Automatic Patch Generation

Michael Lam

December 16, 2007

Abstract

Current static code analysis tools are able to detect errors in programs, but most cannot actually fix the errors. Manual debugging is necessary to fix these issues, but developers make mistakes and often work under tight deadlines. It may be greatly beneficial if analysis tools could make reasonable suggestions about how to fix the problems they detect. Automatic patch suggestion could decrease the amount of wasted developers' time, and improve the general quality of software. This paper presents a survey of the problem, current solutions, related fields, and future work.

1 Introduction

Static analysis tools detect flaws in computer programs before they are executed (ex. at compile time). This is usually done by performing some sort of flow- or type-based analysis on the program's source code. Static analyses can find null pointer errors [6], type errors [10, 9, 7, 8], security errors [14], and many other common mistakes that programmers make when writing software.

Error reporting is a related field to static analysis that deals with how to most effectively report the information gathered by the tool to the programmer. Frequently, error messages are difficult to interpret and occasionally do not point out the true source of the problem.

To aid the programmer's understanding of the error, static analysis tools sometimes include an illustrative example in the error message. Here is a simple example from OCaml:

```
let rec last =
  function
    [v] -> v
  | h::t -> (last t)
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
```

Recently, some authors [10, 14, 7] have explored the possibility of including a suggestion for actually fixing the problem, usually in the form of a code snippet that fixes the issue raised by the static analysis tool. For the example above, the error report might be enhanced by adding the suggestion:

```
Try replacing the highlighted code with:
let rec last =
  function
    [v] -> v
  | h::t -> (last t)
  | [] -> raise (Invalid_argument "")
```

McAdam [10] uses a spell-checker analogy to argue that automatic suggestions are desirable. For an example of this, consider the following input:

```
The quik brown fox jumps over a lazy dog.
```

Current methods of static analysis error reporting, if used by a spell-checker, might produce the following error message:

```
The quik brown ...
       ^
Spelling error at character 7
Cannot follow "qui" with "k"
```

In the context of spell-checking, this is obviously less useful than the list of suggestions that most spell-check engines provide. McAdam uses this to point out the usefulness of suggestion reporting as a better form of static analysis error reporting.

This analogy does have a weak point, however. One might point out that spell checkers use simple pattern matching methods to identify errors, while program checkers are much more complex. This is an important point, and begins to raise the question of whether the suggestion technique is a good general solution for static analysis domains.

Weimer [14] presents a more convincing argument by showing evidence that bug reports submitted with potential patches have an increased chance of being fixed quickly. This shows that fix suggestion has the potential to increase the rate at which bugs are fixed, which will improve the general quality of software.

This paper explores the problem of patch generation and three current solutions, as well as various related fields and possible directions for future research.

2 Problem

It is ultimately impossible to always generate perfect patches. The Curry-Howard isomorphism proposes that programs can be thought of as proofs. Given a proof in normal form, it is possible to create a different proof of the same propositions using repeated applications of the rule of consequence. Thus, when trying to generate code to fit a specification, there are many "valid" alternatives.

From multiple valid alternatives, it's difficult to automatically pick the "correct" one (ie. the one that the programmer intended). No matter how sophisticated a static analysis routine may be, it works only from the information present in the program code. If there is not enough information to deduce the intent of the original programmer, it will be impossible to make a conclusive suggestion. Thus, the existing patch suggestion techniques are heuristic in nature.

3 Current Methods

3.1 Type Inference-based

McAdam [9, 10] describes a system for fixing type errors in functional programs. This system finds possible replacements for type-unsafe expressions by rewriting them according to currying and associativity axioms, performing associative-commutative unification on the resulting forms, and finally performing a partial evaluation to achieve a human-readable and useful output. McAdam implemented this system in a tool for MLj.

The following is an example of a program with an error:

```
structure Test1 = struct
  val intList = [1, 2, 3]
  val intToString = Int.toString
  val _ = map ([1, 2, 3], Int.toString)
end
```

In this case, the arguments to `map()` are in the wrong order and not properly curried. McAdam's system is able to suggest the correct fix for this problem:

```
Error at 6.11-14: Try changing
  map (intList, intToString)
to
  (map intToString intList)
```

Regarding performance, McAdam claims that only about 3ms are added to the compilation process.

Lerner and Grossman [7, 8] describe a similar system called Seminal. Their tool attempts to improve error messages by providing suggestions for re-writing type-unsafe code in a way that resolves the type error. The resulting code may not be logically correct, but is guaranteed to be typesafe.

