Feature Identification: A Novel Approach and a Case Study

Giuliano Antoniol[1,3]
Yann-Gaël Guéhéneuc[3]
[1]RCOST, University of Sannio, Italy
[2]GEODES, DIRO, University of Montreal, Canada
[3]Départment de Génie Informatique, École Polytechnique de Montréal, Canada
antoniol@ieee.org, guehene@iro.umontreal.ca

Abstract

Feature identification is a well-known technique to identify subsets of a program source code activated when exercising a functionality. Several approaches have been proposed to identify features. We present an approach to feature identification and comparison for large object-oriented multi-threaded programs using both static and dynamic data. We use processor emulation, knowledge filtering, and probabilistic ranking to overcome the difficulties of collecting dynamic data, i.e., imprecision and noise. We use model transformations to compare and to visualise identified features. We compare our approach with a naive approach and a concept analysis-based approach using a case study on a real-life large object-oriented multi-threaded program, Mozilla, to show the advantages of our approach. We also use the case study to compare processor emulation with statistical profiling.

Keywords: Program understanding, dynamic and static analyses, feature analysis, meta-modelling.

1 Introduction

Maintenance of legacy software involves costly and tedious program understanding tasks to identify and to understand data structures, functions, methods, objects, and classes and, more generally, any high-level abstractions required by maintainers to perform their tasks. Source code browsing is the most common activity performed during software maintenance because obsolete (or missing) documentation forces maintainers to rely on source code only. Unfortunately, source code browsing becomes very resource consuming as the size and the complexity of programs increase.

An alternative to source code browsing is automated design recovery. Central to design recovery is the recovery of *higher-level abstractions beyond those obtained by examining the system itself* [4]. We propose an approach to support the recovery of higher-level abstractions through program feature identification and comparison. We define a feature of a program as a set of data structures (*i.e.*, fields and classes) and operations (*i.e.*, functions and methods) participating in the realisation of a user-observable functionality in a given scenario. The scenario details the particular conditions of realisation of the functionality: For example in a web browser, accessing a page from the bookmarks corresponds to one feature, adding a Uniform Resource Locator (URL) to the bookmarks is another feature.

In this paper, we describe our approach of building micro-architectures, subsets of a program architecture [10], from feature identification using static and dynamic data. We assume that the program source code is available and that a compiled version of the program can be exercised under different scenarios. We use data collected from the realisation of a functionality, under different scenarios, to filter static data modelled as class diagrams, thus relating classes with features and with scenarios, and highlighting differences among features. Maintainers can use our approach to build and to compare micro-architectures to locate responsibilities and feature differences precisely.

Our approach decomposes in a process and a set of tools to support the process. The process makes use of processor emulation, knowledge filtering, probabilistic ranking, and model transformations to support the analysis of large multi-threaded object-oriented programs and to deal with imprecision and noise.

We evaluate the usefulness of our approach through the analysis of the MOZILLA web browser using two basic scenarios in one case study of features identification and comparison. We compare our approach with a naive approach and a concept analysis-based approach with the same scenarios. We show that our approach is scalable and overcome limitations of existing approaches.

The main contributions of this paper are summarized as an approach to feature identification and comparison in large multi-threaded object-oriented programs using static and dynamic data, knowledge-based filtering, and probabilistic ranking. We present an experimental comparison of our approach when supporting program understanding tasks with a naive `grep`-based approach and a concept analysis-based feature identification technique. Finally, we discuss the impact of the precision of two different tools representative of statistical profilers and of simulation tools on feature identification and, thus, on program understanding tasks.

The remainder of the paper is organised as follows: Section 2 summarises previous work and their limitations; Section 3 presents our approach—the process and the tools—to feature identification and comparison; Section 4 describes the case study with MOZILLA; Section 5 concludes with future work.

2 Related Work

Our work relates to static and dynamic program analysis, to feature analysis, meta-modelling and model transformation techniques. The following sub-sections describe related work in each category.

Static and Dynamic Analysis. Program instrumentation via source-to-source translation is a common technique to collect dynamic data. For example, Ernst *et al.*, with DAIKON, focus on dynamic techniques for discovering invariants in traces [9]. However, DAIKON does not offer front end for C++.

Jeffery and Auguston [13] present the UFO dynamic analysis framework. This framework combines a language and a monitor architecture to ease building new dynamic analysis tools. The main limitations of the UFO framework in the context of our work is the lack of support for C++ and possible implementation and optimisation issues.

Finally, Ernst [8] discusses synergies and dualities between static and dynamic analyses. Reiss and Renieris [17, 18, 19] present an approach to encode dynamic data, to explore program traces, and languages for dynamic instrumentation respectively. The problem of handling large amounts of data when performing dynamic analysis is discussed in several work, for example Hamou-Lhadj *et al.* [11, 12], where the authors present an algorithm for identifying patterns in traces of procedure calls.

Feature Identification. Several works propose feature identification techniques. However, few works attempt to compare features with one another.

In their precursor work, Wilde and Scully [23] propose a technique to identify features by analysing execution traces of test cases. They use two sets of test cases to build two execution traces: An execution trace where a functionality is exercised; An execution trace where the functionality is not. Then, they compare execution traces to identify the feature associated with the functionality in the program. In their work, the authors only use dynamic data to identify features, no static analysis of the program is performed.

Several commonalities exist between the present work and [5, 23]. As detailed in Section 3, our approach is close to Wilde's approach [5], differences being the adoption of a knowledge-base filtering of dynamic data and a modified relevance index accounting for the population sizes of events relevant and irrelevant to a feature of interest.

