Exploring the Use of Deep Learning for Feature Location

Christopher S. Corley
The University of Alabama
Tuscaloosa, AL, USA
cscorley@ua.edu

Kostadin Damevski
Virginia Commonwealth University
Richmond, VA, USA
damevski@acm.org

Nicholas A. Kraft
ABB Corporate Research
Raleigh, NC, USA
nicholas.a.kraft@us.abb.com

***Abstract*—Deep learning models are a class of neural networks. Relative to n-gram models, deep learning models can capture more complex statistical patterns based on smaller training corpora. In this paper we explore the use of a particular deep learning model, document vectors (DVs), for feature location. DVs seem well suited to use with source code, because they both capture the influence of context on each term in a corpus and map terms into a continuous semantic space that encodes semantic relationships such as synonymy. We present preliminary results that show that a feature location technique (FLT) based on DVs can outperform an analogous FLT based on latent Dirichlet allocation (LDA) and then suggest several directions for future work on the use of deep learning models to improve developer effectiveness in feature location.**

***Index Terms*—deep learning; neural networks; document vectors; feature location**

I. Introduction

When starting a maintenance task, software developers commonly need to locate the relevant features in a potentially large and unfamiliar code base. Due to the difficulty and importance of this task, researchers have proposed a number of approaches to improve developers' effectiveness in locating features, largely based upon applying natural language analysis or text retrieval techniques to source code [1]. Most of the proposed feature location techniques have treated source code as an unordered set of natural language terms (i.e., as a bag-of-words), even though recent fundamental results have shown that source code contains context and flow that is even more pronounced than natural language text [2].

In this paper, we explore the use of deep learning, a particular class of neural networks that has shown promising results in modeling natural language, for feature location. In particular, we investigate the efficacy of document vectors [3] (DVs). DVs capture the influence of the surrounding context on each term, which can improve the ranking of results retrieved for a developer query [4]. For example, in the statement `diagram.redraw()` the word *diagram* is relevant to the word *redraw* and this relationship is captured by DVs. Therefore, when querying for *diagram*, program elements where *redraw* is also present are considered more relevant and thus are boosted in the rankings.

Deep learning models such as DVs also create a novel notion of semantic similarity between the source code terms. Semantic similarity is the result of mapping the corpus terms into a continuous semantic space, where synonyms, antonyms, and other semantic relations are encoded and easily composed together.

In our preliminary evaluation, we compare a feature location technique (FLT) based on DVs to an FLT based on latent Dirichlet allocation (LDA) using the benchmark by Dit et al. [5]. The benchmark comprises 633 features from six versions of four open source Java projects, and our results show that for many of the features, the DV-based FLT outperforms the LDA-based FLT. Our results also show that less time is needed for model training and inference in the DV-based FLT as compared to the LDA-based FLT. We also suggest directions for future work on the use of DVs (or other deep learning models) to improve developer effectiveness in feature location.

II. Background

Feature location systems retrieve a ranked list of program elements (e.g., methods or classes) for a developer query. In the *training* phase, feature location systems construct a model of the software, at the granularity of program elements, based on the natural language embedded in identifiers and comments. In the *retrieval* phase, given a natural language query, feature location systems use the model to retrieve the relevant program elements with high similarity to the query.

A. Feature Location Workflow

A feature location system based on deep learning, during its training phase, creates a contextual representation of the natural language terms embedded in the source code. This contextual representation includes influence from terms preceding and following each term, relative to their distance from that term. More intuitively, such models incorporate mutual influence between terms in the same method, while terms that are closer in distance (e.g. occur the same statement) influence each other more strongly.

Deep learning is based on a multi-stage neural network, consisting of several hidden layers in addition to single input and output layers. The input layer consists of an ordered sequences of identifiers extracted from the code. The multiple hidden layers serve to capture the context for each encountered term, representing the complex patterns of term contexts occurring in the corpus. The output layer consists of a vector for each term, which has been shown to carry semantic meaning.

