# Towards s/engineer/bot: Principles for Program Repair Bots

Rijnard van Tonder, Claire Le Goues
*Carnegie Mellon University*
rvt@cs.cmu.edu, clegoues@cs.cmu.edu

***Abstract*—Of the hundreds of billions of dollars spent on developer wages, up to 25% accounts for fixing bugs. Companies like Google save significant human effort and engineering costs with automatic bug detection tools, yet automatically fixing them is still a nascent endeavour. Very recent work (including our own) demonstrates the feasibility of automatic program repair in practice. As automated repair technology matures, it presents great appeal for integration into developer workflows. We believe software bots are a promising vehicle for realizing this integration, as they bridge the gap between human software development and automated processes. We envision repair bots orchestrating automated refactoring and bug fixing. To this end, we explore what building a repair bot entails. We draw on our understanding of patch generation, validation, and real world software development interactions to identify six principles that bear on engineering repair bots and discuss related design challenges for integrating human workflows. Ultimately, this work aims to foster critical focus and interest for making repair bots a reality.**

***Index Terms*—bots, program transformation, automation, software quality, refactoring, automatic program repair**

## I. Introduction

Companies like Google develop and integrate automatic bug detection tools because it saves significant human effort and development costs [1]. A survey by Chen et al. [2] reports that of the hundreds of billions of dollars spent on developer wages, up to 25% accounts for fixing bugs [2]. Automation thus holds huge potential to further reduce human effort by automatically *fixing* bugs. Although adoption is nascent, very recent work demonstrates the feasibility of automatic program repair in practice [3], [4]. As automated repair technology matures, it presents ever more appeal for integration into developer workflows.

We believe software bots are a promising vehicle for realizing this integration, as they bridge the gap between human software development and automated processes. We envision *repair bots* that orchestrate automatic patch generation, application, and validation for performing automated refactoring and bug fixing. To this end, we ask: what does building a repair bot entail? A first paper on the topic by Urli et al. [4] reveal insightful challenges and recommendations while engineering the repair bot `Repairnator`, which fixes CI build errors. They focus particularly on a blueprint design stemming from `Repairnator` and offer a notably concrete example in the design space of repair bot possibilities. The intent of our paper, by contrast, is to elaborate on a higher level view of this design space, abstracting discussion away from a particular language or repair approach.

We identify six principles that bear on engineering repair bots and discuss design elements to address related challenges. Our position draws on our own research broadly concerned with automated program transformation toward software quality [5]–[8]. Our work demonstrates that automated refactoring [8] and program repair [5] can be a reality for mainstream adoption, and our techniques have resulted in over 50 patches merged into highly popular open source projects for over 12 languages. We draw on our understanding of patch generation, validation, and pull request reviews to suggest considerations when engineering repair bots.

The core of automating bug fixes and refactoring seeks *confidence* in desirable program transformations while *minimizing human effort*. The scope and complexity of desirable program transformations will naturally depend on project and developer needs, and in turn bear on the level of validation necessary to argue that a change meets the desired objectives (e.g., improving readability, performance, or correctly fixing a bug). Applying program changes and validating their effect distinguishes repair from auto-formatting tools and linter bots. In our previous work, whether automating simple refactorings or semantic-driven fixes, syntactic manipulation and change validation constituted significant effort (and importance) when submitting patches to upstream repositories. The process consisted of these general phases once we identified a project to repair:

1) Generate a syntactically well-formed patch using one of our techniques.
2) Perform additional patch validation (e.g., type check or run test suites against the patch).
3) Perform auto-format postprocessing.
4) Issue a Pull Request on GitHub.
5) Address reviewer feedback if needed.

Syntax manipulation (1) and change validation (2), are core to automated reasoning and pose unique challenges to repair bot designs. We thus place special emphasis on these concerns in Section II and III respectively. Auto-formatting (3) and pull requests (4) are predominantly mechanical actions that are not necessarily unique to repair bots. They play the role of (a) ensuring code conforms to existing styles and (b) providing an endpoint for maintainers to review and merge code. We discuss these, and other aspects of human developer workflows,

in Section IV. For brevity, this paper consciously elides the inner workings of various repair techniques. Instead, we focus on how general aspects of automated transformation bear on designing and engineering repair bots, giving special attention to challenges for generalizing to multiple languages.