Seminal examines a type-unsafe functional program using a top-down analysis on the syntax tree, starting at the root expression. It replaces each subtree individually with expressions that are guaranteed to be type-safe (called "wildcards"), and if the resulting program typechecks, then that subtree is deemed "interesting." This procedure is repeated recursively for all "interesting" subtrees until a leaf is

`e1`	`raise (DummyExn ())`
`p1`	`_`
`[e1,e2,e3]`	`[e1;e2;e3]`
`let e1 = e2 in e3`	`let rec e1 = e2 in e3`
`let v1:t1 = e1`	`let v1 = e1`
`match e1 with`	`match e1 with`
`  | p1 ->`	`  | p1 ->`
`    match e2 with`	`    (match e2 with`
`    | p2 -> e3`	`    | p2 -> e3`
`    | p3 -> e4`	`    | p3 -> e4)`
`  | p4 -> e5`	`  | p4 -> e5`

Figure 1: Seminal Replacement Examples

reached. Figure 3.1 shows several of the replacement wildcards used by Seminal.

Seminal then compares all of the replacements and ranks them according to several criteria; generally, it prefers smaller changes. Finally, it reports a specified number of these changes to the developer.

There are currently two implementations: 1) an extension to the OCaml compiler for the OCaml language, and 2) an Eclipse plugin for the C++ STL.

These projects have many similarities:

- Both are type inference-based and flow-insensitive.
- Both implementations are for functional languages.
- Both algorithms have some built-in knowledge of common programmer mistakes and use this knowledge to aid in suggestion.

These two methods differ primarily in the method used to generate suggestions. Seminal uses a top-down approach, starting at the top level and quickly narrowing the focus to the "interesting" portions of the program. McAdam's algorithm is more bottom-up, starting with the conflicting types and attempting to unify them into a general solution for a type error.

In addition, McAdam's approach is more localized, since it does not search for other locations involved in the error, which Seminal will find. Seminal also takes care to present errors in terms of types that the programmer has written, while messages generated by McAdam's tool might be confusing since it presents generated types.

3.2 Dataflow- and State Machine-based

Weimer [14] describes a system for generating patches from static analysis error reports. This tool is able to automatically generate missing code or remove extraneous code to produce a program that satisfies a given policy. Weimer also includes a study of the effectiveness of this method across several projects by reporting bug fix rates as correlated with automatic patch suggestion.

The algorithm presented by Weimer is flow- and path-sensitive (as opposed to the inference-based algorithms discussed in the previous section), and the paper contains sections on path predicates and exception handling. These constructs, combined with the fact that a solution may involve adding or removing code, make this process of patch suggestion nontrivial. The algorithm generates multiple correct patches and reports the shortest one.

Weimer applies his algorithm to the error reports generated by FindBugs [6] and Weimer's own WN [15]. In general, the number of bugs submitted with patches that were eventually fixed was equal to or higher than the number of bugs submitted without patches that were eventually fixed.

Finally, Weimer describes his work as a "first step" towards easing developer use of automatically generated patches. He calls for more investigation of various metrics for determining the "best" candidate patch. In addition, Weimer calls for more studies relating error reporting and patch generation techniques to the number of bugs that are eventually fixed.

4 Related Fields

4.1 Security-Related Patching

Several projects have used the term "automatic patch generation" to describe the process of repairing binaries after a malicious attack such as a buffer overflow

[12, 11]. Interestingly, these techniques tend to use the same approaches to fixing problems (eg. slicing and pattern matching) as the patch generation techniques discussed previously. The task here is more straightforward, though, since ground truth is more readily available; the patch generator just assumes the original was correct and that only the currently running version is compromised.

4.2 Error-correcting Compilers

There has been much historical work in error-correcting compilers (ex. [1, 3]). These techniques actually serve a slightly different purpose (with different scope). The main goal of an error-correction routine in a compiler is to continue the compilation process even after finding a syntactic error, and there are few attempts to find or fix semantic or logical errors.

Error-handling techniques fall into two broad categories: 1) correcting and 2) non-correcting. The former techniques try to transform the input into a valid string, while the latter techniques simply discard any portions of the input related to the error. The latter is not helpful for patch generation techniques (deleting all offending code is rarely the correct solution to a problem), but there may be aspects of the former that are useful.

