2014 IEEE International Conference on Software Maintenance and Evolution

Dynamic Inference of Change Contracts

Tien-Duy B. Le[1], Jooyong Yi[2], David Lo[1], Ferdian Thung[1], and Abhik Roychoudhury[2]
[1]School of Information Systems, Singapore Management University, Singapore
[2]School of Computing, National University of Singapore, Singapore
{btdle.2012,davidlo,ferdiant.2013}@smu.edu.sg, {jooyong,abhik}@comp.nus.edu.sg

***Abstract*—Software evolves and thus developers frequently make changes to systems that are logged in version control systems. These changes are often poorly documented – often commit logs are empty or only contain minimal information. Thus, it is often a challenge to understand why certain changes are made especially if they were introduced many months or even years ago. Understanding these changes is important when pertinent questions are raised during future bug fixing or software evolution efforts. Thus, there is a need for an automated approach that can help developers better document changes with little or minimal effort.**

To address this need, we propose a dynamic inference framework that automatically infers change contracts. Recently, change contract is proposed as a formalism to capture the semantics of changes. Different from standard program contract, change contract focuses in expressing the changed behavior between two versions of software systems. Our system infers candidate contracts based on actual changes and developers can further modify these contracts to reflect intended changes. We have performed a preliminary evaluation of our dynamic inference framework on a set of 15 real bug fixing changes from AspectJ with promising results.

I. Introduction

Developing software is an evolving task. In the midst of constant changes of software, developers often want to compare two different versions of code (e.g., the previous code vs. the updated code) to understand and debug a program or for other reasons. When comparing the two versions of code, what developers want to see is beyond the lexical/syntactic differences. It is often semantic differences that developers want to see ultimately. For example, when debugging a program, developers often want to know how a function or a method behaves differently (e.g., returns different values) between two versions. It is desirable that such semantic differences are described in a succinct and comprehensible way, so that they can be easily understood.

Semantic differences are also commonly used as one of information sources when mining software repositories (MSR). For example, Thung et al. categorize different kinds of bugs in machine learning systems by analyzing semantic differences [20]. To extract semantic differences from software repositories, metadata such as commit messages and bug reports are typically used. However, commit messages are often minimal and even empty in many cases. Also, bug reporting is not mandatory. Even in cases where commit messages and bug reports are available, the kinds and the preciseness of semantic differences that can be mined from those metadata are limited.

To meet the need for information about semantic differences, we propose an approach that automatically infers semantic differences between two versions. We use software change contracts recently introduced by Qi et al. [19], [23] which are program-like specifications used as the representation of semantic differences. A change contract can be attached to a method to describe semantic behavioral differences between the previous and updated version of the method. Inferred change contracts can be provided for developers as the summary of semantic changes. By looking at such a summary, not only can developers understand semantic changes at the high level, but they can also handily compare inferred change contracts with the intended changes they want to make. In case the inferred changes do not match the intended changes, developers could make the necessary corrections. Inferred change contracts can also be used by MSR researchers to recover semantic differences automatically especially when information provided in commit logs and bug reports is insufficient.

To infer change contracts, we take a dynamic analysis approach where two versions of a program are instrumented and monitored to gather necessary program states at various program points: entry and exit points of relevant methods and when exceptions are thrown. By comparing the resultant program state logs for the previous and updated version, a change contract can be constructed.

We have performed a preliminary evaluation of our approach on a set of 15 real bug fixing changes from AspectJ. The initial results are promising; in majority of cases (9 cases), the accuracy of an inferred change contract is over 60% (see Section IV for the definition of accuracy). We have also compared our approach with DeltaDoc by Buse and Weimer [5] which can also be used to document changes. We find that our approach outperforms DeltaDoc on each one of the 15 bug fixing changes.

In the remainder of this paper, we describe change contract in Section II. We elaborate our proposed technique in Section III. We present our preliminary experiments in Section IV. We discuss related studies in Section V. We conclude and mention future work in Section VI.

II. Change Contract

Qi et al. introduced change contracts to capture intended changes programmers want to make [19], [23]. Similar to Java Modeling Language (JML) contract [4] that describes program behavior at method boundaries through pre/post conditions, change contracts can describe how a method behaves differently (e.g., returns different values) between two versions.