Chen and Rajlich [3] develop a technique to identify features using Abstract System Dependencies Graph (ASDG). In C, an ASDG models functions and global variables as well as function calls and data flow in a program source code. Maintainers identify, manually, features using the ASDG following a precise process. At the opposite of Wilde and Scully's work, Chen and Rajlich use only static data to identify features and a manual search, which is prohibitive for large multi-threaded object-oriented programs.

Combining previous approaches, Eisenbarth *et al.* [6] use both static and dynamic data to identify features in software components. First, the authors perform a dynamic analysis of the program using profiling techniques to identify feature, similarly to Wilde and Scully's work. Then, they use concept analysis techniques to relate features together and to guide a static analysis to narrow the scope of identified features. Details on concept analysis and feature identification were also published by the same authors in [7]. They apply their approach on two C programs and feature identification was actually performed by means of set operations on concepts.

Salah and Mancoridis [20] use both static and dynamic data to identify features in Java programs. They go beyond feature identification by creating feature-interaction views, which highlight dependencies among features. They define four types of views: Object-interaction view, class-interaction view, feature-interaction view, and feature-implementation view. Each view abstracts preceding views to present only the most relevant data to maintainers. Feature-interaction and implementation views highlight relationships among views. However, maintainers cannot use these views to identify and to highlight differences between features with respect to different scenarios.

Meta-Modelling and Model Transformations. Meta-modelling techniques have been used in several works to model a program source code, for example, Pagel and Winter [16], Sunyé [21]. Also, we concur with Thomas: "Every models need a meta-model" [22].

A meta-model defines a language to create models in some universe of discourse. Models issued from a meta-model can be visualised, compared, and modified at run-time using operations defined at the meta-model level. Model transformations are thus defined more precisely and easily when a meta-model is present. Model transformations are available, for example, in the UMLAUT [21] tool, and are central to the Model-Driven Architecture approach [2, 15].

We build on previous work, in particular on Wilde and Scully's precursor work and on Salah and Mancoridis' work, to identify features in large multi-threaded object-oriented C++ programs and to compare features from different scenarios.

3 Our Approach: Process and Tools

In a Nutshell. A feature links a model of a program architecture with the program dynamic behaviour. First, we build a model of a program architecture. Second, we identify features to provide maintainers with subsets of the program architecture—micro-architectures—representing classes, methods, and fields that take part in a feature of interest. Third, we compare the different features, the micro-architectures, to highlight their differences.

Our feature identification and comparison process uses both static and dynamic data collected from a program source code and from its execution using processor emulation.

We adopt an approach inspired by Wilde [5] to filter and to extract dynamic data during feature identification: We associate events with features using a relevance index, a ranking quantifying the probability that an event is relevant to a feature under study. As in [5], we avoid the use of set manipulations (set difference or intersection). Indeed, approaches based on set manipulations assume that events, relevant and irrelevant, are either present or absent, and that events are registered correctly. Unfortunately, it is very difficult (sometimes impossible) to control all the variables that influence dynamic data collection in multi-threaded or distributed programs, which is a threat to set-based approaches.

A similar threat exist for approaches based on concept analysis: Concept analysis-based approaches, such as Eisenbarth's [7], cannot distinguish irrelevant but interleaved events and require careful examination of lattice regions to identify feature-relevant events.

Figure 1 summarises the process of feature identification and comparison in our approach and details the process of collecting dynamic data, using the IDEF∅[1] graphical language. Several tools were developed/integrated to support the feature identification and comparison process. The authors reused available tools by coupling their respective tools and other open-source tools loosely, thus ensuring independence from specific CASE tools. The following paragraphs detail each activity of the process and associated tools.

[1]http://www.idef.com/IDEF0.html

Program Model Creation. We use static analysis to build a model of a program architecture from its source code. A static analysis parses C++ source code, both headers and body files, resolves and binds the various types, and generates a model of the program architecture expressed in an intermediate representation, the ABSTRACT OBJECT LANGUAGE (AOL) format [1]. Parsers for several programming languages, in particular for C++, exist to create AOL files.

We manipulate AOL representations using the PADL meta-model. PADL is a language-independent meta-model to represent object-oriented programs. It offers a set of entities to represent object-oriented programs: Class, interface, field, method, and their relationships. It also offers a set of parsers to build models from various sources. Parsers can be easily added using the Builder design pattern. Thus, we added a parser for AOL files using the JAVACUP parser generator.

Trace Collection. The child diagram in Figure 1 (bottom) shows the details of the process of collecting dynamic data. Scenarios executions are modelled as traces; Traces are modelled as sequences of intervals [5], which are sequences of events. As in [5] and [7], the definition of an event is blurred intentionally to accommodate different level of granularity: An event may correspond to creating an object, entering a method, or executing a code fragment.

Two families of scenarios are considered: Those related to a feature of interest and those which do not exercise the feature. We exercise each scenario to collect traces. Depending on the scenario, the studied feature, and the locations of the intervals during the scenario, we mark intervals as relevant to a feature or not. Intervals relevant to a feature are likely to contain feature-relevant events. Events relevant to a feature should always be present in feature-relevant intervals. Thus, in absence of *noise* and imprecision, set operations suffice to classify events as relevant or irrelevant to a feature. However, the precise location of an event in time or the identification of an event marking the exact beginning/end of feature-relevant intervals is not always feasible when collecting dynamic data on multi-threaded or distributed programs. Furthermore, feature-relevant events may be interleaved with noise, irrelevant events, and relevant events may be lost because of statistical profiling imprecision. We use processor emulation to collect precise dynamic data when exercising scenarios and probabilistic ranking to classify events as feature relevant or irrelevant.

