DOI: 10.18721/JCSTCS.13204
УДК 004.052.44

AUTOMATIC GENERATION OF SOFTWARE BUG FIXES BASED ON ANALYSIS OF SOFTWARE REPOSITORIES

A. Belskii, V.M. Itsykson

Peter the Great St. Petersburg Polytechnic University,
St. Petersburg, Russian Federation

This paper describes the method, which is developed by the authors to automated correction of software errors, which is based on the analysis of successful project fixes for the ABAP programming language available in open repositories. The method generates the candidates of patches based on predefined templates and ranks the results by the probability of successful application, which is determined by a probabilistic model using machine learning methods. The probabilistic model is formed by training on features, which are extracted from data from successful and unsuccessful patches of ABAP programs in open repositories. The developed method is tested on synthetic examples and real projects with errors in the ABAP language. As a result of the experiments, the method successfully generated some patches, which showed their efficiency. The results in accuracy and efficiency are comparable or superior to the results of experiments in similar works by other authors.

Keywords: Automated program repair, machine learning, Abstract Syntax Tree, logistic regression, gradient descent, ABAP.

Citation: Belskii A., Itsykson V.M. Automatic generation of software bug fixes based on analysis of software repositories. Computing, Telecommunications and Control, 2020, Vol. 13, No. 2, Pp. 35–48. DOI: 10.18721/JCSTCS.13204

This is an open access article under the CC BY-NC 4.0 license (https://creativecommons.org/licenses/by-nc/4.0/).

АВТОМАТИЧЕСКОЕ ФОРМИРОВАНИЕ ИСПРАВЛЕНИЙ ОШИБОК ПРОГРАММНОГО КОДА НА ОСНОВЕ АНАЛИЗА ПРОГРАММНЫХ РЕПОЗИТОРИЕВ

А. Бельский, В.М. Ицыксон

Санкт-Петербургский политехнический университет Петра Великого,
Санкт-Петербург, Российская Федерация

Описан разработанный подход к автоматизированному исправлению программных ошибок на основе анализа успешных исправлений проектов для языка программирования ABAP, имеющихся в открытых репозиториях. Подход основан на генерации кандидатов на исправления (патчей) по заранее определенным шаблонам и ранжирует полученные результаты по вероятности успешного применения, определяемой на основании вероятностной модели, полученной с помощью методов машинного обучения. Вероятностная модель формируется за счет обучения на свойствах, извлекаемых из данных успешных и неуспешных патчей ABAP-программ, доступных в открытых репозиториях. Разработанный подход протестирован как на искусственных примерах, так и на реальных проектах на языке ABAP с ошибками. В результате проведенных экспериментов успешно сформирован ряд патчей, которые показали свою работоспособность. Результаты по точности и эффективности сопоставимы или превосходят результаты экспериментов в аналогичных работах других авторов.

Ключевые слова: автоматическое исправление программ, машинное обучение, абстрактное синтаксическое дерево, логистическая регрессия, градиентный спуск, ABAP.

Ссылка при цитировании: Belskii A., Itsykson V.M. Automatic generation of software bug fixes based on analysis of software repositories // Computing, Telecommunications and Control. 2020. Vol. 13. No. 2. Pp. 35–48. DOI: 10.18721/JCSTCS.13204

Статья открытого доступа, распространяемая по лицензии CC BY-NC 4.0 (https://creative-commons.org/licenses/by-nc/4.0/).

Introduction

In recent years the size of software has been constantly growing and the development cycle is shortening, which usually leads to a total decrease in the quality of software products. This is unacceptable in areas such as embedded systems in medicine, energy, engineering, the financial sector, and others, or even it can lead to significant material losses or danger to human life and health.

To overcome these problems, developers use various methods to improve the quality of software, like testing, verification, or the static analysis of software. However, all common methods of improving the quality of software have certain limitations and they cannot fully guarantee the quality of programs. For example, testing may detect errors in software, but it cannot guarantee that there are no errors in the tested software. In addition, there are entire classes of programs, such as parallel systems, whose behavior may be non-deterministic and whose testing is inefficient. Formal methods, such as deductive verification and static analysis, are still limited by the size of the analyzed programs and can only be applied to a narrow area of software projects.

Recently, software engineering has actively been conducting research in the field of analysis and application of the accumulated experience of millions of programmers in writing hundreds of thousands of software projects. This experience is recorded in software repositories (Version control systems, VCS) as a history of changes to projects and comments to commits, as well as in task and error management systems (issue tracking and bag tracking) as a history of changes to tasks and errors. There are a large number of methods that analyze the accumulated information and extract it from the knowledge, which is used in solving various problems of software engineering. These methods have proven themselves well in various areas of software engineering. Recently, these approaches have been applied in the field of detecting and correcting software errors, using not only the artifacts of analyzed software projects, but also the previously untapped potential of information, which is stored in hundreds of thousands of software repositories and allowing to reuse and generalize the experience of millions of developers.

This paper describes the results of the research in the field of automated error correction of software code based on the experience of analyzing successful fixes (patches) of many projects for the ABAP programming language [1], which is widely used in SAP software products.

