This is an appendix to the paper

"A Modular Platform for the Development of Recommendation Systems in Software Engineering".

by Sebastian Proksch, Sven Amann, Sarah Nadi, and Mira Mezini.
The authors are associated with Technische Universität Darmstadt, Germany.

Identifying Requirements of Static Analyses for Recommendation Systems in Software Engineering

Sebastian Proksch, Sven Amann, Sarah Nadi, Mira Mezini
Software Technology Group
Technische Universität Darmstadt, Germany
{proksch, amann, nadi, mezini}@st.informatik.tu-darmstadt.de

1 Introduction

A multitude of source-code-based *recommendation systems for software engineering* (RSSE) exist. They target a variety of recommendation tasks using different techniques that have specific requirements for the input.

To create a more modular platform for structured research of RSSE, it is necessary to find a common ground on which the requirements of many source-based RSSE are satisfied. As a step in this direction, we elicit the requirements of 23 existing source-code-based techniques and present the results in this appendix. We do not limit the survey to a single line of research, but cover a wide variety of fields. Therefore, we believe that the requirements of future recommendation systems would be similar.

2 Identifying Requirements

We review a recent comprehensive survey [22], the RSSE book [23], and several popular RSSEs to identify requirements. In total, we consider 23 techniques.

For each surveyed recommender, we identify the required input data and find reoccurring analysis challenges, as well as potential locations to simplify the analysis. Table 1 summarizes the results of this process.

2.1 Surveyed Recommender Systems

We now survey the papers to identify requirements for the IR and for reusable static analyses. Due to space limitations, we selected a representative set of typical approaches in RSSEs and do not aim to create a systematic literature review of the area. We group the surveyed recommenders by their recommendation task. Our discussion of each paper or technique focuses on the static analysis. We highlight how the analysis is performed and which of the above code elements are necessary. To facilitate readability, we highlight the code elements in the description of each technique.

Call Recommenders

A call recommender suggests the most likely subsequent method call(s) or parameter(s) to a developer.

McCarey et al. [11] introduced one of the early call recommendation systems that proposed which methods should be used in the current class under edit. The underlying static analysis is based on a simple traversal of the class syntax tree, in which all invoked methods are captured. Due to the simplified encoding, they ignore **statement order** in their analysis. Additionally, the analysis does not differentiate between different methods in a class. Therefore, it is not necessary to **track** the control flow or to **normalize** complex expressions.

Bruch et al. [2] also built a call recommendation system that was later extended by Proksch et al. [20]. The underlying approach in both cases identifies object instances in an intra-class analysis and extracts all method invocations on each instance, as well as a description of the surrounding source code. The original publication considered **object types**, the **enclosing method**, as well as **method calls**. Additionally, they use the **type system** to replace all captured method references with the first occurrence of the method signature in the type hierarchy. The extended work added **definition and paramater sites** as well as the **enclosing class**. The analysis does not preserve **order information** of the captured method invocations. The authors use an advanced **intra-class points-to analysis** to identify all available instances and they implemented a **tracking approach** that follows the control-flow into method invocations in the same class.

Zhang et al. [25] propose a recommender system that focuses on parameter call sites. They extract several features from the structural context that describe the method invocation and its parameters: the **called method**, the **enclosing method**, **methods that are called on the receiver**, and **methods that are called on the parameter**. They do not consider order information and collect all called methods in a set. Method parameters can be nested expressionsl however, they skip parameters if the expression is too complex. Thus, there is an opportunity to improve their technique by applying **normalization** of such expressions. Since they do not mention any extended static analysis in their description, we assume that they heuristically use variable names to distinguish different target objects.

Heinemann et al. [3] mainly use identifiers to predict method calls. They extract all **identifiers** used in the source code and split names on camel-case humps. Even though control structures are not part of their model, they consider **control structure keywords** in the tokenization.

They apply text-mining techniques such as stemming or removing stop words to unify the collected tokens. Their static analysis collects all **method invocations** from the source code, together with the n **preceding tokens**.

Amann et al. [1] implemented a tool that learns correct API usage from interactions of developers in their IDE. When code completion is triggered in the IDE, features are extracted from the structural context around the trigger point (type, definition, enclosing statement, expression type, enclosing method → selected result). The paper does not perform a points-to analysis, but it implements a heuristicfor the identification of definition sites that is based on the tracking of parameter names, variable declarations, and assignments. The enclosing method context is rewritten, therefore, it is necessary to look-up the info in the type system.

