Towards API-Specific Automatic Program Repair

Sebastian Nielebock
Chair of Software Engineering, Faculty of Computer Science
Otto-von-Guericke University Magdeburg, Germany
sebastian.nielebock@ovgu.de

***Abstract*—The domain of Automatic Program Repair (APR) had many research contributions in recent years. So far, most approaches target fixing generic bugs in programs (e.g., off-by-one errors). Nevertheless, recent studies reveal that about 50% of real bugs require API-specific fixes (e.g., adding missing API method calls or correcting method ordering), for which existing APR approaches are not designed. In this paper, we address this problem and introduce the notion of an API-specific program repair mechanism. This mechanism detects erroneous code in a similar way to existing APR approaches. However, to fix such bugs, it uses API-specific information from the erroneous code to search for API usage patterns in other software, with which we could fix the bug. We provide first insights on the applicability of this mechanism and discuss upcoming research challenges.**

***Index Terms*—Automatic Program Repair, API-specific Bugs, Specification Mining**

I. Research Problem of Automatic Program Repair

The burden to fix bugs in software development is both time consuming and costly. Even though automation of bug localization is common (e.g., automated testing or software profiling), bug fixing still requires the programmer's interaction. Even worse, automatic program repair (APR) is undecidable in general [2].

However, in recent years many approaches were published to perform APR. Martin Monperrus classifies these approaches as *behavioral* and *state repair* [27]. *Behavioral repair* approaches directly modify the source code to fix the bug. A list of well-known approaches are the GenPatch algorithm [34], BugMem [16], Pachika [6], GenProg [19], PAR [15], AutoFix [29], Nopol [7], RSRepair [30], LeakFix [9], SearchRepair [14], QACrashFix [10], Angelix [26], HistoricalFix [18], Prophet [23] and DeepFix [12]. In contrast, *state repair* approaches change the surrounding environment of the erroneous program, e.g., by changing input data or the memory state. Some representatives are Mircoreboots [4], DieHard [3], Assure [32], Bolt [17], Input Rectification [22], Armor [5], RCV [24] and Ares [11].

The prevalent trend of these APR approaches is to be applicable for generic bugs, i.e., bugs that may occur in every kind of software, e.g., off-by-one errors. In a recent study on manual bug fixes [37], the authors revealed that 50% of all bugs fall in this category, which explains the success of existing APR approaches. In the other half, at least one API-specific fix was necessary.

As an example of an API-specific fix, one may consider the Lang-38 bug from the defects4J benchmark [13] in Listing 1. This benchmark represents a collection of 395 real-word bugs from six open source projects and their manually created fixes.

```
 1  *** 869,874 ****
 2  --- 869,875 ----
 3    [...]
 4    public StringBuffer format(Calendar vCalendar,
          StringBuffer buf) {
 5     if (mTimeZoneForced) {
 6  +   vCalendar.getTime();
 7      vCalendar = (Calendar) vCalendar.clone();
 8      vCalendar.setTimeZone(mTimeZone);
 9     }
10    [...]
```

Listing 1: Bug Fix Lang-38 from the defects4J benchmark produced with diff. The fix was an addition of the `getTime`-method at line 6.

The issue with Lang-38 is that the `vCalendar` object is not being properly updated before being cloned for further reuse. Thus, an outdated time is formatted. This bug was fixed by adding the `getTime`-method at line 6. According to the API-documentation of the `Calendar` API[1], the call of `get`, `getTime` and three other methods causes an update of the `vCalendar`-variable, and thus fixes the bug.

Here, the intrinsic knowledge to call the `getTime`-method before cloning the `vCalendar`-object was necessary to fix the bug. In general, it is recommended to leverage such API knowledge for APR [20], [37]. In this paper, we introduce the idea of representing API knowledge as *API usage patterns* [31]. These patterns represent common structures of interface application of an API. However, these patterns are not always available, especially if programmers are not familiar with the API. Thus, to fix API-specific bugs automatically, we mine usage patterns, which directly relate to the erroneous code, and modify the code base so that it complies with the usage pattern. This means that our approach represents a behavioral repair method.

