Automated API Property Inference Techniques

Martin P. Robillard, Eric Bodden, David Kawrykow, Mira Mezini, and Tristan Ratchford

Abstract—Frameworks and libraries offer reusable and customizable functionality through Application Programming Interfaces (APIs). Correctly using large and sophisticated APIs can represent a challenge due to hidden assumptions and requirements. Numerous approaches have been developed to infer properties of APIs, intended to guide their use by developers. With each approach come new definitions of API properties, new techniques for inferring these properties, and new ways to assess their correctness and usefulness. This paper provides a comprehensive survey of over a decade of research on automated property inference for APIs. Our survey provides a synthesis of this complex technical field along different dimensions of analysis: properties inferred, mining techniques, and empirical results. In particular, we derive a classification and organization of over 60 techniques into five different categories based on the type of API property inferred: unordered usage patterns, sequential usage patterns, behavioral specifications, migration mappings, and general information.

Index Terms—API property, programming rules, specifications, protocols, interface, data mining, pattern mining, API evolution, API usage pattern

1 Introduction

Large-scale software reuse is often achieved through the use of frameworks and libraries, whose functionality is exported through Application Programming Interfaces (APIs). Although using an API can be as simple as calling a function, in practice it is often much more difficult; the flexibility offered by large APIs translates into sophisticated interface structures that must be accessed by combining interface elements into *usage patterns*, and taking into account constraints and specialized knowledge about the behavior of the API [1]. In brief, correctly using large and sophisticated APIs can represent a challenge due to hidden assumptions and requirements. To compound the problem, the knowledge necessary to properly use an API may not be completely or clearly documented.

In the last decade, numerous techniques have been developed by the research community to automatically infer undocumented properties of APIs. For example, techniques have been developed to infer common function call sequences (and detect incorrect sequences), or to identify valid migration paths between different versions of an API. The general goal of these approaches is to discover useful, but latent, information that can help developers use APIs effectively and correctly. Despite this unifying goal, most approaches have been developed in relative isolation, and differ in the exact nature of properties inferred, the input data required, and the underlying mining technology used.

To consolidate this growing field, this paper surveys over 60 techniques developed in 10 years of research and development on API analysis techniques. Our survey offers a synthesis of API property inference techniques in the form of a set of *dimensions* by which one can compare and understand the various techniques. This conceptual framework enables us to make original observations than can only be derived from an in-depth comparison of groups of approaches along different dimensions.

1.1 Scope of the Survey

We surveyed techniques to support the *automated* inference of API properties. We define an API to be *the interface to a reusable software entity used by multiple clients outside the developing organization, and that can be distributed separately from environment code*. Over the years, a large number of research projects have targeted the derivation of knowledge from programs. We restrict ourselves to techniques that focus on inferring properties for the public interfaces of components. Hence, we exclude from consideration work that reports results at the level of private implementation structures (such as errors in the use of program operators [2] or invariants on local variables [3]).

By *automated analyses*, we consider any analysis that derives, without significant manual intervention, *general* properties of the API that can influence subsequent software development of the API's clients. As such, we exclude work on API component searching and browsing techniques (which are context-specific), code-example retrieval techniques [4], [5] (which are also context-specific), design recovery (which is of interest to API developers, not users), and work on software metrics (which, although applicable to APIs, is not specific to reusable components).

Finally, since one of the goals of our survey is to synthesize existing knowledge about the practical application of API inference techniques, we restrict our coverage to work that

- *M.P. Robillard and D. Kawrykow are with the School of Computer Science, McGill University, #318, McConnell Engineering Building, 3480 University Street, Montréal, QC H3A 2A7, Canada. E-mail: {martin, dkawry}@cs.mcgill.ca.*
- *E. Bodden is with the Secure Software Engineering Group, European Center for Security and Privacy by Design (EC SPRIDE), Technische Universität Darmstadt, Mornewegstr. 30, Darmstadt 64293, Germany. E-mail: bodden@ec-spride.de.*
- *M. Mezini is with the Technische Universität Darmstadt, S2|02 A212 Hochschulstr. 10, Darmstadt 64289, Germany. E-mail: mezini@informatik.tu-darmstadt.de.*
- *T. Ratchford is with IBM Research, One Rogers Street, Cambridge, MA 02142-1203. E-mail: tratch@us.ibm.com*

Manuscript received 17 Oct. 2011; revised 22 May 2012; accepted 13 Sept. 2012; published online 21 Sept. 2012.
Recommended for acceptance by J. Grundy.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-2011-10-0295.
Digital Object Identifier no. 10.1109/TSE.2012.63.

describes implemented techniques that have been applied to existing APIs. Hence, work on the theoretical foundations of some of the analyses presented (e.g., theoretical work on invariant detection [6]) falls outside the scope of the survey.

We catalogued the techniques described in this survey by conducting a methodical review of all papers published between the years 2000 and 2010 in an initial list of 12 publication venues, which led to the eventual identification of over 60 techniques presented in over 20 different venues. The details of the survey protocol are reported in the Appendix.

2 Dimensions of Analysis

The most critical distinction between the works surveyed concerns the nature of the **property inferred**. By *property*, we mean any objectively verifiable fact about an API or its use in practice. We found that the data structure forming the output of proposed property inference techniques was usually not defined formally, and was often directly tied to the inference approach. The lack of standardized formalism for describing API properties creates important challenges when comparing approaches. We mitigate this problem by proposing a classification for properties, following a model inspired from Dwyer et al. [7].

2.1 General Categorization Framework

First, the inference can be of *unordered API usage patterns* that describe which API elements ought to be used together without considering the order of usage. The largest number of approaches, however, take order into account and provides more detailed knowledge about the correct use of an API in the form of *sequential usage patterns*. Other approaches produce *behavioral specifications* that attempt to describe the behavior of the API under specific conditions and, in particular, conditions that lead to erroneous state. The main distinction between the last two categories concerns whether an approach focuses on the programming pattern (sequential usage patterns) or the resultant state of the program (behavioral specifications). Yet other approaches infer various types of *migration mappings* between API elements (for example, equivalent API elements in different versions of an API). Finally, a number of approaches infer *general information* that consists of more idiosyncratic properties of the API. Because of the importance of this dimension, we organize our synthesis along the line of properties inferred.

2.2 Additional Terminology

We observe a lack of uniformity in terminology and definitions for the properties inferred by various approaches. Properties inferred are referred to as, in turn, properties, rules, specifications, patterns, protocols, etc. In many cases, these terms are adorned with various adjectives such as call-usage, usage, temporal, sequential, etc. We observe little consistency in the use of specific terms to the extent that, for example, rules, patterns, or specifications can refer to the same or different concepts.

To provide a consistent use of terminology throughout the survey, we define the terms *pattern*, *rule*, *protocol*, and *specification* as follows.

We consider a **pattern** to be a common way to use an API. Patterns are typically observed from the data as opposed to being formally specified by a developer.

A **rule** is a required way to use the API, as asserted by a developer or analyst. Typically, violating a rule can lead to faults in software that uses the API. Patterns and rules are related in that rules will naturally induce patterns. Alternatively, patterns observed by a tool or developer could be recognized as rules. A number of publications also refer to the inference of API usage **protocols**. We consider that a protocol represents a set of rules associated with a specific component or API.

Specification is another commonly used term in the literature on API property inference. Often we find that authors use this term to refer to rules, as defined above. In contrast, we will use the term *specification* to refer to a stronger property than just API usage rules. In this survey, we will use *specification* to denote rules that also encode information about the *behavior* of a program when an API is used. For example, while a rule could state that function `open` must be called before `read`, a specification might state that calling `read` without a preceding `open` causes an exception. We note that the distinction between a rule and a specification can be subtle in the case where the consequence of violating a rule is clear.

2.3 Structure of the Survey

We follow a parallel structure within Sections 3-7. Each section opens with an overview (subsection 1) that provides a description of the general type of **property inferred**. It then discusses different subcategories along with the classes of approaches that fall into those categories. This section is supported by a table of all the techniques reviewed, in chronological order. The nature of the information presented in the table varies slightly between sections to best describe the most important commonalities and variations between different types of approaches. The tables are indexed by author-date references, and include the name of the tool developed (if available). For ease of reference, all authors or tools mentioned in tables are also highlighted in bold the first time they are mentioned within a paragraph. For readability, we only cite a reviewed technique the first time it appears within a section.

The initial overview is followed by an in-depth discussion of the **mining or analysis techniques** used to infer the respective type of properties (subsection 2). This section does not follow a chronological order, but instead groups approaches by taking into account their commonalities in terms of mining algorithm, which forms our second major dimension of analysis. The third subsection summarizes the **main empirical results** reported for each group of techniques, our third main dimension of analysis. In this subsection, we do not systematically report on the evaluation of each technique. Because different research reports describe widely different evaluation styles and levels of thoroughness, we instead identify the results that can inform the selection, application, or advancement of property inference techniques.

Within each section, we discuss each approach along a number of minor dimensions. First, different techniques also vary in their stated **end goal**. A number of techniques were developed to provide general help or additional

TABLE 1
Works Surveyed: Unordered Usage Patterns

Work	Tool	Goal	Input	Context	Mining
Michail 1999–2000 [8], [9]	CodeWeb	Doc. & Understand.	Client Code	Class	Association Rule
Li & Zhou 2005 [10]	PR-Miner	Bug Detection	Client Code	Function	Association Rule
Livshits & Zimmermann 2005 [11]	DynaMine	Bug Detection	Change History	File	Association Rule
Bruch et al. 2006 [12]	FrUit	Recommendations	Client Code	Class	Association Rule
Bruch et al. 2009 [13]	ICCS	Recommendations	Client Code	Variable	Association Rule/Similarity
Monperrus et al. 2010 [14]	DMMC	Bug Detection	Client Code	Variable	Statistical

documentation to developers trying to understand an API. In other cases, the properties inferred were intended to be directly provided as input to checkers (static or dynamic) with the purpose of detecting violations or ensuring conformance to specifications. Finally, other approaches targeted specific tasks, such as helping developers migrate client code to a new version of an API, or improving the API's documentation. We can also distinguish techniques based on the type of **input data** they require, or even specific attributes of the input data (such as the definition of a code fragment, or **context**, see Section 3). When relevant, the main table in each section also includes a classification of the approaches along these minor dimensions.

3 Unordered Usage Patterns

A basic type of property that can be expressed about an API is that of an *unordered usage pattern*. Conceptually, usage patterns describe typical or common ways to use an API (i.e., to access its elements).

3.1 Overview

Unordered usage patterns describe references to a set of API elements (classes, functions, etc.) observed to co-occur with a certain frequency within a population of *usage contexts* (a lexically defined snippet of client code, such as a function). For this reason most approaches that infer unordered usage patterns use a form of frequent item set mining (see Section 3.2). As an example, for an I/O API it may be possible to detect the usage pattern $\{\text{open}, \text{close}\}$, which indicates that whenever client code calls an API method `open`, it also calls `close`, and vice versa. Being unordered, this pattern does not encode any information about whether `open` should be called before or after `close`.

3.1.1 Goal

Table 1 gives an overview of the inference techniques for unordered usage patterns that we surveyed. A first important distinction concerns the *goal* they pursue. We distinguish between three different goals: documentation and understanding of usage patterns, detection of violations to usage patterns (bug detection), and recommendation of API elements.

Michail was the first to explore the use of association rule mining to detect reuse patterns between a client and its library or framework [8], [9]. Michail's idea, implemented in a tool called **CodeWeb**, was to help developers understand how to reuse classes in a framework by indicating relationships such as "if a class subclasses class `C`, it should also call methods of class `D`." Michail detects these relations by mining client code that uses the API of interest. This preliminary work seeded the idea of using association rule mining on software engineering artifacts, but the absence of a more specific goal for the approach means that it also provides few guidelines for applying the technique to specific tasks. Subsequent techniques all focus on a more specific goal.

Unordered usage patterns can also be used to detect bugs. For example, if an approach determines that API methods `open` and `close` should be called within the same function, then the presence of an unmatched `open` method is evidence of a potential bug. **Li and Zhou** use association rule mining in **PR-Miner**, a tool to automatically detect unordered usage patterns [10]. Once identified, these patterns are considered rules and used to find violations. The assumption is that rule violations can uncover bugs. **DynaMine** [11] shares the same goal. It infers usage patterns by mining the change history of an API's clients. The idea behind DynaMine is to identify erroneous patterns to avoid committing them in the future. The properties inferred by DynaMine are pairwise association rules for methods inserted in a single source file revision. Rules are then checked by instrumenting and executing the client's source code. A third approach that focuses on bug detection is the one of **Monperrus et al.** [14]. They collect statistics about *type-usages*. A type-usage is simply the list of methods called on a variable of a given type in a given client method. They then use this information to detect other client methods that may need to call the missing method. Their idea is implemented in a tool called Detector of Missing Method Calls (**DMMC**).

Along with the emergence of recommender systems for software engineering [15], techniques have been proposed to *recommend* API elements that may be useful in a programming task. Such recommendations are typically produced by detecting unordered usage patterns. Framework Understanding Tool (**FrUiT**) [12] is a tool to help developers learn how to use a framework by providing them with context-sensitive framework reuse rules, mined from existing code examples. FrUiT's underlying principles are a refinement over **Michail**'s ideas with additional engineering enabling the tool to offer specific recommendations to a developer involved in a change task directly from within the IDE. For example, if a user instantiates an `IWizardPage`, FrUiT would recommend making a call to elements like `addPage()`. **Bruch et al.** present an Intelligent Code Completion System (**ICCS**) [13] to rank the methods proposed by an autocompletion system on the basis of programming patterns synthesized from code examples. The key idea of the work is, given a

client method in which a number of API methods have been called on a variable, find other client methods where similar methods have been called on a variable of the same type, and recommend method calls missing within the query context, in order of popularity.

3.1.2 Input

Inferring unordered usage patterns works by analyzing client source code. For example, to infer usage patterns for the Java concurrency API, an approach would look at a large corpus of code snippets that use this API. A corpus can have different origins, but typically consists of open-source projects. For this purpose, so-called *self-hosted projects* (open-source projects that use their own public APIs) are particularly useful because the code examples are segments of production code generally from the same organization that produced the APIs. For example, **Bruch et al. ([12] and [13])** and **Monperrus et al.** use the Eclipse Platform as a self-hosted corpus. **DynaMine** is the only different approach. Instead of relying on a corpus of client source code, it requires the change history of the client code. This history is composed of a stream of change sets stored in a revision control system. DynaMine translates the usage pattern mining problem into an item set mining problem by representing a set of methods committed together into a single file as an item set.

3.1.3 Context

Unordered usage patterns involve a containment relation. If we want to declare that methods `open` and `close` always occur together, we must specify the *context* in which they co-occur. The notion of context maps directly to that of an item set for the purpose of data mining. For example, if the context for an approach is the *function*, an item set consists of all the references to API elements within client functions. Hence, if a client program consists of two functions that call API functions `a` and `b`, and `b` and `c`, respectively, this program would constitute a corpus with two item sets, {`a`, `b`} and {`b`, `c`}.

We distinguish four different types of contexts, in increasing order of granularity: file, class, function, and variable. The class context aggregates all references to API elements within any member of the class (and similarly for the file). Items mined by **CodeWeb** are aggregated at the class level. For example, if any function or method defined in a class A calls a function f, the class as a whole is considered to call the function. **Bruch et al.** also mine class contexts in **FrUiT**. This level of granularity is well adapted to their purpose (framework documentation and recommendations for framework extensions, respectively), but the coarse granularity also means that the approach is noisy. Section 3.2 discusses this aspect in more detail. To deal with noise and to provide better explanations for mined patterns, approaches that work at the class level further annotate elements in item sets with information indicating how the element is meant to be used by clients of the API (e.g., extending a class versus calling one of its methods). With **DynaMine**, **Zimmermann and Livshits** mine changes at the file level because they only consider files that were actually changed in a given client within some time window. Mining at a finer granularity would likely result in very few patterns.

The function context aggregates all references within a function. For example, **PR-Miner** parses functions in C source code to store, as items, identifiers representing functions called, types used, and global variables accessed.

The variable context aggregates all methods called on an object-type variable within a client method. For example, if client method `mc` declares a variable `buffer` of type `Buffer`, an item set will consist of all the methods of `Buffer` called on `buffer` within `mc`. **ICCS** and **DMMC** are two recent approaches working on object-oriented source code that produce item sets for variable contexts.

Context granularity has a critical impact on the nature of the dataset available for mining. Mining broad contexts (class or file) generates fewer but larger items sets. In contrast, mining narrow contexts (function or variable) generates more, but smaller, item sets. The advantage offered by variable-level contexts is that this approach curtails the amount of spurious results (statistically inferred patterns that do not represent a meaningful association). The tradeoff is that the approach is unable to detect usage patterns that involve collaboration between objects of different types.