Raychev et al. [21] solve the task of call recommendation by mapping it to a text mining problem. They model sequences of **methods calls** as sentences. Missing method calls are similar to holes in a sentence that can be filled by calculating likely candidates. They use an intra-procedural static analysis that extracts all sequences of method invocations in a method body. They use an underlying **points-to analysis** to differentiate between different object instances.

Snippet Recommender

Similar to call recommenders, snippet recommenders propose likely completions to the developer. However, the results typically involve multiple statements or method calls and involve more than just the direct element on which completion was triggered.

Nguyen et al. [17] developed *GrouMiner* to find patterns in API usages from source code repositories. Their approach is only based on the syntax tree, they do not consider resolved types. The patterns, called Groums, contain **ordered information** about method calls, object declarations, and control points. They do not extract information about the structural context, but they do make use of the fields that are declared in the class. The analysis does not use a **points-to analysis**, but uses a simple intra-procedural heuristic for the data-flow detection that is built on variable names. In their follow-up work, Nguyen et al. [16] propose the *GraPacc* snippet recommender that uses these patterns. In addition to the extraction of the graph-based features from the Groums of the code under edit, they also tokenize the code to create token-based features to be able to support un-parsable code as well. They only consider keywords and variable names. The underlying models as well as the queries do not contain information about access modifiers. These are only added later during active completion tasks in an IDE (as opposed to offline evaluation).

Prospector (Mandolin et al. [9] and *PARSEWEB* (Thummalapenta et al. [24] are two recommenders that propose ordered sequences that involve different API types. Both recommenders suggest call sequences that show the developer how to get from one API type to another.

Prospector extracts several "elementary" code elements to build its "signature graph", the repository that contains all information that is collected about a framework. They capture **field accesses**, **static and non-static calls**, and **types** and store them as basic "jungloids", composable elements that describe reachable types. By analyzing source code, they also find examples of meaningful **up-casts** and **valid down-casts**, which they add to their repository. **Order** is not explicitly encoded in the input data; their synthesis approach naturally generates the right order, because their Jungloids always contain a tuple that describes how to get from one type to another, an implicit kind of order. They perform an **inter-procedural** and **inter-class analysis** that slices the program to find an elementary Jungloid that creates the target expression. They then **track** the control flow into the subsequent method calls.

PARSEWEB works a little different internally. They rely on examples returned from external code search engines. Since such examples often do not compile, they simply parse the snippets into syntax trees and transform them into a graph-based IR that supports branches but not **concrete control structures**. Their approach considers static and non-static calls and casts. The only **type information** used are the imports in a class. Heuristics are used to infer types in method signatures and method invocations.

The proposed snippets of both approaches contain **variable names**. However, they are not part of the model; unique identifiers are generated on demand.

Hindle et al. [4] provided a more general recommender that is based on natural language processing (NLP) techniques that treat the source code as plain text. While it does not propose complete snippets of source code, it can always propose a next token to write. This could be, for example, a keyword, which makes it a general recommender that is able to do more than just call completion. The technique includes all code elements of the respective language, excluding comments. Based on our understanding of the available description of the work, we assume that the authors either use a parser for each language and tokenize by traversing the AST, or they tokenize based on the grammer (lexing). In both ways, the text is tokenized according to the language specification, which makes it one of the exceptions in our survey that is solely syntax-based and which does not depend on a **resolved type system**. It is important to note that this design decision does not generalize to other NLP-based systems (e.g., [21]). The nature of the approach makes a **points-to analysis** unnecessary. **Normalization** and a **tracking** approach that adds tokens from private helpers would be applicable and the effect on the cross-entropy of the language model could be further evaluated.

Code Search

Code search is quite similar to snippet recommendations. The difference is that proposals point to existing examples

that were observed in repositories or the local workspace, instead of making probabilistic recommendations. The proposals cannot be directly integrated into the current editor, but will point to source code that can be used by the developer to understand a correct usage of the API in question.

Several approaches in this category use interclass tracking to follow the control flow in project-specific files. To be able to this, these approaches require structural context information to "find" the target in a look up and the type hierarchy to find implementations of interface methods.

Zhong et al. [26] create *MAPO*, a code search tool that mines method sequences from API usage examples. Methods calls include constructor calls, static and non-static calls, as well as casts. While they respect the **order of calls** and follow control structures to create all possible sequences, the **control structures** are not actually included in the model. The analysis is intra-procedural and the **enclosing method** is not extracted in the model. They analyze client code, but only include methods in their analysis that are declared in a reusable framework, i.e., included as a dependency, and track into all other method calls. They do this inter-class, as long as the files belong to the same project.