The article is organized as follows: The first section contains a description of the task and a brief overview of the subject area. The second section illustrates the scheme and verbal description of the method, which is developed by the authors. The third section is devoted to the detailed description of the developed method and includes algorithms, a mathematical model, and the technologies and used methods. The fourth section shows the results of testing of the developed method and the analysis of the results. In conclusion, the results are summarized, the results are evaluated and plans for further research are formulated.

Related work

Nowadays, there are various technologies that allow for the automatic generation of bug fixes in programs (patches). The following methods can be the most representative of these technologies.

GenProg [2], relifix [3], Astor [4], and history-based program repair [5] methods, which are based on genetic programming technology. This class of methods is a method of stochastic problem solving, which

is based on the ideas of evolutionary genetics, which include the genotype (the genetic material of an individual) stored in memory, differential reproduction of these genotypes, and variations, which are created by processes, which are similar to the biological processes of mutation and crossover [6].

Methods SemFix [7], JFIX [8], CRSearcher [9], Qlose [10], Semantic program repair using a reference implementation [11], Static automated program repair for heap properties [12], Automated program repair with canonical constraints [13], which are based on the semantic approach. The main idea of these methods is to define a set of restrictions for an expression with an error by applying methods of symbolic program execution [14] and solving these restrictions by using various SMT solvers [15].

Methods R2Fix [16], Prophet [17], ELIXIR [18], Data-Guided Repair of Selection Statements [19], which are based on a class of machine learning methods [20]. The main idea of these methods is to build machine learning models [21], which is based on the source code of programs with errors and their corrections, as well as comments and other data from source code repositories such as GitHub and others. Then the trained model is used for classification tasks, for example, to solve problems of detecting errors in the source code of the program or determining suitable patches that are classified on the same parameters as the error.

The main disadvantage of methods, which are based on genetic programming technology, is the random selection of all possible patch variants without analyzing both the source code context with an error and similar patches. Also, methods, which are based on the semantic approach, already widely analyze the source code context with an error, but do not use the experience of similar patches to strengthen the algorithm for automatic patch generation. At the same time, methods, which are based on the class of machine learning methods, are the closest in implementation to the given task for the authors, since they analyze both the source code context with an error and similar patches.

Thus, the goal of this research is to develop a method to automatically generate bug fixes for software code based on the previously accumulated experience of creating patches. The method has to have an algorithm, which is based on machine learning methods and allows for the automatic generation of patches for various types of errors of the program code without using specifications and other means of automated code generation.

Overview

The main idea of the proposed method is to automatically generate patches for errors in ABAP programs by generating candidate patches based on predefined templates and ranking the results by the probability of successful application, which is determined based on a probabilistic model, which is obtained by using machine learning methods. In turn, the probabilistic model is formed by learning from the data of successful and unsuccessful patches of ABAP programs. The main idea of the method is presented in Fig. 1.

The method contains two main contours – "Machine learning model training" and "Patch generation and ranking", which contain the following seven functional blocks.

Block 1 "Forming an abstract syntax tree". The abstract syntax tree (AST) [22] is based on the source code with an error and a patch. Two independent AST are formed by applying the recursive descent method [23] based on the text of the source code of the ABAP program containing the error and the correction of this error (patch). Further work on the analysis of the source code of the ABAP program is performed on the AST, which gives a more accurate data of the types of elements of the ABAP program (variables, constants, operators, etc.) and their relationships.

Block 2 "Determination of the features of a successful patch". The features of successful patches were formulated to train the probabilistic model, which are determined by analyzing the AST of the source code with an error and the AST of the source code of the patch, which is obtained in block 1. For example, if the program correction was formed by adding a check for an empty variable value before executing the division operator, this feature can be used as a feature of the successful patch and used for training the probabilistic model.

