Feature Identification: An Epidemiological Metaphor

Giuliano Antoniol and Yann-Gaël Guéhéneuc

Abstract—Feature identification is a technique to identify the source code constructs activated when exercising one of the features of a program. We propose new statistical analyses of static and dynamic data to accurately identify features in large multithreaded object-oriented programs. We draw inspiration from epidemiology to improve previous approaches to feature identification and develop an epidemiological metaphor. We build our metaphor on our previous approach to feature identification, in which we use processor emulation, knowledge-based filtering, probabilistic ranking, and metamodeling. We carry out three case studies to assess the usefulness of our metaphor, using the "save a bookmark" feature of Web browsers as an illustration. In the first case study, we compare our approach with three previous approaches (a naive approach, a concept analysis-based approach, and our previous probabilistic approach) in identifying the feature in MOZILLA, a large, real-life, multithreaded object-oriented program. In the second case study, we compare the implementation of the feature in the FIREFOX and MOZILLA Web browsers. In the third case study, we identify the same feature in two more Web browsers, Chimera (in C) and ICEBrowser (in Java), and another feature in JHOTDRAW and XFIG, to highlight the generalizability of our metaphor.

Index Terms—Program understanding, dynamic analysis, static analysis, feature identification, epidemiology, FIREFOX and MOZILLA Web browsers.

1 INTRODUCTION

MAINTENANCE of legacy software involves costly and tedious activities to identify and to understand data structures, functions, methods, objects, classes, and, more generally, any high-level abstraction required by maintainers. Source code browsing is the most common activity performed during maintenance because obsolete or missing documentation forces maintainers to rely on source code only. Unfortunately, source code browsing becomes very time- and resource-consuming when the size and complexity of programs increase.

An alternative to source code browsing is automatic or semiautomatic design recovery. Central to this is the recovery of "higher-level abstractions beyond those obtained by examining a program itself" [1]. We propose an approach to recovering higher-level abstractions through identification and comparison of program features [2], [3]. For example, in a Web browser, accessing a page from the bookmarks corresponds to a feature; adding a Uniform Resource Locator (URL) to the bookmarks is another. Our approach extends our previous work [4] and significantly improves the accuracy of previous approaches.

We assume that the source code of the program being maintained is available and that a compiled version can be executed under different scenarios. We use an epidemiological metaphor to analyze data collected dynamically when exercising a feature under different scenarios to build microarchitectures—subsets of the program architecture [5]—from static and dynamic data, thus relating variables, structures, classes, functions, and methods to features and scenarios. We then use these microarchitectures to highlight differences among features. By using our approach to build and to compare microarchitectures, maintainers can locate precisely the source code constructs responsible for a feature and highlight the differences among features.

Our approach to feature identification relies on a process and a set of supporting tools. The process builds on our previous work [4] using processor emulation, knowledge-based filtering, probabilistic ranking, and model transformations. It shows improved accuracy in identifying features through the use of statistical analyses inspired by epidemiology. We draw a parallel between events observed in execution traces and diseases in a population of individuals. Scenarios are the equivalent of environmental conditions under which certain events occur. As epidemiologists, we try to determine whether a disease (event) is more frequent under certain environmental conditions (scenarios). Events that are more frequent for scenarios in which a feature of interest is exercised are more likely to relate to this feature. The epidemiological analysis at the heart of our process is a major extension of relevance indices from previous work [4], [6]. It improves the accuracy of the feature identification process when there is disorder due to multithreading and imprecision during the collection in dynamic data.

We evaluate our approach through the analysis of large multithreaded object-oriented programs: the FIREFOX and MOZILLA Web browsers. We show, through direct comparison with previous results [4], that the epidemiological analysis overcomes the limitations of existing approaches and, combined with probabilistic ranking, narrows the size of microarchitectures dramatically. We also apply our approach to FIREFOX and compare an identified feature with that of MOZILLA to show the insights gained through feature comparisons. Finally, we use our approach on two C and two Java programs to discuss the generalizability of the epidemiological analysis.

- *G. Antoniol is with the Département d'informatique, École Polytechnique de Montréal, CP 6079, succ. Centre-Ville, Montréal, Quebec, H3C 3A7, Canada. E-mail: antoniol@ieee.org.*
- *Y.-G. Guéhéneuc, is with the Département d'informatique et recherche operationelle, Université de Montréal, CP 6128, succ. Centre-Ville, Montréal, Quebec, H3C 3J7, Canada. E-mail: guehene@iro.umontreal.ca.*

Manuscript received 6 Dec. 2005; revised 10 Apr. 2006; accepted 20 July 2006; published online DD Mmmm, YYYY.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-0322-1205.

The main contributions of this work are:

- A metaphor between feature identification and epidemiology (Section 2.2).
- An exhaustive experimental comparison of our approach with previous work (Section 3.1).
- An experimental study of the differences between FIREFOX and MOZILLA (Section 3.2).
- A discussion and illustration of the generalizability of our metaphor (Section 3.3).

Section 2 describes our process and tools for feature identification and comparison, detailing data collection (Section 2.1), an analogy between feature identification and epidemiology (Section 2.2), and the modeling of features as microarchitectures and their comparisons (Section 2.3). Section 3 presents an experimental comparison using MOZILLA of our approach with other approaches based on `grep`, concept analysis, and probabilistic ranking as well as an experimental study of the architectural and behavioral differences between FIREFOX and MOZILLA. It also sketches the generalizability of our approach to other paradigms and programming languages, illustrated with four programs in C and in Java. Section 4 discusses the metaphor of epidemiology. Section 5 summarizes previous work and its limitations, and Section 6 outlines future work.

2 Feature Identification

We borrow the concept of feature from previous work [3], [6]. A *feature* is defined as a requirement of a program that a user can exercise and which produces an observable behavior. This includes functional requirements, e.g., the "save bookmark" feature of a Web browser, and *observable* nonfunctional requirements, e.g., a security algorithm verifying a user's identity. A feature is described in terms appropriate to the context of the program and is weakly defined intentionally to accommodate any foreseeable situation. *Feature identification* is defined as the activity of identifying the source code constructs implementing a given feature. The source code constructs may be scattered throughout the source code and/or entangled among constructs implementing other features so they are difficult to identify manually. We want to provide maintainers with the source code constructs associated with a feature of interest. We define a *microarchitecture* as the subset of a program architecture—i.e., variables, structures, classes, functions, and methods—that implements a given feature concretely by contributing to the realization of the feature. Maintainers use microarchitectures and their differences to quickly understand the source code constructs activated or not when exercising a particular feature of a program under different scenarios.

Our feature identification and comparison process uses both static and dynamic data collected from a program source code and its executions. A *scenario* provides the conditions under which a feature is exercised. First, we collect dynamic data from different scenarios as *traces*, lists of events composed of variable accesses, object instantiations, function and method calls. We then use an epidemiological metaphor to classify the events in traces as relevant or not to a feature of interest. Also, we rank the relevant events to associate top-ranked events with a feature. Finally, we use *feature-relevant events* to build microarchitectures highlighting the source code constructs that implement the feature.

Fig. 1 summarizes the process of feature identification and comparison in our approach, using the IDEFØ [7] graphical language. The following sections detail each step of the process and associated tools in three parts: data collection (program model creation, trace collection, and knowledge-based filtering), data analysis (epidemiological metaphor and relevance index), and data modeling (feature model creation and comparison). Several tools were developed and integrated to support the process. The authors reused their tools as well as existing open-source tools.

2.1 Data Collection

There are three activities in the initial data collection: 1) program model creation, 2) trace collection, and 3) knowledge-based filtering.

2.1.1 Program Model Creation

We use static analysis to build a model of the architecture of a program from its source code. Such a model is an instance of a dedicated metamodel, which offers all the constituents needed to describe the structure of object-oriented programs: variables, functions, classes, interfaces, fields, methods, parameters, and relationships (such as specialization, association, and aggregation).

A static analyzer parses C++ source code, both headers and implementation files, resolves and binds types, and generates a model of the program architecture expressed in the AOL intermediate representation (ABSTRACT OBJECT LANGUAGE format [8]). Parsers for several programming languages, such as C++, Java, and IDL, exist to create AOL representations.

We manipulate AOL representations using the PADL metamodel. This metamodel provides a set of constituents to describe the structure of object-oriented programs and a set of parsers to build models of programs from code sources in various languages. Parsers can be easily added through the Builder design pattern. We added a parser for AOL files using the JAVACUP parser generator.

2.1.2 Trace Collection

Trace collection depends on executions of feature-relevant and irrelevant scenarios. Scenario executions are modeled as traces. Traces are modeled as sequences of intervals [9], which are sequences of events. The definition of an event is blurred intentionally as in [9] and [3] to accommodate different levels of granularity. An event may correspond to the instantiation of an object, a method call, or the execution of a code fragment.

We consider two families of scenarios: those that exercise a feature of interest and those that do *not*. 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 or irrelevant to the feature. Relevant events should always be present in feature-relevant intervals. Thus, without disorder or imprecision, set operations would suffice to classify events as relevant or irrelevant.

However, precise location of an event in time or identification of an event marking the exact beginning or end of feature-relevant intervals may be very difficult or simply not possible to achieve when collecting dynamic data on multithreaded programs. Indeed, existing tools to collect dynamic data, namely profilers and debuggers, must use either the Marking Instant in Time strategy (MIT, e.g.,