Holmes et al. [5] present the code search tool *Strathcona*. *Strathcona* extracts certain structural context information that describes a certain selected piece of code (**declaring type**, **super type**, **fields of declaring type**, **method declarations** in the fragment, as well as **types/methods/fields referenced** in the fragment).

Moreno et al. [15] propose *MUSE*, a tool that provides developers examples of how to use a particular method. The tool requires the source code of the target API as well as a list of its clients. Whenever a call to one of the target API public methods is observed in the client code, an intra-procedural **backward slice** is created that shows how to get to this call. The slicing builds a program-dependence-graph that requires keeping track of **method calls**, **order**, **variable names**, **assignments**, and **control structures**. Their approach includes all statements and expressions that are allowed in a method body by the Java language; this excludes structural information such as class declarations. When queried, the proposed example contains a ranked list of (rendered) slices that also contains explaining Javadoc, if available. We consider Javadoc as a separate artifact to source code, because while it is maintained in the source code of the framework, it is usually available as a separate download since binary distributions do not contain comments anymore. We believe that their approach can benefit from **normalization** to get rid of ambiguities. However, **points-to** analysis will not necessarily be useful here.

Hummel et al. [6] present *CodeConjurer* that supports developers by searching for working code that follows a specified UML-like syntax. The involved static analysis is based on the structural context only. The internal representation of the search engine only contains **fully-qualified names** for classes and all methods found in examples, as well as a **link to the location** where the file was originally found. The selection and ranking is completely based only on these signatures and identifiers. After including a snippet, **type resolution** of missing types is achieved by guessing the correct location, given the fully-qualified name of a class, and by downloading it to the local workspace. Since the lookup is very simple here, the technique would not require any of the transformations or additional analyses we discuss.

Documentation

Programmers often rely on documentation when learning about an API, but manually created documentation is hard to maintain and is thus often incomplete or outdated. Automated documentation generators try to solve this problem by mining source code repositories. They extract information that describe how to use an API and create documentation that can be consulted by developers to better understand a system. The documentation can have very different forms. We discuss approaches that cover a wide range.

Michail [12] proposed *CodeWeb*, a tool that can be used to identify "reuse relationships" in source code. These tuples can be consulted by the developer to learn about the correct API usage. An example of these rules is "you override X and you also override Y", which is based on information from the **structural context**; another example is "you implement I and override M", which uses a **type-system look-up** to find the required information. The features they consider are **class inheritance**, **member overrides**, and **invocations** (including constructors). However, the captured calls are on a class level though ("existence fact"). The authors later extend their work to include the complete **inheritance hierarchy** in their collected facts [13].

Zhong et al. [27] present *Java Rule Finder*, a tool that infers rules about a correct usage of a framework directly from its source code. The rules are prepared in a textual format and serve as a browsable documentation for developers. To learn the rules, the authors consider extracted facts with information about the **type hierarchy**, **invocations**, **fields**, and **field accesses**. They use these facts to encode information about the source code in a special graph-based notation.They did not use **control structures** and seem to just collect field read/writes, as well as invocations. Therefore, **points-to analysis** and **normalization** are not applicable. **Tracking**, however, seems applicable and might reduce the number of facts that are extracted in their approach, because less method relations need to be stored.

McBurney et al. [10] propose a system that automatically generates the documentation of an API method. Instead of analyzing the implemented behavior in the method, they look at callers of the method and extract descriptive information from there. Their system builds a call graph and uses the page rank algorithm to find regularly called methods. To do this, they need **method calls** and the **enclosing method declaration**. They also identify nouns and verbs in the tokens by splitting the identifiers found

in the signatures, as well as those found in the assigned variables. We believe that **tracking** might potentially improve the quality of the page rank output. **Normalization** might also increase the preciseness of a statement by separating unique steps. However, **points-to analysis** is not applicable.

Anomaly detection

Anomaly detection approaches learn characteristics of a typical/correct program. The underlying models are then used to detect deviations from this established norm. A very related area is bug detection, in which extended static analyses are used to prove the definitive existence of a problem in the code. Due to the similarities of both areas, we will discuss them together.

Monperrus et al. [14] propose *DMMC* to detect missing method calls. They extract object usages that describe how an object instance is used. They encode the **type of the object**, the **enclosing method**, and **all calls** on it. They track the object **inter-procedurally**, but intra-class. They implement a **points-to analysis**. Normalization is not necessary, because their traversal of the syntax tree does not need to handle nested expressions.