3.2 Mining Techniques

All unordered pattern mining approaches follow the same general strategy:

1. Mine a corpus of software engineering artifacts to create a dataset that consists of item sets, where an item set represents API elements used within a usage context.
2. Apply a mining algorithm to discover frequent item sets and (in most cases) generate association rules from these frequent item sets.
3. Apply filtering heuristics to improve the quality of the results.

The previous section described how the different approaches mapped information contained in software artifacts to item sets. We now focus on the pattern mining techniques employed on these item sets and on the heuristics used to improve the results.

3.2.1 Algorithms

All approaches described in this section use association rule mining except the one of **Monperrus et al.** Association Rule Mining (ARM) is a data mining technique that computes frequent subsets in a dataset and attempts to derive rules of the form $A \rightarrow C$, where A is the *antecedent* and C the *consequent*. Such a rule stipulates that if a *transaction* (an individual item set in a dataset) contains the antecedent, it is likely to also contain the consequent. The standard measures of rule quality are the support and confidence. The support is the number of transactions in the dataset that contain $A \cup C$. The confidence is the conditional probability $P(C \subset T \mid A \subset T)$, where T is a transaction.

Most approaches use preexisting ARM algorithms to infer patterns. **Michail [8]** does not specify the mining algorithm or package used for his initial exploration, so we can assume that a simple frequent item set mining algorithm is used. Given his restriction to rules with only one antecedent and one consequent, the algorithm is

reduced to mining co-occuring pairs of elements with a given support and confidence. In his latter attempt (2000) to infer generalized rules (where type subsumption is taken into account), Michail references a specific data mining algorithm [16]. Both **DynaMine** and **ICCS** integrate the Apriori algorithm [17]. For **FrUiT**, **Bruch et al.**, use the Opus algorithm [18], selected for its low memory requirements. **PR-Miner** uses the FPclose frequent item set mining algorithm [19], whose chief characteristic is to mine only *closed* frequent item sets, i.e., where there are no sub item sets that are subsumed by larger item sets with the same support. The authors then use a novel algorithm called ClosedRules to efficiently generate closed association rules from the frequent item sets mined by FPclose.

For generating recommendations for code completion, **Bruch et al.** use two additional mining algorithms in addition to ARM. The first is context-independent and always reports the most frequent methods called on a given type in their corpus. This trivial mining algorithm is used as a baseline. Bruch et al. further experiment with association rule mining, and also develop a new mining algorithm inspired from the idea of k-Nearest Neighbor (kNN) classification. The basic idea of their new algorithm is to find the code snippets most similar to the context for which recommendations are desired, and to generate recommendations based on the item sets found in these snippets.

Finally, **Monperrus et al.** do not use any standard data mining algorithm as part of their approach. Rather, for a given variable x of type T, they generate the entire collection of usages of type T in a given code corpus. From this collection, the authors compute various metrics of similarity and dissimilarity between a type usage and the rest of the collection. The statistical approach used by Monperrus et al. is reminiscent of a bug detection approach originally proposed by Engler et al. [2]. The approach of Engler et al. embodies many of the ideas also found in API property inference techniques, but falls outside the scope of the survey as it targets the detection of errors in source code, with an emphasis on the correct use of variables (as exemplified by the analysis of null checks, or locking variables for concurrent use). However, because of its seminal nature, we briefly describe the technique here and refer to it in the following sections.

In their work on bug detection, Engler et al. analyze the use of various program elements (variables, functions, etc.) and automatically discover programming rules that are instances of a predefined set of rule templates. Examples of rule templates include "do not reference null pointer $<p>$" and "$<a>$ must be paired with $<b>$" [2], where $<p>$, $<a>$, and $<b>$ are *slots* that can be filled by certain types of program elements (a pointer and two functions, respectively). The core of Engler et al's approach consists of checking all instances of a rule and, in cases where violations are found, using a statistically-derived threshold to decide whether the violations invalidate the rule or indicate a bug.

3.2.2 Rule Quality

The results of association rule mining are strongly influenced by the two rule quality thresholds: minimum support and confidence. Table 2 summarizes the minimum support and confidence values selected. These numbers can only be roughly compared because their impact on the results will also be determined by the nature of the dataset and the filtering rules in place. However, even at a superficial level they indicate different design philosophies for detecting usage patterns. **Michail's** thresholds are very low, indicating a desire to find as many rules as possible. In contrast, **PR-Miner** works with strict thresholds, a sensible choice given the application to bug detection. For **DynaMine**, the authors do not use minimum thresholds, but report patterns ranked by decreasing confidence.

TABLE 2
ARM Rule Quality Thresholds

Work	Tool	Sup.	Conf.
Michail 1999 [8]	CodeWeb	3	0.25
Michail 2000 [9]	CodeWeb	15	0.10
Li & Zhou 2005 [10]	PR-Miner	15	0.90
Livshits & Zimmermann 2005 [11]	DynaMine	Floating	
Bruch et al. 2006 [12]	FrUit	10	0.50
Bruch et al. 2009 [13]	ICCS	5	0.70

3.2.3 Filtering Heuristics

All ARM-based approaches report numerous spurious rules. Spurious rules represent co-occurrences of references to API elements that are found in the data but that do not correspond to sensible or useful usage patterns. The standard strategy to reduce the noise in detected usage patterns is to hand-craft filtering heuristics based on knowledge about the approach or the domain.

Michail, **Livshits and Zimmermann**, and **Bruch et al.** [12] employ filtering heuristics to improve the results. For example, Michail removes patterns stating, for example, that a class that calls a library function on type A must also instantiate this type. For **DynaMine**, Livshits and Zimmermann introduce a pattern filtering phase to "greatly reduce the running time of the mining algorithm and significantly reduce the amount of noise it produces" [11, p. 299]. The filters are based on domain knowledge of software repository mining. Examples of filters include ignoring the first addition of a file to a revision control system as part of a transaction. Finally, with **FrUiT**, **Bruch et al.** also apply a number of filtering heuristics. Some of their heuristics are general (such as their removal of "overfitting" rules that add preconditions with little or no gain in confidence), and some are domain-specific, including the same as the example given for Michail's work, where obvious rules are removed.

PR-Miner uses a slightly different approach and prunes spurious results only after violations of a rule (or pattern) are found. When a violation is found, PR-Miner relaxes the constraint that API elements must be used in the same client function, and looks for API elements to complete the rule also in callees. For instance, this technique would avoid falsely violating the rule `{open,close}` within a function `f` if it finds a call to `open` in `f` and the matching call to `close` in a function called by `f`.

3.3 Empirical Findings

Techniques to infer unordered usage patterns were some of the earliest attempts at API property inference, and their assessments were mostly exploratory, consisting of the

application of the technique to one or a few selected systems, and a discussion of the results [8], [9], [11]. Later works include evaluations using cross-validation techniques [13], [14]. All techniques described in this section were evaluated on a small number of systems, so there exist practically no results generalizable between target systems.

In his early work **Michail** applies his approach to two C++ frameworks (ET++ and KDE). Although his reliance on case studies to informally assess the approach limits the generalizability of the results, his observations capture important lessons. First, Michail's interests target the discovery of rules, and as such he applies his approach with very low support and confidence, observing that a filtering stage is necessary for the approach to be feasible. In his case study of KDE, Michail also observes that pattern violations ("detractors") represent uncharacteristic reuse behavior that may be worthwhile to detect, a goal pursued by most following approaches.

Li and Zhou evaluate **PR-Miner** by applying it to three C/C++ systems, reporting on the number and size of rules discovered as a function of the support threshold. More importantly, they also study the violations of patterns reported by the approach, and in this way demonstrate the potential of association rules for detecting bugs. The authors were able to identify 16 bugs in Linux, 6 in PostgreSQL, and 1 in the Apache HTTP server by looking at the top 60 violations detected. PR-miner is also the first (and one of the few) approaches to consider rules with more than one antecedent, and as such demonstrates the feasibility of ARM to discover general usage patterns by removing an important constraint. However, Li and Zhou also note that a large number of the association rules are false positives, even with pruning.

Livshits and Zimmermann evaluate **DynaMine** by applying it to the change history of two Java systems. They focus on the number and nature of patterns detected over the entire change history of both systems. Although DynaMine manages to find a number of usage patterns, this number remains modest. The authors find only 56 patterns in the change history of Eclipse and jEdit using their chosen confidence and support thresholds, only 21 of which are observed to occur at runtime. A casual observation of the patterns reported also shows that the majority involve methods exhibiting a "natural duality," such as `begin-end`, `suspend-resume`, etc. The lesson from this evaluation is that considering change transactions as item sets may yield too little data to find robust patterns.

Bruch et al.'s [13] evaluation of their intelligent code completion system involves a systematic assessment of four different recommendation algorithms for autocompletion using a cross-validation design on data for clients of the SWT toolkit. Specifically, the evaluation compares the recall and precision of recommendations produced with the default Eclipse algorithm (alphabetical), the frequency algorithm (most popular), association rule mining, and their own kNN-inspired algorithm. The evaluation shows that for their dataset, their algorithm offers the best overall performance (precision/recall tradeoff calculated with F1[1]), but that it only offers a performance marginally superior to association rule mining. However, both the kNN-inspired and association rule techniques are shown as much superior to either frequency or alphabetical-based recommendations.

Monperrus et al.'s evaluation of **DMMC** is also conducted by applying the techniques to SWT, in this case to detect missing method calls in SWT clients. The approach is evaluated with a synthetic benchmark by artificially removing missing method calls in client code snippets and using the approach to recommend the missing call. The authors tested two algorithms, one with a coverage, average precision, and average recall of 0.80, 0.84, and 0.78, and one with 0.67, 0.98, and 0.66. Inspection of the results provides the additional insight that although the approach can recommend method calls with excellent performance, it is much less obvious to know how exactly to use the recommended method in that scenario: what arguments to pass in, what to do with the return value, etc. The techniques described in the next section expand the definition of usage pattern to provide additional information that can help answer these questions.

1. $F1 = 2 \cdot (\text{precision} \cdot \text{recall})/(\text{precision} + \text{recall})$.

4 Sequential Usage Patterns

Sequential usage patterns differ from unordered patterns in that they consider the order in which API operations are invoked. For the sequence of method calls `close` → `open`, an unordered approach would not be able to detect any problem, while sequential pattern mining would be able to alert the programmer that `open` should precede `close`.

Most of the property inference approaches discussed in this paper fall into the category of sequential pattern mining. This is not surprising: Although unordered patterns are useful, their detection is easy to implement and mostly limited to variants of frequent item set mining. The extension to sequential patterns introduces many new and challenging research problems, such as how to store abstractions of sequences efficiently and how to infer useful patterns given an observed sequence. As will become evident in this section, there is a large degree of freedom in answering those research questions.

4.1 Overview

Table 3 gives an overview of all surveyed sequential mining techniques in chronological order. For sequential usage patterns, the notion of *context* no longer lends itself to a clean parallel comparison due to the variety and complexity of the mining algorithms employed.

4.1.1 Goal

The motivation for mining sequential API patterns can be expressed as four different goals: mining specifications, detecting bugs, guiding navigation through the API elements, and documenting the API. These distinctions represent the perspective of the authors in describing their technique, but, in practice, these goals largely overlap. The most commonly stated goals are API documentation and bug detection.

Techniques developed for API documentation try to infer some high-level sequential patterns from program code under the assumption that this pattern will have value as documentation. One of the oldest approaches in this

TABLE 3
Works Surveyed: Sequential Usage Patterns (in Chronological Order)

Work	Tool	Goal	Input	Mined Patterns	Representation
Ammons et al. 2002 [20]	–	Spec. Mining	Traces, Human	FSA	FSA
Whaley et al. 2002 [21]	–	Bug Detection	Client Code	FSA	FSA
Yang & Evans 2004 [22], [23]	–	Bug Detection	Traces	Multi-pattern	Response Patterns
Alur et al. 2005 [24]	JIST	Documentation	API Code	FSA	FSA
Mandelin et al. 2005 [25]	Prospector	API Navigation	Client Code	Special purpose	Navigation Path
Salah et al. 2005 [26]	Scenariographer	Documentation	Traces	Reg. Expression	Reg. Expression
Weimer & Necula 2005 [27]	–	Bug Detection	Client Code	Single-pattern	Association Rule
Acharya et al. 2006, 2009 [28], [29]	–	Bug Detection	Client Code	FSA	FSA
Dallmeier et al. 2006 [30]	ADABU	Documentation	Running Cl. Code	FSA	FSA
Liu et al. 2006 [31], [32]	ltRules	Bug Finding	Client Code, Human	Multi-pattern	BLAST [33] patterns
Lo & Khoo 2006 [34]	SMArTIC	Spec. Mining	Traces, Human	FSA	FSA
Yang et al. 2006 [35]	Perracotta	Bug Detection	Traces	Multi-pattern	Alternating Chains
Acharya et al. 2007 [36]	–	Documentation	Client Code	Multi-pattern	Partial Order
Kagdi et al. 2007 [37]	–	Spec. Mining	Client Code	Single-pattern	Association Rule
Quante & Koschke 2007 [38]	–	Spec. Mining	Traces or Cl. Code	FSA	FSA
Ramanathan et al. 2007 [39], [40]	Chronicler	Spec. Mining	Client Code	Precondition	Preconditions
Shoham et al. 2007 [41]	–	Spec. Mining	Client Code	FSA	FSA
Thummalapenta & Xie 2007 [42]	PARSEWEB	API Navigation	Client Code	Special purpose	Navigation Path
Wasylkowski et al. 2007 [43]	JADET	Bug Detection	Client Code	Single-pattern	Association Rule
Walkinshaw et al. 2007/2008 [44], [45]	StateChum	Bug Detection	Running Cl. Code	FSA	FSA
Gabel & Su 2008 [46]	Javert	Bug Detection	Traces	Multi-pattern	Comb. of $(ab^*c)^+$
Lorenzoli et al. [47]	–	Spec. Mining	Traces	FSA	FSA with data constr.
Lo et al. 2008 [48]	–	Bug Detection	Traces	Multi-pattern	Seq. Assoc. Rules
Sankaranarayanan et al. 2008 [49]		Documentation	Running Cl. Code	Multi-pattern	Datalog rules
Zhong et al. 2008 [50]	Java Rule Finder	Documentation	API Code	Special purpose	Program Rule Graph
Gabel & Su 2009 [51]	–	Bug Detection	Traces	Single-pattern	Instances of $(ab^*c)^+$
Lo et al. 2009 [52]	–	Spec. Mining	Traces	Multi-pattern	Quantif. Temp. Rules
Nguyen et al. 2009 [53]	GrouMiner	Documentation	Client Code	Special purpose	Groum
Pradel & Gross 2009 [54], [55]	–	Bug Detection	Traces	FSA	Probabilistic FSA
Thummalapenta & Xie 2009 [56]	CAR-Miner	Bug Detection	Client Code	Single-pattern	Seq. Assoc. Rules
Thummalapenta & Xie 2009 [57]	Alattin	Bug Detection	Client Code	Single-pattern	Association Rule
Wasylkowski & Zeller 2009 [58]	Tikanga	Bug Detection	Client Code	Precondition	Operational Precond.
Zhong et al. 2009 [59]	Doc2Spec	Bug Detection	Comments	Single-pattern	Instances of $(ab^*c)^+$
Gabel & Su 2010 [60]	OCD	Bug Detection	Running Cl. Code	Single-pattern	Instances of $(ab)^+$
Gruska et al. 2010 [61]	checkmycode.org	Bug Detection	Client Code	Single-pattern	Association Rule

category is **JIST** by **Alur et al.** [24], which infers finite-state patterns for Java APIs.

Techniques developed for bug detection typically go one step further: They not only infer patterns, but also use these patterns for anomaly detection. As for unordered patterns, sequential patterns that have high support but nevertheless get violated may indicate bugs. For example, **OCD** by **Gabel and Su** [60] is an efficient runtime tool for inferring and checking simple temporal patterns using a sliding-window technique that considers a limited sequence of events.

Two tools, **Prospector** by **Mandelin et al.** [25] and **PARSEWEB** by **Thummalapenta and Xie** [42], were developed for the purpose of API navigation. Given a user-selected API element, the tools suggest useful ways to navigate the API from that element. Prospector, for example, shows users how to create an object of some target type given an object of another type. Due to the nature of those queries, approaches in this category rely more on data flow than control flow. Nevertheless, they are sequential because they suggest methods to be called in a specific order.

Many other papers describe a sequential pattern mining technique without mentioning a specific goal. In accordance with the terminology used in these papers, we characterize this kind of work simply as specification-mining techniques.