The change contract language is an extension of JML. It is customized to express changes between versions in a concise and intuitive way. Most notably, the change contract language can describe a property like “*whenever* $out > 0$ *holds,* $out' == out + 1$” where out and out' denote output of

1063-6773/14 $31.00 © 2014 IEEE
DOI 10.1109/ICSME.2014.72

IEEE computer society

the previous and the updated program version, respectively. The additional flexibility of relating the program outputs across program versions often leads to concise and intuitive change contract specifications. The grammar of the language is described in [19], [23]. Here, we just illustrate it by means of an example.

An example of a change contract is shown below:

```
public class C {
  /*@ changed_behavior
    @  requires i == 0;
    @  when_signaled (Exception e)
    @         e.getMessage().contains(”Inconsistent State”);
    @  signals (Exception e) false; @*/
  int foo(int i, boolean b);
}
```

Suppose that the above is the change contract for changes made between revision r_1 and r_2 of C.java. Then, the above change contract mandates the following behavioral changes between r_1 and r_2 when the method foo is called with its parameter i set to zero (see the requires clause): whenever method foo of r_1 signals (or raises) an Exception which contains a message “Inconsistent State”, method foo of r_2 should not raise any exception.

As another example, if behavioral changes should occur when the above method foo of r_1 terminates normally without signaling an exception, say returning 0, then one can replace the above when_signaled clause with: “when_ensured \result == 0;”. If the new behavior is that the method returns 1, then one can replace the above signals clause with: “ensures \result == 1;”.

III. Proposed Approach

In our approach, the main idea is to capture differences in the dynamic behaviors of two versions of a code to infer software change contracts. Our framework captures differences in dynamic behaviors of the two software versions by utilizing test cases where the previous and updated versions return different outputs. For example, these test cases might cause the previous version to throw an exception or return unexpected results, but no such problems are observed for the updated version when running these test cases.

Figure 1 depicts the overall process of how we infer change contracts. Our approach takes as inputs two versions of a software program referred to as the previous and updated version respectively. From these versions, we can know the files that are changed. Next, we instrument the changed files in the previous and updated version to record the behaviors of these versions when test cases are run. We instrument each version such that program states are output at entry and exit of relevant methods, and when exceptions are raised in the relevant methods during the executions of the test cases. After we run the test cases, we would have two logs of program states – one for the previous version, and another for the updated version. Each of the logs contains program states for every invocation of changed methods in the changed files.

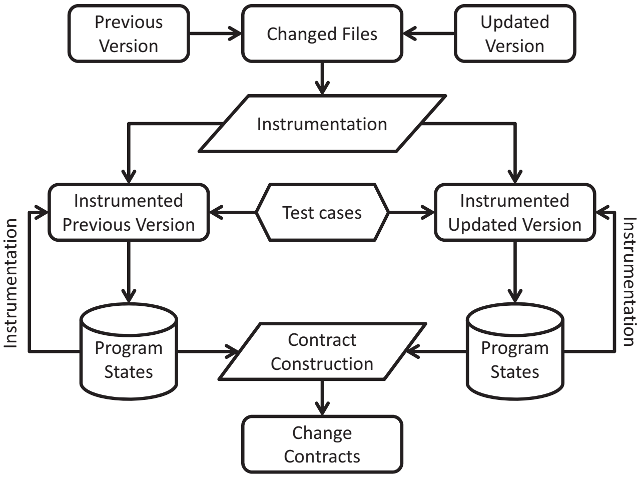

Fig. 1: Overall Process

Finally, we compare program states in these logs to construct software change contracts. If no change contract can be constructed, we expand our search by instrumenting neighboring uninstrumented methods that call the current set of instrumented methods. We do this since at times differences in behavior are exhibited at the caller of a buggy method rather than at the buggy method itself. For example, a buggy method can wrongly return an invalid value (e.g., null), but the problem (e.g., null pointer exception) only occurs when this value is used in other methods. We instrument these neighboring methods and collect additional program states by re-running the test cases. The process is repeated until at least one change contract is constructed.

In our process, there are two primary steps: instrumentation and contract construction. We describe each of these steps in the following paragraphs.

a) Instrumentation: We develop our own instrumentation technique that stores program states into files at particular execution points. By using the Eclipse JDT technology, our technique automatically inserts statements to capture program states at entry points and exit points of selected methods. Furthermore, our instrumentation technique also injects *try-catch* statements to capture program states when exceptions are raised. Table I shows an example of an instrumented method.

