Software Bug Localization with Markov Logic

Sai Zhang Congle Zhang
Department of Computer Science & Engineering
University of Washington, USA
{szhang, clzhang}@cs.washington.edu

ABSTRACT

Software bug localization is the problem of determining buggy statements in a software system. It is a crucial and expensive step in the software debugging process. Interest in it has grown rapidly in recent years, and many approaches have been proposed. However, existing approaches tend to use isolated information to address the problem, and are often ad hoc. In particular, most existing approaches predict the likelihood of a statement being buggy *sequentially* and *separately*.

This paper proposes a well-founded, integrated solution to the software bug localization problem based on Markov logic. Markov logic combines first-order logic and probabilistic graphical models by attaching weights to first-order formulas, and views them as templates for features of Markov networks. We show how a number of salient program features can be seamlessly combined in Markov logic, and how the resulting joint inference can be solved.

We implemented our approach in a debugging system, called MLNDEBUGGER, and evaluated it on 4 small programs. Our initial results demonstrated that our approach achieved higher accuracy than a previous approach.

Categories and Subject Descriptors: D.2.5 [Software Engineering]: Testing and Debugging.
General Terms: Reliability, Experimentation.
Keywords: Automated debugging, Machine learning.

1. INTRODUCTION

Today's software systems suffer from poor reliability. Attempts to reduce the number of bugs in software are estimated to consume 50% to 80% of the development and maintenance effort [3]. Among the tasks required to reduce the number of software bugs, localizing bugs is a time-consuming and difficult component.

Existing approaches. Researchers have proposed many automated bug localization approaches. Some existing approaches use statement coverage information (provided by test suites) to identify likely buggy statements [5]. Other approaches leverage the static program structure information to perform a binary search of the memory state to find potential bug locations [2]. Still other approaches [9] are based on the comparison of statements covered by passed tests and statements covered by failed tests. Several authors have devised, compared and learned similarity measures for use in software bug localization. In a comprehensive study [4], a statement coverage-based approach, called TARANTULA [5], consistently performs better than other state-of-the-art approaches [9] [2].

Most coverage-based approaches [9] [2] [4] share two common limitations. First, they predict the likelihood of a statement being buggy *sequentially* and *separately*. This strategy ignores the inter-statement relation in a program that is potentially crucial. For example, if a statement outputs an incorrect value, then the following statements using that incorrect output value should *not* be blamed for being buggy, even if they produce wrong results. Second, existing approaches tend to use isolated information to localize bugs and block the crosstalk between various information sources (or, it requires substantial engineering effort to integrate information from multiple sources into an existing technique). In reality, however, evidence from different information sources are often strongly intermixed and evidence from one side can help to resolve the uncertainty of the another. For example, prior bug knowledge can be used to localize new bugs, since bugs repeatedly found in the same software system may be similar with one another. The prior knowledge of similar bugs confirmed in the past can be extremely useful to assist bug localization [6]. Clearly, a relational-learning model capable of *joint inference* to exploit those global information can address these limitations, and may open up new avenues for the research of automated software bug localization.

Proposed approach. This paper presents a new bug localization approach based on *Markov logic* [10]. Our approach takes advantage of the recent progress in statistical relational learning (a.k.a. multi-relational data mining), which provides rich representations and efficient inference and learning algorithms for non-i.i.d data. Specifically, we use Markov logic, which combines first-order logic and Markov random fields, with weighted satisfiability testing for efficient inference and a voted perceptron algorithm for criminative learning. When applied to the software bug localization problem, Markov logic permits to combine different information sources into a comprehension solution. We illustrate this in this paper by combining a few salient ones, such as statement coverage, static program structure information, and prior bug knowledge, and applying the resulting method to bug localization.

Contributions. The main contributions of this paper are:

- We proposed the *first* approach that uses *Markov logic* to localize software bugs (Section 3).
- We implemented our approach in a tool, called MLNDEBUGGER, and evaluated it on 4 small programs. The initial results showed that MLNDEBUGGER is a promising approach to software bug localization (Section 4).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

ICSE Companion'14, May 31 – June 7, 2014, Hyderabad, India
Copyright 2014 ACM 978-1-4503-2768-8/14/05...$15.00
http://dx.doi.org/10.1145/2591062.2591099

2. BACKGROUND OF MARKOV LOGIC

A Markov logic network (Markov logic, for short, or MLN) is a well-established probabilistic logic from the machine learning community [10]. Markov logic applies the ideas of a Markov network to first-order logic and enables uncertain inference.