We observe a chronological tendency regarding the goal of the different approaches. Initially, many sequential inference techniques were primarily developed for the general goal of documentation and program understanding. Lately, techniques increasingly focus on bug finding. We surmise that the appeal of a more focused research problem may have been paired with an increased ability to instrument and test-run programs.

4.1.2 Input

Sequential patterns can be derived from a wide variety of inputs. We also distinguish inference techniques by the input they require. The main difference is naturally between *dynamic* and *static* approaches. Dynamic approaches work on data collected from a running program, whereas static approaches work directly on the artifacts related to the API of interest. Within these broad categories there also exist important differences.

Dynamic approaches typically read a single execution trace as input. A tool can read the trace online (while the program is executing) or offline by first recording a trace as the program runs and then reading the trace after the execution has terminated. In Table 3, dynamic approaches (online or offline) are identified as requiring *Traces* as input. Some techniques are not only online; they actually have to run the source code because they heavily interact with the running program (i.e., it is not sufficient to have pre-collected traces). Such approaches are recorded as requiring *Running Client Code*. Examples include **OCD** and **ADABU** [30]. We further note that two dynamic approaches are not fully automatic and require additional input from a *Human* expert. This is the case of the first specification mining approach by **Ammons et al.** [20] and of **SMArTIC** by **Lo and Khoo** [34].

Among the static approaches, we distinguish between the type of artifacts they target. A popular strategy is to analyze source code that uses the API (*Client Code*). This source code does not necessarily need to be executable. **Whaley et al.** [21], for instance, infer finite-state specifications through static code analysis by inferring possible call sequences by analyzing only the program code. Another strategy, employed, for example, by **JIST** and **Java Rule Finder** (**JRF**) [50], is to derive rules by analyzing the code of the API itself. These approaches do not require the code of client applications that use an API. Finally, techniques can also use other artifacts besides source code. One example is **Doc2Spec** by **Zhong et al.** [59], which works on natural-language specifications extracted from Javadoc *comments*.

4.1.3 Mined Patterns

We further distinguish sequential mining approaches by the kind of patterns that they mine (see column "Mined Patterns"). In the table, the column "Representation" provides additional information about the exact representation used.

Single patterns. A significant number of approaches mine instances of a single sequential pattern. Such sequential patterns can consist simply of an ordered pair of API elements (a, b), indicating that the usage of element a should occur before b in a program's execution. The rule that a call to a particular method b should typically follow a call to another method a is frequently written as $a \prec b$. This pattern is also known as the Response pattern [7]. In the remainder of this paper, we will call such patterns *two-letter patterns*. Many approaches fall into this category, for instance, **OCD**, **Alattin** [57], and others [27], [43], [59], [61]. Other approaches go beyond two-letter patterns by aggregating two-letter patterns into larger patterns, such as in **Perracotta** [35]. For instance, from $a \prec b$ and $b \prec c$ one may infer that $a \prec b \prec c$. **Acharya et al.** construct partial orders from two-letter patterns [36]. **Kagdi et al.** use sequential pattern mining to find rules based on the order in which an API is called within the context of a single method [37]. The approach can infer, for instance, that if method `a()` follows `b()` in one client method, then that method should also call `c()`. Other approaches try to mine larger patterns directly [51], [59]. One common pattern of this category is the "resource usage pattern," which can be denoted by a regular expression (ab^*c), where a resource is first created (a), then potentially used many times (b^*), and finally discarded (c). When discussing API properties we will often denote a sequential pattern by its equivalent regular expression. However, we note that hardly any approach reports patterns directly as regular expressions—most approaches report sequential patterns using some other representation, for instance, as a Finite-State Automaton (FSA).

Multiple patterns. A further class of approaches supports mining instances of several patterns at once. Such patterns can be special regular expressions, such as in **Javert** [46], or instances of special temporal patterns such as *Initialization*, *Finalization*, *Push-Pop* (two methods have to be called an equal number of times), *Strict Alternation*, *Function Pair*, and *Adjoining Function*, such as in the work of **Liu et al.** [31], [32]. Interestingly, such patterns can form a partial order, as some patterns imply others. For instance, *Strict Alternation* implies *Push-Pop*. **Yang and Evans** propose an approach that can find a best matching pattern in such cases [22], [23]. **Sankaranarayanan et al.** propose an approach to infer Datalog rules from execution traces [49]. **Lo et al.** present an approach that can mine rules of arbitrary length [52].

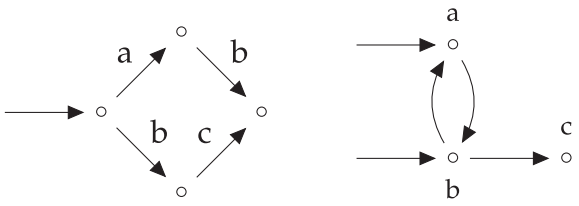
Temporal formulas. Other approaches are also based on mining instances of certain patterns but describe these patterns using temporal formulas. **CAR-Miner** [56], for instance, uses Boolean formulas, while the approach by **Lo et al.** [48] uses a temporal logic.

Preconditions. The tools **Chronicler** by **Ramanathan et al.** [39], [40] and **Tikanga** by **Wasylkowski and Zeller** [58] mine patterns that must match *before* an API element such as a function may be used. Typically, these patterns are sequences of method/function calls. We refer to these patterns as "operational preconditions," a term coined by Wasylkowsky and Zeller [58]. It is also possible to infer simpler data-flow patterns (e.g., that an argument may not be null [39]). Preconditions of this nature can be expressed in CTL [58] or other ad hoc formal languages [39], [40].

Finite-state automata and regular expressions. Many techniques mine API specifications by encoding temporal order as finite-state automata [20], [21], [24], [27], [28], [29], [30], [34], [38], [41], [44], [45], [46], [47], [51], [54], [55]. Some approaches opt to label the finite-state automaton with additional information, for example, with predicates inferred from invoking accessor methods. For instance, the state obtained after creating an empty Vector may be labeled with `isEmpty()`, while the successor state reachable through calling `add(..)` would be labeled with $\neg$`isEmpty()`. **ADABU** makes such a distinction. **Pradel and Gross** [54] label edges with the probabilities of those edges being taken on the observed executions, yielding a so-called probabilistic FSA (PFSA). As **Ammons et al.** show, another option is to compute such a PFSA first, but to then convert it into a regular FSA by removing the probabilities from the edges while at the same time deleting entirely such edges that are labeled with a probability below a certain threshold.

We note that only one of the approaches we surveyed, **Scenariographer** by **Salah et al.** [26], reports patterns in the form of general regular expressions.

Special-purpose representations. The tools **Prospector** and **PARSEWEB** seek to support API navigation, and for this purpose they use specialized representations. Prospector uses so-called "jungloids"; a jungloid is an expression that allows a user to create an object of one type given an object of another type, for example, an abstract syntax tree from a file handle referring to a source code file. A jungloid is an ordered usage pattern because it can comprise several function calls that need to be invoked in the given order. Graph-based object usage models (Groums) are another special-purpose property representation, used in **GrouMiner** by **Nguyen et al.** [53]. Groums associate events in a directed acyclic graph (DAG). In contrast to finite-state automata, this graph can hold special nodes to represent control structures, such as loops and conditionals. Furthermore, edges not only represent sequencing constraints, but also data dependencies. **Zhong et al.**'s tool **Java Rule Finder** encodes temporal rules in a so-called Program Rule Graph (PRG).



(a) Labeled edges (b) Labeled states

Fig. 1. Different automaton models.

4.1.4 Other Considerations

A number of additional aspects must be considered when defining and interpreting sequential API properties.

Edges versus states. Many techniques mine API specifications by encoding temporal order as finite-state automata. When doing so, one has the choice of representing events such as method calls as either states or edges. On the one hand, representing events as edges has the advantage that the language accepted by the resulting state machine is exactly the language of allowed event sequences. On the other hand, inference of a minimal finite-state machine with such a layout is NP-hard [54]. It can therefore make sense to represent events as states because inferring such automata has lower complexity. This representation has the disadvantage, though, that every event is only represented by one single state, even if multiple paths lead to the same event, effectively making the representation context insensitive.

Fig. 1 shows an example of attempting to infer sequential patterns over two input sequences "ab" and "bc." In Fig. 1a, edges are labeled with event names. The automaton can hence keep both inputs apart. Sequences such as "ac" are not part of the automaton's language. In Fig. 1b, this is different. Both b events are identified by the same state. Moreover, according to the traces, b can occur after a but also before c. However, the fact that c can only follow b if this b was not preceded by a gets lost as both prefixes of b are merged in this representation.

The example may make it appear that labeled edges are superior to labeled states. However, the latter representation can be constructed efficiently. Several approaches opt for a state-centric representation, e.g., the work by **Pradel and Gross** and by **Whaley et al.** Nevertheless, it appears more common to represent events by transitions [20], [24], [27], [28], [29], [30], [34], [38], [41], [44], [45], [46], [47], [51].

Allowed versus forbidden sequences. Another question is whether an inferred finite-state machine should represent the language of allowed event sequences or rather the language of forbidden sequences. All of the approaches we surveyed opted for the first choice: Edges in the finite-state machine represent calls that are allowed. Unfortunately, since all of the presented approaches are incomplete, one cannot automatically infer from such a specification that missing edges denote forbidden events: It may be the case that such events are allowed but were not observed on the traces used for learning. This incompleteness causes the inferred specifications to have a potential for yielding false positives when being directly applied to automated bug finding. However, they nevertheless have a high documentation value.

Multi-object properties. Most of the surveyed approaches infer patterns that represent sequencing constraints on a single API element, typically a single reference type. There are some constraints, however, that span multiple types in combination. For instance, one may be interested in inferring that a socket's streams should only be used as long as the socket itself has not been closed. We find that only seven out of the 33 surveyed sequential mining techniques can infer such "multi-object properties" [20], [35], [43], [47], [53], [54], [58]. The reason for this is probably that single-object approaches are much easier to design and implement. Static multi-object approaches not only have to solve the aliasing problem for individual objects but also need to relate multiple objects with each other [62], [63]. Dynamic approaches must use expensive mappings to associate state with multiple combinations of objects [64].

4.2 Mining Techniques

Mining sequential patterns requires more sophisticated analyses than for unordered patterns. We also observe a greater variety in the mining techniques used. The wealth of ideas explored as part of the work on sequential usage pattern inference escapes any obvious categorization. Nevertheless, we can distinguish between three general strategies for engineering a solution: transforming the input data to use a third-party tool as a black box, using a transformational approach involving various combinations of model transformation, clustering, and filtering algorithms, and performing pattern-matching against a predefined set of templates. Naturally, a given approach can combine elements from any of the three strategies. We classify techniques according to their dominant strategy.

4.2.1 Techniques Relying on Third-Party Tools

Techniques in this category use off-the-shelf learners or verification tools in one way or another. Generally, they also preprocess raw input data before providing it to learners, and postprocess the results.

Some approaches use *frequent item set mining*, much like the approaches described in Section 3, but include temporal information in the definition of the elements in the item sets [43], [57], [58], [61]. For example, **Alattin** generates association rules about conditions that must occur before or after a specific API call. **JADET** collects sets of API temporal properties observed in client methods, e.g., $\{hasNext \prec next, get \prec set\}$, from object-specific intraprocedural control-flow graphs and provides those temporal properties to a frequent item set miner.

Other approaches directly mine sequential patterns by using *closed frequent sequential pattern mining* [28], [37], [39], [48], [56]. This mining technique exhibits a higher computational cost than unordered item set mining [65], but has the advantage of retaining useful information like the frequency of an element in its context, the order of the elements, or any context information about the use of the elements. The higher computational cost is compensated for by the time saved in examining fewer false positives and covering more valid cases. Also, more guidance about fixing detected rule violations is provided, e.g., by giving information about where to insert a missing call [65]. Most

of these approaches [34], [37], [39], [48], [56] use the BIDE algorithm [66] to find frequently occurring subsequences of API calls made either on individual objects or across all API calls within a method, or to mine FSAs [28]. **Ramanathan et al.** use both unordered frequent item set mining and sequential pattern mining. The former is used to mine data-flow predicates, the latter to mine control-flow predicates. Similarly, **Acharya et al.** [36] use an off-the-shelf *frequent closed partial-order algorithm* to mine a string database of API method call sequences.

Ammons et al. and **Lo et al.**'s **SMarTic** use the sk-string *FSA learner* [67]. In Ammons et al.'s work the sk-strings algorithm operates on "scenarios"—subsequences of events from execution traces that are related by data flow. The mined PFSA is postprocessed to remove parts with edges with low likelihood of being traversed. Better scalability and accuracy is achieved by **Lo and Kho** [34] by performing some filtering and clustering on the input traces and by applying the learner to each cluster individually. The resulting PFSAs are subsequently merged.

JIST employs a combination of predicate abstraction [68], partial information safety games [69], and the L* learning algorithm [70]. Given an Java class in Jimple format [71] and a set of abstraction predicates that compare a class variable to a constant, a class with only Boolean (or enumerated) variables is produced and the transformed class is rewritten to a symbolic representation compatible with the input format of the model checker NuSMV [72]. The JIST synthesizer implements the L* learning algorithm via CTL model checking queries on this symbolic representation using NuSMV. The synthesis is based on a two-player game where Player 1 tries to find a safe sequence of method calls and Player 2 tries to find a path through a called method that raises an error. A safe interface yields a winning strategy for Player 1.

Sankaranarayanan et al. [49] mine API specifications expressed as *Datalog* programs using Inductive Logic Programming (ILP), a relational data mining technique that seeks to learn Datalog programs given some relational data, a target concept, and background knowledge about the structure of the target program.

Walkinshaw et al. present a semi-automated approach to inferring FSAs from dynamic execution traces that builds on the QSM algorithm [73]. This algorithm infers a finite-state automaton by successively merging states. To avoid overgeneralization, the algorithm poses membership queries to the end-user whenever the resulting machine may otherwise accept or reject too many sequences. Walkinshaw and Bogdanov extend the approach in a follow-up paper [45] to enable users to restrict the behavior of the inferred automaton through LTL formulas in order to reduce the number of user queries. A model-checker is used to determine intermediate automata that violate LTL specifications. The inference engine then uses counter-examples to refine the automaton accordingly.

4.2.2 Transformational Approaches

A large number of approaches [21], [25], [30], [41], [42], [52], [53], [54], [55] do not rely on any off-the-shelf tools, but rather apply a series of custom transformation, filtering, and clustering techniques on data extracted by static and/or dynamic analysis.

The static technique by **Whaley et al.** uses interprocedural analysis and constant propagation to find call sequences to methods that may establish conditions of predicates that guard `throw` statements. The underlying assumption is that programmers of languages with explicit exception handling make use of defensive programming: A component's state is encoded in state variables; state predicates are used to guard calls to operations and cause exceptions to be thrown if satisfied. These sequences are considered illegal and their complement with regard to the set of methods, the sequencing model of which is being analyzed, forms a model of accepted transitions.

While only predicates with simple comparisons of a field with null or constant integers are supported by **Whaley et al.**, **Weimer and Necula's** approach also considers multiple fields and inlines Boolean methods. **Scenariographer** follows a different path to mine sequencing models for API calls. It tries to recognize patterns among strings of symbols representing API elements in dynamically recorded Method Invocation Sequences (MISs) by using the Levenshtein edit-distance to compute bounded canonical sequences, which are subsequently combined and generalized into regular expressions by detecting the longest common subsequence.

Some approaches [38], [53], [54] use control-flow analysis to either derive initial specification FSAs, which are then processed in various ways [38], [53], or group dynamic trace events [54]. **Quante and Koschke** use Object Process Graphs (OPGs)—a projection of an interprocedural flow graph specific to one object, very similar to **JADET's** method models—to represent actual behavior extracted statically or dynamically from a program. Given OPGs for instances of a component, that component's protocol is recovered by performing a set of transformations to eliminate recursion and to merge OPGs into a DFA, which is then minimized. The approach optionally supports additional transformations, depending on the degree of desired generalization.

Unlike the object usage representations of **JADET** and **Quante and Koschke**, the graph-based object usage models used by **GrouMiner** capture the interplay between multiple objects and include control flow structures. The authors' own subgraph matching algorithm (PattExplorer) is based on an observation similar to that of the Apriori association rule mining algorithm: The subgraph of a frequent (sub)graph must also be frequent. Thus, PattExplorer works by building larger candidate patterns from smaller discovered patterns.