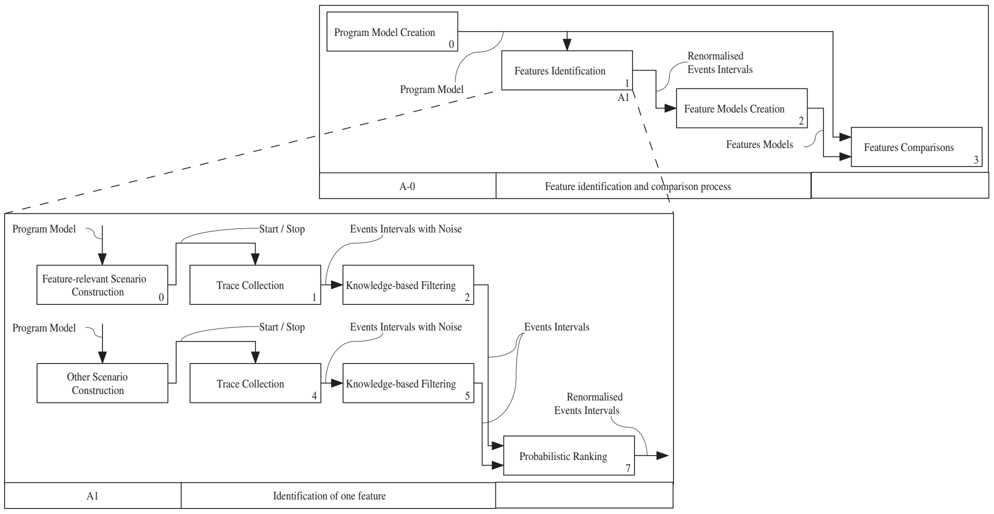

Figure 1. Feature identification and comparison process (detailed feature identification)

VALGRIND[2] is an open-source framework for debugging and profiling x86-Linux programs. VALGRIND simulates an x86 processor and allows the development of specialised plugins for the dynamic analysis of memory management or multi-threads bugs. The collected data are detailed and precise because VALGRIND emulates the processor, at the cost of a reduction in performance of the program under analysis. An interesting plugin for our purposes is CALLGRIND/KCACHEGRIND by Josef Weidendorfer, to collect dynamic data on called methods and functions.

Other approaches to generate trace data exists. One of the more widely used is source code instrumentation. However, in modern multi-language systems, source code instrumentation requires a variety of tools not always easily available. Further difficulties are encountered with threads because thread instrumentation requires source code transformations more complex than mere method entry and exit instrumentation.

Finally, statistical profiling is a suitable choice when accuracy is not essential and overhead must be kept low. We use JPROF to compare statistical profiling with processor emulation. JPROF[3,4] is a statistical profiler developed by Jim Nance for the light profiling of MOZILLA. It instantiates a signal handler (controlled via an environment variable), which generates interrupts at fixed times. At each interrupt, JPROF analyses the call stack and records called methods and functions. Thus, the analysed program runs with little overhead but JPROF requires multiple executions because collected data may include only a subset of all method and function calls due to multitasking and sampling time between interrupts.

[2]http://valgrind.kde.org/

[3]http://lxr.mozilla.org/mozilla/source/tools/jprof/

[4]Statistical profilers, in general, do not collect traces but time spent in functions and methods along with callees and callers, such data can be used to build traces.

Knowledge-based Filtering. In Figure 1, we introduce the concept of knowledge-based filtering. We are not interested in all classes of events, for example, graphical events generated by the mouse or actions of reading/writing from/to an external database or configuration files. We remove events irrelevant for some scenarios of interest to reduce the noise.

We define two families of filters, based on statistical knowledge and on application knowledge. A frequent event which is equally encountered in feature relevant and irrelevant traces is unlikely to improve the maintainers' understanding and can be removed. Application knowledge also helps to reduce the quantity of dynamic data. For example, if a program uses some middleware (*e.g.*, DCOM, CORBA), automatically-generated code, external components (*e.g.*, databases), then these events can be removed if not relevant to the feature of interest directly.

Probabilistic Ranking. We use a probabilistic ranking technique to classify events as relevant or irrelevant to a feature. Let $\mathcal{F}$ be a set of scenarios exercising a feature of interest and $\overline{\mathcal{F}}$ a set of scenarios not exercising the feature. Execution of scenarios in $\mathcal{F}$ produces a set of intervals $\mathcal{I}^*$ containing events relevant to the feature under study. We mark these intervals as relevant via the *Start / Stop* signals in Figure 1. Scenarios in $\overline{\mathcal{F}}$ always produce intervals in $\mathcal{I}$, intervals irrelevant to the feature. However, intervals $\mathcal{I}^*$ ($\mathcal{I}$, respectively) may contain irrelevant (relevant) events. We adopt a strategy derived from Wilde [5] to classify events in $\mathcal{I}^*$ and $\mathcal{I}$ with a statistical relevance index. Three types of events exist: e_i^*, event specific to a feature; $\overline{e}_j$, event always absent in, or irrelevant to, a feature; and e_k, event likely to participate in a feature.