## II. Dealing with Syntax

Many desirable program transformations can be achieved by solely reasoning about and transforming syntax. For example, popular linter tools (e.g., ESLint [9]) include a `--fix` option to automatically fix lint errors. Although syntax transformation can be visually straightforward, two deeper considerations for automation are at play. The first is language generalization: to what extent is syntactic manipulation supported across multiple languages (§II-A)? The second is validating syntax changes: how do criteria for syntactic well-formedness influence automation (§II-B)?

### A. Syntax Generality

Syntax transformation tools can be either language-specific or handle multiple languages. Language-specific tools typically parse programs into concrete syntax trees then output modified syntax (e.g., Clang [10]), while multi-language tools manipulate syntax using a general grammar or abstract representation [11], [12]. Due to syntactic ambiguity and variation in contemporary languages [13], using multi-language tools are less common than language-specific counterparts in mainstream projects.

**Principle #1.** *Syntactic idiosyncracies and ambiguities introduce complexity for automating general (i.e., multi-language) syntax transformation.*

**Approaches.** A repair technique may support multi-language transformation through (1) heterogeneous, language-specific toolchains or (2) multi-language frameworks, or (3) a combination of these. A repair bot must invariably integrate one or more of these approaches, and each imposes constraints on bot generality and maintainability. The first allows unambiguous and accurate syntax transformation, but reduces generality and maintainability for automation: multiple language and compiler toolchains will have to integrate with a repair bot and couple tightly to project languages and versions. A multi-language framework (e.g., [14], [8]) promises improved engineering efficiency and hence bot maintainability, but must be sufficiently expressive to perform sophisticated syntax transformations and powerful enough to resolve syntactic ambiguities for supported languages. As an extreme example, `sed` can manipulate any program text (i.e., it is multi-language), but its regular engine is not expressive enough transform context-free or context-sensitive languages. A combination of approach (1) and (2) may offer a balance in maintainability and expressivity, but is not currently explored in practice.

### B. Syntax Validation

Automated checkers (e.g., linters) typically have the luxury of operating on well-formed syntax. Program transformation by definition goes beyond this "read-only" restriction, and desirable transformations must be, at minimum, syntactically well-formed to imply correctness. A repair bot that incorporates and communicates syntactic well-formedness of automated changes increases human confidence in correctness.

**Principle #2.** *Syntactic changes should be validated against a well-formedness criterion.*

**Approaches.** Compilers validate syntax early in the compilation pipeline, and offer a straightforward approach for ensuring well-formed syntax for a single language. Multiple compiler toolchains are required in a multi-language approach, and imposes a maintainence cost (as in §II-A). Moreover, compiler configuration may require substantial human effort, and compilation may be computationally expensive for large projects; these problems may induce excessive computational cost to validate a simple syntactic change. It would be convenient if a compiler only parsed target source code (to ensure syntactic well-formedness), but compiler pipelines may tightly couple parsing with other phases (i.e., type checking, optimization). An alternative possibility is to relax the well-formedness criterion on syntax and specify it over multiple languages (analogously to adopting an abstract representation for multi-language transformation frameworks) For example, our work on automated refactorings [8] ensures that delimiters (e.g., parentheses, braces, etc.) are balanced in syntax output. This criterion is sufficient for the transformations considered, and holds for multiple languages. Using a relaxed definition of syntactic well-formedness can (a) generalize more easily to multiple languages (improving bot maintainability) and (b) execute early (improving bot efficiency), though the guarantees may be weaker than those of compiler checks. This approach suggests a staged validation process where cheaper syntactic checks take place before potentially more expensive semantics-based ones.

## III. Performing Semantic Validation

Understanding how a change affects program behavior presents greater opportunity to automate deep and complex program fixes. Semantic checks thus provide stronger validation than syntactic ones, particularly because validation can be directed at specific program properties, such as a particular bug kind or fault location. This may require reasoning across procedure boundaries and multiple files, or validating a change against a test suite. Existing work shows that integrating semantic information with automated reasoning is crucial for more sophisticated program fixes [4], [5], [15]; intuitively, automated reasoning benefits from the same semantic information used by engineers to understand and fix bugs (type information, tests, program traces, etc.).