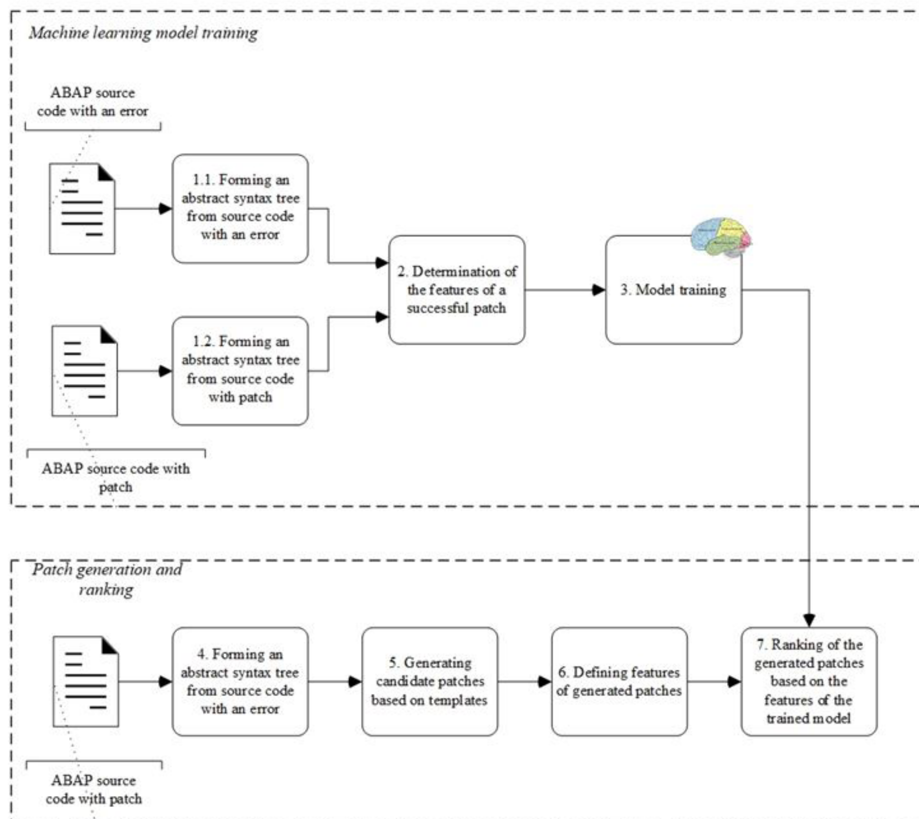

Fig. 1. Scheme of the proposed method

Block 3 "Model training". In this block, the machine learning model is trained based on the features of successful patches, which are obtained in block 2. As a result, the trained model can be used to predict the success rate of any ABAP patch.

Block 4 "Forming an abstract syntax tree from source code with an error". The source code of the ABAP program that needs to be automatically generated for a patch is used to generate the AST, similar to block 1.

Block 5 "Generating candidate patches based on templates". Data of all variables and constants based on the AST is extracted from the program with an error. Furthermore, the array of possible conditions is generated from the data of all variables and constants. Finally, patches are generated using templates based on the received array of variables/constants and the array of possible conditions.

Block 6 "Defining features of generated patches". The features of generated patches in block 5 are determined in the same way as successful patches in block 2 are.

Block 7 "Ranking of the generated patches based on the features of the trained model". The probability of a successful patch is determined for each generated patch based on the trained model, which is obtained in block 3 and the features of the generated patches, which are obtained in block 6. The resulting list of generated patches is sorted in descending order of the probability of a successful patch. Generated patches with the highest probability of successful patches are considered target patches.

Our approach

Let's look more detailed at the stages of the method and the nuances of implementing the methods and models, which are shown in Fig. 1.

Generating AST from source code

The program must be translated into a formalized view to perform the analysis that is suitable for further processing. In this paper, we use an abstract syntax tree. Since there is no official grammar for parser

generators (for example, for ANTLR) for the ABAP language, the authors developed a lightweight parser based on the recursive descent method. The actual formation of the AST from the source code is performed in blocks 1 and 4, which are shown in Fig. 1. The goal of it is to more accurately determine the types of objects and their relationships for further analysis of ABAP programs. The simplified algorithm for parsing ABAP programs is presented as pseudocode in Listing 1.

In line 1 of the algorithm, the input data is the array of $str \in S$, which is the source code lines of the ABAP program. In lines 2-8, the array of lexemes L is generated for each string of *str* in the source code. In lines 9-13, the array of tokens $t \in T$ is formed by defining the following data for each lexem l from the array of lexemes L:

– the token type t_t (header, operator, brackets, number, variable, type), which is defined by assigning each token to a programming language object class;

– the error flag of the error token b_t, is determined by fulfilling the condition: if the source code line *str* contained an error, then all tokens t, which were related to tokens L, will have the value true.

– In lines 14-20, the array of nodes AST P is formed from the token array T. Each node $p \in P$ is the following:

$$\langle p_p | l | t_n | b \rangle \in P,$$

where p_p – a reference to the parent node $p \in P$; l – a lexeme; t_n – a node type, which is determined from the token type t_t; b – the error flag of the node is determined from the error flag of the error token b_t.

The array of nodes AST P is formed using the recursive descent method, which consists of recursively traversing the entire array of tokens $t \in T$ and building their relationships through references p_p according to the grammatical rules of the programming language ABAP, which is shown in Fig. 2.

```
1 Input: S
2 for str in S do {
3     for element in str do {
4           L(l) = element
5     }
6 }
7 for l in L do {
8     t(l) = l
9     t(tt) = defClass(l)
10    t(bt) = defBug(str)
11 }
12 for <l, tt, bt> in T do {
13    P(pp) = defParent(P)
14    P(l) = l
15    P(tn) = tt
16    P(b) = bt
17 }
```