In [5], Wilde maps events and components with a function $\delta()$. We assume a simplified one-to-one mapping so that events and components can be interchanged. Wilde proposes the following relevance index:

$$p_c(e_i) = \frac{N_{\mathcal{F}}(e_i)}{N_{\mathcal{F}}(e_i) + N_{\overline{\mathcal{F}}}(e_i)} \tag{1}$$

where $N_{\mathcal{F}}(e_i)$ is the number of times e_i appears when executing scenarios in $\mathcal{F}$ and $N_{\overline{\mathcal{F}}}(e_i)$ is the number of times the same event is encountered while executing $\overline{\mathcal{F}}$ scenarios. Clearly, p_c ranges between zero and one. It is one, or very close to one, for any feature-relevant event e_i^*, and zero for $\overline{e}_j$. However, in our experience, events irrelevant to a feature are still frequent in traces collected from scenarios in $\mathcal{F}$. Indeed, any scenario is likely to decompose in a few intervals in $\mathcal{I}^*$ surrounded by many intervals in $\mathcal{I}$. Thus, we modify equation 1 to weight the sizes of the different intervals. If $N_{\mathcal{I}^*}$ and $N_{\mathcal{I}}$ are the overall numbers of events in the two sets $\mathcal{I}^*$ and $\mathcal{I}$, then the frequency of e_i in $\mathcal{I}^*$ is $f_{\mathcal{I}^*}(e_i) = \frac{N_{\mathcal{I}^*}(e_i)}{N_{\mathcal{I}^*}}$ and its frequency in $\mathcal{I}$ is $f_{\mathcal{I}}(e_i) = \frac{N_{\mathcal{I}}(e_i)}{N_{\mathcal{I}}}$. We combine the previous equations to propose the following relevance index:

$$r(e_i) = \frac{f_{\mathcal{I}^*}(e_i)}{f_{\mathcal{I}^*}(e_i) + f_{\mathcal{I}}(e_i)} \tag{2}$$

Equation 2 is a *renormalization* of Wilde's equation, where events are re-weighted by population sizes to make events comparable directly. We use Equation 2 to classify events from sets $\mathcal{I}^*$ and $\mathcal{I}$. For a positive threshold t, the set of feature-relevant events is:

$$\mathcal{E}_t^* = \{e_i | r(e_i) \geq t\} \tag{3}$$

$\mathcal{E}_t^*$ size depends on the chosen threshold, a threshold of 100.00% means that events in $\mathcal{E}_t^*$ never appear in $\mathcal{I}$. Equation 3 partitions events in two sets: Those relevant to the feature and those that are unlikely to participate in the feature.

Features Models Creation. We use the program architectural model and the events classified as belonging to $\mathcal{E}_t^*$ to create new models of the program that include only the classes, methods, and fields activated when exercising some scenarios. Such models represent "slices" of the program and are micro-architectures of the program.

We create these micro-architectures by cloning the program architectural model and by removing from this model, using model transformations, all classes, fields, and methods that are not explicitly (directly or indirectly) called when exercising a functionality.

We can create potentially an infinite number of micro-architectures through the executions of different functionalities of the program for different scenarios. We can also create micro-architectures of various *depths*: A micro-architecture may include the classes and only the class exercised or it may include the classes that are related, directly or indirectly, with the classes exercised. Thus, a micro-architecture may be narrow or large arbitrarily, up to the point where it is equal to the complete program architecture.

Features Comparison. We compare and highlight differences among micro-architectures so that maintainers can understand and compare behaviours of different functionalities or of a same functionality with different scenarios. We can use set intersection for features comparison because the micro-architectures are built using events classified uniquely through knowledge filtering and probabilistic ranking.

We use model transformation techniques to highlight differences among micro-architectures: Given an "origin" micro-architecture, we compute the set of model transformations required to transform this micro-architecture in another "destination" micro-architecture. We use this set of transformations to add classes, methods, and fields missing to the "origin" micro-architecture with respect to the "destination" micro-architecture and to distinguish in the "origin" micro-architecture classes, methods, and fields not included in the "destination" micro-architecture. The modifications in the "origin" micro-architecture are described using specific entities of the meta-model to highlight differences between micro-architectures.

Micro-architectures are also models of the PADL meta-model, we develop a Visitor to compare micro-architectures among themselves. Features identification and comparison are independent of the order in which we store classes, fields, and methods.

4 Case Study

Our feature identification and comparison process aims at assisting program understanding tasks of large multi-threaded object-oriented programs by identifying the micro-architectures implementing some features of interest and by highlighting the classes, fields, and methods activated. We realise a study to assess the usefulness of our approach by mimicking a program understanding task assuming no previous knowledge of the program to identify a feature. We need at least two scenarios in our case study to build the intervals sets $\mathcal{I}^*$ and $\mathcal{I}$.

4.1 Object of the Case Study

MOZILLA[5] is an open-source web browser ported on almost all known software and hardware platforms. It is large enough to represent a real world program. MOZILLA size ranges in the millions of lines of code (MLOC). It is developed mostly in C++. C code accounts only for a small fraction of the program. We do not include in our study Java, IDL, XML, HTML and scripting language configuration and support.

The latest MOZILLA versions include more than 10,000 source files for a size of up to 3.70 MLOC decomposing in about 3,500 subdirectories. MOZILLA consists of 90 modules maintained by 50 different module owners. We use version 1.5.1 for our case study.

	Numbers	(MLOC)		Numbers
Header files	8,055	(1.50)	Classes	4,853
C files	1,762	(0.90)	Methods	53,617
C++ files	4,204	(2.00)	Specialisations	5,314
IDL files	2,399	(0.20)	Associations	17,362
XML files	283	(0.12)	Aggregations	6,727
HTML files	2,231	(0.19)		
Java files	56	(0.06)		

Table 1. Mozilla v1.5.1 sizes

Table 1 gives an overview of the sizes of the web browser. Figures reported should be considered as orders of magnitudes rather than as absolute values. Indeed, several factors influence these figures, such as the reverse engineering tools and the parsing techniques used [14] or the way in which certain programming language features are considered. In our case study, we choose to use conservative reverse-engineering techniques: We apply strict reverse-engineering rules such that we classify as classes only entities declared as classes according to the C++ syntax. We consider structures and complex templates (*e.g.*, templates mixed with structures) as outside of the boundary of reverse-engineered models and do not recover their attributes, methods, and files locations.