In addition to Monperrus's classification, we introduce the domain of API-specific and generic repair in Fig. 1. Note that this classification does not represent a holistic picture of APR but rather reveals the need for API-specific repair mechanisms. Some approaches partially address the topic of API-specific repair or use similar ideas (bold and underlined in the classification scheme). In the following, we reveal their limitations for API-specific repair and the main differences according to the proposed approach.

Related behavioral, generic repair methods use patterns only to improve their results or to find API-specific fixes with

[1]https://docs.oracle.com/javase/8/docs/api/java/util/Calendar.html

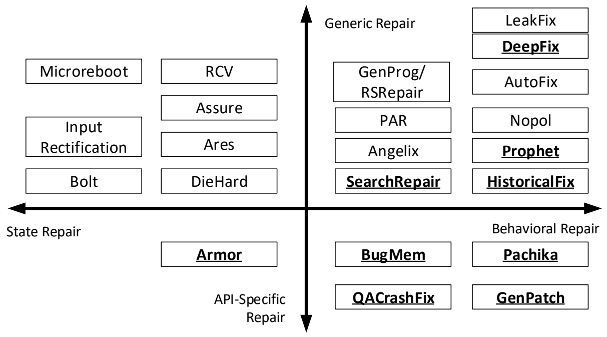

Fig. 1: Classification with the dimension of API-specific repair methods

a different method. HistoricalFix [18] and Prophet [23] use knowledge from bug fix patterns to assess their synthesized patches. Nevertheless, their patterns consider generic bugs rather than API-specific ones. SearchRepair [14] finds patches by searching code that has the same behavior with regard to passing test cases. In contrast to our proposed mechanism, it directly applies found code snippets from other applications to produce patches. DeepFix [12] conducts deep learning of code samples with a recurrent neural network. However, this approach usually targets syntactic instead of semantic bugs.

Existing behavioral, API-specific repair methods either do not represent a fully automated approach or show other limitations to find API-specific fixes. GenPatch [34] describes an algorithm that creates patches by compensating differences between a specification pattern, e.g., an API usage pattern among others, and the actual code. This algorithm expects that the specification patterns are already known, posing the question of how to find related patterns. BugMem [16] memorizes bug fixes from the project's history, to be re-applied if the same error occurs again. Extensions of this approach infer common bug-fix patterns for APR [28], [25]. Pachika [6] analyzes objects' state flows during the program execution and detects anomalies from these flows. It produces a patch according to the usually observed behavior. Due to the monitoring overhead, it only considers anomalies in the erroneous software. QACrashFix [10] uses the information of Q&A-webpages like Stack Overflow to find possible patches for bugs. Here, the approach benefits from API-specific error messages to find relevant Q&A-pages. However, the approach struggles to extract code changes from embedded code in natural language texts whereby the solution space is restricted.

Within the state repair mechanisms only Armor [5] is known to address API-specific APR. Armor leverages existing redundancy in the implementation so that some method calls can be substituted in case of an error. However, developers have to manually define redundant method calls.

Our mechanism addresses these limitations. In the next section, we explain the overall structure of our mechanism, present first results based on the example from Listing 1 and discuss methods for validating our approach.

II. Automatic Program Repair by Mining API Usage Patterns

The thesis addresses the topic of API-specfic automatic program repair by automatic mining of bug-related usage patterns and applying these patterns to overcome API misuses.

Fig. 2 depicts the general workflow of the mechanism and its main contributions. Similarly to other APR approaches, we conduct bug localization (**A**) by running automatic test cases to extract those statements that most likely cause the bug (e.g. by spectrum-based approaches like Tarantula, Ochiai, or DStar [35]). This is promising, as many software projects apply automatic testing, and thus have test suites. As we require only an erroneous code section for patch generation, other localization methods may work as well.