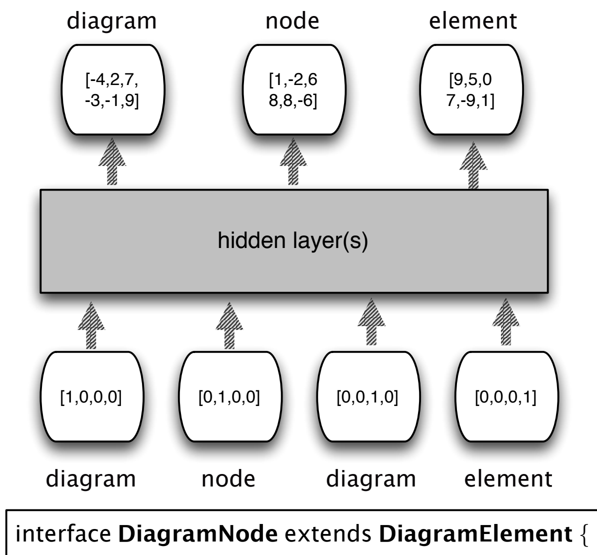

Fig. 1: A deep learning neural network encodes source code identifiers, in the order they appear in the source code, in its input layer. Using a deep structure of hidden layers, each term and its context receives a semantic vector representation. The output layer consists of vector for each term in the corpus; the vector feature size is arbitrary and does not need to relate to the number of terms in the corpus.

An example of this architecture for a single line of code is shown in Figure 1. Recent advances in this area have stemmed from the use of novel neural network architectures, including recurrent neural networks that connect the hidden layers back to the input layer, among other strategies. The systems are trained using backpropagation and gradient descent, techniques common to many neural network based models.

An extension to learning the semantic vector representation of words is the use of an additional vector that will encode the representation of a larger body of text, such as a paragraph or an entire document [3]. While comparing word vectors indicates semantic relations between two terms, comparing two document vectors carries a similar semantic connotation at the document level. For instance, the approach has been applied for determining the sentiment (i.e. positive, negative) of reviews on a popular movie recommendation site.

A number of preprocessing steps are commonly performed before the training phase of feature location systems. The steps commonly used are [6], [7]:

- *Splitting*: separate tokens into constituent words based on common coding style conventions (e.g., the use of camel case or underscores) and on the presence of non-letters (e.g., punctuation or digits)
- *Normalizing*: replace each upper case letter with the corresponding lower case letter
- *Filtering*: remove common words such as articles (e.g., 'an' or 'the'), programming language keywords, standard library entity names, or short words

During the retrieval state of feature location, a similarity measure (e.g. cosine similarity) between the words in the query and words in each program element is computed. The program elements are ranked based on this similarity metric and presented to the developer in descending order.

If document vectors are used then a special inference process is needed to infer a vector representation for the entire query, treated as a document in the corpus. Following this, the vectors of the query and each program element (i.e. method or class) can be compared to produce the final ranking.

B. Semantic Similarity

The result of the deep learning models described in this paper are vectors representing each term in the corpus [8]. Similar vectors can also be computed to represent an entire paragraph or document [3]. Semantic similarity is the notion that these vectors are composable semantically, e.g., the result of the operation *vec("The Eiffel Tower") - vec("Paris") + vec("London")* is closest to *vec("Big Ben")*. This capability is established automatically by the system, without any additional processing or supervised input.

In feature location, as in general information retrieval, the retrieval quality can only be as good as the quality of the developer query. Problems such as the dictionary mismatch problem, as well as the propensity of users to issue short queries, have previously been observed as common difficulties in using feature location tools in the field [9], [10]. Semantic similarity can be a useful capability in mitigating these problems, by performing query recommendation that allows the user to extend his or her queries with terms from the same corpus, or automatic query extension.

To illustrate this capability for source code-based corpora, we provide a set of illustrative examples gathered on the ArgoUML v0.22 in Table I. In many cases semantic similarity provides reasonable results for similar terms, though exceptions exist largely due to the limited appearance of certain words in the corpus. We anticipate that larger corpora, based on larger code bases, or leveraging related code bases or documents, could improve the results of semantic similarity even further.

III. Preliminary Study

In this section we describe the design of a study in which we compare a deep-learning-based FLT to a baseline topic-modeling-based FLT. In particular, we use a DV-based FLT and an LDA-based FLT, respectively.

A. Subject software systems

We employ the dataset of six software systems by Dit et al. [5]. The dataset contains 633 queries for method-level goldsets, as seen in Table II, and was automatically extracted from changesets that relate to the queries (issue reports).