Li et al. [8] present *PR-Miner*, another detector for missing method calls. The tool extracts facts from a method such as the **type of variable declarations**, **variable names**, **assignments**, and **calls** and uses a **prefixing strategy to prevent name collisions** in different scoping levels. By applying frequent items mining, the tool ends up having a list of programming rules. The available source is then checked for violations of these rules. To remove false positives, they track method calls in **children calls** and in the **call-graph of the parent** (in which the current location is a child). They eliminate the violation report if the missing call is found there.

Pradel et al. [19] present an approach to build finite-state-automatons that describe valid protocols for using a specific type. The approach is based on their own framework that can be used to mine method sequences [18]. The concrete mining approach is exchangeable in this framework, but the most precise implementation uses a points-to analysis to identify objects and collects method calls that happen on or with each object. The approach also makes use of prior work by Jaspan et al. [7], which provided an extended **points-to analysis** to judge whether objects are contained in other objects, e.g., in collections. Both analyses work intra-procedurally, so there is an opportunity to apply **tracking** to increase the size of the sequences. Normalization is not applicable.

2.2 Input Requirements

Our survey reveals that each RSSE requires very specific information, but many approaches overlap in what they requirements. In order for `CART` to be useful, it is essential that our IR encodes this information.

One requirement all approaches have in common is that they require **Method Invocations**. For space reasons, we omit this invariant requirement in Table 1. A characteristic shared by almost all approaches is that they need **Resolved Types** [1, 2, 4, 6, 9, 10, 12–15, 20, 26, 27]. One approach uses smart heuristics to infer types [24], which would be unnecessary if the types were resolved. Some of the remaining RSSEs could potentially benefit from resolved types [4, 16, 17, 24]. Some approaches additionally require information about the **Type Hierarchy** [1,2,5,8,10,12–14,20,26,27], because they, for example, look up information from super classes or implemented interfaces.

The majority of the surveyed approaches make use of the **Structural Context**, such as the signatures of declared members. Many approaches use it to group invocations by the enclosing context. Four approaches group per class [4, 11–13], while others use the enclosing method for grouping [1, 2, 5, 6, 8–10, 14, 16, 17, 20, 24–27]. Even text-based approaches such as that used by Hindle et al. [4] could include structural context information in their model, i.e., they could consider complete signatures of methods instead of tokenizing it like the rest of the code.

The structural context is also used by some approaches to identify the reusable API of a type, i.e., the methods that are not project-specific. Zhong et al. [26] call these methods "third party API methods". While it is possible to use the **type hierarchy** to identify methods in a class that override a definition from a library or framework, the intermediate representation should also differentiate method declarations that are reusable from project-specific helper methods, e.g., public or protected methods.

Many approaches consider the **Invocation Order** in addition to **Method Invocations**. Notably, these are all RSSEs that deliver code snippets [4, 9, 15–17, 24, 26], because they need the information to ensure their proposals adhere to correct order. However, other approaches exploit this information as well [3, 15, 19, 21, 26, 27], e.g., to make proposals based on preceding invocations.

Many approaches consider **Control Structures**. Notably, these are again mainly RSSEs delivering code snippets [4, 15–17, 24, 26], which include control structures in their proposals. Others exploit this information to describe the context of invocations [1, 3]. Several other approaches require this information for points-to analysis [2, 14, 19–21].

Two approaches use **Variable Names** to predict call parameters [25] or the methods to invoke [3]. Many others use both **Variable Names** and **Assignments** to analyze data flow [2, 4, 8, 10, 14–17, 19–21, 24, 27]. Some approaches also consider **Casts** when analyzing how instances of certain types can be obtained [9, 15, 24, 26].

Other requirements are used only by a single approach each: First, one NLP-based RSSE uses **Syntax Tokens** [4], such as braces and semicolons, because it tokenizes the entire source code. Note that the technique does not conceptually depend on these tokens, since it treats all tokens

Table 1: Comparison of the surveyed papers. We mark if the respective dimension is used (●), applicable but not used (○), or not fully utilized (◐). A cell is empty if the dimension is not applicable to the approach.