4.2 Scenarios and Task of the Case Study

One of the key functionality of a web browser is the ability to store and to access URL. We consider the following scenarios:

- **Scenario 1**: A user visits an URL: She opens MOZILLA, clicks on a previously book-marked URL, waits for the page to be loaded, and closes the web browser.
- **Scenario 2**: The user acts as above but, once the page is loaded, she saves the URL using the mouse right button and closes the web browser.

[5]http://www.mozilla.org

Sequences of actions between the scenarios are the same but for the book-marking action in **Scenario 2**. Thus, intuitively, all classes activated in **Scenario 1** are present in **Scenario 2** (but not vice-versa).

Program Understanding Task. We formulate the following program understanding task: "Identify classes, fields, and methods activated when an URL is saved in **Scenario 2** with respect to **Scenario 1**".

Objective. We want to assess the usefulness of our approach by identifying feature related to **Scenario 2** with respect to **Scenario 1**.

Assessment. We assess our objective in two steps. First, we tackle the program understanding task with a naive approach based on usual string-matching tools, with a concept analysis-based technique from the literature [7], and with our feature identification and comparison process. We compare and discuss our results with results obtained with the naive and concept analysis-based approach. We also use this case study to compare processor emulation (VALGRIND) with statistical profiling (represented by JPROF).

The activity of compiling a version of MOZILLA required about 30 minutes on a Pentium IV laptop running RedHat enterprise workstation. A single execution, about 5 minutes, is sufficient to gather dynamic data with VALGRIND. VALGRIND raw data are saved in ASCII and are mapped to sets $\mathcal{I}^*$ and $\mathcal{I}$ subsequently.

Several execution are required with JPROF to minimise the probability that relevant data are not sampled due to the granularity of the interrupts and of Linux process context switching. Collecting JPROF profiling data require about 20 minutes for any scenario (20 executions). JPROF data are saved in HTML.

4.3 Results with a Naive Approach

The task of identifying a feature is similar to an information retrieval task. A naive maintainer would attempt to perform the understanding task by searching files with tools supporting string and regular expressions matching, such as the Unix utility `grep`. For example, she would search for files containing the term *add* or synonyms such as *store* and *save*. Several combinations in conjunction or alternative with synonyms and abbreviations to the term bookmark are possible (*e.g.*, *url*, *bookm*, *link*, *ref*). When performing such a process, results are not very encouraging. Data in Table 2 suggests that only a conjunction of terms could reduce reasonably the number of files to be inspected to identify the feature responsible to save a bookmark.

Terms	Occurrences	Terms	Occurrences
add	3,482	*bookm*	15
store	877	*link*	732
save	639	*ref*	2,632
url	782	*uri*	3914

Table 2. Mozilla v1.5.1 term occurrences over 7,411 files (C, C++ and header files)

Various combinations of terms and synonyms such as *add* and *link* produce hits in the hundreds or thousands. A more careful selection of terms would reduce the number of hits drastically. Indeed, by imposing the match of *bookm* and *link*, *ref* or *url*, only 14 hits are returned. Even more promising is the match of *bookm* and *add*, which returns 8 hits.

However, this matches are string-based at file level. There is no way to distinguish a meaningful match with unwanted matches automatically. The maintainer cannot know whether she should also consider one of the other thousands matches or if she missed relevant information because the correct string was not searched.

A more clever approach would be to perform query operations on trace data. Regular expressions matching performed on VALGRIND (JPROF, respectively) data returns limited numbers of hits, depending on the matched string. For example, matchings for the terms *bookm* and *link* return 51 (45) and 44 (18) matches, respectively. However, matchings for other terms, such as *ref* or *add*, return more than 3,000 (500) hits. Thus, even if trace data are available, regular expressions matching is not sufficient to focus the search regardless of the query, *i.e.*, without previous knowledge.

4.4 Results with Concept Analysis

As recognised and proposed in [6, 7], feature identification can be performed using concept analysis. For example, called methods can be considered as the *objects* and scenarios as the *attributes* in some concepts, then the concept lattice represents relations between scenarios, *i.e.*, activated features.

	Method calls (distinct called methods)			
	in $\mathcal{I}^*$		in $\mathcal{I}$	
Startup	0	(0)	92,622,767	(22,507)
Shutdown	0	(0)	24,256,743	(12,007)
Scenario 1	63,154,231	(22,542)	92,781,994	(22,592)
Scenario 2	36,105,209	(17,134)	134,849,884	(26,415)

Table 3. Mozilla v1.5.1 Valgrind $\mathcal{I}^*$ and $\mathcal{I}$ typical sizes when exercising the scenarios

Table 3 reports typical sizes of $\mathcal{I}^*$ and $\mathcal{I}$ in number of events as collected with VALGRIND. It reports the cumulative number of all events observed (*i.e.*, if a method is called 10 times, then 10 events are counted), including Java method calls and C functions but not system calls or system library calls. We are not interested in features related to browser startup and shutdown, so we do not distinguish $\mathcal{I}^*$ and $\mathcal{I}$ in these particular scenarios. We decompose **Scenario 1** and **2** between methods called during browser startup (*i.e.*, $\mathcal{I}$) and the action to access a bookmark (*i.e.*, $\mathcal{I}^*$). The focus of the program understanding task is to identify classes and methods which characterise **Scenario 2**, which are part of the 36,105,209 $\mathcal{I}^*$ method calls.