Listing 1. AST generation algorithm

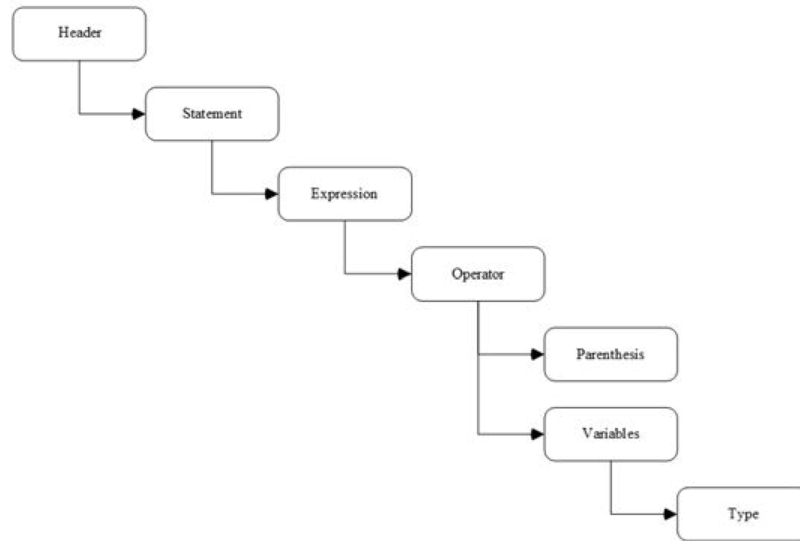

Fig. 2. Grammatical rules of the programming language ABAP

Defining patch features

Patch features are defined in blocks 2 and 6, which is shown in Fig. 1. The authors of the method formulated 15 patch features based on the results of many years of experience working with the ABAP programming language on real projects and creating thousands of bug fixes, as well as on the results of analyzing patches to ABAP programs from open source repositories, which are shown in Table 1. These features are extracted from the source code of ABAP programs with an error and a patch. Previously, to determine the features the method defines the differences between P_{bug} AST source code with an error and P_{patch} AST source code with a patch in the form of node indexes of the beginning of the difference $idx_{start}(P_{patch})$ and the end of the difference $idx_{end}(P_{patch})$. Also, a list of all patch variables $v \in V(P_{patch})$ is defined within $idx_{start}(P_{patch})$ and $idx_{end}(P_{patch})$.

Model training

Model training is performed in block 3 in Fig. 1. There are a number of models with their own advantages and disadvantages to solve classification problems with a teacher in machine learning. The authors of the method chose the logistic regression model [24], because with a small number of properties, this model shows better performance with similar accuracy than other machine learning methods, such as neural networks or the support vector machine. Moreover, the logistic regression model is more convenient to implement and adapt [25], and is also widely used in similar works by other authors.

The following matrix $m \times 15$ is used to train the model:

F_{11}	F_{12}	F_{13}	F_{14}	F_{15}	F_{16}	F_{17}	F_{18}	F_{19}	F_{110}	F_{111}	F_{112}	F_{113}	F_{114}	F_{115}
F_{21}	F_{22}	F_{23}	F_{24}	F_{25}	F_{26}	F_{27}	F_{28}	F_{29}	F_{210}	F_{211}	F_{212}	F_{213}	F_{214}	F_{215}
…	…	…	…	…	…	…	…	…	…	…	…	…	…	…
F_{m1}	F_{m2}	F_{m3}	F_{m4}	F_{m5}	F_{m6}	F_{m7}	F_{m8}	F_{m9}	F_{m10}	F_{m11}	F_{m12}	F_{m13}	F_{m14}	F_{m15}

where m – number of examples in the form of features P_{bug} and P_{patch} (see the section Defining patch features).

Table 1

Patch features

Feature name	Algorithm for determining
Type of error F_1	Determined manually, possible options: 1 – division by 0. 2 – using an empty pointer. 3 – error in the conditional operator. 4 – error in the loop condition
Type of modification the patch F_2	1. Adding a check. If the nodes in the tree P_{patch} within $idx_{start}(P_{patch})$ and $idx_{end}(P_{patch})$, where $\exists l$ = if, то $F_2 = 1$. 2. The change of the if condition. If the nodes in the tree P_{patch} within $idx_{start}(P_{patch})$ and $idx_{end}(P_{patch})$, where $\forall l \neq$ if, but the tree nodes associated with them are $p_p \in P_{patch}$, where $\exists l$ = if, then $F_2 = 2$. 3. The change of the loop condition. If the nodes in the tree P_{patch} within $idx_{start}(P_{patch})$ and $idx_{end}(P_{patch})$, where $\forall l \neq$ loop, but the tree nodes associated with them are $p_p \in P_{patch}$, where $\exists l$ = loop, then $F_2 = 3$. 4. Otherwise $F_2 = 4$
Location of the patch modification F_3	The tree nodes with error P_{bug} are defined by defining $idx_{start}(P_{bug})$ and $idx_{end}(P_{bug})$ of the tree nodes P_{bug}, where $\exists b$ = true. Further, the place where the patch modification occurs is determined by the following rule based on the location data $idx_{start}(P_{patch})$ and $idx_{end}(P_{patch})$ of the tree nodes Ppatch and the location data idxstart(Pbug) and idxend(Pbug) of the tree nodes Pbug: 1. If $idx_{start}(P_{bug}) >= idx_{start}(P_{patch})$ and $idx_{end}(P_{bug}) >= idx_{end}(P_{patch})$, then $F_3 = 0$. 2. If $idx_{start}(P_{bug}) < idx_{start}(P_{patch})$ and $idx_{end}(P_{bug}) >= idx_{end}(P_{patch})$, then $F_3 = 1$. 3. Иначе $F_3 = 2$
If operator is present at the error location F_4	If the tree nodes P_{bug}, where $\exists b$ = true and $\exists l$ = if, then $F_4 = 1$ else 0
Loop operator is present at the error location F_5	If the tree nodes P_{bug}, where $\exists b$ = true and $\exists l$ = loop, then $F_5 = 1$ else 0
/,*,+,- operators are present at the error location F_6	If the tree nodes P_{bug}, where $\exists b$ = true and $\exists l$ = /,*,+,-, then $F_6 = 1$ else 0
Call operator is present at the error location F_7	If the tree nodes P_{bug}, where $\exists b$ = true and $\exists l$ = =>, then $F_7 = 1$ else 0
Variable is present at the if operator at the patch F_8	Defining the tree nodes P_{patch} within $idx_{start}(P_{patch})$ and $idx_{end}(P_{patch})$, where $\exists l$ = if. Further, if in the defined nodes $\exists l = v$, then $F_8 = 1$ else 0
Variable is present at the loop operator at the patch F_9	Defining the tree nodes P_{patch} within $id_{xstart}(P_{patch})$ and $idx_{end}(P_{patch})$, where $\exists l$ = loop. Further, if in the defined nodes $\exists l = v$, then $F_9 = 1$ else 0
Variable is present at the /,*,+,- operators at the patch F_{10}	Defining the tree nodes P_{patch} within $idx_{start}(P_{patch})$ and $idx_{end}(P_{patch})$, where $\exists l$ = /, *,+,-. Further, if in the defined nodes $\exists l = v$, then $F_{10} = 1$ else 0