The landscape for incorporating semantic techniques into program transformation is rich. We can broadly classify such validation strategies and properties by whether the program needs to be run (dynamic) or not (static). We identify a subset of uses in this space that we have found compelling in our own work.

### A. Build Validation

Any syntactic change should still produce a compilable program. Successful compilation is thus a necessary (but not necessarily sufficient) criterion for ensuring a correct program transformation. A successful build means that a set of basic semantic properties are satisfied (e.g., no variable use-before-defines). Because compilation does not require running the program, it may be more efficient than dynamic techniques.

**Principle #3.** *Automated transformations should be validated with a successful build of the program.*

**Approaches.** Building software artifacts can be computationally expensive and time consuming to configure. This burden is especially high for external contributors, who may need to set up a local copy, install dependencies, and then build a project from scratch to validate a change. Continuous Integration (CI) services, like Travis CI,[1] alleviate build reproduction with new changes. CI builds are typically triggered by a pull request made to a project's code host. A passing CI build is essential for communicating good changes during review. However, typical repair approaches need to test for a passing build *before* issuing a pull request. One approach (also noted in [4]) is to replicate the CI environment and build pipeline on separate infrastructure. Forking a repository offers a way to attempt to replicate the build pipeline. Unfortunately, we have found that forking CI builds is unreliable: builds may fail for a number of unrelated reasons (e.g., the fork build cannot authenticate to external endpoints). Replicating the original build for testing changes (but without issuing a pull request) offers an ideal conceptual solution, but one that is not readily available to our knowledge. An alternative approach is to use (1) personal infrastructure, (2) follow the build instructions for the project, and (3) integrate these steps into a validation script. We have opted for this approach in most of our work because it is reliable—unfortunately, step (2) poses a significant barrier to easy automation. A third possibility is to use CI build scripts and attempt to synthesize local build or validation scripts.

The variety of available languages, their compilers, and a plethora of build systems exacerbate the challenges above. CI builds provide a critical abstraction for reducing the complexity of configuring and interfacing a repair bot with this fragmented space. The promise of CI support for external services to easily and reliably validate isolated builds is pivotal to maintainable and efficient repair bots.

### B. Testing

A change that passes all tests for a program is another necessary (but not necessarily sufficient) criterion for a correct program transformation. Tests are inherently dynamic and can incur greater computational cost, but provide greater assurance of a change's correctness with respect to particular program functionality. They may even detect, for example, performance regressions.

**Principle #4.** *Automated transformations should be validated by running changes against available test suites.*

**Approaches.** CI pipelines typically also run test suites against changes, and offer an attractive approach for the same reasons as build validation. The attractiveness of using CI is similarly dampened by the difficulty of easily replicating builds. Local testing may have additional dependencies or rely on external services, and adds additional maintenance concerns for automation. Since running tests can be expensive, incremental testing and caching factors into validation efficiency and bot responsiveness for code changes.

Note that any compiled or tested change implies well-formed syntax (i.e., adopting approaches for Principles 3 and 4 subsumes 2). However, Principle 2 holds independently for simple syntactic or linter fixes that do not require additional semantic validation granted by 3 and 4.

### C. Type Information

Type information enables complex automated fixes and refactorings [5], [16]. On the one hand, types can restrict the set allowable program candidate changes; on the other, they serve as meta-information for querying whether a particular change is legal (e.g., for disambiguiting syntax [13]). Incremental type checking may be sufficient validation for simple changes in lieu of a full program build. Type information can be computed efficiently, and is therefore an attractive validation mechanism.

**Principle #5.** *Computing and exposing type information enables a fast static validation mechanism for automated transformations.*

**Approaches.** Most existing repair tools directly access type information through a program's abstract syntax tree (AST). This approach is straightforward. However, it creates a tight coupling between repair tools and compilers where (a) each repair tool independently computes and uses the same type information as other potential repair tools and (b) each such interface must be maintained independently. This presents a significant challenge for scaling automation to multiple languages. The language server protocol (LSP)[2] presents a new approach for incorporating type information. LSP was developed to provide editor information as a service (including type information lookup). The LSP standardization makes it possible query type information for a project hosted in a central location (where compilation and type information is cached). Such queries are thus efficient and generalize to multiple languages. We see LSP as a promising move towards decoupling static semantics from program syntax and locally managed compilation. These aspects make LSP interfaces a natural consideration for building repair bots.