A program state captured at an execution point contains concrete values of objects and their fields belonging to the Java class containing the execution point. For an object, if the types of its internal fields are not primitive data types, we continue to explore and capture internal values of these fields recursively until reaching fields with primitive data types.

b) Contract Construction: After an instrumented version is run, a program state log is created for each instrumented method. A program state log for a method M is a list of program states that are logged at various invocations of M. Program states are recorded at M's entry and exit or when an exception is raised during an invocation of M. Given program states for two invocations of method M (one in the previous and another in the updated version), a change contract for M is constructed if the program states satisfy one of the two scenarios illustrated in Figure 2.

In the first scenario, the program states at the entries of the two method invocations match, but the program states at

TABLE I: Example of Instrumented Code

```
public Object foo(Object a, Object b) {
 Instrumentation instrumenter = new
     Instrumentation("foo");
 //print program state at entry point
 instrumenter.printState();
 try {
   if (...) {
     //print program state at exit point
     instrumenter.printState();
     return ...;
   }
   //print program state at exit point
   instrumenter.printState();
   return ...;
 }
 catch (RuntimeException e) {
   //print program state
   //when an exception is raised
   instrumenter.printState();
   throw e;
 }
}
```

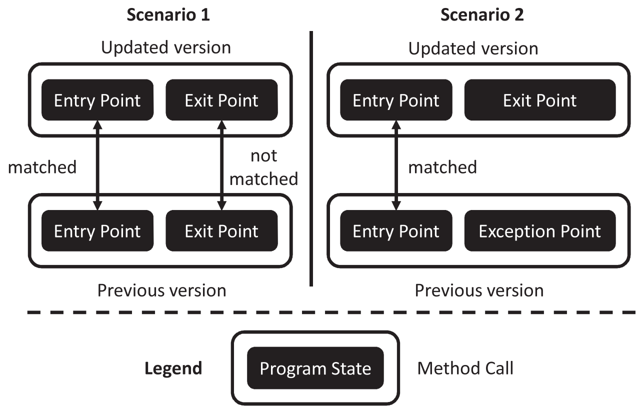

Fig. 2: Scenarios For Change Contract Construction

Algorithm 1: Contract Construction

```
Input: Method $M$
       $call_p(M)$: Invocations of method $M$ in the previous version
       $call_u(M)$: Invocations of method $M$ in the updated version
Output: Change contract of method $M$ or $\{\}$
foreach $c_p$ in $call_p(M)$ do
    foreach $c_u$ in $call_u(M)$ do
        // Check whether program states at M's
           entry point match or not
        if matched($c_p$.entry,$c_u$.entry) is true then
            // Scenario 1
            if matched($c_p$.exit,$c_u$.exit) is false then
                Construct and output a change contract
            // Scenario 2
            if $c_p$.exception == null and
            $c_u$.exception <> null then
                Construct and output a change contract
return $\{\}$
```

the exits of the two method invocations are different. In the second scenario, the program states at the entries of the method invocations match, but one of the invocations (i.e., invocation of M in the previous version) throws an exception. These two scenarios indicate that the behavior of method M has changed. Thus, we create a change contract when either one of the two scenarios occurs.

Algorithm 1 shows the procedure to infer a change contract for method M given two program state logs. It iterates through invocations of method M (in previous and updated versions) and check if any of the two scenarios occurs. If either one occurs, the procedure constructs a change contract as follows:

1) If the first scenario occurs, a change contract is constructed with a requires, a when_ensured, and an ensures clause. The requires clause contains predicates corresponding to the values in the program state at the entry point of M (e.g., i == 0). The when_ensured clause contains a predicate corresponding to the return statement at the exit point of M in the previous version (e.g., \result == 0). The ensures clause contains a predicate corresponding to the return statement at the exit point of M in the updated version (e.g., \result == 1).
2) If the second scenario occurs, a change contract is constructed with a requires, a when_signaled, and a signals clause. The requires clause contains predicates corresponding to the values in the program state at the entry point of M. The when_signaled clause contains a predicate that specifies the message of the exception that is raised in the previous version. The signals clause contains a predicate that specifies that the exception is no longer raised, i.e., signals (Exception e) false.

The output of Algorithm 1 is a set of inferred change contracts for method M. Change contracts that are syntactically the same are merged into one. Note that in this preliminary study we only construct change contracts of specific formats. In the next section, we investigate how close the inferred contracts are to the ideal ones.