We create a formal context $C = (O, A, R)$ for **Scenario 1** and **Scenario 2** following the approach presented in [7]. We consider $\mathcal{I}^*$ and $\mathcal{I}$ for **Scenario 1** and **2** as four sub-scenarios. A, the set of attributes, contains four symbols to represent $\mathcal{I}^*$ and $\mathcal{I}$ for **Scenario 1** and **Scenario 2**. O, the set of objects, contains all *distinct* methods activated when exercising **Scenario 1** and **Scenario 2**. There are about 30,000 distinct methods present in the $\mathcal{I}^*$ and $\mathcal{I}$ sets for **Scenario 1** and **Scenario 2**. R is a binary relation; a pair (o, a) is in R if the method $m \in O$ is called when the sub-scenario $s \in A$ is performed.

The resulting lattice contains 11 concepts (excluding top and bottom); concepts ranges with a minimum of 13,325 and a maximum of 26,613 contained methods. Concepts inspection is difficult given the number of contained methods. A more realistic approach is to inspect set difference of attribute sets (*i.e.*, sets of methods). We target methods used to save one bookmark, which should be present only in one concept, because we are interested in **Scenario 2**. A poor strategy is to consider the union of pairwise set differences between the 11 concepts, because this union contains 15,125 different methods.

A more careful manual inspection identifies in one concept the attributes specific to **Scenario 2** $\mathcal{I}^*$. Thus another approach could be to compute the union of pairwise set differences between this one concept and the other 10 concepts. This approach reduces the number of methods to be inspected, but still the number lies in the range of thousands.

An efficient manual inspection consists in inspecting the lattice to identify concepts corresponding to the $\mathcal{I}^*$ sets of **Scenario 1** and **Scenario 2** and to compute set difference between these two concepts only. Thus, 1,038 methods are retained: These are candidate methods to implement **Scenario 2** specific functionality.

Above considerations support the idea that concept analysis is a powerful tool but results can be discouraging and need to be complemented by heuristics or other approaches to focus the search. Concept analysis is more selective and better narrows the search than a naive approach. However, it suffers of the problem of set difference.

Similar results were obtained when JPROF traces were analysed, JPROF data are not reported due to lack of space.

4.5 Results with our Approach

We use the collected data, in Table 3, to filter MOZILLA architectural model and to extract two micro-architectures corresponding to **Scenario 1** and to **Scenario 2**.

Starting the browser, waiting for the main window to appear, and closing the browser corresponds to a functionality and a feature included in both **Scenario 1** and **Scenario 2** but irrelevant to the features of interest. Mouse and widget events are also noise irrelevant to the program understanding task.

Thus, as explained in Figure 1, we use events in the $\mathcal{I}^*$ and $\mathcal{I}$ sets for **Scenario 1** and **Scenario 2**, in Table 3, to define a knowledge-based filter to reduce noise. We consider methods present in both $\mathcal{I}^*$ and $\mathcal{I}$ sets with a frequency higher than the lower quartile as irrelevant to **Scenario 1** and **Scenario 2**.

The $\mathcal{I}$ and $\mathcal{I}^*$ sets in Table 3 contain 28,654 and 23,733 distinct called methods respectively. Out of 28,654 called methods in $\mathcal{I}$ (23,733 in $\mathcal{I}^*$, respectively), there are 17,758 methods matching the filtering criterion (*i.e.*, present in $\mathcal{I}$ and $\mathcal{I}^*$ with a frequency greater than the lower quartile). We flag these 17,758 methods as irrelevant and exclude them from further consideration. We assume a situation with no previous knowledge (accumulated, domain, or application). We remove simply events that are unlikely to participate in the features (if they participated, they would be hardly ever distinguishable from the noise).

	Method calls (distinct called methods)			
	in $\mathcal{I}^*$		in $\mathcal{I}$	
Startup	0	(0)	1,161,419	(6,995)
Shutdown	0	(0)	3646	(1,824)
Scenario 1	14,428	(5,551)	1,168,879	(6,984)
Scenario 2	6,592	(2,796)	1,178,330	(9,051)

Table 4. Mozilla v1.5.1 Valgrind $\mathcal{I}^*$ and $\mathcal{I}$ filtered sizes

Filtering the noise reduces dynamic data sizes considerably. Table 4 reports the data in Table 3 with irrelevant method calls removed. We use these dynamic data to create a ranked list of methods, according to equation 1 and 2. Both equations highlight 274 events of type e_i^*, *i.e.*, methods that are specific to **Scenario 2**. These methods belongs to about 80 different classes (see Table 6). The actual number of different classes depends on how we count templates, *i.e.*, if we consider `nsCOMPtr<nsIRollupListener>` as atomic or as composed of two classes `nsCOMPtr` and `nsIRollupListener` and thus contributing twice.