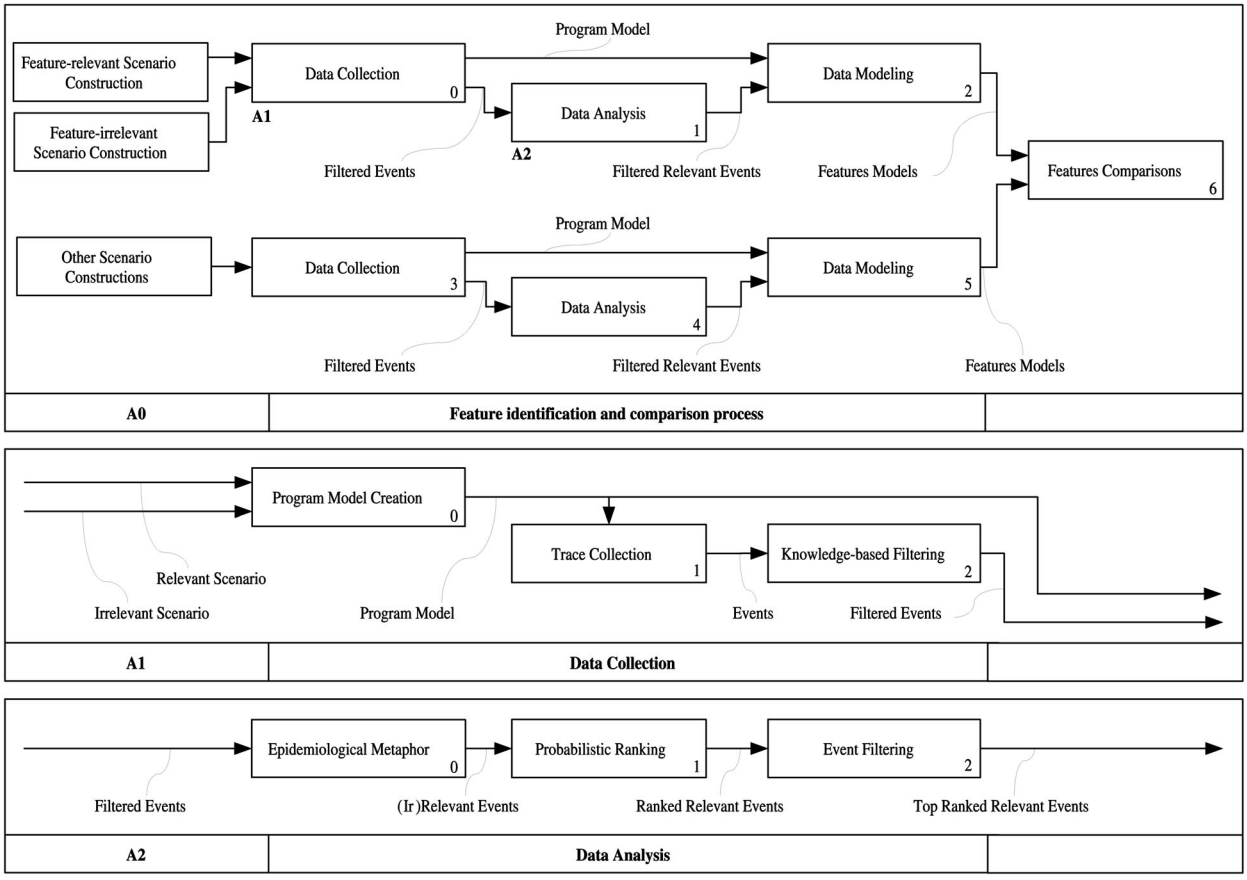

Fig. 1. Feature identification and comparison process (detailed data collection and analysis).

collecting events every 2 milliseconds), the Marking Event strategy (ME, e.g., collecting events related to method calls only) or a hybrid MIT and ME strategy (such as VTUNE, OPROFILE, or GPROF). These strategies can be used to mark events as feature relevant, but are dependent on the scheduler of the operating system—on the order of execution of the processor threads. Thus, regardless of the chosen strategy, events may be collected in different orders in successive, seemingly identical executions. Moreover, tools often use event sampling to reduce the amount of dynamically collected data, the size of the traces. Thus, feature-relevant events may be tangled with irrelevant events because of disorder in multithreading, and relevant events may be lost because of imprecisions during the collection of dynamic data.

We use processor emulation to improve the order *and* the precision of data collection. VALGRIND [10] is an open-source framework for debugging and profiling x86-Linux programs. VALGRIND simulates an x86 processor and allows the development of specialized plugins for the dynamic analysis of memory management or multithread bugs. The collected data are precise because VALGRIND emulates the processor at the cost of a reduction in performance of the program under analysis. We perform data collection with CALLGRIND, an extension to VALGRIND. CALLGRIND and KCACHEGRIND, by Josef Weidendorfer, use the runtime instrumentation of VALGRIND for cache simulation and call-graph generation. Cache simulation and call-graph generation allow for the profiling of shared libraries and dynamically opened plug-ins. The data generated by CALLGRIND can be loaded in KCACHEGRIND for browsing and analysis.

Other approaches also exist to generate dynamic data. Statistical profiling, such as JPROF [11], which uses an MIT strategy, is a suitable choice when accuracy is not essential and overhead must be kept low. Statistical profilers do not collect traces but, rather, time spent in functions and methods along with callees and callers. Such data can be used to recover partial trace information. Another widely used approach is source code instrumentation using the ME strategy. However, in modern multilanguage programs, this approach requires a variety of tools not always readily available. Further difficulties are encountered with threads because a thread-safe instrumentation requires source code transformations more complex than method entry and exit instrumentations. Debuggers can alleviate the burden of source code instrumentation, but do not help with threads because threads are often implemented as lightweight processes and it is often impossible to attach a debugger on a running thread or spawn a debugger for each thread.

Processor emulation collects data in a more orderly and precise manner than statistical profiling and debuggers do. Furthermore, VALGRIND implements a POSIX thread layer, so threads are no longer an issue. To the best of our knowledge, VALGRIND thread implementation is incomplete and programs may misbehave. In our experiments, however, we did not observe any misbehavior, except an expected substantial performance overhead and increased *noise* due to the greater precision. We call *noise* any event that a posteriori knowledge reveals as irrelevant to a feature of interest. The lack of a priori knowledge limits the ability

to filter irrelevant events. Noise is common with modern multithreaded processors, event-driven multithreaded programs, and graphical interfaces, so we use knowledge-based and statistical filtering to reduce it.

2.1.3 Knowledge-Based Filtering

We are not interested in all classes of events collected in traces. For example, graphical events generated by the mouse or actions of reading or writing from or to an external database or configuration files may be considered noise with respect to a feature of interest. We remove events *obviously* irrelevant to some scenarios of interest from the dynamically-collected data and, thus, reduce noise.

We define a family of filters based on application knowledge, which helps to reduce the quantity of dynamic data. For example, if a program uses middleware such as DCOM or CORBA or external components such as databases, the related data can be removed if not directly relevant to the feature.

Furthermore, if an event has already been classified as feature relevant or irrelevant, then it can be removed from the dynamic data. Thus, as knowledge increases and more events are classified, maintainers use this knowledge to narrow the feature through a feedback loop, which reduces the effort to precisely identify the feature.

Thus, knowledge filtering reduces the sets of events. We use an epidemiological metaphor to classify remaining events as relevant or irrelevant to a feature of interest and a relevance index to rank these events.

2.2 Data Analysis

The three activities in data analysis consist of classifying events as relevant or irrelevant to a feature of interest, ranking relevant events, gathering most relevant events, and associating these with a feature.

2.2.1 Epidemiological Metaphor

Epidemiologists study the prevalence of a disease, which is the percentage of individuals infected in a population at risk. The prevalence is calculated as the ratio of the number of individuals infected with the disease (instances of the disease) over the total number of individuals at a given time. The prevalence of a disease varies across time, populations, and environmental conditions. Epidemiologists are interested in verifying if different environmental conditions lead to a statistically significant difference in prevalence.

Different environmental conditions promote different diseases, e.g., the exposure to certain chemical substances may lead to carcinogenesis, while a diet based on vegetables seems to provide protection. For a population and a disease, the ratio of prevalences highlights different environmental conditions. Like epidemiologists, we want to compute the ratio of prevalences of events in different traces to identify feature-relevant events with respect to a given scenario.

We draw a parallel between the prevalence of a disease in a population and the prevalence of events relevant to a feature of interest in a trace. Table 1 summarizes the mapping between the concepts in epidemiology and those in feature identification. A scenario defines the environmental conditions in which a feature is studied. A feature of interest in a given scenario corresponds to a disease under some environmental conditions. Such a feature is characterized dynamically by its events and statically by its microarchitecture, as the "typical" symptoms of a disease may have different manifestations in different individuals.

TABLE 1
Mapping between Concepts in Epidemiology and Concepts in Feature Identification

Concepts	
Epidemiology	Feature Identification
Environmental conditions	Scenario
Disease	Feature (feature-relevant events)
Symptoms of a disease	Micro-architecture
Population	Trace
Sick individuals	Events

A trace is a population, i.e., a set of events that may be relevant or irrelevant to the feature under study, in which events are similar to sick individuals. Our metaphor assumes that any event belongs to a feature—as expected in a program, which corresponds to all individuals in a population having some sickness. A microarchitecture gathers source code constructs activated by the feature in the given scenario, similar to the symptoms common to all individuals suffering from a particular disease under given environmental conditions. It links execution events related to a feature of interest in a given scenario.

The essential difference between epidemiology and feature identification is that in the latter, we cannot distinguish feature-relevant and feature-irrelevant events with one unique trace alone. We need multiple traces from different scenarios and exercising different features to identify feature-relevant events. Feature identification corresponds to an extreme case in which epidemiologists encounter a new species and, therefore, lack the immediate ability to distinguish sick from healthy individuals. Consequently, epidemiologists must categorize individuals according to common distinguishing characteristics and subject individuals in each category to different environmental conditions in order to study their characteristics and identify sick individuals and their diseases.

Let $\mathcal{F}'$ ($\mathcal{F}$, respectively) be a set of scenarios (not) exercising a feature of interest. We classify events as relevant or irrelevant with respect to the many *possible* different features intentionally or *accidentally* exercised in the sets of scenarios $\mathcal{F}'$ and $\mathcal{F}$. Thus, we suppose that $\mathcal{F}'$ and $\mathcal{F}$ are chosen carefully: $\mathcal{F}'$ exercises one and only one feature of interest and $\mathcal{F}$ exercises any feature *but* the feature of interest. (The choice of $\mathcal{F}'$ and $\mathcal{F}$ is beyond the scope of this work; we exemplify the importance of this choice in the case studies in Part 3.) Exercising scenarios in $\mathcal{F}'$ produces a class $\mathcal{C}_{\mathcal{I}'}$, a set of intervals $\mathcal{I}'_i$ containing events relevant to the feature. Scenarios in $\mathcal{F}$ produce $\mathcal{C}_{\mathcal{I}}$, a class of intervals $\mathcal{I}_j$ containing events irrelevant to the feature. Intervals $\mathcal{I}'_i$ ($\mathcal{I}_j$, respectively) are marked as relevant (irrelevant, respectively) during trace collection; however, they may contain irrelevant (relevant, respectively) events due to noise.