ArgoUML is a UML diagramming tool[1]. jEdit is a text editor[2]. JabRef is a BibTeX bibliography management tool[3]. muCommander is a cross-platform file manager[4].

[1] http://argouml.tigris.org/ [2] http://www.jedit.org/
[3] http://jabref.sourceforge.net/ [4] http://www.mucommander.com/

TABLE I: Examples of semantically similar terms and their weight for a deep model trained on the ArgoUML code base. Only terms with weight > 0.6 are included.

Term(s)	*Semantically Similar Terms and Weight*
association	(roles 0.75), (role 0.72), (classifier 0.72), (connection, 0.61)
save	(saved 0.69), (pcs 0.64), (exists 0.63), (projects 0.60), (close 0.61), (file 0.60)
file	(filter 0.78), (zip 0.74), (exists 0.74), (persister 0.71), (files 0.69), (directory 0.69)
file + save	(exists 0.77), (saved 0.74), (filter 0.73), (zip 0.72), (unable 0.67), (projects 0.67), (persister 0.67), (files 0.66), (cant 0.66), (scheme 0.65)
explorer + diagrams - creating	(nodes 0.71), (deletion 0.67), (perspectives 0.65), (perspective 0.60), (updated 0.60), (modified 0.60)

TABLE II: Subject System Sizes and Queries

Subject System	Methods	Queries
ArgoUML v0.22	12353	91
ArgoUML v0.24	13064	52
ArgoUML v0.26.2	16880	209
Jabref v2.6	5357	39
jEdit v4.3	7305	150
muCommander v0.8.5	8799	92
Total	63758	633

B. Setting

We implemented our approach in Python v2.7 using the topic modeling library, Gensim [11] and our ANTLR v3 Java-based tool, Teaser[5], for parsing source code.

To build our corpora, we extract the documents representing methods from every Java file in the snapshot. The text of an inner method (e.g., a method inside an anonymous class) is only attributed to that method, and not the containing one. Comments, literals, and identifiers within a method are considered as text of the method. Block comments immediately preceding an method are also included in this text.

After extracting documents and tokenizing, we split the tokens based on camel case, underscores, and non-letters. We only keep the split tokens; original tokens are discarded. We normalize to lower case before filtering non-letters, English stop words [12], Java keywords, and words shorter than three characters long. We do not stem words. Here, we must be careful to not lose ordering of the words, as this is crucial for the deep learning approach.

For LDA, the approach is straightforward. We train the LDA model and query it using it's built-in inference methods.

In Gensim, the DV deep learning model is known as Doc2Vec. We train this DV model on the corpora, and query it to obtain rankings of methods related to the query. Because DV is a neural network with multiple layers (i.e., the document vector layer and the word vector layer), there are two approaches for measuring document similarity.

For the document-based layer, we can infer the document vector of the query and perform pair-wise similarity of it to the each method's document vector within the model. We also found it useful to consider the word-based layer. For this, we can get the word vector for each word in the query and sum them. We then take the query's summed vector and also perform pair-wise similarity to each document vector in the model.

[5] https://github.com/nkraft/teaser

Regarding configuration of the two models, we choose various sizes of K for each, where K for LDA is the number of topics and for DV is the number of features (i.e., document vector length).

C. Data Collection and Analysis

To evaluate the performance of an FLT we cannot use measures such as precision and recall. This is because the FLT creates the rankings pairwise, causing every entity being searched to appear in the rankings. Poshyvanyk et al. define an effectiveness measure that can be used for FLTs [13]. The effectiveness measure is the rank of the first relevant document and represents the number of source code entities a developer would have to view before reaching a relevant one. The effectiveness measure allows evaluating the FLT by using the mean reciprocal rank (MRR) [14]:

$$MRR = \frac{1}{|Q|}\sum_{i=1}^{|Q|}\frac{1}{e_i} \quad (1)$$

where Q is the set of queries and e_i is the effectiveness measure for some query Q_i.

We also want to explore the differences in terms of computational overhead for using each model. We collect information such as time taken to train the model and the time taken to process the goldsets, i.e., the average time to query and rank each subject system. We train and query all models on a system running ArchLinux with an Intel Core-i7 4Hz processor and 32 GB of memory.