On the contrary to the naive approach, our approach allows ranking, with the relevance index, the classes and methods contributing to a micro-architecture implementing a feature. Clearly, the 274 events of type e_i^* belong to the set of the 1,038 events discovered by concept analysis. The e_i^* events are feature-specific and can also be discovered through set operations.

t	$\mathcal{E}_t^*$ size (*i.e.*, number of methods)	
	Equation 1	Equation 2
100.00%	274	274
99.90%	274	295
99.00%	274	2,273
98.00%	274	2,626
97.00%	274	2,697
96.00%	274	2,760
95.00%	274	2,796
90.00%	274	2,796
50.00%	515	2,796

Table 5. Mozilla v1.5.1 $\mathcal{E}_t^*$ size with different t for Scenario 2 with Valgrind

$\mathcal{E}_1^*$	JPROF	VALGRIND
Added classes	147	10
Removed classes	144	3
Modified behaviour	691	73

Table 6. Mozilla v1.5.1 scenarios overall class differences

Classes and methods counted in Table 5 and Table 6 are the result of feature identification to identify **Scenario 2** micro-architecture. This micro-architecture implements the functionality of interest and thus program understanding is limited to its classes. There are 274 methods participating exclusively to **Scenario 2** (*i.e.*, with a relevance index of 100.00% elements of $\mathcal{E}_1^*$) when using VALGRIND, these methods belong to 10 + 3 + 73 = 86 classes. These 86 classes are the classes that a maintainer must inspect. The 86 classes must be compared to the 878 classes containing the 2,796 methods when using a threshold 50.00%

As shown in Table 5, Equation 2 weights events in a different way than Equation 1. For example, method `nsMenuFrame::IsMenu()` appears one time in $\mathcal{I}$ and two times in $\mathcal{I}^*$, Equation 2 relevance index is 99.92% while Equation 1 index is 66.66%. Further insight is gained when considering method `morkFile::FileIoOpen()`, which appears two times in both $\mathcal{I}$ and $\mathcal{I}^*$ sets in Table 4. Equation 2 gives an index of 99.85% because $\mathcal{I}^*$ contains 6,592 (2,796 distinct) method calls and $\mathcal{I}$ contains 4,698,914 (10,392 distinct) methods calls, while Equation 1 gives an index of 50.00% regardless of the higher frequency of the method in $\mathcal{I}^*$ with respect to $\mathcal{I}$.

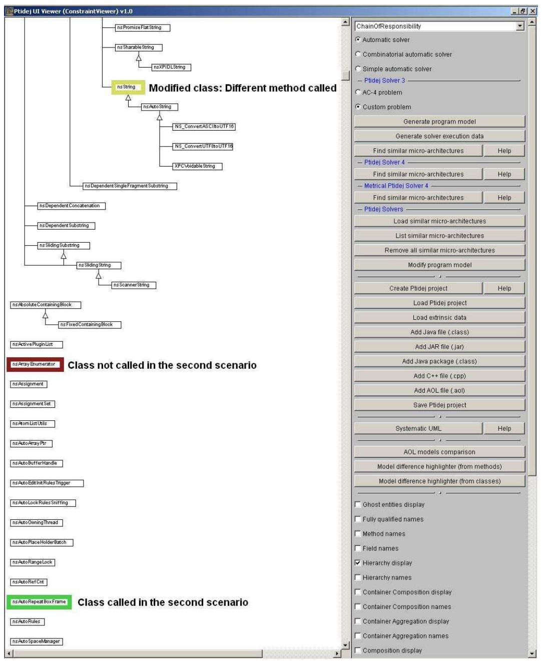

Figure 2. Example of feature comparison for Scenario 1 and Scenario 2

The number of classes and methods that a maintainer must inspect is further reduced substantially if the inspection process is driven by the semantics conveyed by names of classes, fields, and methods. If we apply a strategy similar to the naive approach to classes and methods in Table 6, the number of classes to be inspected is narrowed dramatically. Only one class with modified behaviour is highlighted: Class `nsBookmarksService`, belongs to both **Scenario 1** and **Scenario 2**, in **Scenario 2** it adds to the micro-architecture of **Scenario 1** calls to methods: `AddBookmarkImmediately`, `CreateBookmark`, `CreateBookmarkInContainer`, `InsertResource`, and `getFolderViaHint`. Another modified class `nsComputedDOMStyle` and a method `AddRef` are also highlighted, but we discard the method `AddRef` form inspection immediately because it has no parameter.

The effort required to build mental models and abstractions or to verify conjectures may be alleviated by the adoption of environments and tools supporting program understanding tasks. Figure 2 shows the user-interface of the PTIDEJ tool suite, which highlights differences between the feature of **Scenario 1** and **Scenario 2**. Our approach allows maintainers to identify differences between features precisely. However, the precision of feature comparison is influenced by the precision of the tools collecting the data dynamically: A less precise tool leads to a larger number of classes, methods, and fields to be inspected.

	Scenario 1		Scenario 2	
	JPROF	VALGRIND	JPROF	VALGRIND
Classes	973	2,216	1,179	2,223
Methods	6,170	24,246	6,407	24,409

Table 7. Mozilla v1.5.1 data for Scenario 1 and Scenario 2

4.6 Discussion

Table 7 summarises the data collected with the two tools. Table 7 represents the worst case scenario when $\mathcal{I}^*$ and $\mathcal{I}$ are not distinguished. It is not a surprise that JPROF and VALGRIND results are different. However, such a big difference may represent a serious problem for approaches based solely on set operations or concept analysis. In fact, we cannot be sure that a relevant event is always collected. A careful investigation of **Scenario 2** JPROF dynamic data revealed that out of 20 scenario repetitions, the event `AddBookmarkImmediately` was not sampled four times, which is about a 20.00% error. This fact also calls for more complex and sophisticated approaches with respect to approaches based on set intersection or set differences. The difference between VALGRIND and JPROF, in Table 7, and the above consideration show that JPROF data are under sampled and that more executions are needed to collect data equivalent to VALGRIND. However, further investigation is needed to validate our conjecture and to establish guidelines on how many samples should be reasonably collected if a profiler or statistical sampling are used. Moreover, statistical sampling also affects relevance index values: Under sampling a relevant index potentially results in a larger number of candidate events to inspect during program understanding tasks.