We regard an event e_k as an instance of a disease because the appearance of e_k in an interval $\mathcal{I}'_i$ or $\mathcal{I}_j$ depends on the environmental condition in the region of the trace corresponding to the particular interval. As in epidemiology, we want to compute the ratio of prevalences of event e_k between $\mathcal{I}'_i$ and $\mathcal{I}_j$ to test if an event e_k is statistically more frequent in intervals $\mathcal{I}'_i$ than in intervals $\mathcal{I}_j$. We reformulate our test on the frequencies of events e_k in classes $\mathcal{C}_{\mathcal{I}'}$ and $\mathcal{C}_{\mathcal{I}}$: If $N_{\mathcal{C}_{\mathcal{I}'}}(e_k)$ ($N_{\mathcal{C}_{\mathcal{I}}}(e_k)$, respectively) is the number of times an

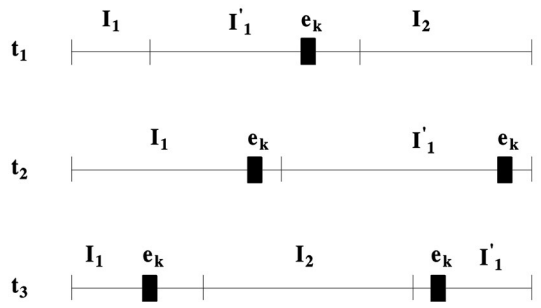

Fig. 2. Use of intervals versus classes.

event e_k appears in $\mathcal{C}_{\mathcal{I}'}$ (in $\mathcal{C}_{\mathcal{I}}$) and $N_{\mathcal{C}_{\mathcal{I}'}}$ ($N_{\mathcal{C}_{\mathcal{I}}}$) is the overall number of events in $\mathcal{C}_{\mathcal{I}'}$ (in $\mathcal{C}_{\mathcal{I}}$), then the frequency of e_k in $\mathcal{C}_{\mathcal{I}'}$ is

$$f_{\mathcal{C}_{\mathcal{I}'}}(e_k) = \frac{N_{\mathcal{C}_{\mathcal{I}}}(e_k)}{N_{\mathcal{C}_{\mathcal{I}'}}},$$

its frequency in $\mathcal{C}_{\mathcal{I}}$ is

$$f_{\mathcal{C}_{\mathcal{I}}}(e_k) = \frac{N_{\mathcal{C}_{\mathcal{I}}}(e_k)}{N_{\mathcal{C}_{\mathcal{I}}}},$$

and the null hypothesis H_0 states that the frequencies $f_{\mathcal{C}_{\mathcal{I}'}}(e_k)$ and $f_{\mathcal{C}_{\mathcal{I}}}(e_k)$ are equal. In other words, e_k belongs to $\mathcal{C}_{\mathcal{I}'}$ or $\mathcal{C}_{\mathcal{I}}$ with the same frequency. However, we observe that very often several scenario executions are available. Thus, we reformulate the hypothesis testing in terms of the intervals in $\mathcal{C}_{\mathcal{I}'}$ and $\mathcal{C}_{\mathcal{I}}$ to better exploit the data available in these intervals.

We test the null hypothesis by testing e_k proportions in the samples corresponding to the intervals $\mathcal{I}'_i \in \mathcal{C}_{\mathcal{I}'}$ and to $\mathcal{I}_j \in \mathcal{C}_{\mathcal{I}}$. Intuitively, the more often we reject H_0, the more likely e_k contributes to the realization of a feature of interest. If we denote with $m = |\mathcal{C}_{\mathcal{I}'}|$ and $n = |\mathcal{C}_{\mathcal{I}}|$ the cardinalities of the two classes, then for each event e_k, we perform $m \times n$ proportion tests. Ideally, for any pair of intervals $\mathcal{I}'_i$ and $\mathcal{I}_j$, we would reject the null hypothesis H_0 that e_k frequency is equal in the two intervals.

We perform tests on pairs of intervals $\mathcal{I}'_i$ and $\mathcal{I}_j$, not on the classes $\mathcal{C}_{\mathcal{I}'}$ and $\mathcal{C}_{\mathcal{I}}$, for greater accuracy. For example, Fig. 2 shows a set of three traces, divided into three intervals in $\mathcal{C}_{\mathcal{I}'}$ and four intervals in $\mathcal{C}_{\mathcal{I}}$. In $\mathcal{C}_{\mathcal{I}'}$, the relevant event e_k appears three times, while in $\mathcal{C}_{\mathcal{I}}$ it appears twice. We could say that e_k appears two times out of three in $\mathcal{C}_{\mathcal{I}'}$ with respect to $\mathcal{C}_{\mathcal{I}}$ and we would conclude that e_k is irrelevant. However, e_k appears in every interval $\mathcal{I}'_i \in \mathcal{C}_{\mathcal{I}'}$, while it appears only in two intervals $\mathcal{I}_j \in \mathcal{C}_{\mathcal{I}}$ over four when comparing pairs of intervals. Thus, we reject H_0 nine times out of 15, with $m \times n = 3 \times 5 = 15$, and classify e_k as relevant.

Proportion testing [12] assumes that the two samples are independent. We build $\mathcal{I}'_i$ and $\mathcal{I}_j$ from the same traces, which could prevent independence. However, we observe that the environmental conditions leading to intervals $\mathcal{I}'_i$ and $\mathcal{I}_j$ are different because they belong either to different scenarios or to a different part of the trace of a same scenario. Moreover, in the absence of noise, the condition when exercising a scenario in $\mathcal{F}'$ does not hold for sets $\mathcal{I}_j$ (similarly, the condition for a scenario in $\mathcal{F}$ does not hold for sets $\mathcal{I}'_i$). Thus, we consider classes $\mathcal{C}_{\mathcal{I}'}$ and $\mathcal{C}_{\mathcal{I}}$ independent.

For a given pair of intervals $\mathcal{I}'_i$ and $\mathcal{I}_j$, we compute the frequency of an event e_k in an interval $\mathcal{I}$ (either $\mathcal{I}'_i$ or $\mathcal{I}_j$) as

$$f(e_k) = \frac{N_{\mathcal{I}}(e_k)}{N_{\mathcal{I}}},$$

where $N_{\mathcal{I}}(e_k)$ is the number of times e_k appears in $\mathcal{I}$ and $N_{\mathcal{I}}$ is the overall number of events in $\mathcal{I}$. The frequency indicates the importance of an event with respect to an interval. It is a measure of the prevalence of this event with respect to the interval. For a selected significance level α, we determine the critical region through a two-sample proportion test by requiring that the test statistic z be greater than the critical value z_α in a single-tailed test [12]. Clearly, not all tests for every pair of intervals $\mathcal{I}'_i$ and $\mathcal{I}_j$ and event e_k reject H_0. We support our expectation that the frequency of e_k is higher in interval $\mathcal{I}'_i$ by carrying out a two-step process.

First, we define a function Ψ

$$\Psi_\alpha(e_k, \mathcal{I}'_i, \mathcal{I}_j) = \begin{cases} 1 & \text{if } H_0 \text{ is rejected} \\ 0 & \text{otherwise,} \end{cases}$$

which, for a given significance level α, returns 1 if and only if the null hypothesis is rejected. We count the number of successes in the $m \times n$ tests with

$$\mathcal{S}_\alpha(e_k) = \sum_{i=1}^{m} \sum_{j=1}^{n} \Psi_\alpha(e_k, \mathcal{I}'_i, \mathcal{I}_j).$$

The closer $\mathcal{S}_\alpha(e_k)$ is to $m \times n$, the higher the likelihood that e_k contributes to the feature.

Second, we decide if e_k is relevant to a feature of interest using a simple voter $v_\theta(\mathcal{S}_\alpha(e_k))$:

$$v_\theta(\mathcal{S}_\alpha(e_k)) = \begin{cases} 1 & \text{if } \mathcal{S}_\alpha(e_k) \geq \theta \\ 0 & \text{otherwise.} \end{cases} \quad (1)$$

For values of θ close to $m \times n$, the voter selects only events that are likely to be relevant to the feature of interest. Values α and θ control the size of the set of events classified as relevant. The smaller the α (e.g., 1.00 percent) and the higher the θ (e.g., $m \times n - 1$), the smaller the set of relevant events.

Using the epidemiological metaphor, we classify all events in $\mathcal{C}_{\mathcal{I}'}$ and $\mathcal{C}_{\mathcal{I}}$ as relevant or irrelevant. We use a relevance index to rank and associate the most relevant events with a feature of interest.

2.2.2 Probabilistic Ranking

In our previous work [4], we introduce a relevance index that is a *renormalization* of Wilde's equation [9] where events are reweighted by the sizes of the classes to make them directly comparable. We summarize this relevance index for completeness.

Wilde proposes the relevance index $p_c(e_k)$, with

$$p_c(e_k) = \frac{N_{\mathcal{C}_{\mathcal{I}'}}(e_k)}{N_{\mathcal{C}_{\mathcal{I}'}}(e_k) + N_{\mathcal{C}_{\mathcal{I}}}(e_k)}, \quad (2)$$

where $N_{\mathcal{C}_{\mathcal{I}'}}(e_k)$ is the number of times e_k appears while executing scenarios in $\mathcal{F}'$ (i.e., counting events contained in class $\mathcal{C}_{\mathcal{I}'}$) and $N_{\mathcal{C}_{\mathcal{I}}}(e_k)$ is the number of times the same event is encountered while executing $\mathcal{F}$ scenarios (i.e., counting events contained in class $\mathcal{C}_{\mathcal{I}}$). Clearly, p_c ranges between 0 and 1. It is 1, or very close to 1, for any feature-relevant event and 0 for irrelevant events.

We modify (2) to weigh the sizes of the different intervals and build from the previous equations the relevance index

$$r(e_i) = \frac{f_{\mathcal{C}_{\mathcal{I}'}}(e_i)}{f_{\mathcal{C}_{\mathcal{I}'}}(e_i) + f_{\mathcal{C}_{\mathcal{I}}}(e_i)}. \quad (3)$$

We inject into (3) the events classified as relevant by the voter $v_\theta(\mathcal{S}_\alpha(e_k))$ in (1) and, then, rank these events with respect to the overall number of possibly relevant or irrelevant events in $\mathcal{C}_{\mathcal{I}'}$ and $\mathcal{C}_{\mathcal{I}}$. Since the number of relevant events might be too large to build the microarchitecture associated to the feature of interest and to be of any help to maintainers, we filter ranked relevant events and keep only those with a higher relevance.

2.2.3 Filtering Relevant Events

We use (4) and a positive threshold t to reduce the set of feature-relevant events:

$$\mathcal{E}'_t = \{e_i | r(e_i) \geq t\}. \quad (4)$$

The size of $\mathcal{E}'_t$ depends on the threshold t. A threshold of 100 percent means that we keep events with 100 percent of relevance with respect to the epidemiological metaphor and the relevance index. Equation (4) limits the number of events that maintainers must consider as relevant to a feature of interest. We use this subset of ranked relevant events to build a microarchitecture representing the feature of interest, which can be displayed against the program architecture and compared to other microarchitectures to help maintainers precisely locate responsibilities and differences.

2.3 Data Modeling and Feature Comparison

The last two activities in the feature identification and comparison process use the subset of filtered relevant events to build and compare models of features in order to highlight architectural and behavioral differences in features and their implementation.

2.3.1 Feature Model Creation

We use the program architectural model and the events in $\mathcal{E}'_t$ to create new models of the program that include only the variables, classes, functions, and methods activated when exercising some scenarios. Such models represent "slices" of the program and are microarchitectures highlighting the constituents that implement a feature.

We create these microarchitectures by cloning the program architectural model and by removing from this model, by means of model transformations, all variables, classes, functions, and methods that are not explicitly (directly or indirectly) called when exercising a feature.