IV. Preliminary Experiment

Dataset & Evaluation Metric. We use AspectJ as our subject program. AspectJ is a compiler for the aspect-oriented extension of Java and it contains about 75k lines of code. In our preliminary experiments, we consider 15 bug fixing changes of AspectJ extracted from the iBugs dataset [6]. We apply our approach to automatically infer change contracts from these changes, focusing on methods whose signatures remain unchanged, and compare the inferred change contracts with ground truth change contracts that we manually construct. To construct each ground truth change contract, we investigate the versions of the code before (previous version) and after (updated version) the change. We also execute test cases that are used to reproduce the bug in the previous and updated versions and analyze the resultant execution traces. Next, we manually craft the change contract based on our analysis of the change and execution traces.

To compare the inferred and ground truth change contracts we extract predicates (e.g., "result == FuzzyBool.NO", "inParameterizedType == true", etc.) that appear in the contracts and see how many of them are the same. These predicates can appear in various clauses in the change contracts, e.g., when_signaled, signals, ensures, etc. In our experiment, predicates in these clauses are usually expressions with "==" and "!=" binary operators where the left and right hand side are identifiers or expressions directly extracted from program source code, change contract special symbols (e.g., \result, etc.), or concrete values. Based on the common predicates, we use the following formula to measure the accuracy of each inferred contract:

$$Acc = \frac{\mid Predicates_{inferred} \cap Predicates_{ground-truth} \mid}{\mid Predicates_{inferred} \cup Predicates_{ground-truth} \mid}$$

In the above formula, $Predicates_{inferred}$ is the set of all predicates in the various clauses of an inferred change contract. $Predicates_{ground-truth}$ is the set of all predicates in the various clauses of a ground-truth change contract.

Results. Our approach infers change contracts for 18 methods from the 15 AspectJ changes. Table II shows the accuracy of our approach for each of the 15 changes. For changes where multiple change contracts are inferred for different methods, we report the average accuracy across the contracts.

From the table, we find that we can achieve an accuracy of up to 77.78%. Also, for the majority of the changes (9 out of the 15 changes), the accuracy is 60% or higher. However for two changes, the accuracy is less than 50%. Thus, the result is promising although there is room for further improvement. By manually inspecting inferred change contracts, we find that the inaccuracies of change contracts are mostly caused by generated predicates in requires clauses as these predicates capture concrete values rather than generalizations of these values.

TABLE II: Accuracy of Change Contracts Inferred by Our approach vs. Accuracy of Change Documentations Inferred by DeltaDoc.

	Our Approach		**DeltaDoc**	
Bug ID	**# Change Contracts**	**Average Accuracy**	**# Change Documentations**	**Average Accuracy**
132349	1	77.78%	0	0.00%
87376	1	66.67%	1	15.38%
120474	1	66.67%	1	33.33%
121616	1	66.67%	0	0.00%
125475	1	66.67%	0	0.00%
131933	1	66.67%	1	33.33%
158624	1	66.67%	0	0.00%
109614	1	62.50%	0	0.00%
100227	1	60.00%	1	25.00%
88652	3	58.99%	0	0.00%
173602	1	57.14%	0	0.00%
130869	1	54.55%	0	0.00%
133307	2	52.78%	0	0.00%
128237	1	57.14%	0	0.00%
122742	1	33.33%	0	0.00%

Comparison with DeltaDoc. Buse and Weimer propose a tool named DeltaDoc to document changes [5]. It can be obtained from https://code.google.com/p/deltadoc/. We apply DeltaDoc to document the 15 AspectJ bug fixing changes, and evaluate it using the same accuracy formula. Table II shows the accuracy of change documentations inferred by DeltaDoc.

From the table, the accuracy of DeltaDoc inferred change documentations are non zero for only 4 of the 15 AspectJ changes. The highest accuracy is only 33.33%. For 4 of the 15 AspectJ change, the accuracy is zero as DeltaDoc inferred change documentations are either empty or trivial and do not capture intended behavioral changes. For 7 of the 15 AspectJ changes, the accuracy is zero as DeltaDoc cannot complete the inference process (e.g., out of memory error). DeltaDoc employs symbolic execution which is expensive and often does not scale for large programs due to path explosion problem. The experiment results show that our approach outperforms DeltaDoc since it is able to document *all* of the AspectJ changes and its accuracy is higher than that of DeltaDoc.