Next, we extract API-information (**B**), e.g., imported classes or used methods, from the erroneous code section. This information is used to find similar code snippets from other software projects that apply the same APIs as used in the erroneous code. In particular, we assume that the bug is caused by a misuse of an API and can be fixed with API usage patterns found in other, correct applications.

With respect to our example in Listing 1, we can extract the information that the erroneous code uses the classes `StringBuffer` and `Calendar`. Afterwards, we search for source files that also utilize these APIs (e.g., via the GitHub Search API[2]). These files serve as input for the mining process.

Recent publications in the domain of specification mining introduce various approaches to mine API usage patterns [31] (**C**). So far, specification mining was mainly used for automatic documentation inference or bug detection. Instead, we plan to use usage patterns for patch generation. In the proposed mechanism, we perform a shepherded specification mining: by restricting input source files to several usages of one particular API, the specification mining task is more likely to find a usage pattern for this particular API, as these appear more frequently in the input data. Thereby, the resulting usage patterns are more closely related to the particular bug.

In a preliminary experiment, we applied Frequent Sequential Pattern Mining (FSPM) from the SPMF-library [8] for source files that also imported the `Calendar` class. We analyzed 1.221 source files from 309 projects acquired from GitHub and obtained 20.486 sequences from these files. Here, a sequence represents a string of nodes from an abstract syntax tree (AST) of each method in the source code file. In order to deal with block statements, such as if-then-else constructs, we add special nodes, indicating the start and the end of statements. To increase the number of frequent sequences, we anonymized variable names in method calls and assignments and restricted the AST nodes to method calls, assignments, loops and conditional-statements.

So far, the results obtained appear to be promising. In detail, we achieved the best results with the CloSpan-algorithm, which aims to find closed frequent sequences [36]. We set the minimal support to 1% and found 127 frequent sequences.

[2]https://developer.github.com/v3/

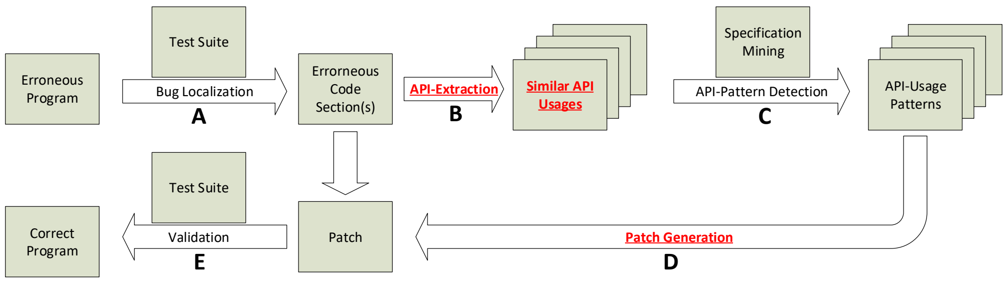

Fig. 2: Workflow of our API-specific program repair mechanism. Contributions of the thesis are highlighted (red, bold and underlined).

Here, the `get`- and the `getTime`-methods (original fix) were the 4th and 18th most frequently applied sequences, respectively. Thus, selecting patterns for APR due to their frequency, would quickly result in patterns, whose insertions at Line 6 in Listing 1 would eventually fix the bug.

Based on mined patterns, the mechanism synthesizes a patch (**D**). So far, we conducted this step manually. To automate it, Weimer's GenPatch algorithm [34] is applicable. GenPatch finds the minimal edit operations, i.e., insertions and deletions, which modify the erroneous code so that it complies with the API usage pattern. However, if the patterns contain methods with input parameters, GenPatch requires manual insertion of these parameters. For entire automation, we need a mapping to the existing variables, e.g., variable `vCalendar`, and a mechanism to infer parameter values, e.g., application of the `get`-method requires an integer value as parameter. We propose to use static analysis of the AST to infer global as well as local variables and to check them exhaustively, if they can call the missing method. For example, the `Calendar`-specific methods `get` and `getTime` can only be called via the `vCalendar`-variable. To infer parameters, a naive approach is to insert the default values of the data types, i.e., we manually insert the value `0` in `get`. This fixes the bug, as calling the `get`-method with an arbitrary parameter triggers the required update of the `vCalendar`-object.