D. Results and Discussion

Table V summarizes the results of each subject system for each model and query approach. We bold which of the three approaches is greatest for each value K at steps of 100. To the right of Table V is Figure 2, a graphical view of the all values of K at steps of 25.

There are a handful of clear trends. For example, the ArgoUML projects perform similarly across each version. For all three versions, the DV word vector summation performs best for many values of K, with few exceptions in favor of

TABLE V: MRR

System	Approach	100	200	300	400	500
ArgoUML v0.22	LDA	0.0175	0.0295	0.0271	**0.0611**	0.0220
	DV Inference	0.0115	0.0105	0.0096	0.0184	0.0162
	DV Summation	**0.0775**	**0.0570**	**0.0625**	0.0587	**0.0601**
ArgoUML v0.24	LDA	0.0441	0.0373	0.0655	**0.0779**	0.0344
	DV Inference	0.0246	0.0152	0.0260	0.0258	0.0380
	DV Summation	**0.0827**	**0.0906**	**0.0874**	0.0691	**0.0942**
ArgoUML v0.26.2	LDA	0.0493	0.0628	**0.0857**	0.0703	**0.0811**
	DV Inference	0.0404	0.0218	0.0290	0.0364	0.0403
	DV Summation	**0.0847**	**0.0890**	0.0813	**0.0834**	0.0805
JabRef v2.6	LDA	0.0055	0.0364	**0.1304**	**0.0781**	**0.0548**
	DV Inference	0.0262	**0.0463**	0.0318	0.0289	0.0234
	DV Summation	**0.0450**	0.0373	0.0455	0.0382	0.0428
jEdit v4.3	LDA	0.0670	0.0432	0.0641	0.0693	0.0607
	DV Inference	0.0341	0.0282	0.0369	0.0354	0.0450
	DV Summation	**0.0872**	**0.0791**	**0.0825**	**0.0814**	**0.0679**
muCommander v0.8.5	LDA	0.0392	0.0217	0.0198	0.0559	0.0329
	DV Inference	**0.0977**	**0.0771**	**0.0800**	**0.0665**	**0.0838**
	DV Summation	0.0652	0.0623	0.0703	0.0606	0.0538

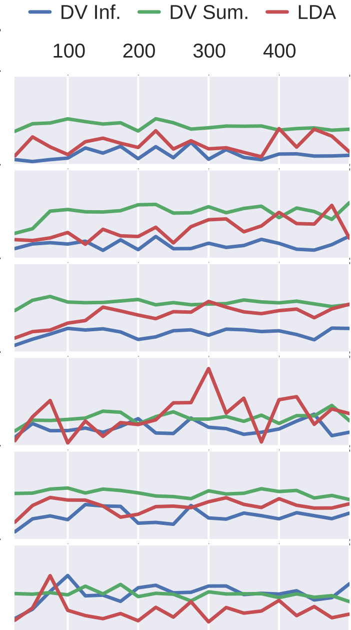

Fig. 2: MRR

TABLE III: Model training times for 100 topics

	LDA	DV
ArgoUML v0.22	0m 58.207s	0m 02.070s
ArgoUML v0.24	1m 05.507s	0m 02.267s
ArgoUML v0.26.2	1m 21.176s	0m 02.736s
JabRef v2.6	0m 29.504s	0m 01.280s
jEdit v4.3	0m 36.701s	0m 01.519s
muCommander v0.8.5	0m 42.897s	0m 01.696s

TABLE IV: Average time to rank per query for 100 topics

	LDA	DV Inf.	DV Sum.
ArgoUML v0.22	0.943362s	0.225868s	2.118835s
ArgoUML v0.24	1.686980s	0.259615s	1.824923s
ArgoUML v0.26.2	0.744468s	0.300956s	2.715062s
JabRef v2.6	0.957000s	0.124128s	0.720589s
jEdit v4.3	0.405700s	0.134080s	1.017060s
muCommander v0.8.5	0.685543s	0.165663s	1.056108s

LDA. Interestingly, DV inference has the worst performance across all three versions. This trend continues with JabRef and jEdit, but does not for muCommander. Surprisingly, DV inference always performs best for muCommander.