### D. Static Analysis

Our work uses static analysis to provide another layer of validation to automated changes [5]. Since static analyses can detect specific property violations (i.e., bugs), they can also validate whether a program change satisfies a desirable property (e.g., removes a bug). Once again, such validation may be a necessary but not sufficient condition; however, static

[1] https://travis-ci.org

[2] https://langserver.org

analyses are special in that they may validate that a particular bug was removed (with more particular guarantee than, e.g., tests). Static analyses are generally fast, finishing in the order of minutes. These features make analyses another significant interface consideration for repair bots.

**Principle #6.** *Static analyses can validate complex changes with respect to desirable program properties and thus enable sophisticated fixes; such validation should accompany automated transformations where applicable.*

**Approaches.** Static analyses complement automated repair. The choice of static analysis will invariably depend on the target language and bug kinds being detected, which is difficult to generalize to program repair bots. Broadly speaking, static analyses impose additional (and likely ad-hoc) integration with a bot architecture, and may impose significant maintenance effort due to varied tools and configurability. We see the biggest opportunity lying in the *standardization* of static analysis tools across languages, such that capabilities and validation guarantees are made explicit. I.e., does the analysis cover the program change explicitly in its model (or could it be missed because the analysis is imprecise)? How correct is the abstraction; could it be unsound? Standardization of bug kinds, and coverage of the analysis, strengthen the automation pipeline for accessible inspection and debugging. Sarif[3] is a recent effort proposing a standardized format for static analysis output offering promise in this direction. We believe analysis capabilities, assumptions, and results should accompany validation in a standardized way. This will enable repair bots to appropriately orchestrate repairs and report validation guarantees to human interfaces.

## IV. Integration with Human Workflows

Automated processes are not necessarily fully autonomous: humans still regularly configure, deploy, and monitor automation. Repair bots are no exception, and their effectiveness depends on successful integration with human processes of software development. We outline general considerations behind this integration toward the goal of minimizing human effort and enabling a push-button approach to approving automated changes.

**Configuration.** Developer needs differ across projects, and exposing the right bot configuration options impacts effectiveness. One key challenge we have faced is determining which files are valid targets for refactoring or repair. In some cases, this is easy. For example, the `vendor` folder for Go projects contains dependencies that should generally be ignored. In others projects, external dependencies are included without any special indication. A further difficulty lies in ignoring files that explicitly test for buggy functionality. Blacklisting targets is one option that must be communicated to repair bots. Other options include timeouts, and filtering files with specific extensions.

**Postprocessing.** Projects adopt their own styles and conventions. Auto-formatting tools ensure consistent styles (e.g., indentation, line length, and comments). Using syntax-manipulating tools (both our own and others) can break these stylistic conventions, which are typically outside the scope of the tool to correct. We have found it effective to apply postprocessing using existing auto-formatters after syntax changes. To minimize human effort we advocate auto-formatting postprocessing in repair bots to maintain readability.

**Code Host Integration and Deployment.** Code hosts like GitHub and GitLab are natural platforms for software bots because they (a) store the source of truth for a software project and (b) expose API endpoints for performing actions automatically. The choice of code host thus deeply affects how bots are engineered, configured, and deployed. Code host features are constantly changing and improving, and we do not attempt to give a full breakdown of relevant features here. Nevertheless, we have found appealing ones to include: (a) push-button approval for suggested changes and pull requests [17], (b) a GUI indicating progress and completion of checks for bots [18], and (c) a well-designed and documented API for interacting with code host (e.g., issuing pull requests, adding tags to pull requests, or adding reviewers).

**Review.** Humans are the gatekeepers of program changes, even automated ones. The review process for a project therefore factors into building and deploying a repair bot. An interesting consideration for automated changes is quantifying the size of the change and potential review burden. Large refactors resulting from a conceptually simple change may appear intimidating for review. Conversely, extensive change validation (e.g., passing builds and tests) can encourage quick review of changes. Changes, and their validation, should thus be automatically tagged with attributes to streamline review efficiency.