Finally, we validate found patches by re-running the whole test suite (**E**). We consider a patch valid, if all test cases pass.

So far, we introduced the notion of API-specific APR on the Lang-38 example from the defects4J benchmark. In order to provide evidence for the approach's applicability to other bugs and APIs, we are currently working on a larger study. APR is usually validated on existing bug benchmarks [13], [21], [33]. We plan to evaluate our mechanism with the MuBench benchmark, which contains real-world API-specific bugs [1].

The following section highlights further challenges.

III. Further Work

Even though the approach of API-specific APR is promising, we identify three different research challenges.

Extracting API-specific information: API-specific information is crucial to mine meaningful API usage patterns. If we use too coarse API information for the code search, we may find many API usages, and thus patterns, which are extraneous to the bug. In contrast, with too detailed information, we may struggle to find related code samples for the mining step at all. Even though API usage pattern recommender systems like MAPO [38] give first insights, from whose information specification mining benefits, we suggest further investigation of whether this is applicable to APR.

Automating Patch Generation: As discussed before, the algorithm by Weimer [34] enables us to create fixes automatically based on API usage patterns. Nevertheless, this process is not fully automated, as for example method parameters cannot be automatically inferred. Next to the previously mentioned approach of using default values for unknown parameters, further work will consider more sophisticated approaches. For instance, we can use Frequent Itemset Mining to find frequently applied method parameters from other applications.

Dealing with False Positive Patterns: A common problem of specification mining algorithms is that they find many false positive patterns. These false positives represent relations between statements that do not exist. Thus, applying the complete pattern may either not fix the bug or introduce new ones. As false positives usually represent a false combination of true patterns, we suggest accepting partial compliance with a pattern as patch candidates.

We will address these challenges in ongoing research. In addition, as specification mining can be too time-consuming in practice, storing and combining usage patterns for later re-use can avoid repetitive mining.

IV. Conclusion

In this paper, we introduced the idea of an API-specific repair mechanism. Most existing approaches focus on creating fixes for generic bugs, such as off-by-one errors. Nevertheless, recent publications reveal that 50% of programming bugs require at least one API-specific repair action.

Thus, the thesis addresses the topic of automatic repair of API-specific bugs (e.g., false method order, missing method calls). The proposed approach captures the intrinsic knowledge of how to use an API in the form of API usage patterns. To get these patterns, algorithms from the domain of specification mining are the most promising. In particular, we search in other applications, which also use a particular API, for API usage patterns, which, if applied, are likely to fix the bug.

Within this paper, we give an example of an API-specific bug, for which we are able to find usage patterns by frequent sequence mining, which would eventually fix the bug. Finally, we highlighted our further research directions.

Acknowledgment

I would like to thank my advisor, Frank Ortmeier, for his support and guidance.

References

[1] S. Amann, S. Nadi, H. A. Nguyen, T. N. Nguyen, and M. Mezini, "MUBench: A Benchmark for API-misuse Detectors," in *Proceedings of the 13th Working Conference on Mining Software Repositories (MSR)*. IEEE/ACM, 2016, pp. 464–467.

[2] A. Arcuri, "Evolutionary Repair of Faulty Software," *Applied Soft Computing*, vol. 11, no. 4, pp. 3494–3514, 2011.

[3] E. D. Berger and B. G. Zorn, "DieHard: Probabilistic Memory Safety for Unsafe Languages," in *ACM SIGPLAN Notices*, vol. 41, no. 6, ACM. ACM, 2006, pp. 158–168.

[4] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox, "Microreboot-a technique for cheap recovery." in *Proceedings of the 6th Symposium on Operating Systems, Design & Implementation (OSDI)*, ACM. ACM, 2004, pp. 31–44.