5 Conclusion

We proposed an approach to feature identification and comparison based on consolidated tools and techniques such as parsing, processor emulation, and reverse engineering, to integrate static and dynamic data to support program understanding of large multi-threaded object-oriented programs. We compared the usefulness of our approach to identify classes, fields, and methods supporting a functionality of the MOZILLA web browser (3.70 MLOC) to save a visited page with a naive `grep`-based approach and with an approach relaying on concept analysis.

Our approach allows reducing the quantity of data to process and thus has no scalability issues. It produces a ranked list of methods and classes participating in a feature and it can be extended to study feature evolution easily. It allows extracting and studying any micro-architecture and comparing micro-architectures evolutions.

We will devote future work to study the influence of increased depth of micro-architectures, to define results of features comparisons precisely, and to study alternative means, such as layered views, to improve features inspections.

References

[1] G. Antoniol, R. Fiutem, and L. Cristoforetti. Using metrics to identify design patterns in object-oriented software. In *Proceedings of 5 th International Symposium on Software Metrics - METRICS98*, pages 23–34, Bethesda MD, Nov 20-21 1998.

[2] J. Bézivin. From object composition to model transformation with the MDA. In B. Meyer, editor, *proceedings of the 39th International Conference on Technology of Object-Oriented Languages and Systems*, page 0348. IEEE Computer Society Press, August 2001.

[3] K. Chen and V. Rajlich. Case study of feature location using dependence graph. In A. von Mayrhauser and H. Gall, editors, *proceedings of the 8th International Workshop on Program Comprehension*, pages 241–252. IEEE Computer Society Press, June 2000.

[4] E. Chikofsky and J. C. II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, Jan 1990.

[5] D. Edwards, S. Simmons, and N. Wilde. An approach to feature location in distributed systems. Technical report, Software Engineering Research Center, 2004.

[6] T. Eisenbarth, R. Koschke, and D. Simon. Aiding program comprehension by static and dynamic feature analysis. In G. Canfora and A. A. Andrews, editors, *proceedings of the 8th International Conference on Software Maintenance*, pages 602–611. IEEE Computer Society Press, November 2001.

[7] T. Eisenbarth, R. Koschke, and D. Simon. Locating features in source code. *IEEE Transactions on Software Engineering*, pages 210–224, March 2003.

[8] M. Ernst. Static and dynamic analysis: synergy and duality. In *ICSE Workshop on Dynamic Analysis (WODA 2003)*, Portland, OR, USA, May 2003.

[9] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *International Conference on Software Engineering*, pages 213–224, 1999.

[10] Y.-G. Guéhéneuc and N. Jussien. Using explanations for design-patterns identification. In C. Bessière, editor, *proceedings of the 1st IJCAI workshop on Modeling and Solving Problems with Constraints*, pages 57–64. AAAI Press, August 2001.

[11] A. Hamou-Lhadj and T. Lethbridge. Compression techniques to simplify the analysis of large execution traces. In *Proceedings of the 10th International Workshop on Program Comprehension*, pages 159–168, Paris, France, Jun 26-29 2002.

[12] A. Hamou-Lhadj and T. Lethbridge. An efficient algorithm for detecting patterns in traces of procedure calls. In *ICSE Workshop on Dynamic Analysis (WODA 2003)*, Portland, OR, USA, May 2003.

[13] C. Jeffery and M. Auguston. Some axioms and issues in the ufo dynamic analysis framework. In *ICSE Workshop on Dynamic Analysis (WODA 2003)*, Portland, OR, USA, May 2003.

[14] R. Kollmann, P. Selonen, E. Stroulia, T. Systa, and A. Zundorf. A study on the current state of the art in tool-supported uml-based static reverse engineering. In *Proceedings of IEEE Working Conference on Reverse Engineering*, pages 22–33, Richmond, Virginia, USA, October 29 - November 1 2002.

[15] Object Management Group. Model driven architecture, January 2005. www.omg.org/mda/.

[16] B.-U. Pagel and M. Winter. Towards pattern-based tools. In F. Buschmann, editor, *proceedings of 1st european conference on Pattern Languages of Programs*. Preliminary conference proceedings, July 1996.

[17] S. Reiss and M. Renieris. Languages for dynamic instrumentation. In *ICSE Workshop on Dynamic Analysis (WODA 2003)*, Portland, OR, USA, May 2003.

[18] S. P. Reiss and M. Renieris. Encoding program executions. In *International Conference on Software Engineering*, pages 221–230, 2001.

[19] M. Renieris and S. P. Reiss. ALMOST: Exploring program traces. In *NPIVM Workshop*, pages 70–77, Kansas City, MO, USA, Nov 1999.

[20] M. Salah and S. Mancoridis. A hierarchy of dynamic software views: From object-interactions to feature-interactions. In M. Harman and B. Korel, editors, *proceedings of the 20th International Conference on Software Maintenance*, pages 72–81. IEEE Computer Society Press, September 2004.

[21] G. Sunyé, A. L. Guennec, and J.-M. Jézéquel. Design patterns application in UML. In E. Bertino, editor, *proceedings of the 14th European Conference for Object-Oriented Programming*, pages 44–62. Springer-Verlag, June 2000.

[22] D. Thomas. Reflective software engineering – From MOPS to AOSD. *Journal of Object Technology*, 1(4):17–26, September 2002.

[23] N. Wilde and M. C. Scully. Software reconnaissance: Mapping program features to code. In K. H. Bennett and N. Chapin, editors, *Journal of Software Maintenance: Research and Practice*, pages 49–62. John Wiley & Sons, January-February 1995.