Variable is present at the call operator at the patch F_{11}	Defining the tree nodes P_{patch} within $idx_{start}(P_{patch})$ and $idx_{end}(P_{patch})$, where $\exists l$ = =>. Further, if in the defined nodes $\exists l = v$, then F_{11} = 1 else 0
Variable is present at the if operator the error location F_{12}	Defining the tree nodes P_{bug}, where $\exists b$ = true, and $\exists l$ = if. Further, if in the defined nodes $\exists l = v$, then F_{12} = 1 else 0
Variable is present at the loop operator the error location F_9	Defining the tree nodes P_{bug}, where $\exists b$ = true, and $\exists l$ = loop. Further, if in the defined nodes $\exists l = v$, then F_{13} = 1 else 0
Variable is present at the /,*,+,- operators the error location F_{10}	Defining the tree nodes P_{bug}, where $\exists b$ = true, and $\exists l$ = /,*,+,-. Further, if in the defined nodes $\exists l = v$, then F_{14} = 1 else 0
Variable is present at the call operator the error location F_{11}	Defining the tree nodes P_{bug}, where $\exists b$ = true, and $\exists l$ = =>. Further, if in the defined nodes $\exists l = v$, then F_{15} = 1 else 0

The training is performed for the logistic regression model:

$$prediction = \frac{1}{1 + e^{-\theta \times F}}.$$

The main idea of training a logistic regression model is to determine coefficients θ for features F successful patches (see the section Defining patch features), which can then be used to build a forecast prediction for any generated patches ABAP programs based on their features. The coefficients θ are determined using the gradient descent method [26], according to which the following calculations are performed simultaneously:

$$\theta_0 = \theta_0 - \alpha \times \frac{1}{m}\left(prediction - y\right),$$

$$\theta_1 = \theta_1 - \alpha \times \frac{1}{m}\left(prediction - y\right) \times F_1 + \frac{\lambda}{m} \times \theta_1,$$

$$\ldots$$

$$\theta_{15} = \theta_{15} - \alpha \times \frac{1}{m}\left(prediction - y\right) \times F_{15} + \frac{\lambda}{m} \times \theta_{15},$$

where y – the result of successful application P_{patch} to P_{bug} (it is set manually, 0 – unsuccessful patch, 1 – successful patch); α – the coefficient of speed of learning (it is set manually and is used to regulate the accuracy and speed of the determination process θ); λ – the regularization coefficient (it is set manually and used to reduce the likelihood of model overfitting).

When calculating the coefficients θ the cost function J is also calculated, which should tend to zero at each iteration of the calculation and reflects the progress and correctness of the gradient descent method:

$$J = \frac{1}{m} \times \sum_{i=1}^{m}\left(-y_i \times \log\left(prediction\right) - \left(1 - y_i\right) \times \log\left(1 - prediction\right)\right) + \frac{\lambda}{2 \times m} \times \sum_{j=2}^{15} \theta_j^2.$$

1 **Input**: P_{fix}

2 $v_{fix} = defVarible(P_{fix})$

3 **for** v_{fix1} **in** V_{fix} **do** {

4   **for** v_{fix2} **in** V_{fix} **do** {

5         $CND_{fix}(cnd_{fix}) = v_{fix1} > v_{fix2}$

6         $CND_{fix}(cnd_{fix}) = v_{fix1} < v_{fix2}$

7         $CND_{fix}(cnd_{fix}) = v_{fix1} = v_{fix2}$

8         $CND_{fix}(cnd_{fix}) = v_{fix1} \neq v_{fix2}$

9   }

10     $CND_{fix}(cnd_{fix}) = v_{fix1}$ *is initial*

11     $CND_{fix}(cnd_{fix}) = v_{fix1}$ *is not initial*

12 }

13 **for** cnd_{fix} **in** CND_{fix} **do** {

14   *GeneratePatchAddIf* (cnd_{fix})

15   *GeneratePatchEditIf* (cnd_{fix})

16   *GeneratePatchEditCycle* (cnd_{fix})

17 }

Listing 2. Algorithm for generating candidate patches based on templates

Generating candidate patches based on templates

The generation of patch candidates by templates is performed in block 5 in the method diagram in Fig. 1. The generation of patch candidates by predefined templates is performed from the source code objects of the ABAP program with an error. This algorithm is presented in Listing 2.