A second trend is how few features (K) DV needs to perform well. As shown in Figure 2, many of the projects achieve high DV summation performance by $K = 100$, and plateau after that. By contrast, LDA can require as many as 300 to 500 topics. This observation is noteworthy because fewer topics generally implies less training time.

Tables III and IV summarize the time taken to train each model on the corpus and the average query and rank time for all queries in the goldset.

Little time is required to train a DV model — training finishes in under 3 seconds for all systems. On the other hand, training an LDA model can take up to about 90 seconds. The average time to query and rank all queries in the goldset shows that DV inference is fastest, while the word vector summation is on par or worse than LDA.

IV. Related Work

Statistical natural language models, such as the n-gram model, have seen widespread use for a variety of natural language processing tasks due to their simplicity and effectiveness when trained with a substantial corpus of text. Software engineering researchers have shown n-gram models to be even more effective for source code than for natural language documents [2].

Recent research in natural language modeling has introduced deep learning methods, consisting of specific classes and architectures of neural networks, that can produce capable statistical models of natural language text able to capture more complex patterns while being trained using smaller corpora relative to the n-gram model [3], [8]. Using a set of optimizations strategies such models can be built at reasonable computational costs. In this paper we examine the effectiveness and potential applicability of these deep learning models for the problem of feature location in software engineering.

White et al. [15] reported promising results in applying deep learning models for source code, outperforming models based on similar n-gram configurations on code completion tasks. This work establishes the use of deep learning for software engineering problems, sketching out several avenues for future use of such models, including code completion and the building

of synonym dictionaries, among others. Our work explores these models for the feature location problem during software maintenance. A notable difference to the prior experiments conducted by White et al. is that the deep models in this paper are based solely on terms found in identifiers and comments in the source code of a single software project, while White et al. considered a larger corpus consisting of the entire source listing of several combined projects.

Previous feature location techniques that incorporated surrounding context in the model have shown high degrees of effectiveness. Examples include the use of verb and direct object pairs [16] and statement level markov random fields [4]. Deep learning approaches allow the inclusion of broader context than these previous approaches.

Howard at al. [17] utilized rules based on natural language text in leading method comments to mine synonym dictionaries from source code repositories. One of the additional capability of deep language models is the ability to mine such terms at a greater scale relative to the their approach.

V. Conclusions and Future Work

In this work we present a preliminary study of a deep-learning-based FLT using document vectors (DV). We find that training the DV model has low computational overhead (i.e., is fast), while maintaining accuracy on par with LDA. We find DV to be a promising solution to implementing smarter developer search tools in the IDE, a task to which more computationally-intensive models such as LDA are less well-suited.

One direction for future work is to explore the effects of parameter tuning on the performance of DV. Multiple studies [18] show that selection of appropriate parameter values is key to the performance of an LDA-based FLT. Thus, the first question to address is whether the same is true of a DV-based FLT, and if so, the next question is how to select DV parameters for a particular subject system. The results of our study show that a DV-based FLT can provide better accuracy than an LDA-based FLT while requiring fewer computational resources, and more intelligent parameter selection for DV could further improve its accuracy.

Another direction for future work is to use the semantic relations encoded by a DV model for query recommendation or refinement. We would like to investigate the efficacy of these semantic relations for recommending query terms based on a partial query. That is, while a developer is entering a query, a recommender could use a DV model of the subject system to recommend additional query terms. Such an approach would be similar (in function) to the ones implemented in the Sando [19] and CONQUER [20]. Similarly, given its relative computational efficiency, using DV as the basis for an IDE-based search tool is a realistic goal.

Further directions for future work include extending the DV model to incorporate program structure information, an approach which has shown useful when combined with text retrieval models [21], or extending the DV model to incorporate natural language information such as part-of-speech tags [16] or phrasal representations [22].

References

[1] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, “Feature location in source code: a taxonomy and survey,” *Journal of Software: Evolution and Process*, vol. 25, no. 1, pp. 53–95, 2013.

[2] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, “On the Naturalness of Software,” in *Proceedings of the 34th ACM/IEEE International Conference on Software Engineering*, 2012, pp. 837–847.

[3] Q. Le and T. Mikolov, “Distributed Representations of Sentences and Documents,” in *Proceedings of the 31st International Conference on Machine Learning*, T. Jebara and E. P. Xing, Eds., 2014, pp. 1188–1196.