## V. Discussion and Conclusion

Various project-specific needs will impose different requirements and constraints on automated changes, whether lightweight syntactic changes (Section II), or semantic-driven fixes (Section III). At a high level, the principles behind syntactic transformation, change validation, and capitalizing on external tools (e.g., LSP type provision and static analyses) encourage *decoupling concerns* to achieve maintainable and reusable architectures for building repair bots based on need. Whatever the flavor of automated change, the expectation of repair bots is that they integrate successfully and efficiently with human workflows; configurability, code hosts, project style, and review processes bear on their deployment (Section IV). Repair bots lie in the unique intersection of the principles and challenges outlined above; we look forward to their development for realizing end-to-end program repair in the future.

## Acknowledgments

This work is partially supported under NSF grant number CCF-750116. All statements are those of the authors, and do not necessarily reflect the views of the funding agency.

[3]http://docs.oasis-open.org/sarif/sarif/v2.0/csprd01/sarif-v2.0-csprd01.html

## References


[1] C. Sadowski, E. Aftandilian, A. Eagle, L. Miller-Cushon, and C. Jaspan, "Lessons from building static analysis tools at google," *Commun. ACM*, vol. 61, no. 4, pp. 58–66, 2018.

[2] S. Chen, W. K. Fuchs, and J. Chung, "Reversible debugging using program instrumentation," *IEEE Trans. Software Eng.*, vol. 27, no. 8, pp. 715–727, 2001.

[3] "Getafix: How Facebook tools learn to fix bugs automatically," https://code.fb.com/developer-tools/getafix-how-facebook-tools-learn-to-fix-bugs-automatically/.

[4] S. Urli, Z. Yu, L. Seinturier, and M. Monperrus, "How to design a program repair bot?: insights from the repairnator project," in *International Conference on Software Engineering: Software Engineering in Practice*, 2018, pp. 95–104.

[5] R. van Tonder and C. Le Goues, "Static automated program repair for heap properties," in *International Conference on Software Engineering*, ser. ICSE '18, May 2018, pp. 151–162.

[6] R. van Tonder, J. Kotheimer, and C. Le Goues, "Semantic crash bucketing," in *International Conference on Automated Software Engineering*, ser. ASE '18, 2018.

[7] R. van Tonder and C. Le Goues, "Defending against the attack of the micro-clones," in *International Conference on Program Comprehension*, ser. ICPC Short '16. IEEE Computer Society, May 2016, pp. 1–4.

[8] R. van Tonder and C. Le Goues, "Lightweight multi-language syntax transformation with parser parser combinators," in *Conference on Programming language Design and Implementation*, ser. PLDI '19, to appear, 2019.

[9] "ESLint," https://eslint.org, 2018, online; accessed 31 January 2019.

[10] "Clang's refactoring engine," https://clang.llvm.org/docs/RefactoringEngine.html, 2018, online; accessed 11 October 2018.

[11] P. Klint, T. van der Storm, and J. J. Vinju, "RASCAL: A domain specific language for source code analysis and manipulation," in *International Working Conference on Source Code Analysis and Manipulation*, ser. SCAM '09, 2009, pp. 168–177.

[12] J. R. Cordy, "The TXL source transformation language," *Sci. Comput. Program.*, vol. 61, no. 3, pp. 190–210, 2006.

[13] E. A. T. Merks, J. M. Dyck, and R. D. Cameron, "Language design for program manipulation," *IEEE Trans. Software Eng.*, vol. 18, no. 1, pp. 19–32, 1992.

[14] J. Koppel, V. Premtoon, and A. Solar-Lezama, "One tool, many languages: language-parametric transformation with incremental parametric syntax," *PACMPL*, vol. 2, no. OOPSLA, pp. 122:1–122:28, 2018.

[15] X.-B. D. Le, D. H. Chu, D. Lo, C. Le Goues, and W. Visser, "S3: Syntax- and semantic-guided repair synthesis via programming by examples," in *Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE '17. ACM, Sep. 2017, pp. 593–604.

[16] K. Sagonas and T. Avgerinos, "Automatic refactoring of erlang programs," in *International Conference on Principles and Practice of Declarative Programming*, ser. PPDP '09, 2009, pp. 13–24.

[17] "GitHub suggested changes," https://help.github.com/articles/incorporating-feedback-in-your-pull-request/#applying-a-suggested-change, 2018, online; accessed 11 October 2018.

[18] "GitHub status checks," https://help.github.com/articles/about-required-status-checks/, 2018, online; accessed 11 October 2018.