Threats to Validity. We need to acknowledge several threats to the validity of our experiment results. Threats to internal validity relates to errors in our experiments. We derive the ground truth change contracts manually. We have carefully infer these contracts. However, there might be errors that we did not notice. Due to the lack of available ground truth, authors of many past studies also infer ground truth data manually by themselves, e.g., [9]. Threats to external validity relates to the generalizability of our findings. In this preliminary study, we have only investigated 15 bug fixing changes from one software system. In the future, we want to reduce this threat further by investigating more changes from additional software systems. Threats to construct validity relates to the suitability of our evaluation metric. In this paper, we make use of accuracy, which is a standard metric used before in many studies [10].

V. Related Work

The most related work is DeltaDoc proposed by Buse and Weimer [5]. It also infers semantic differences between two program versions at the method level, and report those differences, as in the following example where φ represents an input condition:

```
// An inferred DeltaDoc of method m:
When calling m(), If φ, return 1 Instead of return 0.
```

Despite the syntactic similarity between a change contract and a DeltaDoc, they are not directly exchangeable due to their different semantics. As mentioned in Section II, a change contract can describe a property like

$$\textit{whenever } result == 0 \textit{ holds, } result' == result + 1$$

where $result$ and $result'$ denote the return value of the previous and the updated program version, respectively. Notice that such additional ability of describing an output-output relationship simplifies the description; the input condition φ can be greatly simplified, and often can even be removed entirely, as in this example. Meanwhile, DeltaDoc, lacking in such ability, uses ad hoc heuristics to simplify φ, in exchange for losing the preciseness of an inferred DeltaDoc.

Also, the current DeltaDoc inference system cannot cover a pattern such as "exceptions unexpectedly thrown in the previous version are resolved in the updated version", which can be detected by our system. This is mainly because the DeltaDoc inference system compares the behaviors of two versions at the statement level – e.g., how the path condition of the return statement changes across versions – while we perform comparison outside the method body.

There are a number of studies on specification mining and inference. The most popular specification mining technique is Daikon which infers invariants at program entries and exits and specific program points of interest [7]. Many other studies infer specifications in the form of a finite state machine, e.g., [1], [13], [18], [16], [3], [2]. Ammons et al. is one of the pioneers that infer specifications in the form of finite state machines [1]. They collect execution traces from programs and make use of an algorithm named sk-string to construct a finite state machine that abstracts the execution traces. One of the latest works is the work by Beschastnikh et al. which propose a framework named InvariMint that unifies various specification mining techniques that extract specifications in the form of finite state machines [2]. There are also other specification mining techniques that infer specifications in the form of rules, e.g., [12], [22], [21], [17]. Li et al. extract rules from source code methods by employing a combination of lightweight static analysis and data mining [12]. Their proposed technique converts each method into a multi-set of program elements (e.g., method calls) and use these multi-sets to infer rules that govern the invocations of these program elements (e.g., `close` must be called after `open`). One of the latest works is the work by Lo et al. which mines specifications in the form of temporal rules, expressible in Linear Temporal Logic, enriched with Daikon invariants [17]. Yet, other studies infer specifications in the form of sequence diagrams, e.g., [15], [24], [14], [11], [8]. Kumar et al. infer specifications in the form of Message Sequence Chart (MSC) graph from traces of distributed systems [11]. One of the latest works is the work by Fahland et al. which infers sequence diagrams in the format of branching-time Live Sequence Charts [8]. A specification mining technique recovers specifications for a particular version of a code. Different from specification mining techniques, we recover specifications of changes from one version of a code to another. Thus, our work is orthogonal and complements existing specification mining studies.

VI. Conclusion and Future Work

In this work, we propose a dynamic analysis approach to infer change contracts. Change contract is recently proposed as a formalism to capture behavioral changes between two versions of a code. To infer change contracts from two versions of a code (previous and updated version), we instrument the two versions, execute test cases, and collect program states at input and output of relevant methods and when exceptions are raised in these methods. Program states for the previous version and those for the updated version are then compared to construct change contracts. We have performed a preliminary experiment to evaluate the effectiveness of our approach on 15 real AspectJ bug fixing changes. Our experiments show that we can infer change contracts with an accuracy of up to 77.78%. Also, for the majority of the changes (9 out of the 15 changes), the accuracy is 60% or higher. Furthermore, the accuracy of our approach outperforms that of DeltaDoc, a state-of-the-art change documentation inference approach, for each change.