Informally, Markov logic is a first-order knowledge base with a weight attached to each formula. Here, a first-order knowledge base (KB) can be seen as a set of hard constraints on the set of possible worlds: if a world violates even one formula, it has zero probability. The basic idea of MLNs is to *soften* these constraints: when a world violates one formula in the KB it is deemed *less probable*, but *not impossible*. The fewer formulas a world violates, the more probable it is. Each formula has an associated weight that reflects how strong a constraint it is: the higher the weight, the greater the difference in probability between a world that satisfies the formula and one that does not, other things being equal.

The weight of each first-order formula can be inferred by using standard training techniques from the machine learning community. Tools such as Alchemy [1] provide fully-automated support for such purpose. After a Markov logic network has been trained, it can answer arbitrary queries of the form "What is the probability that predicate p holds given that formula F does?"

When applying Markov logic to localize a software bug, we assign a predicate p_{stmt} to each statement *stmt*, which represents whether *stmt* is buggy or not. Given the observations of the target software (i.e., reflected by the conjunction of all held formulas), we estimate the probability that p_{stmt} holds.

3. APPROACH

Markov logic [10] provides an ideal presentation for reasoning with complex probabilistic representation. In this section, we describe how to apply it to the problem domain of software bug localization. We first introduce a simple programming model and a number of program features used in this paper (Section 3.1). Then, we present how to encode the problem into a multivariate Markov logic network model with each predicate in the model as a bug predictor (Section 3.2). Finally, we show how to perform inference on the model to identify likely buggy statements (Section 3.3).

3.1 Programming Model and Program Features

Programming Model. A software system S consists of a sequence of statements *stmts*; each statement in *stmts* takes a set of input variables, performs certain computation, and assigns the computed values to the output variables. A bug occurs when a statement assigns an incorrect value to one of its output variables, and that incorrect output variable is later used by other statements until certain undesired behavior is observed.

In practice, many statements can be executed when a bug exhibits. The goal of our approach is to identify the *root* statement(s) for the observed bug.

Program Features. Many program features can be used to predict a bug's location. In our approach, we use three of them.

1. Statement coverage. Software testing is an integral part in modern software development. The statement coverage information by passed or failed test executions reflects a software system's dynamic behaviors.

Informally, given a test suite T for a software system S, a test t in T exercises a subset of *stmts* and gives test execution result: pass/fail and code coverage information. A test t passes if the actual output is the same as the expected output for t; otherwise, t fails. The test suite T may contain multiple failed tests, indicating potential bugs in the tested software. Often, a statement s covered by failed tests is more likely to be buggy than statements only covered by passed tests.

2. Program structure. Statement coverage information reflects the *dynamic* behaviors of a software system at runtime. On the other hand, from a *static* viewpoint, each statement may also depend on other statements via control flow or data flow dependence.

We further explore such static program structure information to obtain inter-statement dependence relations as program features.

3. Prior bug knowledge. Previous research [6] indicates the existence of *recurring bugs*. Bugs repeatedly found in the same software system as it evolves can be similar with one another. For a new bug, it is likely that a similar bug has been confirmed or even fixed in the past. Knowledge about previous bugs can be used to assist bug localization. Our approach extracts three common beliefs about recurring bugs as program features and uses them as prior.

3.2 Modeling Buggy Behavior

With Markov logic, we can encode program features into a joint inference system and conduct inference. The model is composed of first-order logic formulas, with predicates and negations. We assign each statement a predicate s_i. When $s_i = true$, it indicates that the i-th statement is buggy. The goal of software bug localization is to estimate the marginal distribution $p(s_i)$, and then prioritize buggy statements by this distribution.

We next encode three categories of program features described in Section 3.1 as first-logic formulas to reason about buggy program behaviors.

Encoding statement coverage. We use variable t_k to indicate whether a test k fails. That is, $t_k = true\ (false)$, when the test fails (passes).