Pradel and Gross group dynamic events by their caller method into "object collaborations"—sequences of (o, m) pairs, where o is the receiver and m the method signature. Collaborations are split into object roles—unordered sets of methods called on the same object. Next, techniques are applied to abstract over irrelevant differences between collaborations, facilitating their clustering into patterns. Roles are projected onto the most general type providing their methods; objects that play the same role are merged into one artificial object, and collaborations are split such

that one collaboration only contains methods from the same package. The resulting collaborations are clustered into patterns whenever their objects have the same roles, and patterns are filtered out if they have many objects or occur rarely in traces and method bodies. Finally, collaboration patterns are mapped to PFSA by mapping methods to states and connecting them by an edge if they are called consecutively.

Whaley et al. and **Dallmeier et al.** use static analysis to distinguish between the state-preserving (inspectors) and state-modifying (mutators) methods of a class, but use this information in different ways. Whaley et al. do the classification individually per each class field, and the methods are instrumented to record their type. Training client programs are then executed with the instrumented methods and observed sequences of external state-modifying calls are recorded with FSAs (one per field) as they occur. State-preserving method calls are not recorded as nodes in the FSAs, but rather associated with the states in the corresponding FSA. Dallmeier et al. instrument mutators to call all inspectors before and after their execution. As a result, the execution of the program protocols transitions are of the kind s_1 mutator s_2. In a further step, concrete values in s_i are mapped to abstract domains, and the abstract state machines of individual instances of a class are merged into a state machine for that class. In follow-up work, Dallmeier et al. even designed a novel tool, TAUTOKO, that leverages test generation techniques to cover unobserved program behavior and thereby extend the inferred models with additional transitions [74].

Shoham et al. use abstract interpretation with a combined domain for aliasing (the heap abstraction) and event sequences (history abstraction) to collect summarized abstract API usage event traces for objects of a particular type. The history abstraction is represented as automata. The analysis is parameterized by a heap abstraction (flow-insensitive versus flow-sensitive) and by a merge operator for the history abstraction (merge all traces reaching a program point versus merge only traces that share a common recent past). Automata clustering and unification are exploited to reduce noise in the collected traces.

Lorenzoli et al. and **Lo et al.** [52] mine expressive classes of specifications encoded in extended FSAs. The GK-tail algorithm [47] produces FSAs annotated with data constraints (called Extended FSAs, or EFSAs). The algorithm first merges traces that only differ by their data values, abstracting from those values through predicates (produced by Daikon [3]). Next, it generates an EFSA by first embedding the input traces through transitions and states, and then subsequently merging states that fulfill some equivalence relation. Lo et al. infer quantified binary temporal rules with equality constraints (QBEC) by performing a set of sophisticated custom processing steps on input traces for inferring temporal properties and combines those with frequent item set mining for the inference of statistically relevant qualified properties.

Prospector and **PARSEWEB** both infer *source* $\rightarrow$ *destination* paths but use different approaches. Prospector builds a signature graph to find legal paths. Nodes in this graph are types and edges represent ways of getting from one type to another, e.g., via field accesses, method outputs, inheritance relations, or downcasts. Legal downcasts for the API are found by mining the API's client code and are used to enrich the signature graph with new edges, resulting in what is called the jungloid graph. Knowledge about legal downcasts is important as otherwise too many call chains that exist in practice will be missed. PARSEWEB uses examples fetched from the Google Code search engine to extract DAGs that record the control flow between statements that result in a transformation from one object type to another (method invocation, constructor, casts). Signatures of these methods are used to annotate the DAG nodes as sources or targets. Subsequently, frequent method invocation sequences that can transform an object of the source type to an object of the destination type are extracted by calculating the shortest path from a source node to a destination node. The MISs are then clustered using a custom clustering algorithm and are ranked based on frequency (higher frequency means higher rank) and length (shorter lengths mean higher rank).

4.2.3 Template-Based Techniques

Template-based techniques discover instances of a fixed set of pattern templates. The rationale is that the specifications of interesting properties follow a few well-defined patterns [7]. Pattern matching has its cost, which requires the size and the alphabet of the specification patterns to be restricted. Hence, most approaches target two-letter patterns.

Liu et al. (**LtRules**) [31], [32] instantiate rule templates by replacing symbolic function names in templates with concrete API function names and by feeding the instantiated rules to the BLAST model checker to filter invalid ones.

To find matching event pairs for the alternate pattern, **Perracotta** uses a matrix $M[n \times n]$, where n is the number of events in a dynamic trace and $M(i, j)$ stores the current state of a state machine that represents the alternating pattern for the pair of events (i, j). When an event x is read from the trace, both the row and column corresponding to $\mathrm{index}(x)$ of M are updated. At the end of the trace, the inferred properties are those whose corresponding state machines are in accepting states. This core technique is complemented with a set of heuristics to cope with imperfect traces and to increase precision as well as a chaining technique to combine two-letter alternating patterns to alternating chains.

Gabel and Su [51] improve the scalability of pattern matching approaches to support the mining of three-letter patterns of the form $(ab^*c)^+$ by employing Binary Decision Diagram (BDDs). The BDD-based algorithm is also used in their subsequent work on **Javert**, which builds up arbitrary large temporal specifications by chaining instances of a predefined set of patterns. In their more recent work on online inference, **Gabel and Su** [60] use a sliding-window queue approach over dynamic sequences of API method calls during the program's execution to mine two-letter regular expressions.

Doc2Spec combines specification template instantiation with the *Named Entity Recognition (NER)* NPL technique to extract specifications from JavaDoc documentation. Using the JavaDoc method descriptions, NER extracts action-

resource code element pairs, whereby actions are mapped to predefined categories: creation, manipulation, closure, lock, and unlock, eventually using a synonym dictionary. Using the class/interface information from the JavaDoc documentation, for each resource type the set of action categories is generated, including both those extracted directly from that class' methods and from its superclasses. Each such group is rendered into an automata based on predefined specification templates.

Zhong et al. [50] propose the **Java Rule Finder** to mine instances of two-letter patterns from source code of Java libraries. Patterns are of the form a must (not) precede b, and instances of those patterns are encoded in a so-called Program Rule Graph. JRF only analyzes the code of the API at hand; it requires no access to client code.

As a final example of the use of the pattern-matching strategy, **Acharya et al.** [28] statically analyze an API's client code to detect instantiations of a set of manually specified generic temporal properties of abstract operations on the output of an API call, e.g., checking the return value or checking error code variables.

4.3 Empirical Findings

Most of the techniques described in this section were the object of empirical evaluation, but much variety exists in the specific focus of the evaluation. Authors sometimes evaluate the performance of the approach as a whole, the impact of specific design decisions of the approach, the meaningfulness of the results, or any combinations of these factors. Given the number of approaches discussed in this section, we provide a general road map of the empirical evaluation conducted and highlight noteworthy results.

4.3.1 Runtime Performance Evaluation

Roughly half of the surveyed papers report on *the performance of the respective approaches/tools* [21], [24], [30], [34], [35], [43], [46], [48], [51], [52], [54], [59], [60], [61]. Most papers report performance numbers for selected target systems to show that the technique is fast enough to be of practical use [24], [30], [35], [51], [59], [60], [61]. In other cases, the authors try to empirically [21], [34], [46], [54] or analytically [52] correlate the size of the input data or features of the algorithm to the time needed for mining or violation detection. **Lo et al.** [48] provide a noteworthy instance of performance evaluation in that their experiments to evaluate the scalability of the approach are performed not only on synthetic data, but also on standard datasets used as benchmarks for mining algorithms in the data mining community.

4.3.2 Design Decision Assessment

Also about half of the works surveyed in this section attempt to assess the *contribution of distinct features of the respective algorithms to the overall technique* [20], [21], [24], [27], [28], [34], [35], [46], [48], [51], [54], [56], [57], [60], [61].

Most of them do so by finding evidence for the usefulness of certain components of their algorithms [20], [24], [27], [28], [35], [39], [40], [51], [56], [57], [61]. For example, **Gruska et al.** discuss examples of detected anomalies to show that some issues can only be found by a cross-project analysis. **Ramanathan et al.** analyze the distribution of the size of the set of mined properties to indicate that the inferred properties are readable and understandable by developers. **Lo et al.** [52] indirectly justify the proposed algorithm's features by showing that a significant portion of the rules integrated into Microsoft's Static Driver Verifier fall in the class of quantified temporal properties with equality constraints.

Other authors implement several versions of their algorithm with different feature sets [21], [34], [39], [40], [47], [48], [51]. **SMARTiC**, for instance, is evaluated with different configurations to show how the proposed techniques actually contribute to improving accuracy or to increase stability against erroneous traces, and how they affect the scalability of the approach. **Gabel and Su** [51] compare two versions of their BDD-based algorithm, a precise one and an approximate one, with an algorithm with explicit state tracking which corresponds to the algorithm proposed by **Yang et al.** The latter is selected as a representative of other approaches [2], [27], [35], [40]. The comparison shows that the BDD-based algorithm outperforms the explicit state tracking algorithm in terms of time and space when mining three-letter patterns. **Walkinshaw et al.** compare the version of their tool that uses a model checker as an automated oracle with the version based on user queries only [44], and show that the number of user queries can be significantly reduced.

Others empirically correlate thresholds or characteristics related to certain features to observations about the results such as the false positive ratio or scalability [39], [40], [46], [54]. In particular, **Lo et al.** [48] show that their algorithm is statistically sound and complete, i.e., 1) all mined rules are significant and 2) all significant rules are mined, for some given statistical significance threshold and assuming that input traces are complete and representative.

4.3.3 Result Quality Evaluation

The quality of mined specifications is evaluated with respect to their validity and their effectiveness in serving the goal for which they are mined.

In a large number of papers [20], [21], [24], [25], [27], [28], [30], [35], [36], [41], [42], [43], [46], [47], [48], [51], [52], [53], [54], [60], the quality of the mined API specifications is judged by discussing example specifications and by manually inspecting the code.

The number of the examined specifications varies greatly, but is generally only a fraction of the actually mined specifications. This selectivity leads to varying degrees of threats to the generalizability of the findings. For example, 4 out of 1,100 generated models are discussed by **Dallmeier et al.** Only four specifications mined for one particular function of the X11 API are manually classified as real or false by **Acharya et al.** [35]. A thorough validation of the mined specifications is performed for **Perracotta**; all 142 mined properties that remain after several heuristics are applied are evaluated.

A more objective way to judge the quality of the mined specifications is by comparing them to specifications that are explicitly documented. **Ammons et al.** compare specifications mined for the X11 API with those documented in a manual. **Weimer and Necula** compare mined and documented specifications for three selected classes of the Hibernate framework; **Lo et al.** [48] check the mined multi-event rules

for the JBoss transaction component against the JBoss documentation. **Yang et al.** also compare one of the specifications mined by **Perracotta** for JBoss with an object interaction diagram from the J2EE (JTA) specification.

A different strategy to judge the quality of the mined specifications is to rely on some stated properties of the specifications. For instance, **Lo et al.** [48] define several properties of temporal rules mined by their approach, such as significance (defined by support and confidence thresholds), monotonicity, and redundancy, to prune the search space. **Pradel and Gross**, who mine specifications for JDK using traces from 10 DaCapo benchmarks, consider as nonaccidental only those that are mined from traces of more than one benchmark (35 percent). **Gabel and Su** [46] classify as real specifications only those sequences whose events are connected by data flow.

Relatively few papers quantify the false positive/negative ratio for the mined specifications [28], [39], [40], [59]. On the contrary, detected violations are generally classified more precisely by quantifying the false positive ratio [43], [36], [59], [57], [58], [59], [61]. As with specifications, the judgment is often based on code inspection. **Thummalapenta and Xie** [56] undertake an additional step and verify their classification by submitting bug reports for the defects found and reporting feedback from developers.

As also noticed by Lo and Khoo [75] and more recently by Pradel et al. [76], manual judgment of the effectiveness of specification miners is subjective and introduces threats to the validity of the evaluation. Yet, very few papers describe an effort to avoid this risk by automating (part of) the evaluation process. These include work by **Lo and Khoo**, **Lo et al.** [48], and **Quante and Koschke**. To evaluate **SMarTic**, **Lo and Khoo** use the evaluation technique proposed by the same authors in previous work [75]: A given specification is used to construct clients that exercise the specification to generate "correct traces." The latter are used to mine an automaton that serves as the reference against which to compare the automaton mined by arbitrary traces in terms of precision, recall, and cosimulation. In their work, **Lo et al.** [48] run the mined rules and buggy software through a model checker, judging the quality of the mined rules by their capability to discover the known bugs.

Among all approaches surveyed in this section, only **Prospector** is evaluated in a user study to gain insights into the actual effect of the approach on the quality of the development process.

The evaluations surveyed often do not include an empirical comparison to similar approaches. Such a comparison is indeed not easy to perform. Approaches often address slightly different problems, make different assumptions about the input data, and provide results in different formats. Also, experiments are often performed on different datasets; the criteria for classifying results is different and often not described precisely. Finally, often competing tools are not available for use in experiments. Hence, when an empirical comparison is performed [24], [25], [27], [34], [36], [48], [51], [56], [57], related approaches are generally simulated/reimplemented to enable a comparison on the same dataset. **Weimer and Necula**, for instance, use three Hibernate classes for which specifications are available to perform a qualitative comparison of their tool against five other approaches [2], [20], [21], [24] which make different assumptions and produce different kinds of specifications. **CAR-miner** is empirically compared to Weimer and Necula's technique. **Quante and Koschke** use the evaluation technique by **Lo and Khoo** to perform an automated comparative evaluation of several versions of their technique against basic approaches to protocol learning such as the k-tails learner [77] and the sk-strings learner [67]. Unlike the previous approaches, the results created by **PARSEWEB** are compared against results directly obtained from **Prospector** and Strathcona [4].

4.3.4 Reported Accuracy

The level of rigor employed in evaluating approaches to inferring sequential usage patterns varies. Perhaps not surprisingly, most of the time only results that are at least indicative that some valid knowledge was found in an enormous space of possibilities are reported. There is, however, no convergence on metrics, such as false positive rates, that assess the quality of the properties inferred. Overall, when such metrics are reported, the accuracy is low, especially for flagged violations detected from mined specifications.

We observe that more accurate results are reported for approaches that use transformational techniques. For instance, **Weimer and Necula** report a much lower false positive rate (0.08-0.13) of the detected anomalies than approaches using off-the-shelf learners, whose false positive rates vary between 0.30 (**Ammons et al.**), 0.60 (**Whaley et al.**), 1.0 (**JIST**), and between 0.51 and 0.93 among top-10 ranked anomalies (**JADET**). Likewise, low precision is reported for **Tikanga** (between 0.16 and 0.80 when inspecting the top 25 percent of entries) and by **Gruska et al.** (only 0.22 for the top-ranked 25 percent of the detected anomalies).

As expected, the results of transformational approaches improve with increased precision of the underlying analysis, as indicated by comparing the false positive rate of **Weimer and Necula**'s tool with **JIST**. Yet, the real value of the improvements cannot be objectively appreciated without systematically evaluating the performance/scalability cost eventually implied by the increased analysis precision.

Significant improvements seem possible for techniques using off-the-shelf learners when clever filtering and clustering is employed. This phenomenon is illustrated by the comparison of the metrics for **SMArTiC** and the approach by **Ammons et al.** (0.50 false positives and 0.01 false negatives versus 0.80 false positives and 0.50 false negatives) [34]. Also, looking for specific kinds of specifications such as preconditions [40], alternative patterns [56], [57], and conditional specifications [28] seems to lower the noise. For example, **Ramanathan et al.**, **Thummalapenta and Xie** (both **Alattin** and **CAR-Miner**), and **Acharya et al.** [29] report lower false positive rates for detected anomalies than more general approaches

The accuracy of approaches using template-based techniques is also low. For instance, **Yang et al.** report that only roughly 40 percent of the properties inferred by **Perracotta** are "useful." Likewise, for **Javert**, **Gabel and Su** report a false positives rate of approximately 0.50, which they argue to be still low relative to previous approaches. While **Zhong et al.** [59] report good precision and recall for the specifications mined by **Doc2Spec** (at around 0.80),

using them to find bugs produces roughly 74 percent of false positives.

5 Behavioral Specifications

A number of approaches have been developed to infer API properties that describe not how API elements can be combined into usage patterns, but rather the valid and invalid *behavior* of the API.

5.1 Overview

The two main points of distinction between the approaches surveyed concern the nature and complexity of the specifications, and the goal of the approach.

5.1.1 Nature and Complexity of Inferred Specifications

One form of behavioral description is through *contracts*, such as preconditions, postconditions, and invariants defined over an abstract data type or class [78]. A typical example of a precondition is that a value passed as an argument to a function should not be a null reference. We consider that design contracts constitute a form of behavioral description of the API in that they state something about the behavior of the API if the contract is either met or (more typically) not met. **Houdini** is a tool for inferring basic invariants, preconditions, and postconditions about the methods of a Java program [79]. **Chronicler** infers control- and data-flow preconditions for functions in C/C++ function calls [39].