Among correcting techniques, there is also variation of scope. Some techniques look only at the immediately surrounding context, while others look more broadly at the entire program. Usually, the local search is performed first, and the second is used as a last resort if the first fails. Other techniques work interactively, prompting the developer to correct syntax errors as the compiler runs and restarting from a previous nearby point once the error is fixed.

This idea of interactive issue resolution might serve as a way of integrating patch generation with development environments. As a program is compiled, the developer could be presented with an interactive dialog, showing issues found by a static analysis tool along with various possible patches. When the developer chooses a patch (or makes manual changes), compilation would continue.

4.3 Compiler Error Post-processing

STLFilt [19] is a tool for post-processing compiler error messages in order to make them more readable (and hopefully by extension more usable). Weimer [14] points out that this type of solution is really not complete since it does not provide a solution of any sort. However, tools like this may be useful in generating human-readable output for suggestions, particularly in formatting patch code.

4.4 Program Slicing

"Slicing" is a technique that builds a set of all program points around any piece ("slicing criterion") of a program. In this context, slicing could be used to obtain all portions of a program involved in an error, in the hope that this information can improve type explanation.

Slicing has been a research topic since the early 1980s [16]. There are various ways of extracting an intraprocedural slice: dataflow analysis, information-flow analysis, and program dependence graphs (PDG). PDGs are fastest method [13]. Interprocedural slicing is difficult, and the first attempts were not complete. There were also problems with slicing in the context of composite datatypes and pointers, as well as interprocess communication. Most of these problems had been addressed by the time of the 1995 Tip survey paper [13].

Program slicing is a well-developed field, introduced around 1980 and continued today with research in the areas of conditional, dynamic, and object-oriented slicing. Various authors [4, 5] have also used slicing to detect type errors. However, there appears to be no work on using slicing to automatically generate bug patches. It's possible that performing slicing before running a patch generation tool could reduce the amount of code the generation tool would have to analyze.

4.5 Type Explanation

There is a large body of literature on "type explanation," which describes techniques used to increase the amount of information given in error messages about the cause of type errors [2, 18, 17]. This sort of analysis could form the basis of a bug fix suggestion engine.

Most type explanation is based in the Hindley-Milner style polymorphism of functional languages. Type checkers for such a language produce error messages when the actual type of an expression does not match the expected type of that expression. More generally, the checker reports an error when two uses of an expression or variable are found to have conflicting types.

Type checking presents two difficulties in reporting the errors to the developer. First of all, most of the information the messages contain is inferred by the type inference engine, so it is difficult for the developer to understand the context of the reported information in relation to the code that they are familiar with. The second difficulty is that the error could be at either use site, and there is no good way of automatically determining which site needs to be changed to fix the problem.

To date, there doesn't appear to be any work in using type explanation techniques to augment patch generation routines. It seems that it would be useful to know all possible sites of an error, and this knowledge might lead to a more correct solution to a problem (ie. changing a single non-local site rather than multiple local sites).

5 Future Work

Many questions remain in this area:

- Most of the existing techniques boil down to pattern matching and replacing. The patterns are seeded by human programmers. Are there any bug domains for which an automatic fix is possible, but it cannot be generated simply by a pattern match and replace?

- Is there a more general way of generating bug patches? What role might annotations play in a scheme like this?

- Are these types of analyses efficient? Does this even matter? Is it possible for integrated development environments to run these types of analyses in the background during the idle times?

- Extrapolated even further, is it possible to perform patching automatically on code that is about to be executed? What sort of effects would this have on speed, reliability, and security?

Along with the above questions, it's certainly possible to extend current solutions. Lerner and Grossman's Seminal could be improved by the addition of more search patterns and change strategies. It could also be interesting to add a plugin structure for adding new strategies, so that any programmer could add a patch generation routine for errors observed in their specialized domain. Weimer's tool might be improved by comparing various metrics for determining edit distances in order to see if different metrics work better in different situations.

All existing patch generation tools would benefit from more testing on a wider range of source programs. Also, integrating any of the discussed solutions into an IDE would allow them to be used more seamlessly in a real development environment, perhaps providing a better demonstration of their usefulness. Finally, all of the methods discussed could be made more useful by creating implementations for new languages (all are currently implemented only for one or two languages).

Even though suggestion techniques can be improved, however, it's not clear that this is necessarily a good thing. Perhaps in practice programmers will ignore the suggestions, and the computational time spent generating the patch will be wasted. On the other hand, programmers might tend to apply patches blindly without verifying their correctness or side effects. Finally, it's always possible that fixing one bug might introduce another (or several others) of a different type that is not detectable by the analysis used to detect the first error. If this is the case, consequences of the first bug might be less severe than the consequences of the second bug, and thus the patch may negatively affect reliability.