Line 2 defines the array of variables V_{fix} of the tree nodes P_{fix}, which was obtained by forming AST (see the section Generating AST from source code) from the text of the program to automatically generate the patch for. The array of variables V_{fix} is determined from l of the tree nodes $p_{fix} \in P_{fix}$, where $\exists b$ = true and $\exists t_n$ = Variable. In lines 3-14, the array of conditions CND_{fix} is generated by executing the Cartesian product of the array of variables V_{fix} and the array of degrees of comparison (>, <, =, ≠, is initial, is not initial). In lines 15-17, patch candidates $P_{fixpatch}$ are generated by adding a check (if statement) with the condition cnd_{fix} from the array of conditions CND_{fix} before the error location. In lines 15-18, patch candidates $P_{fixpatch}$ are generated by replacing a condition in the check statement (if) with cnd_{fix} from the array of conditions CND_{fix} in the error location. In lines 15-19, patch candidates $P_{fixpatch}$ are generated by changing the condition in the loop operator to cnd_{fix} from the array of conditions CND_{fix} in the error location.

Further, the features for the generated patch candidates $P_{fixpatch}$ are defined (see the section Defining patch features) and the application success rate is determined (see the section Ranking generated patches based on features and the trained logistic regression model).

Ranking generated patches based on features and the trained logistic regression model

The ranking of generated patches based on features and the trained logistic regression model is performed in block 7 in Fig. 1. The success rate $prediction_{fixpatch}$ is determined for each generated patch candidate $P_{fixpatch}$ (see the section Generating candidate patches based on templates) by applying the trained logistic regression model:

$$prediction_{fixpatch} = \frac{1}{1+e^{-\theta \times F_{fixpatch}}},$$

where θ obtained in the process of training the logistic regression model (see the section Model training); $F_{fixpatch}$ obtained in the process of defining patch features for patch candidates $P_{fixpatch}$ (see the section Defining patch features).

Further, $P_{fixpatch}$ with the maximum value of $prediction_{fixpatch}$ is selected, which means that the candidate patches with the highest probability of application success are selected based on the analysis of existing patches.

Evaluation

The method was tested on 10 projects in the ABAP language with errors. Some of the examples were prepared by the authors in accordance with the required types of errors for evaluating the method's performance, while the other part – real projects. The test results are shown in table 2.

Table 2

The results of the test method

Name of the source code example	Type of error	Number of lines of source code	Number of candidate patches generated	Execution time, sec	The patch was successfully generated
ABAPException.abap[1]	Division by 0	34	300	66	Yes
mycalculator.abap[2]	Division by 0	25	100	14	Yes
SubRoutines.abap[3]	Division by 0	59	1200	836	Yes
AbapRep_usingclassHana.abap[4]	Calling a function using an empty pointer	27	800	371	No
zma_dp_strategy.prog.abap[5]	Calling a function using an empty pointer	33	700	285	Yes
zcl_pi_static.clas.abap[6]	Calling a function using an empty pointer	46	100	12	Yes
TestCodeWithIfBug.abap[7]	Error in the if operator	17	200	37	No
TestCodeWithIfBug2.abap[8]	Error in the if operator	11	50	9	Да
TestCodeWithCycleBug.abap[9]	Error in the loop operator	13	20	8	1

[1] https://github.com/naveenkumarbaskaran/SAP_ABAP19Jan/blob/efc47953337bb8fbaeee506ee9a3c701bfa4f498/ABAPException.abap
[2] https://github.com/naveenkumarbaskaran/SAP_ABAP19Jan/blob/master/mycalculator.abap
[3] https://github.com/naveenkumarbaskaran/SAP_ABAP19Jan/blob/master/SubRoutines.abap
[4] https://github.com/naveenkumarbaskaran/SAP_ABAP19Jan/blob/master/AbapRep_usingclassHana.abap
[5] https://github.com/Huargh/OO-Design-Patterns-in-ABAP/blob/master/src/zma_dp_strategy.prog.abap
[6] https://github.com/ivangurin/abapPI/blob/5f30db0cc7a408a759ad833fe14f6e803b1b46bf/src/zcl_pi_static.clas.abap
[7] https://github.com/AlekseiBelskii/AlexB/blob/master/TestCodeWithIfBug.abap
[8] https://github.com/AlekseiBelskii/AlexB/blob/master/TestCodeWithIfBug2.abap
[9] https://github.com/AlekseiBelskii/AlexB/blob/master/TestCodeWithCycleBug.abap

TestCodeWithCycleBug2.abap[1]	Error in the loop operator	13	50	12	0
		276	3520	1650	6/10

The first column shows the project name with an error and a link to Github source code repository. The second column shows the type of error that patches were generated for. The third column shows the number of lines of source code with an error. The fourth column contains the number of error correction candidate patches generated for each project. The number of candidate patches was formed in an amount, which was enough to get the expected result. The fifth column shows the time it took to generate candidate patches for each project with an error. The last column shows whether patches were successfully generated for each project with an error or not. Patch is considered successfully generated if the desired patch is found among all the generated patch candidates with the highest probability of success $prediction_{fixpatch}$.