A richer form of behavioral description is that of contracts to which an action is attached that describes the consequence of accessing an API without respecting the contract. For example, **Axiom Meister** infers properties, called *axioms*, such as [80, p. 718]

```
requires key! = null
    otherwise ArgumentNullException
```

Taken to a further extent, properties concerned with behavior in erroneous conditions can describe only the preconditions that lead to exceptional behavior. In particular, **Buse and Weimer**'s exception documentation reverse-engineering approach [81] produces for each method/exception-type pair a *path predicate* that describes constraints on the values of the method's variables that will result in a control-flow path ending in an exception of that type to be raised. Specifically, Buse and Weimer's approach infers, for a given API method, 1) possible exception types that might be thrown (directly or indirectly), 2) predicates over paths that might cause these exceptions to be thrown, 3) human-readable strings describing these paths. Although in many cases variables in such path predicates will correspond to private implementation structures of the framework or library, the approach also produces cases that describe conditions on parameters which are relevant to API users.

Finally, descriptions of API behavior can take the form of an elaborate formal language to express facts about the behavior of the API. This approach is illustrated by the work of **Henkel and Diwan** [82], [83], who infer *algebraic specifications* [84] or equations expressing truths about call sequences to the public methods of classes. Essentially, an algebraic specification consists of a series of public method calls to an object of a given class, with symbolic variables storing the states of the object and the returned values from those method calls. For example, for a stack, one algebraic specification might say that a call sequence `x = new Stack(); x.pop();` will throw an exception. Another might say that the overall state of `x` is unchanged after a call chain `x.push(y); x.pop();`. These algebraic specifications are similar to axioms and path predicates in that they provide information about uses of the API that will result in exceptional or erroneous behavior. This is different from the temporal properties described in Section 4, which describe sequences of operations with little focus on program state. The approach of **Ghezzi et al.** [85], implemented in a tool called **SPY**, also produces rules describing valid transitions in a stateful model of the API's behavior. In their case, the final specification is called *intensional behavior model*.

5.1.2 Goal

The two main lines of arguments used to motivate the work surveyed in this section are as follows: First, manually authored API documentation is incomplete or wrong and should be complemented with the output of automated techniques. We refer to this goal as a general *documentation goal*. Second, practical use of automated verification tools requires numerous program annotations (such as preconditions) that are too onerous and error-prone to produce manually, so these annotations should be automatically generated. Although annotations are also a form of documentation, we refer to this goal as *bug detection* because the ultimate use of the annotations is as input to verification tools.

Three of the approaches we surveyed produce output exclusively for documentation. Claiming that the documentation of library classes is not always clear, **Henkel and Diwan**'s goal is to complement natural-language documentation with automatically-derived formal specifications. Similarly, according to **Ghezzi et al.**, the output of **SPY** "may help understand what the component does when no formal specification is available" [85, p. 430]. The goal of **Buse and Weimer**'s approach is to discover information about the potential causes of exceptions to better describe exception clauses in API reference documentation. In all three cases, the approach is not used for automated bug detection.

Although their general claimed goal is to produce human-readable specifications, both **Tillmann et al.** and **Ramanathan et al.** also use their inferred specifications to detect bugs.

Finally, the goal of **Houdini** is to automatically produce *annotations* for the ESC/Java static program checker [86]. ESC/Java requires manually specified input in the form of annotations describing invariants or pre/postconditions. It can then check whether a program (or module) is consistent with these annotations. The motivation for Houdini was to partially automate the otherwise tedious and difficult task of writing annotations for ESC/Java.

Ideally, specifications produced to help programmers program correctly should also be usable for automated bug detection. Unfortunately, we do not observe that this is the case in practice. In the works surveyed, there exists an inverse relation between the complexity of the specification inferred and their use in verification. In Table 4, only the authors of approaches that produce the simplest types of

TABLE 4
Works Surveyed: Behavioral Specifications

Work	Tool	Goal	Property	Input	Analysis
Flanagan & Leino 2001 [79]	Houdini	Bug Detection	Contract	API Code	Symbolic Exec.
Henkel & Diwan 2003–2007 [82], [83]	–	Documentation	Spec. Language	API Interface	Dynamic
Tillmann et al. 2006 [80]	Axiom Meister	Doc. & Bug Detection	Contract + Action	API Code	Symbolic Exec.
Ramanathan et al. 2007 [39] (*see Table 3*)	Chronicler	Doc. & Bug Detection	Contract	Client Code	Static
Buse & Weimer 2008 [81]	–	Documentation	Exception Path	API Code	Static & Symbolic Exec.
Ghezzi et al. 2009 [85]	SPY	Documentation	Spec. Language	API Interface	Dynamic

specifications (contracts and their variants) report using them for bug detection.

5.2 Analysis Techniques

Five of the six approaches surveyed in this section infer properties through various analyses of the API source code (or simply its interface). In this way, they contrast with most other approaches, which instead mine artifacts that use the API.

We distinguish between two general strategies used to generate behavioral specifications of APIs. With the *conjecture/refute* strategy, the idea is to synthetically generate a large number of potential specifications, and then to use different analyses to refute, or invalidate, specifications unlikely to hold in practice. We note that the conjecture/refute strategy is not unique to behavioral specification inference, but was also used by more general invariant detection techniques and, in particular, Daikon [3].

The second strategy is to systematically explore the behavior of an API method or function by symbolically executing it. Symbolic execution [87] attempts to explore all possible execution paths, whereby each path is characterized by a set of constraints on the inputs (the method's arguments and the initial state of the heap) called the *path condition*. A path may terminate normally or have an exceptional result.

5.2.1 Conjecture/Refute Strategy

Houdini, **Henkel and Diwan**'s approach, and **SPY** follow the conjecture/refute strategy. To generate conjectures, Houdini systematically generates a predefined set of conjecture templates for all types and routines declared in the target program. For example, for a field `f` holding a reference type, the invariant `f != null` is conjectured. For a field of integral type, a richer set of value-comparison invariants is generated using six comparison operators ($<$, $>$, etc.), and a set of expressions including all fields declared before `f`, as well as *interesting values* such as 0, 1, etc. The conjecture generation approach of Henkel and Diwan and SPY is designed to systematically explore the state space of the class for which specifications are generated. Their respective approaches, therefore, create an instantiation transition for the class (a constructor call), and then systematically construct a sequence of invocations of increasing complexity on this instance. There the two approaches diverge: While Henkel and Diwan's focuses on the detection of invalid sequences, the goal of SPY is to model behavioral equivalence of method-call sequences on objects and analytically transform this model into a specification.

To invalidate the conjectured invariants, **Houdini** simply adds all annotations to the target program and invokes the ESC/Java checker. Refuted annotations are removed and the process is repeated until a fixed point is reached. In contrast, **Henkel and Diwan**'s tool and **SPY** directly execute code corresponding to the synthetically generated transitions. In the case of Henkel and Diwan's tool, when executions result in an exception, the corresponding sequence is flagged as invalid.

5.2.2 Symbolic Execution Strategy

Axiom Meister and **Buse and Weimer**'s approach both use symbolic execution to infer behavioral knowledge about the target API element. Axiom Meister's symbolic execution produces a set of path conditions and resulting program state, which are then transformed into the final specifications using a process called *observational abstraction*. The idea is to convert path conditions expressed using implementation-specific constraints (e.g., on local variables) into statements expressed only in terms of the class's interface. The use of symbolic execution by Buse and Weimer is similar, with two notable exceptions. First, their approach only identifies paths that result in an exception. Second, they do not abstract the constraints into higher level abstractions; instead, they use a heuristic process to translate the path conditions into more human-readable statements. For example, the constraint `x instanceof T` becomes "`x` is a `T`" [81, p. 276]

5.2.3 Other Strategy

Chronicler's approach, described in Section 4.2, uses a combination of mining and static analysis techniques of client code. In this way it is different from the other approaches described in this section, which only analyze the API code itself. However, in terms of the two strategies described above, it is more closely related to the symbolic execution strategy, given its use of constraint propagation as an underlying technique to infer preconditions.

5.3 Empirical Findings

Evaluations of behavioral specification inference approaches have focused on the feasibility and scalability as well as the quality of the specifications produced.

5.3.1 Feasibility and Scalability

The reports on the empirical assessment of **Houdini**, **Henkel and Diwan**'s approach, and **Axiom Meister** focus on the general feasibility of the tools using a few small or medium-size API classes as input. Hence, the reported results are mostly useful to provide a general "feel" for how the tools operate. The main lesson from the reported

TABLE 5
Works Surveyed: Migration Mappings

Work	Tool	Mappings	Input	Technique
Dig et al. 2006 [91]	RefactoringCrawler	Refactorings	API Snapshots	Sim. Metrics + Ref. Analysis
Xing & Stroulia 2007 [92]	Diff-CatchUp	Refactorings	API Snapshots + Client Code	Sim. Metrics + UMLDiff [93]
Schäfer et al. 2008 [94]	–	1-to-1	Client Code	Association Rules
Dagenais & Robillard 2008 [90], [95]	SemDiff	1-to-1	Change History	Conf. Metrics
Wu et al. 2010 [96]	AURA	1-to-m, n-to-1	API Snapshots	Sim. Metrics + Conf. Metrics
Zhong et al. 2010 [97]	MAM	m-to-n	Client Code	Sim. Metrics + ATGs
Nguyen et al. 2010 [98]	LibSync	Edit Scripts	Client Code	GROUMs + Frequent Itemsets

TABLE 6
Works Surveyed: Other Approaches

Work	Tool	Goal	Input	Property
Holmes & Walker 2007 [102]	PopCon	Doc. & Understand.	Client Code	Popularity
Saul et al. 2007 [105]	FRAN	Doc. & Understand	API/Client Code	Related elements
Thummalapenta & Xie 2008 [100]	SpotWeb	Doc. & Understand.	Client Code	Popularity
Stylos et al. 2009 [101]	Jadeite	Doc. & Understand.	Web pages	Popularity
Mileva et al. 2009 [103]	Aktari	Doc. & Understand.	Client Projects	Popularity
Long et al. 2009 [104]	Altair	Doc. & Understand.	API Code	Related elements

experience with Houdini is that using a program checker to systematically evaluate a large number of synthetically generated annotations is marginally scalable. For example, to apply Houdini to a 36,000 LOC program requires 62 hours (or 6.2s/LOC) [79, p. 512]. Henkel and Diwan's approach requires time in the same order of magnitude, with about 13.3-15.3s/LOC (depending on optimization parameters).[2] In contrast, the tools based on symbolic execution show at least one order of magnitude faster runtime performance, with Axiom Meister requiring about 0.0035-0.4662s/LOC and **Buse and Weimer**'s approach 0.0012-0.011s/LOC (depending on the class or project tested).[3] In a nutshell, we see that conjecture/refute-based systems do not quite meet the challenge of being able to complete the analysis of a reasonably sized system in the 12-16 hours between work days [79], while in the case of the two approaches based on symbolic execution, this goal can be achieved.

5.3.2 Specification Quality

Henkel and Diwan also assess the quality of the algebraic specifications produced for 13 small classes (eight of which were created by the authors) by manually inspecting the axioms produced and evaluating their correctness. Overall, the approach is found to produce directly usable correct axioms in 76-98 percent of the cases. For example, for Java's `LinkedList` class, the approach produces 139 axioms, 24 of which are judged correct but imperfect (too general or too specific), and one of which is incorrect. **Tillmann et al.** also manually assess the output of **Axiom Meister**, and also report a very low number of false positives (four invalid axioms over 88 axioms generated for six classes). **Ghezzi et al.** compare their approach, **SPY**, with that of Henkel and Diwan in terms of the correctness of the specifications generated for a number of simple container classes. They find that both approaches produce no false specifications for their benchmarks, but that Henkel and Diwan's approach is unable to produce specifications in 0-64 percent of test cases (depending on the target API class), whereas SPY is able to generate specifications in all cases. They conclude that SPY is more generally applicable.

Buse and Weimer evaluate the quality of their generated exception documentation by applying their tool to 10 medium-sized open-source Java programs and comparing the output of their tool with the exception documentation already present in the source code of their target systems (a total of 951 documented exceptions across all 10 systems). The authors find that the generated exception documentation was the same as or better than the existing documentation in 85 percent of cases where human readable documentation was present, according to a manual assessment.

6 Migration Mappings

The techniques discussed in this section infer migration mappings to support updating clients of an API when the API evolves with backward-incompatible changes. Alternatively, mappings can also be inferred between different, but equivalent, APIs.

6.1 Overview

Migration mappings link elements declared by one API with corresponding elements in a different API, or in a different version of the same API. Table 5 lists approaches for inferring migration mappings. Table 6 lists other approaches.

6.1.1 Goal

The techniques surveyed in this section aim to discover a *mapping* (I, R, G), which consists of an initial set I of elements in the current API version and their replacements R in a different version. Optionally, some optional migration guidelines G can be provided to give further instructions on how to replace the references to elements in I with references to the elements in R. Guidelines can take the

2. We estimated this value from the Hashtable class, using 578 uncommented nonblank LOC.

3. The runtime performance is a function of the limits imposed on the symbolic execution, e.g., with respect to loop unrolling and recursion. We use the numbers reported by the authors as indicative of the configuration in which they expect the approach to be used.

form of code examples or edit scripts (series of steps required to update a client). Because the goal of all approaches surveyed is the same, we do not have a column to that effect in the table.

We note that techniques for inferring mappings between API versions provide a solution to a specific subproblem of the general problem of inferring knowledge from software evolution data. For example, techniques have also been developed to discover the origin of a section of code [88], or to infer transformation rules describing the evolution or refactoring of an API [89]. These techniques fall outside the scope of our survey as described in Section 1. A detailed review of such approaches is available in a separate publication [90].

6.1.2 Nature of Mined Mappings

We can roughly categorize migration mapping inference techniques based on the cardinalities of the mappings they infer.

The most basic mappings are those that correspond to simple API refactorings, such as renaming API elements or moving them to a different module. Such minor changes have been shown to account for roughly 85 percent of all changes breaking backwards compatibility in an API's evolution [99]. These refactorings are discovered by API-refactoring-detection tools such as **Xing and Stroulia**'s **Diff-CatchUp** [92] and **Dig et al.**'s **RefactoringCrawler** [91].

Other techniques go beyond standard refactorings by discovering more general mappings between API elements. **Dagenais and Robillard**'s **SemDiff** [95] and an approach by **Schäfer et al**. [94] discover mappings for a single input API method (SemDiff) or an input class, field, or method (Schäfer et al.). **Wu et al.**'s **AURA** [96] furthers the state of the art by detecting one-to-many and many-to-one method mappings between versions of an API. **Zhong et al.**'s **MAM** technique [97] detects one-to-one type mappings and many-to-many method mappings between two equivalent versions of an API, where one version is implemented in Java and the other one in C#.

Finally, some techniques also present some kind of migration guidelines that illustrate how references to the current element can be replaced by references to its target element(s). **Nguyen et al.**'s **LibSync** [98] falls in this category by discovering so-called edit scripts.

6.2 Inference Techniques

Mapping inference techniques can be distinguished by the general strategy they use to infer mappings, the nature of the input to the technique, and their main choice of algorithm for implementing the technique. In the case of implementation design, we distinguish between the use of custom heuristics versus general data-mining techniques.

6.2.1 General Strategy

Two main strategies are employed by mapping inference techniques. A first strategy is to analyze the pairwise textual similarity between elements in two versions of an API. The intuition is that if two elements only vary slightly in their name, one might be an evolved version of another. For example, `openFile` in one version of the API might map to `open_file` in a different version.

A second, and more complex, strategy assumes the presence of two equivalent versions of a source code context using two different versions of an API. For simplicity, we will refer to *old* and *migrated* code. The intuition underlying this strategy is that if the API usage context C_o in the old version references a set of API elements from the old API version, and its equivalent usage context C_m within the migrated version references some other set of elements from the target API version, then the set difference between C_m and C_o will contain the mapping between elements of the two versions of the API (along with some noise). As a simplistic example, if a `main` method in one version of a client program calls `open` and `close` and a later version of `main` compiled with a new version of the API calls `open2` and `close`, then it is likely that `open2` replaces `open`. In this case, the context is the `main` method.

Most of the approaches we surveyed combine these two strategies, but in very different ratios. Early refactoring detection approaches (**RefactoringCrawler** and **Diff-CatchUp**) rely primarily on textual similarity to generate a list of mappings. In contrast, **SemDiff** relies almost exclusively on the second strategy, reverting to textual similarity analysis only to break ties in ranked recommendations. The other approaches combine textual similarity and the other strategy in a more balanced way.