We can potentially create an infinite number of microarchitectures through the execution of different features of the program for different scenarios. We can also create microarchitectures of various *widths*: A microarchitecture may include only the variables, classes, functions, and methods exercised, or it may also include those that are related, directly or indirectly. Thus, a microarchitecture may be small or expanded, as desired, up to the point where it is equal to the complete program architecture.

PADL offers facilities to clone and visit models using the Prototype and Visitor design patterns. We use these facilities to build a Visitor dedicated to the construction of microarchitectures representing features from a program architectural model and from subset $\mathcal{E}'_t$.

2.3.2 Feature Comparison

We compare and highlight differences among microarchitectures so that maintainers can understand and compare the behavior of different features or of the same feature with different scenarios or across different versions. We can use set intersection to compare microarchitectures because these are built using sets of filtered ranked-relevant events, in which disorder and imprecision have been dealt with by processor emulation and the epidemiological metaphor.

We use model transformation techniques to highlight differences among microarchitectures. From a given "origin" microarchitecture, we compute the set of model transformations required to transform it into another "destination" microarchitecture. We use this set of transformations to add to the "origin" microarchitecture variables, classes, functions, and methods missing with respect to the "destination" microarchitecture and to distinguish in the "origin" microarchitecture variables, classes, functions, and methods not included in the "destination" microarchitecture. The modifications in the "origin" microarchitecture are described using specific entities of the metamodel to highlight differences between microarchitectures visually.

Since microarchitectures are also models of the PADL metamodel, we develop a Visitor to compare microarchitectures among themselves. Feature identification and comparison are independent of the order in which we store variables, classes, functions, and methods.

3 Case Studies

The goal of our feature identification and comparison process is to assist program understanding tasks in large multithreaded object-oriented programs by identifying the microarchitectures implementing some features of interest and by highlighting the variables, classes, functions, and methods activated when exercising a feature. We use the following three case studies to assess the usefulness of our process:

1. We mimic the program understanding task of identifying the microarchitecture implementing the feature "save a bookmark" of MOZILLA [13]. We compare our approach with those presented in previous works.
2. We compare the implementation of the previous feature "save a bookmark" in FIREFOX [14] and in MOZILLA to assess the similarities and the differences between features of these related programs.
3. We reproduce the previous case studies with two Web browsers in C and in Java: CHIMERA [15] and ICEBROWSER [16], to support the generalizability of our process and of the epidemiological metaphor. We perform other feature identification tasks with two graphical editors, JHOTDRAW [17] in Java and XFIG [18] in C, to further strengthen the generalizability.

We did not have prior knowledge of the implementations of the programs used in the three case studies. In each case study, we restrict feature identification to classes, functions, and methods because of limitations in the technical capabilities of CALLGRIND. Events collected through CALLGRIND are function and methods calls, from which we infer exercised classes using the program architectural model.

TABLE 2
Mozilla v1.5.1 Sizes

	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)		
rdf files	185	(0.02)		

3.1 Case Study 1

In this first case study, we compare our process of feature identification with previous works by Eisenbarth et al. [3], and add a comparison of the accuracy to our previous approach [4] and Wilde's [9].

3.1.1 Setup

MOZILLA is an open-source Web browser ported on almost all software and hardware platforms. It is sufficiently large and mature to represent a real-world program. Its size ranges in the millions of lines of code (MLOC). It is developed mostly in C++, with C code accounting for only a small fraction of the program. We do not include in our study Java, IDL, XML, HTML, and configuration language and support. MOZILLA version 1.5.1 includes more than 14,000 source files for a size of up to 4.40 MLOC decomposing into about 3,060 subdirectories.

Table 2 gives an overview of the size of the Web browser. Reported figures are orders of magnitude rather than absolute values. Indeed, several factors influence these figures, such as reverse engineering tools, parsing techniques [19], and certain programming language features. In our case studies, we choose conservative reverse-engineering techniques. We apply strict reverse-engineering rules such that we categorize as classes only entities declared as such according to the C++ syntax. C structures and templates cannot be expressed as AOL entities because AOL does not provide dedicated representations of structures or templates. We map structures and templates to AOL classes annotated with specific comments. We also consider templates mixed with structures as outside of the boundary of the reverse-engineered models and do not recover their attributes, methods, and file locations.

3.1.2 Objective and Hypothesis

We hypothesize that the number of classes in the micro-architecture reported using the new process is smaller than the same number in microarchitectures identified with previous approaches, using the same traces.

One key feature of Web browsers is the ability to store URLs. We are interested in feature F: the variables, classes, functions, and methods activated when a URL is saved. We seek to identify the microarchitecture implementing feature F in MOZILLA. We consider two families of scenarios:

- $\mathcal{F}$, which includes 10 scenarios similar to: Users visit an URL. For example, users open a Web browser, click on a previously bookmarked URL, wait for the page to load, and close the Web browser.
- $\mathcal{F}'$, which includes one scenario where: Users, once the page is loaded, save the URL. For example, users perform the appropriate actions, via mouse and graphic objects interactions, to save the bookmark.

Sequences of actions between $\mathcal{F}'$ and $\mathcal{F}$ are equal except for the bookmarking action in $\mathcal{F}'$. Thus, intuitively, all variables, classes, functions, and methods activated in $\mathcal{F}$ are present in $\mathcal{F}'$, but not vice-versa; the difference corresponds to feature F. We build and compare two microarchitectures: $\mu A_{\mathcal{F}'}$, which corresponds to the scenario in $\mathcal{F}'$, and $\mu A_{\mathcal{F}}$, which corresponds to the scenarios in $\mathcal{F}$, to identify events relevant to feature F.

We assess our hypothesis by comparing the sizes of the microarchitectures implementing feature F when identified first using a naive approach based on the usual `grep` string-matching tool, then using a concept analysis-based technique from the literature [3], and finally using our previous relevance index [4].

TABLE 3
Mozilla v1.5.1 Term Occurrences over 7,411 Files (C, C++, and Header Files)

Terms	Occurrences	Terms	Occurrences
add	3,482	*bookm*	15
store	877	*link*	732
save	639	*ref*	2,632
url	782	*uri*	3,914

TABLE 4
Typical Sizes of $\mathcal{I}$ and $\mathcal{I}'$ when Exercising $\mathcal{F}'$ and $\mathcal{F}$ in Mozilla v1.5.1 with Valgrind

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

3.1.3 Results with a Naive Approach

Naive maintainers would attempt to perform the understanding task by searching files with tools supporting strings and regular expressions matching, such as the Unix utility `grep`. For example, they would search for files containing the term *add* or synonyms such as *store* and *save*. Several combinations in conjunction or disjunction with synonyms and abbreviations to the term bookmark are possible (e.g., *url*, *uri*—Universal Resource Identification, *bookm*, *link*, *ref*). Data in Table 3 shows that only a conjunction of terms reduces the number of files to be inspected to identify the feature responsible for saving a bookmark. However, these matches are string-based at the file level. It is impossible to automatically distinguish a meaningful match from unwanted matches.

3.1.4 Results with Concept Analysis

Table 4 reports typical sizes of $\mathcal{C}_{\mathcal{I}'}$ and $\mathcal{C}_{\mathcal{I}}$ in number of events as collected with VALGRIND. The focus of the

TABLE 5
Filtered Sizes of $\mathcal{I}$ and $\mathcal{I}'$ for Mozilla v1.5.1

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

TABLE 6
Sizes of $\mathcal{E}'_t$ with the Above $\mathcal{I}'$ for Mozilla v1.5.1

t	$\mathcal{E}'_t$ sizes (*i.e.*, number of methods)		
	Relevance index		Epidemiological metaphor (using Equation 1)
	Wilde's Equation 2	Our Equation 3	
100.00%	274	274	272
99.90%	274	295	272
99.00%	274	2,273	274
98.00%	274	2,626	281
97.00%	274	2,697	282
96.00%	274	2,760	291
95.00%	274	2,796	291
90.00%	274	2,796	304
85.00%	274	2,796	310
50.00%	515	2,796	310

program understanding task is to identify classes and methods which characterize feature F, which are part of the 36,105,209 $\mathcal{C}_{\mathcal{I}'}$ method calls. We create a formal context $C = (O, A, R)$ for $\mathcal{F}'$ and $\mathcal{F}$ following the approach presented in [3]. We consider $\mathcal{C}_{\mathcal{I}'}$ and $\mathcal{C}_{\mathcal{I}}$ for $\mathcal{F}'$ and $\mathcal{F}$ as four subscenarios and A, the set of attributes, contains four symbols to represent these subscenarios. O, the set of objects, contains all *distinct* methods activated when exercising $\mathcal{F}'$ and $\mathcal{F}$. There are about 30,000 distinct methods present in $\mathcal{C}_{\mathcal{I}'}$ and $\mathcal{C}_{\mathcal{I}}$. R is a binary relation; a pair (o, a) is in R if the method $m \in O$ is called when the subscenario $s \in A$ is performed. The resulting lattice contains 11 concepts (excluding top and bottom); concepts range from a minimum of 13,325 methods to a maximum of 26,613. We identify manually, in the lattice, concepts corresponding to the $\mathcal{I}'_i$ and $\mathcal{I}_j$ intervals of $\mathcal{F}'$ and $\mathcal{F}$. We compute set difference between these concepts only and retain 1,038 candidate methods to feature F. Concept analysis is a powerful tool, but results need a manual and tedious inspection to focus feature identification. It narrows features better than a naive approach, but suffers from the problem of set difference.

3.1.5 Results with Probabilistic Ranking

In a previous work [4], we used knowledge-based filtering and relevance index, as reviewed in Section 2.1. First, we use events in the $\mathcal{C}_{\mathcal{I}'}$ and $\mathcal{C}_{\mathcal{I}}$ classes for $\mathcal{F}'$ and $\mathcal{F}$ (reported in Table 4) to define a knowledge-based filter and thus reduce noise. Table 5 reports the data in Table 4 with filtered method calls. Second, we use this data to create a ranked list of methods according to (2) and (3). Both

TABLE 7
Classes Exercised in Feature F ($\mathcal{E}_1^*$ in Fig. 6) when Comparing $\mu A_{\mathcal{F}'}$ with Respect to $\mu A_{\mathcal{F}}$

Exercised Classes	With our relevance index from Equation 3	With the epidemiological metaphor using Equation 1
In $\mu A_{\mathcal{F}'}$, not in $\mu A_{\mathcal{F}}$	10	44
In $\mu A_{\mathcal{F}}$, not in $\mu A_{\mathcal{F}'}$	3	330
With differing methods calls	73	20

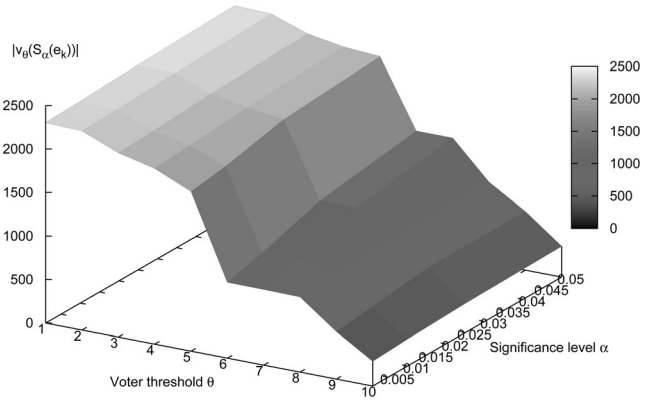

Fig. 3. Number of methods classified as relevant with the epidemiological metaphor in function of α and θ.

equations highlight 274 relevant events, i.e., methods that are particular to $\mathcal{F}'$, with $\mathcal{E}_1^*$. These methods belong to about 80 different classes. Classes and methods counted in Table 7 form a microarchitecture implementing feature F and program comprehension is limited to this. Unlike the naive approach, probabilistic ranking makes it possible to classify classes and methods contributing to a microarchitecture implementing a feature. The 274 methods belong to the set of 1,038 methods identified by concept analysis.