In the future, we plan to improve the effectiveness of our approach to achieve a higher accuracy. In this preliminary study, a method can have multiple inferred contracts and we only merge contracts that are syntactically the same; in the future, we plan to develop a technique that can semantically merge and simplify non-conflicting contracts. Also, we are investigating ways to combine symbolic execution with dynamic analysis to mine contracts more accurately. We are also interested in integrating automated test case generation methods to our approach in order to generate change contracts without the availability of test cases. We also plan to perform additional experiments on more real changes from various systems. Furthermore, we plan to investigate whether it is possible to evaluate contracts generated by our approach using other accuracy metrics (e.g., contract subsumption metric).

References

[1] G. Ammons, R. Bodík, and J. R. Larus, "Mining specifications," in *POPL*, 2002.

[2] I. Beschastnikh, Y. Brun, J. Abrahamson, M. D. Ernst, and A. Krishnamurthy, "Unifying fsm-inference algorithms through declarative specification," in *ICSE*, 2013.

[3] I. Beschastnikh, Y. Brun, S. Schneider, M. Sloan, and M. D. Ernst, "Leveraging existing instrumentation to automatically infer invariant-constrained models," in *SIGSOFT FSE*, 2011, pp. 267–277.

[4] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll, "An overview of JML tools and applications," *STTT*, vol. 7, no. 3, pp. 212–232, 2005.

[5] R. P. Buse and W. R. Weimer, "Automatically documenting program changes," in *ASE*, 2010, pp. 33–42.

[6] V. Dallmeier and T. Zimmermann, "Extraction of bug localization benchmarks from history," in *ASE*, 2007.

[7] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The daikon system for dynamic detection of likely invariants," *Sci. Comput. Program.*, vol. 69, no. 1-3, 2007.

[8] D. Fahland, D. Lo, and S. Maoz, "Mining branching-time scenarios," in *ASE*, 2013, pp. 443–453.

[9] A. Gokhale, V. Ganapathy, and Y. Padmanaban, "Inferring likely mappings between apis," in *ICSE*, 2013, pp. 82–91.

[10] J. Han and M. Kamber, *Data Mining Concepts and Techniques*, 2nd ed. Morgan Kaufmann, 2006.

[11] S. Kumar, S.-C. Khoo, A. Roychoudhury, and D. Lo, "Inferring class level specifications for distributed systems," in *ICSE*, 2012.

[12] Z. Li and Y. Zhou, "Pr-miner: automatically extracting implicit programming rules and detecting violations in large software code," in *ESEC/SIGSOFT FSE*, 2005.

[13] D. Lo and S.-C. Khoo, "Smartic: towards building an accurate, robust and scalable specification miner," in *SIGSOFT FSE*, 2006, pp. 265–275.

[14] D. Lo and S. Maoz, "Scenario-based and value-based specification mining: better together," *Autom. Softw. Eng.*, vol. 19, no. 4, 2012.

[15] D. Lo, S. Maoz, and S.-C. Khoo, "Mining modal scenario-based specifications from execution traces of reactive systems," in *ASE*, 2007.

[16] D. Lo, L. Mariani, and M. Pezzè, "Automatic steering of behavioral model inference," in *ESEC/SIGSOFT FSE*, 2009, pp. 345–354.

[17] D. Lo, G. Ramalingam, V. P. Ranganath, and K. Vaswani, "Mining quantified temporal rules: Formalism, algorithms, and evaluation," *Sci. Comput. Program.*, vol. 77, no. 6, pp. 743–759, 2012.

[18] D. Lorenzoli, L. Mariani, and M. Pezzè, "Automatic generation of software behavioral models," in *ICSE*, 2008, pp. 501–510.

[19] D. Qi, J. Yi, and A. Roychoudhury, "Software change contracts," in *SIGSOFT FSE*, 2012.

[20] F. Thung, S. Wang, D. Lo, and L. Jiang, "An empirical study of bugs in machine learning systems," in *ISSRE*, 2012, pp. 271–280.

[21] A. Wasylkowski and A. Zeller, "Mining temporal specifications from object usage," in *ASE*, 2009, pp. 295–306.

[22] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das, "Perracotta: mining temporal api rules from imperfect traces," in *ICSE*, 2006.

[23] J. Yi, D. Qi, S. H. Tan, and A. Roychoudhury, "Expressing and checking intended changes via software change contracts," in *ISSTA*, 2013.

[24] T. Ziadi, M. A. A. da Silva, L.-M. Hillah, and M. Ziane, "A fully dynamic approach to the reverse engineering of uml sequence diagrams," in *ICECCS*, 2011, pp. 107–116.