The method was tested on a stand with the following characteristics: Intel Core i3-7100U 2.40 Ghz, 4.00 Gb RAM, Windows 10. As a result of the experiments, 6 patches were successfully found for 10 programs with an error of 1650 seconds, which indicates the reality of using machine learning methods for automatic patch generation, but at the same time, the obtained accuracy and the speed indicate the necessity for additional tests, better training of the logistic regression model, increasing the power of the test stand, as well as other improvements to the method. These improvements are expected to be developed and implemented in future works.

Conclusion

During the research, the method was developed to automatically generate bug fixes for ABAP programs based on the analysis of existing patches, which generates candidate patches for ABAP programs and ranks the results using machine learning methods. The obtained preliminary test results suggest that using machine learning methods to solve problems of automatic error correction in programs is a promising direction for software engineering. Directions for further development of the work:

- conducting deeper testing of the method on a wider set of real projects;
- extending the method to support new programming languages;
- extending the set of the extracted features and the list of error types to fix;
- use more complex machine learning models to improve the performance of the method.

REFERENCES

1. SAP SE. ABAP—Keyword Documentation. Available: *https://help.sap.com/doc/abapdocu_latest_index_htm/latest/en-US/index.htm –2019.*

2. **Le Goues C., et al.** Genprog: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, 2012, Vol. 38, No. 1, P. 54.

3. **Tan S.H., Roychoudhury A.** relifix: Automated repair of software regressions. *Proceedings of the 37th International Conference on Software Engineering*. IEEE Press, 2015, Vol. 1, Pp. 471–482.

4. **Martinez M., Monperrus M.** Astor: A program repair library for java. *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 2016, Pp. 441–444.

5. **Le X.B.D., Lo D., Le Goues C.** *History driven automated program repair*. 2016.

6. **Forrest S.** Genetic algorithms: Principles of natural selection applied to computation. *Science*, 1993, Vol. 261, Pp. 872–878.

[1] https://github.com/AlekseiBelskii/AlexB/blob/master/TestCodeWithCycleBug2.abap

7. **Nguyen H.D.T., et al.** Semfix: Program repair via semantic analysis. *Proceedings of the 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, Pp. 772–781.

8. **Le X.B.D., et al.** JFIX: semantics-based repair of Java programs via symbolic PathFinder. *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ACM, 2017, Pp. 376–379.

9. **Wang Y., et al.** CRSearcher: Searching Code Database for Repairing Bugs. *Proceedings of the 9th Asia-Pacific Symposium on Internetware*, ACM, 2017, P. 16.

10. **D'Antoni L., Samanta R., Singh R.** Qlose: Program repair with quantitative objectives. *International Conference on Computer Aided Verification*. Springer, Cham, 2016, Pp. 383–401.

11. **Mechtaev S., et al.** Semantic Program Repair Using a Reference Implementation. *Proceedings of ICSE*, 2018.

12. **van Tonder R., Le Goues C.** *Static Automated Program Repair for Heap Properties*, 2018.

13. **Hill A., Păsăreanu C.S., Stolee K.T.** Automated program repair with canonical constraints. *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. ACM, 2018, Pp. 339–341.

14. **Cadar C., et al.** KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. *OSDI*, 2008, Vol. 8, Pp. 209–224.

15. **De Moura L., Bjørner N.** Z3: An efficient SMT solver. *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, Berlin, Heidelberg, 2008, Pp. 337–340.

16. **Liu C., et al.** R2Fix: Automatically generating bug fixes from bug reports. *Proceedings of the 6th International Conference on Software Testing, Verification and Validation*, IEEE, 2013, Pp. 282–291.

17. **Long F., Rinard M.** Automatic patch generation by learning correct code. *ACM SIGPLAN Notices*, 2016, Vol. 51, No. 1, Pp. 298–312.

18. **Saha R.K., et al.** Elixir: Effective object-oriented program repair. *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, 2017, Pp. 648–659.

19. **Gopinath D., et al.** Data-guided repair of selection statements. *Proceedings of the 36th International Conference on Software En*gineering**. ACM, 2014, Pp. 243-253.**

20. **Dietterich T.G.** *Machine learning. Encyclopedia of Computer Science*. John Wiley and Sons Ltd., GBR, 2003, Pp. 1056–1059.

21. **Witten I.H., et al.** *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2016.

22. **Cui B., et al.** Code comparison system based on abstract syntax tree. *Proceedings of the 3rd IEEE International Conference on Broadband Network and Multimedia Technology (IC-BNMT)*. IEEE, 2010, Pp. 668–673.

23. **Matthew S. Davis.** An object oriented approach to constructing recursive descent parsers. *SIGPLAN Not.*, 2000, 35, 2, Pp. 29–35. DOI: https://doi.org/10.1145/345105.345113 – 2000

24. **Kleinbaum D.G., et al.** *Logistic regression*. New York: Springer-Verlag, 2002.

25. **Kalantar B., et al.** Assessment of the effects of training data selection on the landslide susceptibility mapping: A comparison between support vector machine (SVM), logistic regression (LR) and artificial neural networks (ANN). *Geomatics, Natural Hazards and Risk*, 2018, Vol. 9, No. 1, Pp. 49–69.

26. **Ruder S.** An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*. 2016.