3.1.6 Results with Epidemiological Metaphor

We use the epidemiological metaphor to classify events as relevant or irrelevant to feature F before ranking relevant events with the relevance index exemplified in the previous subsection.

With a significance level $\alpha = 0.01$ and a threshold $\theta = 10$ for the number of rejected tests, the voter in (1) classifies 310 methods as relevant to feature F, with t equals 50 percent. These 310 relevant methods should be compared with the 2,796 methods from our (3) and with the 515 methods from Wilde's (2).

We rank these 310 methods with the relevance index in (3) and identify 272 methods with an index of 100.00 percent. Table 6 shows the sizes of $\mathcal{E}_t^*$ when ranking methods with different threshold values. The use of the epidemiological metaphor dramatically reduces the set of relevant events from 0.73 percent when $t = 100\%$ with respect to (2) and (3) to 40 percent when $t = 50\%$ with respect to (2) and to 88.95 percent when $t = 50\%$ with respect to (3). Fig. 3 shows the variation of the number of methods classified as relevant with the epidemiological metaphor in function of α and θ, when exercising the one scenario in $\mathcal{F}'$ and the scenarios in $\mathcal{F}$. The number of relevant events decreases as θ closes in on $m \times n$ (1×10). This number also decreases as we reduce the size α of the critical region.

TABLE 8
Firefox 1.0.7 Sizes

	Numbers	(MLOC)		Numbers
Header files	5,147	(0.87)	Classes	5,438
C files	1,503	(0.90)	Methods	57,748
C++ files	4,297	(1.98)	Specialisations	4,667
IDL files	1,394	(0.13)	Associations	10,397
XML files	330	(0.11)	Aggregations	5,035
HTML files	2,239	(0.18)		
Java files	62	(0.07)		
rdf files	228	(0.11)		

The microarchitecture $\mu A_{\mathcal{F}'}$ associated with feature F, when exercising $\mathcal{F}'$, contains 66 classes defining the 272 methods with $t = 100\%$. The microarchitecture $\mu A_{\mathcal{F}}$ created when exercising $\mathcal{F}$ contains 352 classes. The comparison of the two microarchitectures reveals that $\mu A_{\mathcal{F}'}$ contains 44 classes not present in $\mu A_{\mathcal{F}}$, while $\mu A_{\mathcal{F}}$ contains 330 classes absent from $\mu A_{\mathcal{F}'}$, as reported in Table 7 $(352 - 330 + 44 = 66)$. Maintainers do not need to consider the 330 absent classes because they do not play a role in feature F.

A method-level comparison reveals that 127 methods in $\mu A_{\mathcal{F}}$ are not called in $\mu A_{\mathcal{F}'}$ and that there are 58 methods called in $\mu A_{\mathcal{F}'}$ with respect to $\mu A_{\mathcal{F}}$. A deeper analysis shows that 20 classes are exercised differently. Thus, maintainers would manually analyze either the 44 classes in $\mu A'_{\mathcal{F}}$ (and not in $\mu A_{\mathcal{F}}$) or the 20 classes exercised differently.

3.1.7 Conclusion

The use of the epidemiological metaphor dramatically decreases the number of methods in comparison to previous works and thus the size of the microarchitecture implementing feature F to be analyzed by maintainers. In the next case study, we show that our approach is also useful in comparing the implementation of feature F, called $\mathrm{F_{Mozilla}}$, in FIREFOX, a different, but related, Web browser.

3.2 Case Study 2

The second case study consists of comparing the microarchitecture associated with feature F in the FIREFOX and MOZILLA Web browsers.

3.2.1 Setup

FIREFOX is the next generation of Web browser built by the MOZILLA foundation based on its experience with MOZILLA. It is ported on almost all known software and hardware platforms and is also large enough to represent real-world programs. FIREFOX's size ranges in the millions of lines of code (MLOC). It is also mostly developed in C++. The considered version, 1.0.7, includes more than 10,000 source files for a size of up to 3.75 MLOC decomposing in about 3,000 subdirectories. Table 8 gives an overview of the size of the Web browser. The restrictions regarding the technique used to reverse-engineer MOZILLA also apply to FIREFOX.

3.2.2 Objective and Hypothesis

We assess the usefulness of our approach when comparing feature F in the Web browsers FIREFOX and MOZILLA. We hypothesize that our approach allows for identifying the classes involved in feature F in both Web browsers.

Tables 2 and 8 summarize data on the two Web browsers but do not report the differences between the two Web browsers. FIREFOX contains 1,173 classes that are not present in MOZILLA, which in turn contains 1,758 that do not appear in FIREFOX. The differences between the Web browsers are actually even greater because we compute these using string matching on class names and do not include class and method semantics. In addition, differences are also likely to be greater between the Web browsers as FIREFOX continues to be actively developed by its community while MOZILLA is in maintenance.

For this case study, we apply the families of scenarios $\mathcal{F}'$ and $\mathcal{F}$, from the previous case study, to the FIREFOX Web browser. We obtain a feature $\mathrm{F_{Firefox}}$ which we compare to the feature $\mathrm{F_{Mozilla}}$ identified in the previous case study. We expect many differences between features $\mathrm{F_{Firefox}}$ and $\mathrm{F_{Mozilla}}$, because of the many differences between the two Web browsers.

3.2.3 Results

Feature $\mathrm{F_{Firefox}}$ includes 227 methods for 83 classes. Feature $\mathrm{F_{Mozilla}}$ includes 310 methods for 87 classes. Among these classes and methods, 264 methods (and 67 classes) are present when exercising MOZILLA but not FIREFOX and 208 methods (and 63 classes) vice-versa. Fig. 5 shows some of the differences between features $\mathrm{F_{Firefox}}$ and $\mathrm{F_{Mozilla}}$. Some classes exist in FIREFOX but not in MOZILLA, while some exist in both. In addition, Web browsers perform different method calls.

We perform a deeper analysis of the differences between features $\mathrm{F_{Firefox}}$ and $\mathrm{F_{Mozilla}}$. Fig. 6 shows a summary of the classes and methods involved in saving a bookmark in FIREFOX and in MOZILLA and their relationships. The same class `nsBookmarksService` is the main actor in both features, but it has been refactored in FIREFOX to reduce the length of the call chain to save a bookmark (three method calls versus one). In MOZILLA, `AddBookmarkImmediately()` calls `CreateBookmarkInContainer()`, which calls `CreateBookmark()`, while in FIREFOX there is a unique call to `CreateBookmark()`. Thus, it seems that the developers of FIREFOX grouped the feature F into one method.

A further manual comparison of the classes `nsBookmarksService` in the two Web browsers shows that all methods in this class for MOZILLA still exist in FIREFOX, even though they are no longer used. The `nsIBookmarksService` IDL interface even specifies that some available method skeleton should be removed. Developers state in file `nsIBookmarksService.idl` of directory `browser/components/bookmarks/public`, line 113, "xxxpch: to be removed." immediately before the IDL declaration of method signature `addBookmarkImmediately()`. A comment in line 1305 in file `nsBookmarksService.cpp` further states that: "xxxpch: useless caller of AddBookmarkImmediately..." just before the method skeleton `AddBookmarkImmediately`. We conclude that the implementations of feature F in FIREFOX and MOZILLA are similar, but that in FIREFOX, developers have performed some unfinished refactorings.

3.2.4 Conclusion

Our approach allows for the identification and comparison of the microarchitectures implementing the feature F in FIREFOX and MOZILLA and, thus, helps maintainers to

understand the architecture and behavior of the two programs with respect to each other. With our approach to feature identification, maintainers can gain much insight into the implementation of FIREFOX.

3.3 Case Study 3

The third case study concerns the generalization of our approach. We developed our process to identify features in large, multithreaded C++ programs. The process and the metaphor, however, also apply to other paradigms and object-oriented programming languages. We briefly present four examples, using both the previous feature F and a new feature, to illustrate the generalizability of our approach to C and Java programs. The new feature G concerns the "draw circle" capability of the graphic programs JHOTDRAW and XFIG.

3.3.1 Feature F and C

CHIMERA [15] is a Web browser dating back to the early 1990s. Written by John Kilburg and others, it is an X/ATHENA Web client for UNIX-based workstations. It is entirely developed in C. Its 2.0a19 version contains about 38 thousands of lines of code (KLOC), organized in 75 C files and 38 header files. Its compiled executable contains 413 functions. Despite its minimalist approach, CHIMERA implements the core features found in full-fledged Web browsers, including feature F.

We replicate the task of identifying feature F in CHIMERA to support the generalizability of our approach. We use the following scenarios: "open and close CHIMERA," "access a link through the bookmarks," and "save a bookmark." We exercise the feature-relevant scenario "save a bookmark" once, while we exercise the other scenarios twice.

Of the 413 functions in CHIMERA compiled executable, our approach retains 54, of which 12 are ranked as relevant to feature F at 100 percent. The semantics of the names of the functions draws attention to three functions immediately: `BMDAddMark()`, `BookmarkAdd()`, and `BMWrite()`. Caller-callee relationships show a chain of function calls in which `BMDAddMark()` calls `BookmarkAdd()`, which, in turn, relies on `BMWrite()`. Another existing function, `BMCreate()` is not part of the functions contributing to feature F in CHIMERA because it also participates in the "access a link through the bookmarks" feature and is thus filtered out. Our approach therefore precisely identifies the functions implementing feature F in a C program.

3.3.2 Feature F and Java

ICEBROWSER [16] is a Web browser produced by the company Icesoft. It is entirely written in Java and divides into three reference implementations using either AWT, an extended version of AWT, or SWING as graphic libraries. The considered version 6.1.2 of the enhanced AWT reference implementation, like CHIMERA, provides all the features found in Web browsers such as MOZILLA. In particular, it provides a "save a bookmark" feature. This version contains 381 classes, 168 interfaces, 3195 fields, and 6,777 methods (excluding standard Java libraries) for about 57,000 KLOC, when including proprietary libraries.