A study to examine programmer application of automatically generated patches (possibly correlated with future bugs on the same code segments), would help to reveal whether the suggestions are actually helping. A well-designed study would also explore the breakdown of developer debugging time to see how these tools affect the amount of time spent on the various aspects of debugging.

Finally, the greatest unanswered question is whether the bugs addressed by these sorts of analyses are really important. Static analysis techniques are generally designed to be conservative and sound, and usually generate many false positives. Even if a bug is not a false positive, occasionally it doesn't matter since the offending path is rarely or never executed. If patch generation is effective only for these cases, its relevance is questionable.

To be a viable research topic, automatic patch generation needs to be shown to effectively fix bugs that are both serious and otherwise hard to correctly address. Since it is difficult to quantify the seriousness of a bug, and since studies involving actual developers on real-world projects are difficult to justify for employers, it may be hard to build a strong case for automatic patch generation.

6 Conclusions

Automatic patch generation is feasible, but current techniques are limited to well-defined scopes and problem domains. Simply put, they can only suggest good solutions to very specific types of problems, usually involving common programmer mistakes. Future work may involve extending current techniques, integrating tools from related fields, and justifying the usefulness of patch generation with empirical studies.

References

[1] A. Aho and T. Peterson. A minimum distance error-correcting parser for context-free languages. *SIAM Journal on Computing*, 1:305–312, 1972.

[2] M. Beaven and R. Stansifer. Explaining type errors in polymorphic languages. *ACM Lett. Program. Lang. Syst.*, 2(1-4):17–30, 1993.

[3] P. Degano and C. Priami. Comparison of syntactic error handling in LR parsers. *Software - Practice and Experience*, 25(6):657–679, 1995.

[4] T. B. Dinesh and F. Tip. A slicing-based approach for locating type errors. In *264*, page 24. Centrum voor Wiskunde en Informatica (CWI), ISSN 1386-369X, 31 1998.

[5] C. Haack and J. B. Wells. Type error slicing in implicitly typed higher-order languages. In *In Proc. 12th European Symposium on Programming, Lecture Notes in Computer Science, Warsaw, Poland, April 2003.* Springer-Verlag., 2003.

[6] D. Hovemeyer and W. Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12):92–106, 2004.

[7] B. Lerner, D. Grossman, and C. Chambers. Seminal: searching for ml type-error messages. In *ML '06: Proceedings of the 2006 workshop on ML*, pages 63–73, New York, NY, USA, 2006. ACM.

[8] B. S. Lerner, M. Flower, D. Grossman, and C. Chambers. Searching for type-error messages. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 425–434, New York, NY, USA, 2007. ACM.

[9] B. McAdam. How to repair type errors automatically. In *In Scottish Functional Programming Workshop, Stirling, U.K., pages 121–135, August 2001.*, 2001.

[10] B. McAdam. *Repairing Type Errors in Functional Programs.* PhD thesis, Laboratory for Foundatations of Computer Science, The University of Edinburgh, 2001.

[11] S. Sidiroglou and A. Keromytis. Countering network worms through automatic patch generation, 2003.

[12] A. Smirnov and T. Chiueh. Automatic patch generation for buffer overflow attacks. *Information Assurance and Security, 2007. IAS 2007. Third International Symposium on*, pages 165–170, 29-31 Aug. 2007.

[13] F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995.

[14] W. Weimer. Patches as better bug reports. In *GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering*, pages 181–190, New York, NY, USA, 2006. ACM.

[15] W. Weimer and G. Necula. Finding and preventing run-time error handling mistakes. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications.*, pages 419–431, 2004.

[16] M. Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, pages 439–449. IEEE Computer Society Press, 1981.

[17] J. Yang, G. Michaelson, and P. Trinder. Explaining polymorphic types, 2001.

[18] J. Yang, J. Wells, P. Trinder, and G. Michaelson. Improved type error reporting. In *In M. Mohnen and P. Koopman, editors, Proceedings of 12th International Workshop on Implementation of Functional Languages, pages 71–86, Aachner Informatik-Berichte, 2000 23*, 2000.

[19] L. Zolman. An stl error message decryptor for visual c++. *C/C++ Users Journal*, 2001.