Intuitively, if a statement s_i is covered by failed tests, it is more likely to be buggy than statements that are not. On the other hand, if s_i is only covered by passed tests, it becomes less suspicious. We express these two observations as follows:

$$\neg t_k \wedge \text{cover}(t_k, s_i) \Rightarrow s_i \tag{1}$$

$$t_k \wedge \text{cover}(t_k, s_i) \Rightarrow \neg s_i \tag{2}$$

where cover(t_k, s_i) is a predicate that returns true if test t_k covers statement s_i.

Encoding program structure. Additional evidences can be gained by exploring a program's static structure. Specifically, for two statements s_i and s_j, if s_i assigns a *wrong* value to its output variable, and s_j uses that wrong variable as its input (i.e., s_j has *data dependence* on s_i); statement s_j should not be treated as buggy, even if it produces an incorrect output. We encode such inter-statement *data dependence* relation as follows:

$$s_i \wedge \text{dataDep}(s_i, s_j) \Rightarrow \neg s_j \tag{3}$$

where dataDep(s_i, s_j) is a predicate that returns true if s_j is data dependent on s_i.

Similarly, if a branching statement (i.e., an `if`, `else`, `while`, or `for` statement) is buggy, all statements having control dependence on it should not be blamed for being buggy, because the decision that makes the execution fail has already been made earlier in the branching statement before the code after the predicate is executed. We encode such inter-statement *control dependence* relation as follows:

$$s_i \wedge \text{isBranch}(s_j) \wedge \text{controlDep}(s_i, s_j) \Rightarrow \neg s_j \tag{4}$$

where isBranch(s_j) is a predicate that returns true if s_j is a branch-

ing statement and controlDep(s_i, s_j) is a predicate that returns true if s_j has *control dependence* on s_i.

Encoding prior bug knowledge. Our approach also incorporates prior knowledge by representing a common observation that statements that were buggy before are more likely to be buggy. We encode such prior knowledge as follows:

$$\text{wasBuggy}(s) \Rightarrow s \quad (5)$$

where *wasBuggy*(s) is a predicate that returns true if statement s was buggy before.

Our approach permits users to further enrich program features by writing first-order formulas, but our experiments did not use this capability.

3.3 Weight Inference

Many algorithms can be used to train a markov logic model. Our approach uses a simple and effective algorithm proposed in the machine learning community and implemented in the mature Alchemy toolkit [1].

Our current inference algorithm assumes that there is only one bug in a tested program. This is based on a common practice that in modern software development, programmers often run the regression test suite after a small amount of code changes. When a test fails, programmers usually need to fix it before implementing new features.

Under this assumption, MLNDEBUGGER does not need estimate the likelihood of two different statements being buggy simultaneously, and thus the search space in inferring the Markov logic model becomes linear, instead of exponentially large, to the number of statements. This permits us to perform inference efficiently even with a simple, exhaustive algorithm.

To learn the weight of each formula, our approach combines two strategies: (1) for the statement coverage formulas, we use the weight as the ratio of failed tests it is executed. The intuition is that a statement is more likely to be buggy if it is covered by more failed tests. (2) for other program structure and prior knowledge formulas, we use a unified weight λ, and exhaustively search for the best λ among the value set: $\{0, 0.001, 0.01, 0.1, 1, 10\}$ to maximize the likelihood of each buggy statement in the training set.

4. EXPERIMENTS

4.1 Tool Implementation

We have implemented our approach in an automated bug localization tool, called MLNDEBUGGER. MLNDEBUGGER is built on top of the Alchemy toolkit [1]. It takes the program source code, a set of passed and failed tests as **input**. MLNDEBUGGER first instruments the program to collect statement coverage information by executing the tests. Then, it uses the standard data-flow and control-flow analysis algorithms [7] to statically compute inter-statement relations. MLNDEBUGGER learns the probabilistic Markov logic model, performs inference to rank each statement according to the likelihood of being buggy, and finally **outputs** a ranked list of suspicious statements that may contain bugs.