Again, we replicate the task of identifying feature F in ICEBROWSER. We use the following scenarios: "open and close ICEBROWSER," "access a link through the bookmarks," and "save a bookmark." We exercise the feature-relevant scenario "save a bookmark" once, while we exercise the other scenarios twice.

Without any knowledge-based filtering, our approach retains 111 methods, one order of magnitude less than the number of methods in the implementation, out of which 50 are ranked with a relevance index of 100 percent, belonging to nine different classes. Among the nine classes, four belong to the implementation of the Web browser per se, while the other five belong to the standard Java class libraries (for example, class `java.util.List`). The four classes in the implementation declare 12 methods among the 50 ranked 100 percent, of which eight implement feature F directly: the constructors `Bookmark()` and `BookmarksDialog()` and the methods `actionPerformed()`, `finished()`, `init()`, `restoreBookmarks()`, `saveBookmarks()`, and `updateChanges()`, of the class `BookmarksDialog`. Therefore, our approach is also useful with small-size Java programs to identify the classes and methods implementing a feature of interest.

3.3.3 Feature G in C

XFIG [18] is a menu-driven graphic tool to draw and manipulate graphical objects interactively, under the X Window system. It is written entirely in C. The 3.2.4 version of XFIG contains about 90 KLOC, divided into 109 C files and 83 header files. Static code analysis reports 1,937 functions, of which 1,911 are linked into the executable binary.

We apply our approach to identify the functions implementing feature G "draw a circle." We execute the following two feature-relevant scenarios. In the first scenario, we draw a circle, save a file, and exit. In the second scenario, we reproduce the previous scenario but draw two circles. Feature irrelevant scenarios are "open and close XFIG" and "open XFIG, draw a rectangle, save, and exit."

When applying our approach and its epidemiological metaphor, 20 functions are retained, out of which 18 are ranked as 100 percent relevant. The semantics of the function names and a study of the execution traces using a debugger support that the following seven identified functions implement feature G: `add_ellipse()`, `circle_ellipse_by-radius_drawing_selected()`, `create_circleby-rad()`, `create_ellipse()`, `draw_ellipse()`, `init_circlebyradius_drawing()`, and `write_ellipse()`. Thus, we show that we can apply our approach successfully to another paradigm, program application, and feature.

3.3.4 Feature G in Java

As in XFIG, JHOTDRAW is a graphical tool with an advanced user interface to draw and manipulate graphical objects. It is written in Java and its 5.1 version contains 136 classes, 19 interfaces, 362 fields, and 1,380 methods for a total of 4,582 KLOC (excluding standard Java class libraries).

As with XFIG, we apply our approach to identify the classes and methods implementing feature G when exercising the scenario "draw a circle" and the feature irrelevant scenarios "open and close JHOTDRAW," "draw a rectangle," and "animate some text."

Of the 1,380 methods, we identify 14 methods as participating in feature G, among which we rank 12 as being 100 percent relevant to the feature. These 12 methods belong to the five classes `AttributeFigure`, `FigureAttributs`, `EllipseFigure`, `AbstractFigure`, and `ColorMap`. A closer study of the implementations of class `EllipseFigure` (subclass of `AttributeFigure`, subclass of `AbstractFigure`) and of its two ranked methods `drawBackground()` and `drawFrame()` concretely high-

lights their role in the implementation of feature G. Our approach thus dramatically reduces the search for the method implementing feature G.

3.3.5 Conclusion

Although CHIMERA, ICEBROWSER, JHOTDRAW, and XFIG are smaller programs than FIREFOX and MOZILLA, Case Study 3 still supports the generalizability of our approach and of the epidemiological metaphor. This shows that our approach can be applied to other object-oriented programming languages as well as to procedural paradigms to substantially narrow the number of classes, methods, and functions to be inspected by maintainers.

4 DISCUSSION

The actual number of different classes in C++ programs depends on how templates are counted. Since we consider `nsCOMPtr<nsIRollupListener>` as consisting of two classes, `nsCOMPtr` and `nsIRollupListener`, we count class `nsCOMPtr` twice.

The use of the epidemiological metaphor and of the relevance index limits the impact of the precision of the tools collecting the data dynamically: A less precise tool leads to a greater number of variables, classes, functions, and methods, but this number is compensated by our approach.

The number of classes and methods that a maintainer must inspect is further substantially reduced if the inspection process is driven by the semantics conveyed by the names of classes, functions, and methods, as exemplified in the case studies. For example, if we apply a strategy similar to the naive approach to events relevant to feature F, the number of classes to be inspected is narrowed dramatically. Only one class with modified behavior is highlighted, class `nsBookmarksService`, belonging to $\mathcal{F}'$, as well as the methods `AddBookmarkImmediately()`, `CreateBookmark()`, `CreateBookmarkInContainer()`, `InsertResource()`, and `getFolderViaHint()`.

The complexity of this approach depends upon the size of sets $\mathcal{I}'_i$ and $\mathcal{I}_j$ and the number of these sets in classes $\mathcal{C}_{\mathcal{I}'}$ and $\mathcal{C}_{\mathcal{I}}$. In real cases, only a few sets $\mathcal{I}'_i$ are collected and, typically, more intervals $\mathcal{I}_j$ are available. Thus, building the reduced set of relevant events is linear in the number of $\mathcal{I}_j$ intervals. Equations 2 and 3 compute in linear time with the size of the data and, thus, scalability is not an issue.

We reuse the traces collected in our previous work [4] to build the microarchitectures $\mu A_{\mathcal{F}'}$ and $\mu A_{\mathcal{F}}$. However, the specifications of $\mu A_{\mathcal{F}}$, access a bookmark stored in the bookmark list, are larger than necessary. We observe that events related to accessing a URL by typing it directly in the URL top-bar are not strictly related to the microarchitecture $\mu A_{\mathcal{F}}$. We add one realization of the scenario "accessing a URL by typing" to the scenarios used to build microarchitecture $\mu A_{\mathcal{F}}$ and obtain a microarchitecture with a size of 69 classes, to be compared to the previous 352 classes.

Equation (3), which ranks relevant events, may be reinterpreted more closely to the epidemiological metaphor. As epidemiologists, we may be more interested in the prevalence of a given disease in a subpopulation that already manifests a certain class of problems than we are in the complete at-risk population. This corresponds, for example, to studying the prevalence of lung cancer in the population of people affected by *any type of* cancer.

In this case, we would consider the conditional frequencies computed in relation to relevant events, not the frequencies in relation to the overall classes $\mathcal{C}_{\mathcal{I}'}$ and $\mathcal{C}_{\mathcal{I}}$. Thus, we do not alter the number of relevant events in Table 6 ranked 100 percent but, rather, we better differentiate events not ranked 100 percent relevant. For example, if we use the same threshold 50 percent in Table 6, we retain 278 events, to be compared to the 310 previously ranked relevant events.

Figs. 4 and 6 show the user interface of the PTIDEJ [20] tool suite, which highlights differences between a microarchitecture built from a scenario in $\mathcal{F}$ and the microarchitecture built from the scenario in $\mathcal{F}'$. The epidemiological metaphor allows maintainers to identify differences between features accurately and rapidly. We hypothesize that the effort required to build mental models and abstractions or to verify conjectures on the feature F is alleviated by the accuracy of the microarchitectures representing feature F, built with the epidemiological metaphor.

More theoretical work is required to prove the generalizability of our approach to different programs, programming languages, and paradigms. The three case studies presented in Part 3 (using programs CHIMERA, FIREFOX, ICEBROWSER, JHOTDRAW, MOZILLA, and XFIG) support, but do not prove, the generality of our approach and the epidemiological metaphor. A proof would require defining traces abstractly as well as characterizing the disorder and imprecision caused by the different strategies of trace collection. This is beyond the scope of this study.

5 RELATED WORK

Our research relates to static and dynamic program analysis, feature analysis, and metamodeling and model transformation techniques. The following sections describe related work in each category and compare it to our approach.

5.1 Static and Dynamic Analysis

Program instrumentation via source-to-source translation is a common technique used to collect dynamic data. For example, the work of Ernst et al. focuses on dynamic techniques for discovering invariants in traces [21]. A program is instrumented via the DAIKON front-end by source-to-source translation. For each instrumentation point, the values of all the variables in the scope are saved. DAIKON does not offer a front-end for C++ and does not provide enough data on method calls and variable accesses for method-level feature analysis.

Ball and Larus [22] introduce techniques for program profiling and trace collection at the intraprocedural level, in particular, an efficient intraprocedural path numbering algorithm that collects detailed data at the procedural level. Melski and Reps [23] address interprocedural path profiling by modifying the Ball-Larus algorithm for context-sensitive interprocedural profiling. These approaches based on path numbering may be prohibitive given the theoretically exponential number of interprocedural paths and the impossibility of representing such paths with limited-length machine words. To the best of our knowledge, no empirical data has been reported on the performance (time and space) of these approaches.

Harrold et al. [24] present an empirical analysis of relationships among program spectra, i.e., distributions of program execution paths, and program behavior. They experiment with seven small C programs to analyze the correlation between programs, faulty programs, inputs, and

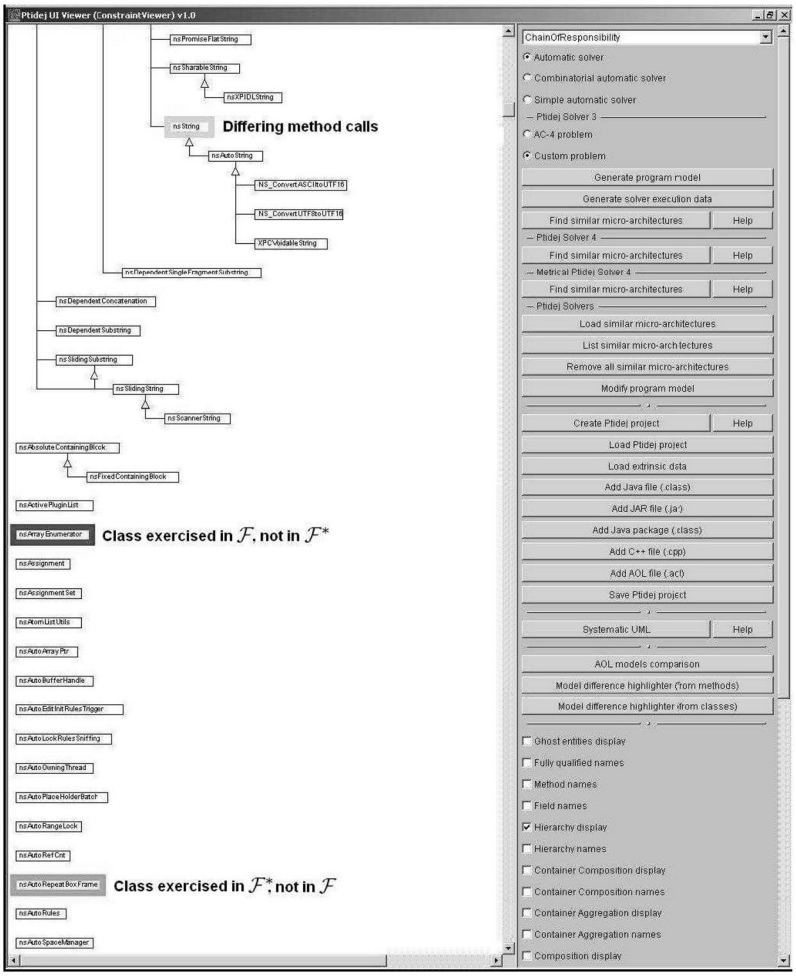

Fig. 4. Example of the visualization of $\mathrm{F}_{\mathrm{Mozilla}}$.

