie and David Notkin. Automatic extraction of sliced object state machines for component interfaces. In *Proc. 3rd Workshop on Specification and Verification of Component-Based Systems*, pages 39–46, 2004.

[219] Jinlin Yang and David Evans. Dynamically inferring temporal properties. In *Proc. of Workshop on Program analysis for software tools and engineering*, pages 23–28, New York, NY, USA, 2004. ACM Press.

[220] Jinlin Yang, David Evans, Deepali Bhardwaj, Thirumalesh Bhat, and Manuvir Das. Perracotta: mining temporal API rules from imperfect traces. In *Proc. of 28th ICSE*, pages 282–291, New York, NY, USA, 2006. ACM Press.

[221] Daniel M. Yellin and Robert E. Strom. Interfaces, protocols, and the semi-automatic construction of software adaptors. In *Proc. of 9th OOPSLA*, pages 176–190, 1994.

[222] Daniel M. Yellin and Robert E. Strom. Protocol specifications and component adaptors. *ACM Trans. Program. Lang. Syst.*, 19(2):292–333, 1997.

[223] Andy Zaidman. *Scalability Solutions for Program Comprehension Through Dynamic Analysis*. PhD thesis, University of Antwerp, Antwerp, the Netherlands, September 2006.

[224] Andy Zaidman, Toon Calders, Serge Demeyer, and Jan Paredaens. Applying webmining techniques to execution traces to support the program comprehension process. In *Proc. of 9th CSMR*, pages 134–142, 2005.

[225] Andy Zaidman and Serge Demeyer. Managing trace data volume through a heuristical clustering process based on event execution frequency. In *Proc. of 8th CSMR*, pages 329–338, 2004.

[226] Andy Zaidman, Serge Demeyer, Bram Adams, Kris De Schutter, Ghislain Hoffman, and Bernard De Ruyck. Regaining lost knowledge through dynamic analysis and aspect orientation. In *Proc. of 10th CSMR*, pages 91–102, 2006.

[227] Andreas Zendler. *Elemente der experimentellen Softwaretechnik*. March 2000.

[228] Wolf Zimmermann and Michael Schaarschmidt. Automatic checking of component protocols in component-based systems. In *5th Intl. Symp. on Software Composition, SC 2006*, volume 4089 of *LNCS*, pages 1–17. Springer, March 2006.

Index