6.2.2 Input

From the perspective of the input to the mapping analysis techniques, there are three categories of approaches. Our categorization relies on the difference between considering all intermediate versions of source code contained in a revision control system (the *change history*), or only fully built versions of a software system (which we call a *snapshot*).

We distinguish between techniques that analyze entire *snapshots* of the API, techniques that analyze client snapshots (or a combination of API and client snapshots), and techniques that analyze an API's change history. The first category includes **RefactoringCrawler** and **AURA**. The second category includes **Schäfer et al.**'s approach, **LibSync**, and **Zhong et al.**'s **MAM**. **Diff-CatchUp** is actually a hybrid approach because it uses primarily API snapshots as input. However, if the results are not satisfactory, client snapshots can be analyzed as well. **SemDiff** constitutes its own category since it does not analyze API or client snapshots, but rather the change history of the API's clients.

6.2.3 Custom Heuristics

A subcategory of approaches that use custom heuristics consists of **RefactoringCrawler**, **Diff-CatchUp**, and **MAM**, which apply *text similarity metrics* to the signatures of API elements. RefactoringCrawler and DiffCatch-Up infer mappings between refactored versions of an API, whereas MAM finds mappings between APIs implemented for Java and C#.

After identifying the most similar API element pairs by performing a syntactic comparison between all API element pairs across the two API versions, **RefactoringCrawler** validates each refactoring candidate (e_o, e_m) by checking whether references to and from the old element e_o are similar to the references to and from its candidate replacement element e_m. Due to heavy use of similarity

metrics, in cases where e_o is not simply refactored into a single other element e_m, RefactoringCrawler will most likely not infer an appropriate mapping for e_o.

Diff-CatchUp uses mappings detected by UMLDiff, an earlier refactoring-detection tool [93] that detects refactorings through a hierarchical pairwise comparison of the programs' packages, classes, methods, and fields in program snapshots. If a user deems the detected refactorings to be invalid, Diff-CatchUp can analyze user-provided client code to identify additional one-to-one replacement candidates. **Diff-CatchUp** ranks candidates based on four similarity metrics that assess the candidate's name, inheritance relations, references, and associations to other elements.

For an existing pair of projects P_J, P_C that implement the same functionality in Java and C#, **MAM** first aligns elements in P_J and P_C using textual similarity. It then builds and compares data dependency graphs for each aligned client method pair (M_J, M_C). Essentially, MAM finds a mapping if cohesive groups of methods invoked within M_J and M_C share similar names and if all of their input and output variables types were aligned in the previous step.

SemDiff does not relate API elements by the syntactical similarity of their signatures. It rather identifies possible candidate replacements for a method m_o by analyzing the change history of a client of the API that has already been migrated within a self-hosted corpus. SemDiff makes the hypothesis that, generally, calls to deleted methods will be replaced in the same change set by one or more calls to methods that provide a similar functionality. Having identified such change sets, SemDiff then ranks all detected replacements for m_o with respect to a popularity-based metric, as well as name similarity, and presents these as a ranked list to the user.

AURA is a hybrid approach that combines textual similarity analysis of API element signatures similar to that used by **RefactoringCrawler** with call dependency similarity analysis. Unlike **SemDiff**, AURA does not analyze call dependencies in the change history but rather in two snapshots of the API. As opposed to simply returning a ranking list of potential replacements for a method, AURA takes steps to identify likely coreplacements, or groups of methods that replace a single target method. Once a main replacement is identified, AURA finds potential co-replacements by finding references that were frequently co-inserted along with the main replacement.

6.2.4 General Data Mining

Schäfer et al.'s approach collects API element references in the old version of clients, determines how those element references were replaced in the new versions of the clients, and populates a database with all possible one-to-one usage replacement rules that might be derived from these replacements. Association rule mining is applied on the rule database and high-quality rules are returned. The tool then uses text similarity metrics to filter out multiple rules with the same antecedent.

LibSync represents API usages as directed acyclic graphs called GROUMs that basically capture reference- and inheritance-based usage of API methods and types, as well as various control structures and dataflow dependencies between them. Essentially, given some mapping $I \rightarrow R$, LibSync first identifies GROUMs describing usages of the elements in I in the old versions of client code, and then computes edit scripts to describe how those usages differ from the usages of the elements in R in the new versions of its client code. GROUM-based edit scripts for all mappings $I \rightarrow R$ are provided to frequent item set mining to generalize common edit operations. This strategy is different from **MAM**, which applies a custom similarity-based algorithm to the graph-based representation of usage data rather off-the-shelf frequent item set mining.

6.3 Empirical Findings

The work on migration mapping detection offers interesting commonalities in the design of evaluation studies. The results of the evaluation also exhibit relatively good precision.

6.3.1 Study Design

A good amount of the work on the detection of migration mappings has been evaluated in comparison with similar approaches. This kind of comparative work is rare in the body of work on API property inference mining. We can presume that the focused goal and relatively uniform nature of the property inferred has facilitated this development.

RefactoringCrawler, the approach by **Schäfer et al.**, and **AURA** use a common set of target APIs as their subject systems (eclipse.ui, Struts, and JHotDraw). This uniformity facilitates mutual comparison. RefactoringCrawler is a noteworthy baseline because all of the other approaches are compared directly or indirectly against it. **Wu et al.** compare AURA against Schäfer et al.'s approach on two of the three common subject systems. **SemDiff** and RefactoringCrawler are applied to two versions of the same API (Eclipse JDT Core, versions 3.1 and 3.3). Wu et al. also replicate SemDiff's evaluation and compare AURA's recommendations against a subset of scenarios for which SemDiff makes recommendations.

The other approaches use idiosyncratic benchmarks. **Zhong et al.** evaluate **MAM** by running it on 15 open-source projects for which both Java and C# implementations were available; **Nguyen et al.** evaluate the quality of **LibSync**'s edit scripts by running it on two snapshots taken from one development branch of JBoss; **Diff-CatchUp** is evaluated by using two small to medium-sized APIs (HTMLUnit and JFreeChart). The techniques are not compared against any other.

Concerning the assessment of the quality of the produced migration mappings, we also identify two categories of work. Most of the work discussed in this section employs manual inspection of the results to evaluate their correctness. This evaluation is based on the authors' own knowledge or on knowledge obtained from relevant documentation. For instance, the accuracy of **MAM** is assessed by manually evaluating the first 30 mappings for each target system (in alphabetical order); its recall is measured by comparing correct mappings mined by it against those hand-coded by the developers of Java2CSharp,[4] an existing Java to C# translation tool. **RefactoringCrawler**'s recommendations are

4. j2cstranslator.wiki.sourceforge.net.

compared against a set of refactorings documented by the authors in the context of a previous study using the same subject systems. **Schäfer et al.** manually compare the mappings of their tool against refactorings proposed by RefactoringCrawler.

A different strategy was developed to mitigate the threat of investigator bias induced by the manual assessment of mappings by the authors of an approach. For a number of approaches, the authors detect API migration mappings for clients and verify these mappings by either trying them out or by comparing them with versions of the client code that have actually been migrated by its original developers. This approach is illustrated by the evaluation of **SemDiff**. **Dagenais and Robillard** take three clients programs developed on Eclipse 3.1 (Mylyn, JBoss IDE, jdt.debug.ui), recompile them against Eclipse 3.3, collect the number of missing references to API methods, and use SemDiff to recommend mappings. The correctness of SemDiff's recommendations is determined by assessing how the new versions of the clients has actually been updated. Similarly, the quality of edit scripts produced by **LibSync** is evaluated by checking whether they were also applied on another development branch of JBoss. The advantage of this approach is that the evidence for correctness is strong. The disadvantage is that it can be difficult to find migrated clients. This style of evaluation also produces overconservative results because even in cases where many valid mappings could apply, the migrated code will only implement one. In a slightly different variant of this approach, **Diff-CatchUp** is used to fix compilation errors that arise when source code using the subject APIs (HTMLUnit and JFreeChart) is recompiled against newer versions of those APIs. This type of evaluation is weaker because compilable code may not necessarily be correct.

6.3.2 Reported Results

Overall, the precision of mappings produced by the approaches is very good. In contrast to sequential pattern mining techniques (Section 4) that generate many spurious rules, most migration mappings inferred actually correspond to valid knowledge. This performance can partially be explained by the narrower focus of the problem solved.

RefactoringCrawler detects refactorings with 0.90-1.0 precision and 0.86-1.0 recall according to the assessment strategy outlined above.

The evaluations of the technique by **Schäfer et al.** and **Diff-CatchUp** seek to characterize the overall precision rather than recall. The evaluation shows that **Diff-CatchUp**'s recommendations for two subject systems are accurate in 93 and 77 percent of cases, respectively. The approach by **Schäfer et al.** detects rules with a precision of over 0.85. Schäfer et al. also find that 25 percent of all correct rules are not induced by refactorings, and that refactorings detected by their technique do not fully overlap with those detected by **Dig et al.** For two out of three common subject systems, **AURA** detects about 50 percent more method replacement rules than Schäfer et al.'s technique, with a precision of close to 0.95. According to **Wu et al.**, AURA's use of syntactic similarity between methods allows it to detect rules not covered by Schäfer et al.. Wu et al. also find that rules detected by AURA and those detected by Schäfer et al. do not fully overlap.

In total, **SemDiff** produces relevant recommendations for 33 out of 37 broken method calls in clients, versus just 6 by **RefactoringCrawler**. **Wu et al.** replicate SemDiff's evaluation and compare **AURA**'s recommendations against those of SemDiff for 14 of the 37 broken method calls. On these 14 methods, SemDiff produces recommendations with 1.0 precision, while AURA's is 0.92.

The correctness of the mapped relations produced by **MAM** is 0.13 for one of the 15 subject projects, and between 0.73 and 1.0 for the remaining projects. The authors find that MAM detects over 70 percent of all type and method mappings declared in Java2CSharp, as well as 25 additional correct method mappings.

7 General Information

A number of approaches were designed to automatically collect general API-related information that can only loosely be defined as a property of the API.

For example, a number of approaches were designed to collect API usage statistics [100], [101], [102], [103]. Given a large enough corpus of usage contexts, usage statistics can be considered to be a form of general property that can inform, for example, the presentation of API documentation [101].

Approaches have also been proposed to find API elements related to a *seed* or query element [104], [105]. The idea is to provide developers with insights about API elements which may be related to elements already of interest (the seed). The property inferred in this case is a mapping $e_s \rightarrow \{e_{r_1}, \ldots, e_{r_n}\}$ between the seed e_s and its related elements. Although, in practice, related elements are provided on-demand as the result of a query, in principle it is also possible to systematically precompute related sets for all API elements. For this reason, we consider that such mappings are also a general property of the API. We note that although sets of related API elements have the same formal structure as sets of co-occurring elements as described in Section 3, they are distinct in that they do not necessarily represent usage patterns.

For all the approaches described in this section, the goal remains general: to help improve API documentation and guide developers in using the API.

7.1 Overview

Because mining techniques used for the approaches surveyed in this section are relatively simple and closely tied to the approach in question, we discuss the mining technique directly in the overview.

7.1.1 Popularity Statistics

One of the simplest ways to analyze an API's usage is to count the number of times its elements are referenced in various contexts. Four tools illustrate the potential of this approach by providing different perspective on popularity statistics.

The **PopCon** Eclipse plug-in [102] takes as input projects that use the API of interest and, for each project, creates a database on the structural relationships between client code and the API. Using the analysis engine of the Strathcona example recommendation tool [4], PopCon extracts four

relationships: inheritance, overriding, method calls, and field references. These steps are the same as performed by all other approaches that analyze client code (see Section 3), except that instead of mining the database, PopCon offers various views of usage statistics.

SpotWeb [100] refines the idea of popularity statistics by attempting to detect popular and unpopular API elements ("hotspots" and "coldspots," respectively). The novelty of SpotWeb includes that it automatically derives its corpus from examples returned from a code search engine. SpotWeb parses and analyzes the retrieved code examples to compute, for each API element, properties called *usage metrics*. Usage metrics are an elaborate form of popularity metric. The computation of usage metrics takes into account the various ways in which API elements can be accessed in client code. For example, for a class, the usage metric takes into account the number of constructor call sites and the number of times the class is extended; for methods, the metric takes into account the number of times the method has been invoked, overridden, or implemented.

While **SpotWeb** explores new popularity metrics, **Jadeite** [101] investigates new means of presentation for API usage statistics. Essentially an enhanced version of a JavaDoc-viewer, Jadeite provides a number of features intended to facilitate the discovery of information on how to start using an API. One of the features provided is to display the names of API elements in a font size proportional to their popularity. The underlying assumption is that this feature can help developers quickly identify elements they may need to start using the API. As opposed to the API usage statistics computed by **PopCon** or **SpotWeb**, Jadeite measures the popularity of API elements by searching the web and counting the number of webpages mentioning that element.

Finally, **Aktari** [103] is a tool to help developers select the best version of a library to use based on the popularity of choices made by previous developers. Aktari mines a corpus of projects and extracts information about which versions are the most popular and which libraries were most reverted to; these are the two properties Aktari infers about an API. Aktari works by analyzing a project's configuration file for dependent libraries (with version number).

7.1.2 Within-API Mappings

The four tools described above compute popularity properties. In contrast, two approaches have been proposed to calculate within-API mappings.

Saul et al. developed an algorithm called **FRAN** to discover the API elements most related to an element of interest e [105]. The approach is motivated by the assumption that developers often need to find related API functions when using an API. To issue recommendations, the approach analyzes a call graph of source code that uses the API and computes relatedness using formulas inspired from the web search community and, in particular, Kleinberg's Hypertext Induced Topic Selection (HITS) algorithm [106]. In essence, FRAN determines a base set of functions in the same layer (or relative call-graph depth) as the query function e, then analyzes the call-graph topology around e to calculate a score of "authority" defined by the probability that a programmer randomly exploring the code would visit a particular function given its dependency graph, and then returns a user-specified number of top authorities. In terms of input, the approach does not specifically require client applications, but assumes that API functions of interest are called by the rest of the API, so the approach indeed requires some sort of client code. This is the same assumption made by **SemDiff** [90], [95], for example (see Section 6).

Altair [104] works just like **FRAN**: It takes as input an API function f and automatically proposes other functions f_i that are "related" to f, i.e., would be added as "see-also" functions in f's documentation. Altair computes relatedness by finding functions f_i that access the same primitive variables as f and that use similar composite types as f (i.e., C structs and records).

7.2 Empirical Findings

PopCon and **Aktari** were developed as proof of concepts, and were not formally evaluated. Descriptions of the tools include examples of applications to real open-source systems, with illustrations of the type of aggregated data and insights that can be obtained with them. The Font Size feature of **Jadeite** [101] is assessed as part of a small user study with seven student participants. As the study shows, participants were between two and three times faster at finding key API classes when using Jadeite, compared to a control group that used JavaDoc. In brief, the above projects mostly focus on exploring different ways to provide API usage information to developers. In contrast, **Thummalapenta and Xie** assess the effectiveness of their hotspot detection approach by comparing the hotspot classes with benchmark classes, either obtained from API documentation or from a comparable approach developed by Viljamaa [107]. In the first case, they compare the hotspots detected in Log4j and JUnit with the main classes described in tutorials for these APIs, and indeed find that the main classes detected by their approach are also the ones described as "starting points" in the documentation. In the second case, the authors compare their results with those of a case study of hotspots in JUnit conducted by Viljamaa [107], but this comparison, focusing on only one small system, provides few insights into the effectiveness of the approach except to show that the results are different.

Both the **FRAN** and **Altair** recommendation algorithms are evaluated through benchmark-based experiments. Specifically, the approaches are systematically applied and the results compared with an independently developed oracle. In both cases, the performance of the approach is compared with different algorithms.

To evaluate **FRAN**, the authors apply it to 330 functions of an Apache project (the Apache Portability Runtime, or APR) and calculate various sanity metrics, such as the number of functions returned. More importantly, the authors analyze whether the functions recommended as related to the seed function correspond to the functions in the same module as the query, based on a modular decomposition defined by the project's developers and specified in the documentation. From this analysis, they derive precision/recall measures. The performance of FRAN is then compared to two other algorithms, an association rule-based approach (FRIAR) also developed

by the authors, and a topology analysis of software dependencies algorithm originally proposed by Robillard [108], [109] (even though this algorithm was not designed to elicit programming patterns and is only partially reimplemented). The authors find that in the case of APR, FRAN can provide recommendation sets that match the modular decomposition with F1 values in the range 0.03-0.5, depending on the query function.