4.2 Subject Programs

Our experiments use 4 subject programs from the Siemens suite [11]. Figure 1 summarizes the characteristics of the programs. Each program is associated with a comprehensive test suite and has a number of buggy versions, which contain typical software bugs such as operator and operand mutations, missing and extraneous code, and constant value mutation. Each buggy version contains one bug, but may have multiple buggy statements that should be localized. The bug causes a number of tests to fail in each version.

Program	LOC	# Methods	# Versions	# Tests
replace	562	21	31	5542
schedule	365	18	9	2650
tcas	173	9	41	1608
printtokens	562	18	7	4130

Figure 1: The 4 programs in our evaluation. Column "LOC" represents the number of lines of code. Column "# Methods" represents the number of methods. Column "# Versions" represents the number of buggy versions. Column "# Tests" represents the number of available tests. For each subject, all its buggy versions use the same test suite.

4.3 Methodology

Experiment steps and environment. For each program, we randomly split buggy versions into three subsets so that they were distributed as evenly as possible. We then used 2 subsets for learning the Markov logic model, and used the rest 1 subset as the test data. Our system was run on each test buggy version separately using the learned model, and the output ranked statement list was manually checked to determine its correctness.

Our experiments were conducted on a 2.67GHz Intel Core PC with 12GB physical memory (3GB is allocated for JVM), running Ubuntu 10.04 LTS.

Accuracy measurement. We use the following metric to evaluate a bug localization system's accuracy: we assign a score to the *top k* ranked statements that is the percentage of actual buggy statements they contain. Specifically, suppose our tool outputs a ranked statement list S, and $bug(S)$ is a function that returns the number of actual buggy statements covered by S, and $bugs_{total}$ is the total number of actual buggy statements. Thus, the score of S is defined as:

$$scores(S) = \frac{|bug(S)|}{bugs_{total}} \times 100\%$$

A higher score indicates that the output ranked statement list S is accurate , since it reflects that more irrelevant statements in the program are ignored before the buggy statements are localized.

Baseline for comparison. We choose the well-established TARANTULA approach [5] as the baseline for comparison. As described in a comprehensive study [4], TARANTULA outperforms three other related approaches in bug localization, albeit that it only uses the statement coverage information.

TARANTULA uses a heuristic to infer likely buggy statements from the statement coverage evidence. It counts the number of passed tests and failed tests that cover a statement, and ranks a statement's suspiciousness heuristically as follows:

$$\frac{\%fail(s)}{\%fail(s) + \%pass(s)}$$

where $\%fail(s)$ is the ratio of the number of failed tests that cover statement s to the total number of failed tests in the test suite. $\%pass(s)$, likewise, is the ratio of the number of passed tests that cover statement s to the total number of passed tests in the test suite.

4.4 Results

Figure 2 shows our experimental results. In each figure, the dash curve is the result of TARANTULA [5] ,and the solid curve is the result of MLNDEBUGGER.

Figure 2 clearly indicates that MLNDEBUGGER remarkably outperforms the TARANTULA approach on average. Specifically, for

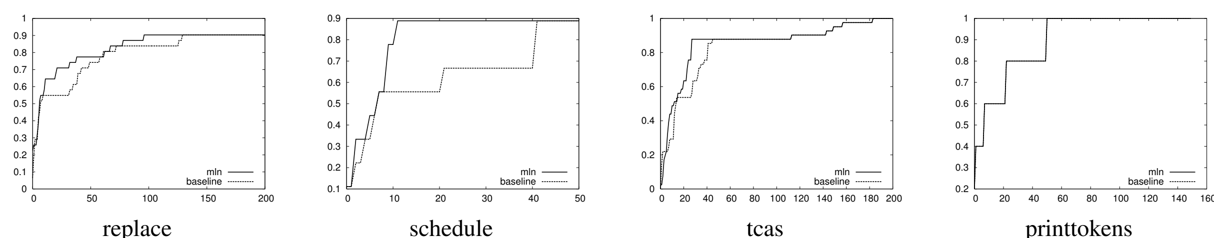

Figure 2: Results of using our MLNDEBUGGER system to localize bugs in 4 small programs. In each figure, the horizontal axis number represents the top *k* suspicious statements returned by a bug localization approach, and the vertical axis number represents the percentage of actual buggy ones they have covered (i.e., the *scores* value of the corresponding returned statement list). The dash curve is the result of TARANTULA [5] *averaged* across all tested buggy versions, and the solid curve is the result of MLNDEBUGGER *averaged* across the same tested buggy versions. Our MLNDEBUGGER outperforms TARANTULA on three out of four programs, and ties with TARANTULA on the printtokens program.

three out of the four programs (tcas, replace, and schedule), MLNDEBUGGER achieves much higher accuracy. For the rest printtokens program, MLNDEBUGGER achieves the same accuracy as TARANTULA does.

