Using AI to Automate the Modernization of Legacy Software Applications

Vikram Nitin
vikram.nitin@cs.columbia.edu
Columbia University
New York, NY, USA

Abstract

The task of modernizing legacy software has gained increasing attention in recent years. Old code is prone to security vulnerabilities, and is difficult to maintain and upgrade. Manual approaches to modernize legacy software involve intensive human effort and are challenging to scale up. Thus, there is an urgent need to develop automated techniques to modernize old code. In this proposal, we shall look at three aspects of this problem. The first is the conversion of legacy monolithic software architectures to modern microservice architectures. The second is the translation of code written in older programming languages like C, to code written in modern programming languages like Rust. The third is the detection of bugs that arise during modernization. We look at three prior papers (written by this author) that address each of these three aspects of application modernization. For each of these, we also present some ideas and directions for further research.

CCS Concepts

• **Software and its engineering** → **Software evolution**; **Maintaining software**; Software version control.

Keywords

Modernization, Software, Legacy, Microservice, Translation, Rust

ACM Reference Format:
Vikram Nitin. 2024. Using AI to Automate the Modernization of Legacy Software Applications. In *39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24), October 27-November 1, 2024, Sacramento, CA, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3691620.3695610

1 Introduction

Software systems are used everywhere in the modern world, from banking applications, to airline reservations, to weather forecasting. However, under the hood, many of these software systems run on highly complex code written decades ago, that is prone to security vulnerabilities, unable to scale effectively, and difficult to maintain. As some recent high-profile failures [5] have shown, this problem is extremely pervasive and can have massive financial impact. By some estimates, the "technical debt" associated with maintaining legacy code costs the USA over a trillion dollars annually [20]. Consequently, there is a pressing need to modernize legacy code. Modernization of legacy code has different aspects.

Topic 1 - Monolith to Microservice Refactoring: One such aspect is converting monolithic applications to microservices. Businesses are gradually migrating their existing applications to the cloud as their enterprise applications outgrow their monolithic architectures. This allows them to leverage better scalability and reliability, faster development, easier maintenance and deployment, and better fault isolation, among others [30]. This migration requires a decomposition of monolithic applications into loosely connected compositions of specialized microservices [10]. However, modernizing an enterprise application with entrenched technology stacks is a challenging task. Manual approaches to modernization consume a significant amount of engineering time and resources [1] and a complete rebuild is rarely feasible. Therefore, big enterprises frequently advocate for partitioning applications by identifying functional boundaries in the code that may be extracted as microservices.

Topic 2 - Code Translation: Another aspect of modernization is converting code written in older programming languages to newer programming languages. This task is crucial because old programming languages like C and JavaScript have issues related to memory safety and type safety. New programming languages like Rust, Go, and TypeScript have been developed to overcome the limitations of older ones. For instance, Rust adopts a memory model based on ownership that completely eliminates memory safety errors for a subset of the language. The problem of C to Rust conversion in particular has seen a lot of interest in recent times [32].

Topic 3 - Detecting Bugs in Modernized Code: Software modernization typically takes extensive effort by human experts. This is not only costly and time-intensive, but also error-prone. If the modernization process is not done with care, the modernized software might contain hidden bugs and/or security vulnerabilities. Thus, there is a pressing need to develop tools to automate parts of this process, as well as build systems that can help developers detect bugs that arise in the process of modernization.

In this paper, we shall study the problem of using AI to automate these two aspects of software modernization - microservice refactoring, and code translation. We shall then look at techniques to detect bugs that might arise in the process of application modernization.

2 Background

2.1 Monolith to Microservice Refactoring

Typical microservice extraction approaches start by converting a monolithic application into a call-graph or control-flow graph representation. Then, an off-the-shelf graph clustering algorithm is used to find loosely connected functional boundaries and make partitioning recommendations. The performance of these graph clustering algorithms is directly influenced by the expressiveness and completeness of the graphs they use.

Popular approaches such as CoGCN [8] and microservice extractor [1] use context-insensitive static-analysis to build a call graph for analysis. Certain other approaches construct a dynamic call graph by instrumenting the program to amass light-weight traces as the application's many use-cases are exercised [14, 15].

2.2 Code Translation

Traditionally, translating code between languages has been accomplished with transpilers [9, 12]. These are language-specific rule-based systems that use intricately designed algorithms and heuristics to produce code in the target language that is guaranteed to be functionally identical to the code in the source language.