	Input Requirements									Transform. & Analyses			
	Resolved Types	Type Hierarchy	Struct. Context	Invocation Order	Ctrl. Structures	Variables/Fields	Assignments	Casts	Other	Tracking	Points-to	Nested Express.	Slicing
Call Completion													
Bruch et al. [2]	●	●	●		●	●	●			●	●		
Proksch et al. [20]	●	●	●		●	●	●			●	●		
Raychev et al. [21]	●			●	●	●	●			○	●		
McCarey et al. [11]	●		●										
Heinemann et al. [3]	●			●	●	●				○		○	
Amann et al. [1]	●	●	●		●					○	◐		
Zhang et al. [25]	●		●			●				○	◐	○	
Snippet Recommender													
Nguyen et al. [17]	○		●	●	●	●	●			○	◐	○	
Nguyen et al. [16]	○		●	●	●	●	●		●	○	◐	○	
Mandelin et al. [9]	●		●	◐				●		●			●
Thummalap. et al. [24]	◐		●	●	●	●	●	●		●			
Hindle et al. [4]	○		◐	●	●	●	●	●	●	○		○	
Code Search													
Zhong et al. [26]	●	●	●	●	●			●		●			
Holmes et al. [5]	●	●	●							○			
Moreno et al. [15]	●			●	●	●	●	●	●	○		○	●
Hummel et al. [6]	●		●						●				
Documentation													
Michail [12, 13]	●	●	●							○			
Zhong et al. [27]	●	●	●	●		●	●			○			
McBurney et al. [10]	●	●	●			●	●			○		○	
Anomaly Detection													
Monperrus et al. [14]	●	●	●		●	●	●			●	●		
Li et al. [8]	●	●	●			●	●			●			
Pradel et al. [19]	●			●	●	●	●			○	●		

the same. Second, one approach requires JavaDoc **Comments** [15] from frameworks to enrich their proposed snippets with documentation. Third, one approach **Links to Source** [6] in a public repository, to be able to fetch the complete source code and additional files from the original repository.

2.3 Analysis Challenges and Tasks

In our survey we realized that all RSSEs that consider control or data flow in their analysis need to resolve the evaluation order of sub expressions in **Nested Expressions**, like nested calls `m1(m2())` or chained calls `o.m1().m2()`. This task is often intertwined with the RSSE's specific analysis, increasing its complexity and decreasing its maintainability. We can mitigate this, if we resolve the actual order once, in a *normalization* step, and implement the analyses on the normalized information.

Furthermore, many approaches reimplement static analyses: Several approaches employ **Points-to Analysis** [2, 14, 19–21] to distinguish unique object instances in their approach and some implement heuristics to achieve a similar goal [1, 16, 17, 25]. Others apply a **Tracking** analysis to follow the control flow into invoked methods [2, 8, 9, 14, 20, 24, 26]. Others do not apply such a technique, even though they might benefit from it [1, 3–5, 10, 12, 13, 15–17, 21, 25, 27]. Two approaches use **Slicing** to extract relevant parts of code snippets [9, 15]. Tool smiths could save much effort, if they could reuse a respective analysis.

2.4 Future Requirements

We identify further requirements that are, to the best of our knowledge, currently not required by any recommender system, but might be of interest to future work: A widely unexplored dimension of RSSEs is the **Framework Version** some input source code depends on. Since APIs might change between versions, this information should be considered when applying RSSEs in practice. **Generic Type Parameters**, as supported by Java and C#, are another potentially beneficial information for RSSEs. Finally, most surveyed approaches use source code from repositories as their input. The only exception being Amann et al. [1], who capture **Source Code under Edit** in the IDE. To support this approach, the static analyses need to handle potentially non-compiling code and the input data should capture both incomplete tokens and the cursor location.

3 Conclusion

In this appendix, we conducted a survey of 23 RSSE. We introduced each approach and focused on the static analysis to identify requirements for the input. We abstracted the concrete requirements, introduced several high-level requirements, and categorized for each paper if the requirements are applicable.

We found that the static analyses are very different and that they range from a simple traversal of the abstract syntax tree to sophisticated analyses of data flow and control flow properties of a program. The identified requirements can guide researchers that create unified tools or platforms for the development of RSSE.

References

[1] S. Amann, S. Proksch, and M. Mezini. Method-call Recommendations from Implicit Developer Feedback. In *Proceedings of the 1st International Workshop on CrowdSourcing in Software Engineering*. ACM, 2014.

[2] M. Bruch, M. Monperrus, and M. Mezini. Learning from examples to improve code completion systems. In *Proc. of ESEC/FSE*. ACM, 2009.