[5] A. Carzaniga, A. Gorla, A. Mattavelli, N. Perino, and M. Pezze, "Automatic Recovery from Runtime Failures," in *Proceedings of the 35th International Conference on Software Engineering (ICSE)*, ACM/IEEE. IEEE Press, 2013, pp. 782–791.

[6] V. Dallmeier, A. Zeller, and B. Meyer, "Generating Fixes From Object Behavior Anomalies," in *Proceedings of the 24th International Conference on Automated Software Engineering (ASE)*, IEEE/ACM. IEEE/ACM, 2009, pp. 550–554.

[7] F. DeMarco, J. Xuan, D. Le Berre, and M. Monperrus, "Automatic Repair of Buggy If Conditions and Missing Preconditions with SMT," in *Proceedings of the 6th International Workshop on Constraints in Software Testing, Verification, and Analysis (CSTVA)*. ACM, 2014, pp. 30–39.

[8] P. Fournier-Viger, J. C.-W. Lin, A. Gomariz, T. Gueniche, A. Soltani, Z. Deng, and H. T. Lam, "The SPMF Open-Source Data Mining Library Version 2," in *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer, 2016, pp. 36–40.

[9] Q. Gao, Y. Xiong, Y. Mi, L. Zhang, W. Yang, Z. Zhou, B. Xie, and H. Mei, "Safe Memory-leak fixing for C Programs," in *Proceedings of the 37th International Conference on Software Engineering (ICSE)*. IEEE/ACM, 2015, pp. 459–470.

[10] Q. Gao, H. Zhang, J. Wang, Y. Xiong, L. Zhang, and H. Mei, "Fixing Recurring Crash Bugs via Analyzing Q&A Sites," in *Proceedings of the 30th International Conference on Automated Software Engineering (ASE)*. IEEE/ACM, 2015, pp. 307–318.

[11] T. Gu, C. Sun, X. Ma, J. Lü, and Z. Su, "Automatic Runtime Recovery via Error Handler Synthesis," in *Proceedings of the 31st International Conference on Automated Software Engineering (ASE)*, IEEE/ACM. IEEE/ACM, 2016, pp. 684–695.

[12] R. Gupta, S. Pal, A. Kanade, and S. Shevade, "DeepFix: Fixing Common C Language Errors by Deep Learning," in *Proceedings of the 31st Conference on Artificial Intelligence (AAAI)*. AAAI, 2017.

[13] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2014, pp. 437–440.

[14] Y. Ke, K. T. Stolee, C. Le Goues, and Y. Brun, "Repairing Programs with Semantic Code Search," in *Proceedings of the 30th International Conference on Automated Software Engineering (ASE)*. IEEE/ACM, 2015, pp. 295–306.

[15] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic Patch Generation Learned from Human-written Patches," in *Proceedings of the 35th International Conference on Software Engineering (ICSE)*. IEEE/ACM, 2013, pp. 802–811.

[16] S. Kim, K. Pan, and E. J. Whitehead Jr, "Memories of Bug Fixes," in *Proceedings of the 14th International Symposium on Foundations of Software Engineering (FSE)*. ACM, 2006, pp. 35–45.

[17] M. Kling, S. Misailovic, M. Carbin, and M. Rinard, "Bolt: On-demand Infinite Loop Escape in Unmodified Binaries," in *ACM SIGPLAN Notices*, vol. 47, no. 10. ACM, 2012, pp. 431–450.

[18] X. B. D. Le, D. Lo, and C. Le Goues, "History Driven Program Repair," in *Proceedings of the 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER),*, vol. 1. IEEE, 2016, pp. 213–224.

[19] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "A Systematic Study of Automated Program Repair: Fixing 55 out of 105 Bugs for $8 each," in *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, IEEE. IEEE Press, 2012, pp. 3–13.

[20] C. Le Goues, S. Forrest, and W. Weimer, "Current Challenges in Automatic Software Repair," *Software Quality Journal*, vol. 21, no. 3, pp. 421–443, 2013.