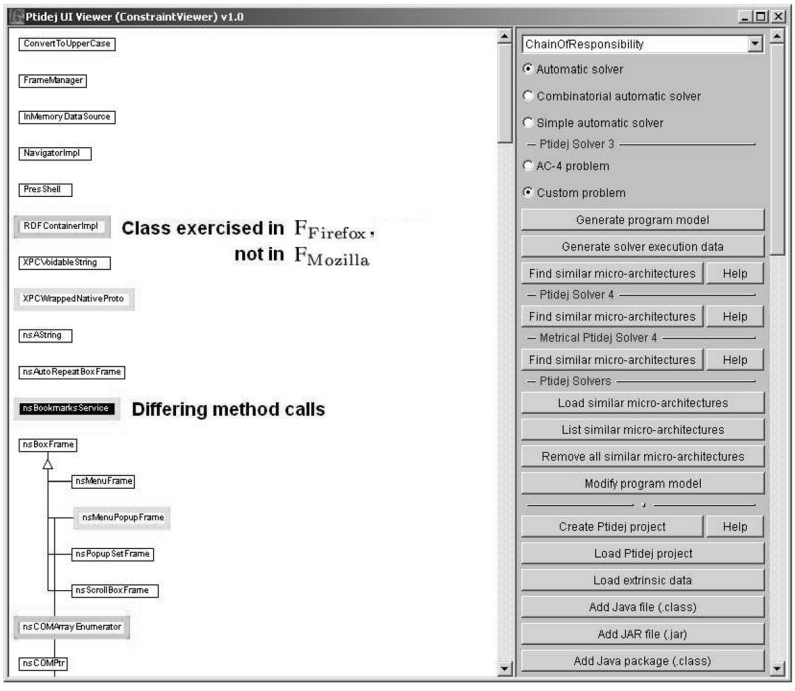

Fig. 5. Excerpt of the comparison of features $\mathrm{F}_{\mathrm{Firefox}}$ and $\mathrm{F}_{\mathrm{Mozilla}}$.

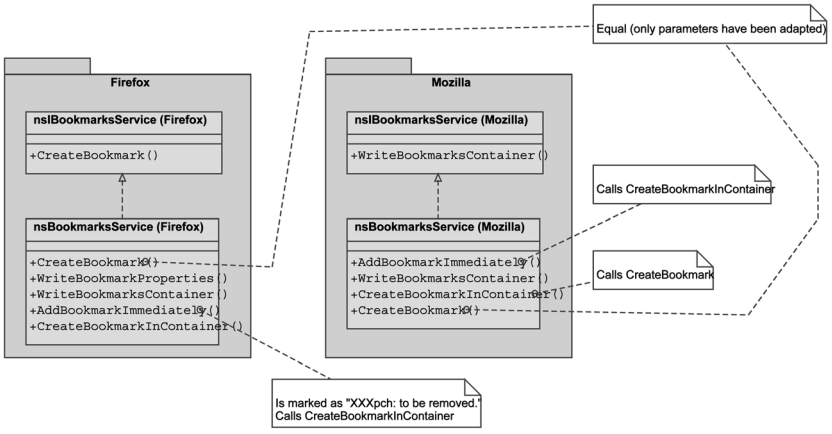

Fig. 6. Summarized architectural differences between FIREFOX and MOZILLA.

program spectra. They conclude that the comparison of program spectra can be used to highlight fault root causes in programs. However, their tools for computing the spectra of programs work for C and Java only and it is still unclear whether program spectra can be efficiently computed on large software.

Jeffery and Auguston [25] present the UFO dynamic analysis framework, which combines a language and a monitor architecture to simplify building new dynamic analysis tools. UFO offers a precise behavioral model, a declarative monitor language, dynamic analyses both at runtime and postmortem, and automatic instrumentation through the use of a virtual machine for the Icon and Unicon programming languages. UFO also works on a subset of C. The main limitations of the UFO framework in the context of our work is the lack of support for C++ and possible performance issues.

Finally, Ernst [26] discusses synergies and dualities between static and dynamic analyses. Reiss and Renieris [27], [28], [29] present an approach to encode dynamic data and explore program traces. They also propose languages for dynamic instrumentation. The problem of handling large amounts of data when performing dynamic analysis is discussed in several works, such as that by Hamou-Lhadj and Lethbridge [30], [31], where the authors present an algorithm for identifying patterns in traces of procedure calls.

Studies on open-source programs are becoming increasingly popular because of the importance and of the quality of these kinds of programs. Recently, Gyimóthy et al. [32] studied several versions of MOZILLA to validate object-oriented metrics for fault prediction. Koru and Tian [33] also related high-change modules and modules with highest measurement values on MOZILLA and OPENOFFICE. However, we have not found any work that combines dynamic and static analyses of large multithreaded C++ programs.

5.2 Feature Identification

Although, researchers have proposed many feature identification approaches, none of these approaches can be applied directly to multithreaded programs.

In their pioneering work, Wilde et al. [6], [9] present an approach to identifying features by analyzing execution traces. They use two sets of test cases to build two execution traces, one where a feature is exercised and another where the feature is not. They compare execution traces to identify the feature associated with the feature in the program. They only use dynamic data from program executions to identify features; they do not perform static analysis of the program. We use a knowledge-based filtering technique for dynamic data and an epidemiological metaphor to classify relevant and irrelevant events for a feature of interest.

Chen and Rajlich [34] developed an approach to identify features using Abstract System Dependencies Graphs (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 features using the ASDG following a precise manual process. In contrast to Wilde et al.'s work, Chen and Rajlich use only static data to identify features and a manual process. We believe that any manual identification process is prohibitive for large multithreaded object-oriented programs.

Eisenbarth et al. [3], [35] combine previous approaches by using both static and dynamic data to identify features. First, the authors perform a dynamic analysis of a program using profiling techniques to identify features, similar to Wilde et al.'s work. They then apply concept analysis techniques to link features together and guide a static analysis that narrows the scope of identified features. They apply their approach to two C programs. However, feature identification is actually performed by means of set operations on concepts, which impedes the identification in the case of multithreading, where relevant and irrelevant events are entangled in traces.

Salah and Mancoridis [36] 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, class-interaction, feature-interaction, and feature-implementation. Each view abstracts preceding views to present only the most relevant data to maintainers. Feature-interaction and feature-implementation views highlight relationships among views. This work was recently extended to allow feature identification and evolution analysis in MOZILLA [37]. However, maintainers cannot use these views to identify or highlight differences among features from different scenarios.

Eisenberg et al. [38] introduce an approach to feature identification using test cases. First, they partition test cases in feature-specific subsets manually and use them to generate traces. Then, they collect traces and rank methods using heuristic criteria. They develop a tool for the Java programming language and apply this to three different programs. They do not discuss multithreaded programs.

Greevy et al. [39] study the evolution of object-oriented software entities from a feature point of view. They analyze

TABLE 9
Comparison of Related Work (Analysis Techniques, Left, and Feature Identification Techniques, Right)

	DAIKON [21]	Ball-Larus [22], [23]	Program Spectra [24]	UFO [25]	Wilde [6], [9]	ASDG [34]	Eisenbarth [35], [3]	Salah [36]	Eisenberg [38]	Greevy [39]	Our Approach
C++	×	×	×	×	✓	✓	×	×	×	×	✓
Method-level Granularity	×	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Dynamic Data	✓	✓	✓	✓	✓	×	✓	✓	✓	✓	✓
Static Data	×	×	×	×	×	✓	✓	✓	×	×	✓
Multi-threading	×	×	×	×	×	×	×	×	×	×	✓
Scalability		✓	×	×	✓	×	×	✓	✓	✓	✓
Knowledge-based Filtering	N/A	N/A	N/A	N/A	✓	×	×	×	×	×	✓
Automatic Identification					✓	×	×	✓	×	✓	✓
Modeling					×	×	×	✓	×	✓	✓
Comparison					×	×	×	×	×	✓	✓

statically and dynamically (i.e., feature traces) multiple versions of a program to obtain data on the evolution of class roles. They classify, over time, class roles as feature specific, not belonging to any features, infrastructural, and group feature.

Feature identification is similar to dynamic program slicing [40], in that both techniques seek to identify source code constructs responsible for a certain program behavior. While feature identification uses dynamic events to identify a microarchitecture implementing a feature of interest through its user-observable behavior, dynamic program slicing uses static analyzes (such as dynamic dependence graphs) to identify the program statements affecting the value of a variable for given program inputs.

5.3 Metamodeling and Model Transformations

Metamodeling techniques are often used to model program source code (for example, Pagel and Winter [41], Sunyé et al. [42]). We concur with Thomas that "[e]very model needs a metamodel" [43].

A metamodel defines a language to create models in some universe of discourse. Models issued from a metamodel can be visualized, compared, and modified at runtime programmatically, using operations defined in the metamodel. Model transformations are thus defined more precisely and easily when a metamodel is present. Model transformations are available, for example, in the UMLAUT [42] tool, and are central to the Model-Driven Architecture approach [44], [45].

5.4 Comparison to Previous Work and Criteria

Our work builds on and extends previous work, in particular [3], [6], [36]. Like other authors, we consider the problem of feature identification using static and dynamic data and we use probabilistic ranking; however, to the best of our knowledge, no previous work has obtained results as good as ours for similar feature identification tasks on large, multithreaded, object-oriented programs.

In this paper, we draw on our previous work [4] and substantially extend it to improve accuracy through the use of the epidemiological metaphor and statistical analyses.

Table 9 shows a comparison of our approach with related work on static and dynamic analysis and feature identification. In addition to the greater accuracy obtained through the epidemiological metaphor, as illustrated by the case studies in Part 3, our approach substantially improves upon previous works through the identification, modeling, and comparison, in large, multithreaded C++ programs, of method-level features, using both dynamic and static data with a scalable epidemiological metaphor, supplemented by knowledge-based filtering and a relevance index.