The experimental results suggest that joint inference consistently outperforms isolated inference. Our Markov logic-based bug localization model permits to use different information (e.g., statement coverage, static program structure, and prior bug knowledge) to resolve the uncertainty of the another. Crosstalk between various information sources is the key reason for achieving better results than a single heuristic as TARANTULA uses [5].

MLNDEBUGGER tied with TARANTULA on the printtoken program due to the bug characteristics. In printtoken, most bugs occur in the constant declaration statements. However, after a C program is compiled into binary code, such constant declaration statements are erased. Thus, the standard data flow analysis algorithm we use cannot collect the corresponding statement coverage and inter-statement dependence information correctly, and MLNDEBUGGER falls back to use isolate information. This is not a limitation of our Markov logic model. Instead, such result reflects the necessity of performing joint inference in the software bug localization task.

Time and memory cost. MLNDEBUGGER runs in a practical amount of time. For each program evaluated, MLNDEBUGGER used less than 200 seconds with 200MB memory consumption.

System extensibility. MLNDEBUGGER is built on top of the Alchemy toolkit and permits easy extension. Users can define new Markov logic rules (rather than writing low-level implementation code), and plug them into the system with new information sources.

5. CONCLUSION AND FUTURE WORK

This paper presents a new approach (and its tool implementation, called MLNDEBUGGER) to address the software bug localization problem. MLNDEBUGGER uses Markov logic to represent observations and relations between potentially buggy program statements and other knowledge bases. MLNDEBUGGER uses joint inference to estimate the probability of a statement being buggy. Our initial experiments showed that MLNDEBUGGER is a promising approach.

We believe that *statistical relational learning* and *joint inference* have the potential to address the demanding need of eliminating software bugs and improving software quality [8]. As our future work, we plan to develop new algorithms to quickly and cheaply infer our MLN-based probabilistic model. Exact inference may be slow for large commercial software. We are interested in answering two open research questions: (1) how many executions are sufficient to build a sufficiently accurate probabilistic model? and (2) can we use approximate inference for software bug localization at a lower cost but still achieve reasonably good results. We also plan to enhance MLNDEBUGGER's results with richer *contextually-relevant* debugging clues. One possible way is to combine MLNDEBUGGER with a failure explanation tool like FailureDoc [12] to provide richer debugging information.

6. ACKNOWLEDGMENTS

We thank Pedro Domingos and Aniruddh Nath for helpful discussion about Markov logic. This work was supported in part by NSF grants CCF-1016701 and CCF-0963757.

7. REFERENCES

[1] Alchemy - Open Source AI. http://alchemy.cs.washington.edu/.

[2] H. Cleve and A. Zeller. Locating causes of program failures. In *ICSE*, 2005.

[3] J. S. Collofello and S. N. Woodfield. Evaluating the effectiveness of reliability-assurance techniques. *Journal of Systems and Software*, 9(3):191–195, 1989.

[4] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *ASE*, 2005.

[5] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *ICSE*, 2002.

[6] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. Al-Kofahi, and T. N. Nguyen. Recurring bug fixes in object-oriented programs. In *ICSE*, 2010.

[7] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., 1999.

[8] A. V. Nori and S. K. Rajamani. Program analysis and machine learning: A win-win deal. In *SAS*, 2011.

[9] M. Renieris and S. Reiss. Fault localization with nearest neighbor queries. In *ASE*, 2003.

[10] M. Richardson and P. Domingos. Markov logic networks. In *Machine Learning*, page 2006, 2006.

[11] The Software-artifact Infrastructure Repository. http://sir.unl.edu/.

[12] S. Zhang, C. Zhang, and M. D. Ernst. Automated documentation inference to explain failed tests. In *ASE*.