[21] C. Le Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. Devanbu, S. Forrest, and W. Weimer, "The ManyBugs and IntroClass Benchmarks for Automated Repair of C programs," *IEEE Transactions on Software Engineering*, vol. 41, no. 12, pp. 1236–1256, 2015.

[22] F. Long, V. Ganesh, M. Carbin, S. Sidiroglou, and M. Rinard, "Automatic Input Rectification," in *Proceedings of the 34th International Conference on Software Engineering (ICSE)*. IEEE/ACM, 2012, pp. 80–90.

[23] F. Long and M. Rinard, "Automatic Patch Generation by Learning Correct Code," in *ACM SIGPLAN Notices*, vol. 51, no. 1. ACM, 2016, pp. 298–312.

[24] F. Long, S. Sidiroglou-Douskos, and M. Rinard, "Automatic Runtime Error Repair and Containment via Recovery Shepherding," in *ACM SIGPLAN Notices*, vol. 49, no. 6. ACM, 2014, pp. 227–238.

[25] M. Martinez and M. Monperrus, "Mining Software Repair Models for Reasoning on the Search Space of Automated Program Fixing," *Springer Empirical Software Engineering*, vol. 20, no. 1, pp. 176–205, 2015.

[26] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis," in *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, IEEE/ACM. IEEE Press, 2016, pp. 691–701.

[27] M. Monperrus, "Automatic Software Repair: A Bibliography," *University of Lille, Tech. Rep. hal-01206501*, 2015.

[28] H. Osman, M. Lungu, and O. Nierstrasz, "Mining Frequent Bug-Fix Code Changes," in *Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE)*. IEEE, 2014, pp. 343–347.

[29] Y. Pei, C. A. Furia, M. Nordio, Y. Wei, B. Meyer, and A. Zeller, "Automated Fixing of Programs with Contracts," *IEEE Transactions on Software Engineering*, vol. 40, no. 5, pp. 427–449, 2014.

[30] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang, "The Strength of Random Search on Automated Program Repair," in *Proceedings of the 36th International Conference on Software Engineering (ICSE)*. ACM/IEEE, 2014, pp. 254–265.

[31] M. P. Robillard, E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford, "Automated API Property Inference Techniques," *IEEE Transactions on Software Engineering*, vol. 39, no. 5, pp. 613–637, 2013.

[32] S. Sidiroglou, O. Laadan, C. Perez, N. Viennot, J. Nieh, and A. D. Keromytis, "Assure: Automatic Software Self-healing Using Rescue Points," *ACM SIGARCH Computer Architecture News*, vol. 37, no. 1, pp. 37–48, 2009.

[33] S. H. Tan, J. Yi, S. Mechtaev, A. Roychoudhury *et al.*, "Codeflaws: A Programming Competition Benchmark for Evaluating Automated Program Repair Tools," in *Proceedings of the 39th International Conference on Software Engineering Companion*. IEEE Press, 2017, pp. 180–182.

[34] W. Weimer, "Patches as Better Bug Reports," in *Proceedings of the 5th International Conference on Generative Programming and Component Engineering (GPCE)*. ACM, 2006, pp. 181–190.

[35] W. E. Wong, V. Debroy, R. Gao, and Y. Li, "The DStar Method for Effective Software Fault Localization," *IEEE Transactions on Reliability*, vol. 63, no. 1, pp. 290–308, 2014.

[36] X. Yan, J. Han, and R. Afshar, "CloSpan: Mining: Closed Sequential Patterns in Large Datasets," in *Proceedings of the 2003 SIAM International Conference on Data Mining (SDM)*. SIAM, 2003, pp. 166–177.

[37] H. Zhong and Z. Su, "An Empirical Study on Real Bug Fixes," in *Proceedings of the 37th International Conference on Software Engineering (ICSE)*, IEEE/ACM. IEEE Press, 2015, pp. 913–923.

[38] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei, "MAPO: Mining and Recommending API Usage Patterns," *Proceedings of the 23rd European Conference on Object-Oriented Programming (ECOOP)*, pp. 318–343, 2009.