For its evaluation of see-also recommendation, **Altair**'s output is compared against the three approaches mentioned above: Robillard's Suade [108], **FRAN**, and FRIAR [105]. Although the authors present detailed results for a few selected queries, the evaluation does not include project-wide precision/recall numbers.

The main insights that stem from the evaluation of **FRAN**/FRIAR and **Altair** are that, *as illustrated by the case of the Apache Portable Runtime Library*, 1) API analysis can recommend sets of related functions that roughly map to the "natural" modular decomposition, but 2) the performance varies tremendously depending on the input seed, and 3) there is little consistency between approaches. Ultimately, one of the major challenges for approaches that recommend "related" functions is that, in the general case, "relatedness" remains subjective and context-sensitive.

8 Conclusions

This survey offers a systematic review of over 60 API property inference techniques along several dimensions: nature of the property inferred, goal of the approach, mining technique, input data required, chronology. Our classification and synthesis not only provide a conceptual framework for describing and comparing approaches, it also brings to light many novel observations about the field.

First, we observe a lack of uniformity in terminology and definitions for the properties inferred by various approaches. In the literature, properties inferred from APIs are referred to, almost interchangeably, as patterns, rules, specifications, or protocols. As demonstrated by our review, there is also very little uniformity in the definition of data structures for representing properties inferred from APIs. All too often, such properties are simply defined as the output of an approach. Indeed, a close study of the properties inferred by various approaches shows that it is very difficult to decouple the definition of properties from their generation technique and intended use. Even the simplest types of properties, unordered usage patterns and migration mappings, differ slightly between techniques according to design and implementation details. Even performance considerations can impact the formalization of properties, such as the choice of edges or nodes to represent events in finite-state automata representations of sequential patterns.

With this survey, we hope to bring some order to this complex field by proposing a simple terminology for API properties and a basic conceptual framework for comparing properties and the techniques that can infer them. We found it very challenging to elaborate a classification framework that would be, at the same time, simple, conceptually coherent, and a good fit for the variety of approaches surveyed. After many trials and errors, we opted to work with the nature of the property inferred as the dominant dimension. This strategy works well: The approaches grouped within a main category share many characteristics besides the nature of the property inferred. We note that, in particular, there exists a relation between the type of the property inferred, the nature of the input data, and the mining techniques used to infer the properties. That such relations would exist is perhaps not surprising, but our review enables us to document it more exactly. In particular, unordered properties (Section 3), including mappings (Section 6), are typically inferred from mining client data. When it comes to inferring knowledge about order (Section 4), dynamic approaches (or simulation) are most popular since flow-sensitive static analyses are still hard to scale up to analyzing large programs. For behavioral specifications (Section 5), symbolic execution and some form of static program verification techniques are required.

No categorization is perfect and, inevitably, some of the category definitions within our dimensions of analysis lead to classifications that are not all crisp. One particular challenge lies in the distinction between sequential usage patterns and behavioral specifications. In both cases, a type of automaton is inferred, and the distinction lies in whether the approach focuses on the programming pattern (Section 4) or the resultant state of the program (Section 5). Another distinction is that sequential pattern mining approaches are mostly data mining-based, whereas behavioral specifications are derived from source code analysis, but there again there are exceptions. Another challenge consists in categorizing the few approaches that cannot be strongly associated with any of the first four types of properties. Within the "General Information" category, we notice two small but cohesive clusters of approaches. Nevertheless, because development work in these areas is not as active as in the other cases, we aggregate all remaining work in a more generic category.

The survey also allows us to make hypotheses about the engineering aspects most likely to have impact on the results. For example, different techniques to infer unordered usage patterns rely on very similar data mining technology, and their performance is usually determined through the careful configuration of pre and postprocessing customizations, and in particular in the choice of mining contexts and filtering heuristics. In the case of sequential mining, much more engineering goes directly into the inference process and similarly, in the case of the work on behavioral specification mining and migration mapping mining, where we note important distinctions in the strategies employed and their consequences.

Our survey also highlights interesting evolutionary trends. For instance, Table 3 shows that in the early days mining approaches for sequential patterns focused on finite-state machine models. Around 2007, this situation changed, with a large number of approaches that focus on different kinds of representations, mostly simpler temporal rules or combinations thereof. The goal of the techniques also seems to be refined over the years, with more general goals in the early years (Documentation, Specification Mining) followed by a burst of work specifically on bug finding in the last years. For unordered patterns, the foundational work also had an ill-defined goal, with subsequent work more focused. Interestingly, for mappings, it is the nature of the problem that changed.

The earlier approaches focused on the narrower problem of inferring refactorings, with later approaches tackling the inference of more general migration mappings.

In closing, a careful study of property inference techniques inevitably raises the question: What value does knowledge about API properties provide? Intuitively, a valuable property should provide *new* knowledge that can be *acted upon* by developers working with APIs. The criteria to infer new knowledge implies a challenge: that of inferring nonobvious or surprising information. As illustrated by the seminal work of Michail [8], many patterns automatically mined from client code will be obvious to anyone with a basic programming knowledge, and hence of little value. To generate properties that can be acted upon means not only documenting properties in a format fit for that purpose (e.g., machine-checkable for bug detection approaches), but also with a sufficiently high rate of true positives to be actually usable, either by humans or by tools.

The basic way to assess the value of properties is through empirical evaluation. Although most of the approaches we surveyed were the object of some form of empirical evaluation, the thoroughness and style of assessment varies greatly. In fact, we find the area is lacking a systematic evaluation methodology. As noted by Gabel and Su [46], "the very motivation for the work, the lack of well-documented specifications, makes validating the findings difficult and subjective." Even when documentation is available, it is rarely expressed in a precise formal language, and hence it is a matter of subjective interpretation. Empirical comparisons to similar approaches are difficult, partly because the tools rarely solve exactly the same problem or produce directly comparable properties. With this survey, we provide a foundation to organize and compare API property inference techniques, which should facilitate the further development of standardized evaluation techniques and artifacts.

Appendix

Methodology

We collected references based on a methodical review of the literature. In a first phase, we manually reviewed the title and abstract of every single paper published since the year 2000 (inclusive) in a number of selected venues. The year 2000 was chosen to provide at least a comprehensive 10-year horizon on the topic and because we were not aware of any applied work on API analysis prior to that date. We initially reviewed the contents of the venues listed below. The list includes the number of formally reviewed papers (i.e., appearing in a table) found in each venue:

- The ACM/IEEE International Conference on Software Engineering (ICSE): 14.
- The ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE) and Joint Meeting of the European Software Engineering Conference and the The ACM SIGSOFT International Symposium on the Foundations of Software Engineering (ESEC/FSE): 10.
- The ACM Transactions on Software Engineering and Methodology: 1.
- The IEEE Transactions on Software Engineering: 2.
- The ACM SIGSOFT/SIGPLAN International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA): 1.
- The European Conference on Object-Oriented Programming (ECOOP): 3.
- The IEEE/ACM International Conference on Automated Software Engineering (ASE): 10.
- The Working Conference on Reverse Engineering (WCRE): 3.
- The International Symposium on Software Testing and Analysis (ISSTA): 4.
- The ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL): 2.
- The Automated Software Engineering: An International Journal (Springer): 0.

We chose these venues to include mainstream publication venues in software engineering and venues we knew had published work relevant to the survey. This phase resulted in the identification of approximately 160 potentially relevant references.

In a second phase, each author individually reviewed a subset of the articles to assess its relevance, categorize the type of properties inferred by the technique described in the paper, and study the list of references to add additional relevant papers and review them following an iterative process. This phase led to the identification of relevant publications in 13 additional venues. At this point, we did not consider new venues for methodical year-by-year review because all new venues identified in the second phase were the publication vehicle for a single relevant paper (except in one case, with two relevant papers).

We then studied each paper in detail, removing from the survey any paper that fell outside the scope according to the definitions in Section 1.

Acknowledgments

This work has been made possible by the generous support of the Alexander von Humboldt Foundation, the German Federal Ministry of Education and Research (BMBF) within EC SPRIDE, and by the Hessian LOEWE excellence initiative within CASED. The authors are also grateful to Barthélémy Dagenais, Michael Pradel, and Thomas Zimmermann for their valuable comments on this paper.

References

[1] M. Robillard and R. DeLine, "A Field Study of API Learning Obstacles," *Empirical Software Eng.*, vol. 16, no. 6, pp. 703-732, 2011.

[2] D. Engler, D.Y. Chen, S. Hallem, A. Chou, and B. Chelf, "Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code," *Proc. 18th ACM Symp. Operating Systems Principles*, pp. 57-72, 2001.

[3] M.D. Ernst, J. Cockrell, W.G. Griswold, and D. Notkin, "Dynamically Discovering Likely Program Invariants to Support Program Evolution," *Proc. 21st ACM/IEEE Int'l Conf. Software Eng.*, pp. 213-224, 1999.

[4] R. Holmes, R.J. Walker, and G.C. Murphy, "Approximate Structural Context Matching: An Approach to Recommend Relevant Examples," *IEEE Trans. Software Eng.*, vol. 32, no. 12, pp. 952-970, Dec. 2006.

[5] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei, "MAPO: Mining and Recommending API Usage Patterns," *Proc. 23rd European Conf. Object-Oriented Programming*, pp. 318-343, 2009.

[6] N. Bjørner, A. Browne, and Z. Manna, "Automatic Generation of Invariants and Intermediate Assertions," *Theoretical Computer Science,* vol. 173, pp. 49-87, 1997.

[7] M.B. Dwyer, G.S. Avrunin, and J.C. Corbett, "Patterns in Property Specifications for Finite-State Verification," *Proc. 21st ACM/IEEE Int'l Conf. Software Eng.,* pp. 411-420, 1999.

[8] A. Michail, "Data Mining Library Reuse Patterns in User-Selected Applications," *Proc. 14th IEEE Int'l Conf. Automated Software Eng.,* pp. 24-33, 1999.

[9] A. Michail, "Data Mining Library Reuse Patterns Using Generalized Association Rules," *Proc. 22nd ACM/IEEE Int'l Conf. Software Eng.,* pp. 167-176, 2000.

[10] Z. Li and Y. Zhou, "PR-Miner: Automatically Extracting Implicit Programming Rules and Detecting Violations in Large Software Code," *Proc. Joint Meeting 10th European Software Eng. Conf. and 13th ACM SIGSOFT Int'l Symp. Foundations of Software Eng.,* pp. 306-315, 2005.

[11] B. Livshits and T. Zimmermann, "DynaMine: Finding Common Error Patterns by Mining Software Revision Histories," *Proc. Joint Meeting 10th European Software Eng. Conf. and 13th ACM SIGSOFT Int'l Symp. Foundations of Software Eng.,* pp. 296-305, 2005.

[12] M. Bruch, T. Schäfer, and M. Mezini, "FrUiT: IDE Support for Framework Understanding," *Proc. OOPSLA Eclipse Technology eXchange,* pp. 55-59, 2006.

[13] M. Bruch, M. Monperrus, and M. Mezini, "Learning from Examples to Improve Code Completion Systems," *Proc. Joint Meeting Seventh European Software Eng. Conf. and Seventh ACM SIGSOFT Symp. Foundations of Software Eng.,* pp. 213-222, 2009.

[14] M. Monperrus, M. Bruch, and M. Mezini, "Detecting Missing Method Calls in Object-Oriented Software," *Proc. 24th European Conf. Object-Oriented Programming,* pp. 2-25, 2010.

[15] M.P. Robillard, R.J. Walker, and T. Zimmermann, "Recommendation Systems for Software Engineering," *IEEE Software,* vol. 27, no. 4, pp. 80-86, July/Aug. 2010.

[16] R. Srikant and R. Agrawal, "Mining Generalized Association Rules," *Proc. 21st Int'l Conf. Very Large Data Bases,* pp. 407-419, 1995.

[17] R. Agrawal and R. Srikant, "Fast Algorithms for Mining Association Rules in Large Databases," *Proc. 20th Int'l Conf. Very Large Data Bases,* pp. 487-499, 1994.

[18] G.I. Webb and S. Zhang, "Beyond Association Rules: Generalized Rule Discovery," *Knowledge Discovery and Data Mining,* Kluwer Academic Publishers, 2003.

[19] G. Grahne and J. Zhu, "Efficiently Using Prefix-Trees in Mining Frequent Item Sets," *Proc. Workshop Frequent Item Set Mining Implementations,* 2003.

[20] G. Ammons, R. Bodík, and J.R. Larus, "Mining Specifications," *Proc. 29th ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages,* pp. 4-16, 2002.

[21] J. Whaley, M.C. Martin, and M.S. Lam, "Automatic Extraction of Object-Oriented Component Interfaces," *Proc. ACM Int'l Symp. Software Testing and Analysis,* pp. 218-228, 2002.

[22] J. Yang and D. Evans, "Automatically Inferring Temporal Properties for Program Evolution," *Proc. IEEE Int'l Symp. Software Reliability Eng.,* pp. 340-351, 2004.

[23] J. Yang and D. Evans, "Dynamically Inferring Temporal Properties," *Proc. Fifth ACM SIGPLAN-SIGSOFT Workshop Program Analysis for Software Tools and Eng.,* pp. 23-28, 2004.

[24] R. Alur, P. Černý, P. Madhusudan, and W. Nam, "Synthesis of Interface Specifications for Java Classes," *Proc. 32nd ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages,* pp. 98-109, 2005.

[25] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman, "Jungloid Mining: Helping to Navigate the API Jungle," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation,* pp. 48-61, 2005.

[26] M. Salah, T. Denton, S. Mancoridis, A. Shokoufandeh, and F.I. Vokolos, "Scenariographer: A Tool for Reverse Engineering Class Usage Scenarios from Method Invocation Sequences," *Proc. 21st IEEE Int'l Conf. Software Maintenance,* pp. 155-164, 2005.

[27] W. Weimer and G.C. Necula, "Mining Temporal Specifications for Error Detection," *Proc. 11th Int'l Conf. Tools and Algorithms for the Construction and Analysis of Systems,* pp. 461-476, 2005.

[28] M. Acharya and T. Xie, "Mining API Error-Handling Specifications from Source Code," *Proc. 12th Int'l Conf. Fundamental Approaches to Software Eng.,* pp. 370-384, 2009.

[29] M. Acharya, T. Xie, and J. Xu, "Mining Interface Specifications for Generating Checkable Robustness Properties," *Proc. 17th IEEE Int'l Symp. Software Reliability Eng.,* pp. 311-320, 2006.

[30] V. Dallmeier, C. Lindig, A. Wasylkowski, and A. Zeller, "Mining Object Behavior with ADABU," *Proc. Int'l Workshop Dynamic Systems Analysis,* pp. 17-24, 2006.

[31] C. Liu, E. Ye, and D.J. Richardson, "LtRules: An Automated Software Library Usage Rule Extraction Tool," *Proc. 28th ACM/IEEE Int'l Conf. Software Eng.,* pp. 823-826, 2006.

[32] C. Liu, E. Ye, and D.J. Richardson, "Software Library Usage Pattern Extraction Using a Software Model Checker," *Proc. 21st IEEE/ACM Int'l Conf. Automated Software Eng.,* 2006.

[33] T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Software Verification with BLAST," *Proc. 10th Int'l Conf. Model Checking Software,* pp. 235-239, 2003.

[34] D. Lo and S. Khoo, "SMArTIC: Towards Building an Accurate, Robust and Scalable Specification Miner," *Proc. 14th ACM SIGSOFT Int'l Symp. Foundations of Software Eng.,* pp. 265-275, 2006.

[35] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das, "Perracotta: Mining Temporal API Rules from Imperfect Traces," *Proc. 28th ACM/IEEE Int'l Conf. Software Eng.,* pp. 282-291, 2006.

[36] M. Acharya, T. Xie, J. Pei, and J. Xu, "Mining API Patterns as Partial Orders from Source Code: From Usage Scenarios to Specifications," *Proc. Sixth Joint Meeting European Software Eng. Conf. and ACM SIGSOFT Int'l Symp. Foundations of Software Eng.,* pp. 25-34, 2007.

[37] H. Kagdi, M.L. Collard, and J.I. Maletic, "An Approach to Mining Call-Usage Patterns with Syntactic Context," *Proc. 22nd IEEE/ACM Int'l Conf. Automated Software Eng.,* pp. 457-460, 2007.

[38] J. Quante and R. Koschke, "Dynamic Protocol Recovery," *Proc. Working Conf. Reverse Eng.,* pp. 219-228, 2007.

[39] M.K. Ramanathan, A. Grama, and S. Jagannathan, "Static Specification Inference Using Predicate Mining," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation,* pp. 123-134, 2007.

[40] M.K. Ramanathan, A. Grama, and S. Jagannathan, "Path-Sensitive Inference of Function Precedence Protocols," *Proc. 29th ACM/IEEE Int'l Conf. Software Eng.,* pp. 240-250, 2007.