Received 10.03.2020.

СПИСОК ЛИТЕРАТУРЫ

1. **SAP SE.** ABAP—Keyword Documentation // URL: https://help.sap.com/doc/abapdocu_latest_index_htm/latest/en-US/index.htm–2019.

2. **Le Goues C., et al.** Genprog: A generic method for automatic software repair // IEEE Transactions on Software Engineering. 2012. Vol. 38. No. 1. P. 54.

3. **Tan S.H., Roychoudhury A.** relifix: Automated repair of software regressions // Proc. of the 37th Internat. Conf. on Software Engineering. IEEE Press, 2015. Vol. 1. Pp. 471–482.

4. . **Martinez M., Monperrus M.** Astor: A program repair library for java // Proc. of the 25th Internat. Symp. on Software Testing and Analysis. ACM, 2016. Pp. 441–444.

5. **Le X.B.D., Lo D., Le Goues C.** History driven automated program repair. 2016.

6. **Forrest S.** Genetic algorithms: Principles of natural selection applied to computation. Science, 1993, Vol. 261, Pp. 872–878.

7. **Nguyen H.D.T., et al.** Semfix: Program repair via semantic analysis // Proc. of the 35th Internat. Conf. on Software Engineering. IEEE, 2013. Pp. 772–781.

8. **Le X.B.D., et al.** JFIX: semantics-based repair of Java programs via symbolic PathFinder // Proc. of the 26th ACM SIGSOFT Internat. Symp. on Software Testing and Analysis. ACM, 2017. Pp. 376–379.

9. **Wang Y., et al.** CRSearcher: Searching Code Database for Repairing Bugs // Proc. of the 9th Asia-Pacific Symp. on Internetware. ACM, 2017. P. 16.

10. **D'Antoni L., Samanta R., Singh R.** Qlose: Program repair with quantitative objectives // Internat. Conf. on Computer Aided Verification. Springer, Cham, 2016. Pp. 383–401.

11. **Mechtaev S., et al.** Semantic Program Repair Using a Reference Implementation // Proc. of ICSE. 2018.

12. **van Tonder R., Le Goues C.** Static Automated Program Repair for Heap Properties. 2018.

13. **Hill A., Păsăreanu C.S., Stolee K.T.** Automated program repair with canonical constraints // Proc. of the 40th Internat. Conf. on Software Engineering: Companion Proceeedings. ACM, 2018. Pp. 339–341.

14. **Cadar C., et al.** KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs // OSDI. 2008. Vol. 8. Pp. 209–224.

15. **De Moura L., Bjørner N.** Z3: An efficient SMT solver // Internat. Conf. on Tools and Algorithms for the Construction and Analysis of Systems. Springer, Berlin, Heidelberg, 2008. Pp. 337–340.

16. **Liu C., et al.** R2Fix: Automatically generating bug fixes from bug reports // Proc. of the 6th Internat. Conf. on Software Testing, Verification and Validation. IEEE, 2013. Pp. 282–291.

17. **Long F., Rinard M.** Automatic patch generation by learning correct code // ACM SIGPLAN Notices. 2016. Vol. 51. No. 1. Pp. 298–312.

18. **Saha R.K., et al.** Elixir: Effective object-oriented program repair // 2017 32nd IEEE/ACM Internat. Conf. on Automated Software Engineering. IEEE, 2017. Pp. 648–659.

19. **Gopinath D., et al.** Data-guided repair of selection statements // Proc. of the 36th Internat. Conf. on Software Engineering. ACM, 2014. Pp. 243–253.

20. **Dietterich T.G.** Machine learning. Encyclopedia of Computer Science. John Wiley and Sons Ltd., GBR, 2003. Pp. 1056–1059.

21. **Witten I.H., et al.** Data Mining: Practical machine learning tools and techniques. Morgan Kaufmann, 2016.

22. **Cui B., et al.** Code comparison system based on abstract syntax tree // Proc. of the 3rd IEEE Internat. Conf. on Broadband Network and Multimedia Technology. IEEE, 2010. Pp. 668–673.

23. **Matthew S. Davis.** An object oriented approach to constructing recursive descent parsers // SIGPLAN Not. 2000. 35. 2. Pp. 29–35. DOI: https://doi.org/10.1145/345105.345113 – 2000

24. **Kleinbaum D.G., et al.** Logistic regression. New York: Springer-Verlag, 2002.

25. **Kalantar B., et al.** Assessment of the effects of training data selection on the landslide susceptibility mapping: A comparison between support vector machine (SVM), logistic regression (LR) and artificial neural networks (ANN) // Geomatics, Natural Hazards and Risk. 2018. Vol. 9. No. 1. Pp. 49–69.

26. **Ruder S.** An overview of gradient descent optimization algorithms // arXiv preprint arXiv:1609.04747. 2016.

Статья поступила в редакцию 10.03.2020.

THE AUTHORS / СВЕДЕНИЯ ОБ АВТОРАХ

Belskii Aleksei
Бельский Алексей
E-mail: belskii.alexey@gmail.com

Itsykson Vladimir M.
Ицыксон Владимир Михайлович
E-mail: vlad@icc.spbstu.ru

© Санкт-Петербургский политехнический университет Петра Великого, 2020