[4] E. Hill, B. Sisman, and A. Kak, “On the use of positional proximity in IR-based feature location,” in *Proc. of IEEE Conf. on Software Maintenance, Reengineering and Reverse Engineering*, 2014, pp. 318–322.

[5] B. Dit, A. Holtzhauer, D. Poshyvanyk, and H. Kagdi, “A dataset from change history to support evaluation of software maintenance tasks,” in *Proceedings of the 10th IEEE Working Conference on Mining Software Repositories*, 2013, pp. 131–134.

[6] A. Marcus, A. Sergeyev, V. Rajlich, and J. I. Maletic, “An information retrieval approach to concept location in source code,” in *Proc. of 11th Working Conf. on Reverse Engineering*, 2004, pp. 214–223.

[7] A. Marcus and T. Menzies, “Software is data too,” in *Proceedings of the FSE/SDP workshop on Future of software engineering research*. ACM, 2010, pp. 229–232.

[8] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, “Distributed Representations of Words and Phrases and their Compositionality,” in *Advances in Neural Information Processing Systems 26*, 2013, pp. 3111–3119.

[9] S. Haiduc and A. Marcus, “On the Effect of the Query in IR-based Concept Location,” in *Proceedings of the 19th IEEE International Conference on Program Comprehension*, 2011, pp. 234–237.

[10] K. Damevski, D. Shepherd, and L. Pollock, “A field study of how developers locate features in source code,” *Empirical Software Engineering*, pp. 1–24, 2015.

[11] R. Řehůřek and P. Sojka, “Software framework for topic modelling with large corpora,” in *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*, May 2010, pp. 45–50.

[12] C. Fox, “Lexical analysis and stoplists,” in *Information Retrieval: Data Structures and Algorithms*, W. Frakes and R. Baeza-Yates, Eds. Prentice-Hall, 1992, pp. 102–130.

[13] D. Poshyvanyk, Y.-G. Guéhéneuc, A. Marcus, G. Antoniol, and V. Rajlich, “Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval,” *IEEE Trans. on Software Engineering*, vol. 33, no. 6, pp. 420–432, 2007.

[14] E. M. Voorhees, “The trec-8 question answering track report.” in *TREC*, vol. 99, 1999, pp. 77–82.

[15] M. White, C. Vendome, M. Linares-Vásquez, and D. Poshyvanyk, “Toward deep learning software repositories,” in *Proceedings of the 12th Working Conference on Mining Software Repositories*, 2015.

[16] D. Shepherd, Z. P. Fry, E. Hill, L. Pollock, and K. Vijay-Shanker, “Using natural language program analysis to locate and understand action-oriented concerns.” ACM Press, 2007, p. 212.

[17] M. J. Howard, S. Gupta, L. Pollock, and K. Vijay-Shanker, “Automatically Mining Software-based, Semantically-similar Words from Comment-code Mappings,” in *Proceedings of the 10th Working Conference on Mining Software Repositories*, 2013, pp. 377–386.

[18] L. R. Biggers, C. Bocovich, R. Capshaw, B. P. Eddy, L. H. Etzkorn, and N. A. Kraft, “Configuring latent dirichlet allocation based feature location,” *Empirical Software Engineering*, vol. 19, no. 3, pp. 465–500, 2014.

[19] D. Shepherd, K. Damevski, B. Ropski, and T. Fritz, “Sando: an extensible local code search framework,” in *Proc. of ACM SIGSOFT 20th Int'l Sym. on the Foundations of Software Engineering*, 2012.

[20] M. Roldan-Vega, G. Mallet, E. Hill, and J. Fails, “CONQUER: A tool for nl-based query refinement and contextualizing source code search results,” in *Proceedings of the 29th IEEE International Conference on Software Maintenance*, 2013.

[21] B. Bassett and N. A. Kraft, “Structural information based term weighting in text retrieval for feature location,” in *Proc. 21st IEEE Int'l Conf. Program Comprehension*, 2013, pp. 133–141.

[22] E. Hill, L. Pollock, and K. Vijay-Shanker, “Improving source code search with natural language phrasal representations of method signatures,” in *Proceedings of the 26th IEEE International Conference on Automated Software Engineering*, 2011.