6 CONCLUSION

We proposed an approach, inspired by epidemiology, to feature identification and comparison, using consolidated tools and techniques such as parsing, processor emulation, and reverse-engineering techniques, to integrate static and dynamic data to support program understanding of large, multithreaded, object-oriented programs. We compared the accuracy of our approach in identifying variables, classes, functions, and methods supporting the "save a bookmark" feature of the MOZILLA Web browser, with a naive `grep`-based approach and an approach relying on concept analysis. Accuracy was also evaluated by using our approach in identifying features of CHIMERA, FIREFOX, ICEBROWSER, JHOTDRAW, and XFIG.

Our approach reduces the quantity of data to process and thus does not have scalability issues. It produces a ranked list of events participating in a feature and it allows the study of feature evolution. Extracted microarchitectures are valuable means to document programs and to support maintenance and program understanding tasks.

Future work will characterize the influence of noise and uncertainty when collecting data, study increased width of microarchitectures, define results of features comparisons precisely, and study alternative means, such as layered views, to improve features inspections. We also plan to measure the performances of people when using our approach to perform various program understanding tasks, study the influence of increased widths, and apply our approach to locate defects.

ACKNOWLEDGMENTS

Giuliano Antoniol was partially supported by the NSERC Canada Research Chair in Software Change and Evolution. Yann-Gaël Guéhéneuc was partially supported by an NSERC Discovery Grant.

REFERENCES

[1] E. Chikofsky and J.H. Cross II, "Reverse Engineering and Design Recovery: A Taxonomy," *IEEE Software,* vol. 7, no. 1, pp. 13-17, Jan. 1990.

[2] J. Concling and M. Bergen, "Software Reconnaissance: Mapping Program Features to Code," *J. Software Maintenance: Research and Practice,* vol. 7, no. 1, pp. 49-62, Jan. 1995.
[3] R. Koschke and D. Simon, "Locating Features in Source Code," *IEEE Trans. Software Eng.,* vol. 29, no. 3, pp. 210-224, Mar. 2003.
[4] G. Antoniol and Y.-G. Guéhéneuc, "Feature Identification: A Novel Approach and a Case Study," *Proc. 21st Int'l Conf. Software Maintenance,* Sept. 2005.
[5] Y.-G. Guéhéneuc and N. Jussien, "Using Explanations for Design-Patterns Identification," *Proc. First IJCAI Workshop Modeling and Solving Problems with Constraints,* pp. 57-64, Aug. 2001.
[6] N. Wilde and M.C. Scully, "Software Reconnaissance: Mapping Program Features to Code," *J. Software Maintenance: Research and Practice,* pp. 49-62, Jan-Feb. 1995.
[7] IDEFØ graphical language, http://www.idef.com/IDEF0.html, 2006.
[8] G. Antoniol, R. Fiutem, and L. Cristoforetti, "Using Metrics to Identify Design Patterns in Object-Oriented Software," *Proc. Fifth Int'l Symp. Software Metrics,* pp. 23-34, Nov. 1998.
[9] D. Edwards, S. Simmons, and N. Wilde, "An Approach to Feature Location in Distributed Systems," Technical Report SERC-TR-270, Software Eng. Research Center, 2004.
[10] Valgrind, http://valgrind.kde.org/, 2006.
[11] JProf, http://lxr.mozilla.org/mozilla/source/tools/jprof/, 2006.
[12] G.W. Snedecor and G.W. Cochran, *Statistical Methods.* Iowa State Univ. Press, 1989.
[13] Mozilla, http://www.mozilla.org, 2006.
[14] Firefox, http://www.mozilla.com/firefox/, 2006.
[15] Chimera, http://www.chimera.org/two/, 2006.
[16] ICEBrowser, http://www.icesoft.com/products/icebrowser.html, 2006.
[17] JHotDraw, http://www.jhotdraw.org/, 2006.
[18] Xfig, http://xfig.org/art2.html, 2006.
[19] 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," *Proc. Ninth Working Conf. Reverse Eng.,* E. Burd and A. van Deursen, eds., pp. 22-33, Oct-Nov. 2002.
[20] Y.-G. Guéhéneuc, "Ptidej," *A Tool Suite To Evaluate and to Enhance the Quality of Object-Oriented Programs,* since, [Online]. Available: http://www.ptidej.net/ July 2001.
[21] M.D. Ernst, J. Cockrell, W.G. Griswold, and D. Notkin, "Dynamically Discovering Likely Program Invariants to Support Program Evolution," *Proc. 21st Int'l Conf. Software Eng.,* pp. 213-224, 1999.
[22] T. Ball and J.R. Larus, "Efficient Path Profiling," *Proc. 29th Int'l Symp. Microarchitecture,* pp. 46-57 Dec. 1996.
[23] D. Melski and T.W. Reps, "Interprocedural Path Profiling," *Proc. 14th Conf. Computational Complexity,* pp. 47-62, 1999.
[24] M.J. Harrold, G. Rothermel, R. Wu, and L. Yi, "An Empirical Investigation of Program Spectra," *Proc. First Workshop Program Analysis for Software Tools and Eng.,* pp. 83-90, June 1998.
[25] C. Jeffery and M. Auguston, "Some Axioms and Issues in the UFO Dynamic Analysis Framework," *Proc. First ICSE Workshop Dynamic Analysis,* J.E. Cook and M.D. Ernst, eds., May 2003.
[26] M.D. Ernst, "Static and Dynamic Analysis: Synergy and Duality," *Proc. First ICSE Workshop Dynamic Analysis,* J.E. Cook and M.D. Ernst eds., May 2003.
[27] S. Reiss, M. Renieris, "Languages for Dynamic Instrumentation," *Proc. First ICSE Workshop Dynamic Analysis,* J.E. Cook and M.D. Ernst, eds., May 2003.
[28] S.P. Reiss and M. Renieris, "Encoding Program Executions," *Proc. 23rd Int'l Conf. Software Eng.,* pp. 221-230, May 2001.
[29] M. Renieris and S.P. Reiss, "ALMOST: Exploring Program Traces," *Proc. First Conf. Information and Knowledge Management/ Workshop New Paradigms in Information Visualization and Manipulation,* pp. 70-77, Nov. 1999.
[30] A. Hamou-Lhadj and T. Lethbridge, "Compression Techniques to Simplify the Analysis of Large Execution Traces," *Proc. 10th Int'l Workshop Program Comprehension,* pp. 159-168, June 2002.
[31] A. Hamou-Lhadj and T.C. Lethbridge, "An Efficient Algorithm for Detecting Patterns in Traces of Procedure Calls," *Proc. First Int'l Conf. Software Eng. Workshop Dynamic Analysis,* May 2003.
[32] T. Gyimóthy, R. Ferenc, and I. Siket, "Empirical Validation of Object Oriented Metrics on Open Source Software for Fault Prediction," *IEEE Trans. Software Eng.,* vol. 31, no. 10, pp. 897-910, Oct. 2005.
[33] A.G. Koru and J.J. Tian, "Comparing High-Change Modules and Modules with the Highest Measurement Values in Two Large-Scale Open-Source Products," *IEEE Trans. Software Eng.,* vol. 31, no. 8, pp. 625-642, Aug. 2005.
[34] K. Chen and V. Rajlich, "Case Study of Feature Location Using Dependence Graph," *Proc. Eight Int'l Program Comprehension,* pp. 241-252, June 2000.
[35] T. Eisenbarth, R. Koschke, and D. Simon, "Aiding Program Comprehension by Static and Dynamic Feature Analysis," *Proc. Eight Int'l Conf. Software Maintenance,* pp. 602-611, Nov. 2001.
[36] M. Salah, S. Mancoridis, "A Hierarchy of Dynamic Software Views: From Object-Interactions to Feature-Interactions," *Proc. 20th Int'l Conf. Software Maintenance,* pp. 72-81, Sept. 2004.
[37] M. Salah, S. Mancordis, G. Antoniol, and M.D. Penta, "Towards Employing Use-Cases and Dynamic Analysis to Comprehend Mozilla," *Proc. 21st Int'l Conf. Software Maintenance,* pp. 639-642, Sept. 2005.
[38] A.D. Eisenberg and K.D. Volder, "Dynamic Feature Traces: Finding Features in Unfamiliar Code," *Proc. 21st Int'l Conf. Software Maintenance,* pp. 337-346, Sept. 2005.
[39] O. Greevy, S. Ducasse, and T. Girba, "Analyzing Feature Traces to Incorporate the Semantic of Changes," *Proc. 21st Int'l Conf. Software Maintenance,* pp. 347-356, Sept. 2005.
[40] H. Agrawal and J.R. Horgan, "Dynamic Program Slicing," *Proc. Fourth Conf. Programming Language Design and Implementation,* pp. 246-256, June 1990.
[41] B.-U. Pagel and M. Winter, "Towards Pattern-Based Tools," *Proc. First European Conf. Pattern Languages of Programs,* July 1996.
[42] G. Sunyé, A.L. Guennec, and J.-M. Jézéquel, "Design Patterns Application in UML," *Proc. 14th European Conf. Object-Oriented Programming,* pp. 44-62, June 2000.
[43] D. Thomas, "Reflective Software Engineering—From MOPS to AOSD," *J. Object Technology,* vol. 1, no. 4, pp. 17-26, Sept. 2002.
[44] J. Bézivin, "From Object Composition to Model Transformation with the MDA," *Proc. 39th Int'l Conf. Technology of Object-Oriented Languages and Systems,* p. 348, Aug. 2001.
[45] Object Management Group, Model Driven Architecture, www.omg.org/mda/, Jan. 2005.

Giuliano Antoniol received the degree in electronics engineering from the Universitá di Padova in 1982 and the PhD degree in electrical engineering from the École Polytechnique de Montréal in 2004. He has worked in companies, research institutions, and universities. In 2005 he was awarded the Canada Research Chair Tier 1 in Software Change and Evolution. Giuliano Antoniol has published more than 90 papers in journals and international conferences. He served as a member of the program committees of international conferences and workshops such as the International Conference on Software Maintenance, the International Workshop on Program Comprehension, and the International Symposium on Software Metrics. He is presently a member of the editorial board of the J*ournal of Software Testing Verification and Reliability*, the *Journal of Information and Software Technology*, the *Journal of Empirical Software Engineering*, and the J*ournal of Software Quality*.

Yann-Gaël Guéhéneuc received the engineering diploma from the École des Mines of Nantes, France, in 1998 and the PhD degree in software engineering from the University of Nantes (under Professor Pierre Cointe's supervision) in 2003. His PhD thesis was funded by Object Technology International, Inc. (now IBM Ottawa Labs.), where he worked in 1999 and 2000. He is an assistant professor with the Department of Computing Science and Operations Research at the University of Montreal, where he leads the Ptidej team to evaluate and enhance the quality of object-oriented programs by promoting the use of patterns at the language, design, and architectural levels. His research interests are program understanding and program quality during development and maintenance, in particular through the use and the identification of recurring patterns. He is also interested in empirical software engineering, where he uses eye trackers to understand and to develop theories about program comprehension. He has published many papers in international conferences and journals.