[3] L. Heinemann, V. Bauer, M. Herrmannsdoerfer, and B. Hummel. Identifier-based context-dependent API Method Recommendation. In *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*. IEEE, 2012.

[4] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu. On the naturalness of software. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 837–847. IEEE, 2012.

[5] R. Holmes, R. J. Walker, and G. C. Murphy. Approximate structural context matching: An approach to recommend relevant examples. *Software Engineering, IEEE Transactions on*, 32(12):952–970, 2006.

[6] O. Hummel, W. Janjic, and C. Atkinson. Code conjurer: Pulling reusable software out of thin air. *Software, IEEE*, 25(5):45–52, 2008.

[7] C. Jaspan and J. Aldrich. *Checking framework interactions with relationships*. Springer, 2009.

[8] Z. Li and Y. Zhou. Pr-miner: automatically extracting implicit programming rules and detecting violations in large software code. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*. ACM, 2005.

[9] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid mining: helping to navigate the api jungle. *ACM SIGPLAN Notices*, 40(6):48–61, 2005.

[10] P. W. McBurney and C. McMillan. Automatic documentation generation via source code summarization of method context. In *Proceedings of the 22Nd International Conference on Program Comprehension*. ACM, 2014.

[11] F. Mccarey, M. Ó. Cinnéide, and N. Kushmerick. Rascal: A recommender agent for agile reuse. *Artificial Intelligence Review*, 24(3-4):253–276, 2005.

[12] A. Michail. Data mining library reuse patterns in user-selected applications. In *Automated Software Engineering, 1999. 14th IEEE International Conference on.*, pages 24–33. IEEE, 1999.

[13] A. Michail. Data mining library reuse patterns using generalized association rules. In *Proceedings of the 22nd international conference on Software engineering*, pages 167–176. ACM, 2000.

[14] M. Monperrus, M. Bruch, and M. Mezini. Detecting missing method calls in object-oriented software. In *ECOOP 2010–Object-Oriented Programming*. Springer, 2010.

[15] L. Moreno, G. Bavota, M. Di Penta, R. Oliveto, and A. Marcus. How can i use this method? In *Proceedings of the 37th International Conference on Software Engineering*. IEEE, 2015.

[16] A. T. Nguyen, T. T. Nguyen, H. A. Nguyen, A. Tamrawi, H. V. Nguyen, J. Al-Kofahi, and T. N. Nguyen. Graph-based pattern-oriented, context-sensitive source code completion. In *Proceedings of the 34th International Conference on Software Engineering*, pages 69–79. IEEE Press, 2012.

[17] 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. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 383–392. ACM, 2009.

[18] M. Pradel, P. Bichsel, and T. R. Gross. A framework for the evaluation of specification miners based on finite state machines. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pages 1–10. IEEE, 2010.

[19] M. Pradel, C. Jaspan, J. Aldrich, and T. R. Gross. Statically checking api protocol conformance with mined multi-object specifications. In *Proceedings of the 34th International Conference on Software Engineering*, pages 925–935. IEEE Press, 2012.

[20] S. Proksch, J. Lerch, and M. Mezini. Intelligent Code Completion with Bayesian Networks. *ACM Transactions on Software Engineering and Methodology*, 2015.

[21] V. Raychev, M. Vechev, and E. Yahav. Code Completion with Statistical Language Models. In *Proceedings of the 35th Conference on Programming Language Design and Implementation*. ACM, 2014.

[22] M. P. Robillard, E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford. Automated api property inference techniques. *Software Engineering, IEEE Transactions on*, 39(5):613–637, 2013.

[23] M. P. Robillard, W. Maalej, R. J. Walker, and T. Zimmermann. *Recommendation Systems in Software Engineering*. Springer, 2014.

[24] S. Thummalapenta and T. Xie. Parseweb: a programmer assistant for reusing open source code on the web. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 204–213. ACM, 2007.

[25] C. Zhang, J. Yang, Y. Zhang, J. Fan, X. Zhang, J. Zhao, and P. Ou. Automatic parameter recommendation for practical api usage. In *Proceedings of the 34th International Conference on Software Engineering*, pages 826–836. IEEE Press, 2012.

[26] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei. Mapo: Mining and recommending api usage patterns. In *ECOOP 2009–Object-Oriented Programming*, pages 318–343. Springer, 2009.

[27] H. Zhong, L. Zhang, and H. Mei. Inferring specifications of object oriented apis from api source code. In *Software Engineering Conference, 2008. APSEC'08. 15th Asia-Pacific*. IEEE, 2008.