[41] S. Shoham, E. Yahav, S. Fink, and M. Pistoia, "Static Specification Mining Using Automata-Based Abstractions," *Proc. ACM Int'l Symp. Software Testing and Analysis,* pp. 174-184, 2007.

[42] S. Thummalapenta and T. Xie, "Parseweb: A Programmer Assistant for Reusing Open Source Code on the Web," *Proc. 22nd IEEE/ACM Int'l Conf. Automated Software Eng.,* pp. 204-213, 2007.

[43] A. Wasylkowski, A. Zeller, and C. Lindig, "Detecting Object Usage Anomalies," *Proc. Sixth Joint Meeting European Software Eng. Conf. and ACM SIGSOFT Int'l Symp. Foundations of Software Eng.,* pp. 35-44, 2007.

[44] N. Walkinshaw, K. Bogdanov, M. Holcombe, and S. Salahuddin, "Reverse Engineering State Machines by Interactive Grammar Inference," *Proc. Working Conf. Reverse Eng.,* pp. 209-218, 2007.

[45] N. Walkinshaw and K. Bogdanov, "Inferring Finite-State Models with Temporal Constraints," *Proc. 23rd IEEE/ACM Int'l Conf. Automated Software Eng.,* pp. 248-257, 2008.

[46] M. Gabel and Z. Su, "Javert: Fully Automatic Mining of General Temporal Properties from Dynamic Traces," *Proc. 16th ACM SIGSOFT Int'l Symp. Foundations of Software Eng.,* pp. 339-349, 2008.

[47] D. Lorenzoli, L. Mariani, and M. Pezzè, "Automatic Generation of Software Behavioral Models," *Proc. 30th ACM/IEEE Int'l Conf. Software Eng.,* pp. 501-510, 2008.

[48] D. Lo, S.C. Khoo, and C. Liu, "Mining Temporal Rules for Software Maintenance," *J. Software Maintenance and Evolution: Research and Practice,* vol. 20, no. 4, pp. 227-247, 2008.

[49] S. Sankaranarayanan, F. Ivančı, and A. Gupta, "Mining Library Specifications Using Inductive Logic Programming," *Proc. 13th ACM/IEEE Int'l Conf. Software Eng.,* pp. 131-140, 2008.

[50] H. Zhong, L. Zhang, and H. Mei, "Inferring Specifications of Object Oriented APIs from API Source Code," *Proc. 15th Asia-Pacific Software Eng. Conf.,* pp. 221-228, 2008.

[51] M. Gabel and Z. Su, "Symbolic Mining of Temporal Specifications," *Proc. 30th ACM/IEEE Int'l Conf. Software Eng.,* pp. 51-60, 2008.

[52] D. Lo, G. Ramalingam, V.P. Ranganath, and K. Vaswani, "Mining Quantified Temporal Rules: Formalism, Algorithms, and Evaluation," *Proc. Working Conf. Reverse Eng.*, pp. 62-71, 2009.

[53] T.T. Nguyen, H.A. Nguyen, N.H. Pham, J.M. Al-Kofahi, and T.N. Nguyen, "Graph-Based Mining of Multiple Object Usage Patterns," *Proc. Seventh Joint Meeting European Software Eng. Conf. and ACM SIGSOFT Symp. Foundations of Software Eng.*, pp. 383-392, 2009.

[54] M. Pradel and T.R. Gross, "Automatic Generation of Object Usage Specifications from Large Method Traces," *Proc. 24th IEEE/ACM Int'l Conf. Automated Software Eng.*, pp. 371-382, 2009.

[55] M. Pradel, "Dynamically Inferring, Refining, and Checking API Usage Protocols," *Proc. 24th ACM SIGPLAN Conf. Object-Oriented Programming Systems, Languages and Applications—Companion*, pp. 773-774, 2009.

[56] S. Thummalapenta and T. Xie, "Mining Exception-Handling Rules as Sequence Association Rules," *Proc. 31st ACM/IEEE Int'l Conf. Software Eng.*, pp. 496-506, 2009.

[57] S. Thummalapenta and T. Xie, "Alattin: Mining Alternative Patterns for Detecting Neglected Conditions," *Proc. 24th IEEE/ACM Int'l Conf. Automated Software Eng.*, pp. 283-294, 2009.

[58] A. Wasylkowski and A. Zeller, "Mining Temporal Specifications from Object Usage," *Proc. 24th IEEE/ACM Int'l Conf. Automated Software Eng.*, pp. 295-306, 2009.

[59] H. Zhong, L. Zhang, T. Xie, and H. Mei, "Inferring Resource Specifications from Natural Language API Documentation," *Proc. 24th IEEE/ACM Int'l Conf. Automated Software Eng.*, pp. 307-318, 2009.

[60] M. Gabel and Z. Su, "Online Inference and Enforcement of Temporal Properties," *Proc. 32nd ACM/IEEE Int'l Conf. Software Eng.*, pp. 15-24, 2010.

[61] N. Gruska, A. Wasylkowski, and A. Zeller, "Learning from 6,000 Projects: Lightweight Cross-Project Anomaly Detection," *Proc. 19th ACM Int'l Symp. Software Testing and Analysis*, pp. 119-130, 2010.

[62] N.A. Naeem and O. Lhoták, "Typestate-Like Analysis of Multiple Interacting Objects," *Proc. ACM Int'l Conf. Object-Oriented Programming Systems, Languages and Applications*, pp. 347-366, 2008.

[63] E. Bodden, P. Lam, and L. Hendren, "Partially Evaluating Finite-State Runtime Monitors Ahead of Time," *ACM Trans. Programming Languages and Systems*, vol. 34, 2012.

[64] C. Lee, F. Chen, and G. Roşu, "Mining Parametric Specifications," *Proc. 33rd ACM/IEEE Int'l Conf. Software Eng.*, pp. 591-600, 2011.

[65] H. Kagdi, M.L. Collard, and J.I. Maletic, "Comparing Approaches to Mining Source Code for Call-Usage Patterns," *Proc. Fourth Int'l Workshop Mining Software Repositories*, 2007.

[66] J. Wang and J. Han, "BIDE: Efficient Mining of Frequent Closed Sequences," *Proc. 20th Int'l Conf. Data Eng.*, pp. 79-90, 2004.

[67] A.V. Raman and J.D. Patrick, "The sk-Strings Method for Inferring PFSA," *Proc. 14th Int'l Conf. Machine Learning Workshop Automata Induction, Grammatical Inference and Language Acquisition*, 1997.

[68] P. Cousot and R. Cousot, "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints," *Proc. Fourth ACM Symp. Principles of Programming Languages*, pp. 238-252, 1977.

[69] J.H. Reif, "Universal Games of Incomplete Information," *Proc. ACM Symp. Theory of Computing*, pp. 288-308, 1979.

[70] D. Angluin, "Learning Regular Sets from Queries and Counterexamples," *Information and Computation/Information and Control*, vol. 75, pp. 87-106, 1987.

[71] R. Vallée-Rai, P. Co, E. Gagnon, L.J. Hendren, P. Lam, and V. Sundaresan, "Soot—A Java Bytecode Optimization Framework," *Proc. Conf. Centre for Advanced Studies on Collaborative Research*, p. 13, 1999.

[72] A. Cimatti, E.M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, *NuSMV 2: An OpenSource Tool for Symbolic Model Checking*, 2002.

[73] P. Dupont, B. Lambeau, C. Damas, and A. van Lamsweerde, "The QSM Algorithm and Its Application to Software Behavior Model Induction," *Applied Artificial Intelligence*, vol. 22, no. 1/2, pp. 77-115, 2008.

[74] V. Dallmeier, N. Knopp, C. Mallon, S. Hack, and A. Zeller, "Generating Test Cases for Specification Mining," *Proc. ACM Int'l Symp. Software Testing and Analysis*, pp. 85-96, 2010.

[75] D. Lo and S.-C. Khoo, "QUARK: Empirical Assessment of Automaton-Based Specification Miners," *Proc. 13th Working Conf. Reverse Eng.*, pp. 51-60, 2006.

[76] M. Pradel, P. Bichsel, and T.R. Gross, "A Framework for the Evaluation of Specification Miners Based on Finite State Machines," *Proc. 26th IEEE Int'l Conf. Software Maintenance*, pp. 1-10, 2010.

[77] A.W. Biermann and J.A. Feldman, "On the Synthesis of Finite-State Machines from Samples of Their Behavior," *IEEE Trans. Computers*, vol. 21, no. 6, pp. 592-597, June 1972.

[78] B. Meyer, "Applying 'Design by Contract'," *Computer*, vol. 25, no. 10, pp. 40-51, Oct. 1992.

[79] C. Flanagan and K.R.M. Leino, "Houdini, an Annotation Assistant for ESC/Java," *Proc. Int'l Symp. Formal Methods Europe*, pp. 500-517, 2001.

[80] N. Tillmann, F. Chen, and W. Schulte, "Discovering Likely Method Specifications," *Proc. Eighth Int'l Conf. Formal Eng.*, pp. 717-736, 2006.

[81] R.P.L. Buse and W.R. Weimer, "Automatic Documentation Inference for Exceptions," *Proc. ACM Int'l Symp. Software Testing and Analysis*, pp. 273-282, 2008.

[82] J. Henkel and A. Diwan, "Discovering Algebraic Specifications from Java Classes," *Proc. 17th European Conf. Object-Oriented Programming*, 2003.

[83] J. Henkel, C. Reichenbach, and A. Diwan, "Discovering Documentation for Java Container Classes," *IEEE Trans. Software Eng.*, vol. 33, no. 8, pp. 526-543, Aug. 2007.

[84] J. Guttag and J. Horning, "The Algebraic Specification of Abstract Data Types," *Acta Informatica*, vol. 10, no. 1, pp. 27-52, 1978.

[85] C. Ghezzi, A. Mocci, and M. Monga, "Synthesizing Intensional Behavior Models by Graph Transformation," *Proc. 31st ACM/IEEE Int'l Conf. Software Eng.*, pp. 430-440, 2009.

[86] K.R.M. Leino, J.B. Saxe, and R. Stata, "Checking Java Programs via Guarded Commands," *Proc. 13th European Conf. Object-Oriented Programming*, pp. 110-111, 1999.

[87] J.C. King, "Symbolic Execution and Program Testing," *Comm. ACM*, vol. 19, pp. 385-394, July 1976.

[88] M.W. Godfrey and L. Zou, "Using Origin Analysis to Detect Merging and Splitting of Source Code Entities," *IEEE Trans. Software Eng.*, vol. 31, no. 2, pp. 166-181, Feb. 2005.

[89] M. Kim, D. Notkin, and D. Grossman, "Automatic Inference of Structural Changes for Matching across Program Versions," *Proc. 29th ACM/IEEE Int'l Conf. Software Eng.*, pp. 333-343, 2007.

[90] B. Dagenais and M.P. Robillard, "Recommending Adaptive Changes for Framework Evolution," *ACM Trans. Software Eng. and Methodology*, vol. 20, no. 4, 2011.

[91] D. Dig, C. Comertoglu, D. Marinov, R. Johnson, and D. Thomas, "Automated Detection of Refactorings in Evolving Components," *Proc. 20th European Conf. Object-Oriented Programming*, pp. 404-428, 2006.

[92] Z. Xing and E. Stroulia, "API-Evolution Support with Diff-CatchUp," *IEEE Trans. Software Eng.*, vol. 33, no. 12 pp. 818-836, Dec. 2007.

[93] Z. Xing and E. Stroulia, "Differencing Logical UML Models," *Automated Software Eng.*, vol. 14, no. 2, pp. 215-259, 2007.

[94] T. Schäfer, J. Jonas, and M. Mezini, "Mining Framework Usage Changes from Instantiation Code," *Proc. 30th ACM/IEEE Int'l Conf. Software Eng.*, pp. 471-480, 2008.

[95] B. Dagenais and M.P. Robillard, "Recommending Adaptive Changes for Framework Evolution," *Proc. 30th ACM/IEEE Int'l Conf. Software Eng.*, pp. 481-490, 2008.

[96] W. Wu, Y.G. Guéhéneuc, G. Antoniol, and M. Kim, "AURA: A Hybrid Approach to Identify Framework Evolution," *Proc. 32nd ACM/IEEE Int'l Conf. Software Eng.*, pp. 325-334, 2010.

[97] H. Zhong, S. Thummalapenta, T. Xie, L. Zhang, and Q. Wang, "Mining API Mapping for Language Migration," *Proc. 32nd ACM/IEEE Int'l Conf. Software Eng.*, pp. 195-204, 2010.

[98] H. Nguyen, T. Nguyen, G. Wilson Jr, A. Nguyen, M. Kim, and T. Nguyen, "A Graph-Based Approach to API Usage Adaptation," *Proc. 25th ACM Int'l Conf. Object-Oriented Programming Systems, Languages and Applications*, pp. 302-321, 2010.

[99] D. Dig and R. Johnson, "How Do APIs Evolve? A Story of Refactoring," *J. Software Maintenance and Evolution: Research and Practice*, vol. 18, no. 2, pp. 83-107, 2006.

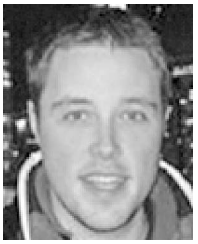
[100] S. Thummalapenta and T. Xie, "SpotWeb: Detecting Framework Hotspots and Coldspots via Mining Open Source Code on the Web," *Proc. 23rd IEEE/ACM Int'l Conf. Automated Software Eng.*, pp. 327-336, 2008.
[101] J. Stylos, A. Faulring, Z. Yang, and B.A. Myers, "Improving API Documentation Using API Usage Information," *Proc. IEEE Symp. Visual Languages and Human-Centric Computing*, pp. 119-126, 2009.
[102] R. Holmes and R.J. Walker, "Informing Eclipse API Production and Consumption," *Proc. OOPSLA Workshop: Eclipse Technology eXchange*, pp. 70-74, 2007.
[103] Y.M. Mileva, V. Dallmeier, M. Burger, and A. Zeller, "Mining Trends of Library Usage," *Proc. Joint ERCIM Workshop Software Evolution and Int'l Workshop Principles of Software Evolution*, pp. 57-62, 2009.
[104] F. Long, X. Wang, and Y. Cai, "API Hyperlinking via Structural Overlap," *Proc. Seventh Joint Meeting European Software Eng. Conf. and ACM SIGSOFT Symp. Foundations of Software Eng.*, pp. 203-212, 2009.
[105] Z.M. Saul, V. Filkov, P. Devanbu, and C. Bird, "Recommending Random Walks," *Proc. Sixth Joint Meeting European Software Eng. Conf. and ACM SIGSOFT Int'l Symp. Foundations of Software Eng.*, pp. 15-24, 2007.
[106] J.M. Kleinberg, "Authoritative Sources in a Hyperlinked Environment," *J. ACM*, vol. 46, pp. 604-632, 1999.
[107] J. Viljamaa, "Reverse Engineering Framework Reuse Interfaces," *Proc. Joint Meeting Ninth European Software Eng. Conf. and 11th ACM SIGSOFT Int'l Symp. Foundations of Software Eng.*, pp. 217-226, 2003.
[108] M.P. Robillard, "Automatic Generation of Suggestions for Program Investigation," *Proc. Joint Meeting 10th European Software Eng. Conf. and 13th ACM SIGSOFT Int'l Symp. Foundations of Software Eng.*, pp. 11-20, 2005.
[109] M.P. Robillard, "Topology Analysis of Software Dependencies," *ACM Trans. Software Eng. and Methodology*, vol. 17, no. 4, 2008.

Martin P. Robillard received the BEng degree from the École Polytechnique de Montréal, and the MSc and PhD degrees in computer science from the University of British Columbia. He is an associate professor of computer science at McGill University.

Eric Bodden received the PhD degree from the Sable Research Group from McGill University, under the supervision of Laurie Hendren. He heads the Secure Software Engineering Group at the European Center for Security and Privacy by Design (EC SPRIDE), Darmstadt, Germany. Previously, he was a postdoctoral researcher in the Software Technology Group of the Technische Universität Darmstadt. His thesis was on evaluating runtime monitors ahead of time.

David Kawrykow received the MSc degree in computer science from McGill University.

Mira Mezini received the diploma degree in computer science from the University of Tirana, Albania, and the PhD degree in computer science from the University of Siegen, Germany. She is a professor of computer science at the Technische Universität Darmstadt, Germany, where she heads the Software Technology Lab.



Tristan Ratchford received the BA degree in economics and computer science and the MSc degree in computer science from McGill University. He is a software engineer at the IBM T.J. Watson Research Center, Cambridge, Massachusetts.