Over the past few years, Large Language Models (LLMs) have been employed to tackle a range of code-related tasks, including code generation, code summarization, bug fixing, and test case generation. An expanding body of research [2, 25–27, 34] focuses on leveraging LLMs for automatic code translation between programming languages.

2.3 Detecting Bugs in Modernized Code

Since this topic is very broad in scope, we restrict our attention to the specific problem of *detecting bugs in the Rust language*. Rust has emerged as a popular replacement for C in kernels, drivers and embedded system software [13, 31]. Rust has two subsets - *safe* Rust, and *unsafe* Rust. If we stay within the scope of safe Rust, we are guaranteed to have memory safety. However, Rust's memory safety guarantees can be too restrictive for some applications, so Rust provides an "escape hatch" in the form of unsafe Rust, which allows writing code that bypasses some memory safety checks. The added flexibility comes at the cost of potential memory safety-related bugs.

This is particularly relevant when converting C to Rust, because unsafe Rust provides support for operations involving *pointers*. Pointers are a core feature of the C language, and are used extensively in low level software like kernels and drivers. When this software is converted to Rust, the resulting code will likely require the use of unsafe Rust. Memory safety bugs from the original C code can spill over into the new unsafe Rust code.

Checking the safety of unsafe Rust is an active area of research [4, 18, 19], with multiple reported vulnerabilities [28, 33], and automated tools that perform static [3, 6, 7, 17], and dynamic [21, 29] analysis to check for vulnerabilities.

3 Proposed Approaches

3.1 CARGO - Monolith to Microservice Refactoring [22]

3.1.1 Motivation. Prior *static* analysis approaches for refactoring used context-insensitive static analysis, which is incomplete and imprecise. Further, these techniques use off-the-shelf static analysis

[1]https://aws.amazon.com/microservice-extractor/

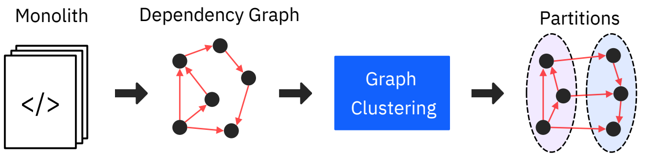

Figure 1: A high-level diagram showing the components of CARGO

tools such as SOOT [16] and WALA [2] which are known to ignore crucial features used by enterprise java applications such as dependency injection [11]. *Dynamic* approaches, on the other hand, need to be run with several inputs to reach all parts of the program. This leads to an incomplete representation of the program. Further, all existing approaches ignore the interconnections between applications and databases. If these interactions are not taken into account, the partitioning guidelines could force the adoption of distributed database transactions, which are notoriously difficult to implement and maintain.

3.1.2 Proposed Approach. To address the above challenges, we propose a microservice partitioning and refinement tool that statically analyzes JEE applications to build a comprehensive, highly-precise, context-sensitive system dependency graph (SDG). Our SDG is a rich graphical abstraction of a Java EE application with various crucial relationships that exist in the application, namely: call-return edges, data-flow edges, heap-dependency edges, and database transaction edges. Our work also proposes a novel community detection algorithm called **C**ontext-sensitive l**A**bel p**R**opa**G**ati**O**n [22] (abbreviated as CARGO, serving as an acronym for our general tool) that isolates "snapshots" of the system dependency graph under various contexts to discover and/or to refine existing partitions into *highly-cohesive* and *loosely-coupled* compositions of the program which can be implemented as microservices. This is shown in Figure 1.

3.1.3 Initial Results. We evaluate CARGO on 5 monolithic applications (4 open source and 1 proprietary) a suite of architectural metrics. Further, using CARGO's partitions, we modernize a benchmark monolithic application (daytrader) to evaluate its real-world performance in terms of latency and throughput. Our key findings are listed below:

- CARGO is able to almost eliminate distributed database transactions, both when applied in conjunction with a baseline approach and when applied stand-alone.
- We use CARGO to refine the partitions produced by a baseline approach, and find that the resulting microservice application has 11% less latency and 120% more throughput than the baseline.
- When we compare CARGO to baseline partitioning approaches on 4 architectural metrics, CARGO is better than the baselines on 3 metrics and inferior on 1 metric.

3.1.4 Future Plan. There are two main directions for future work:
- *Textual Semantics:* Although CARGO learns program semantics from the structure of the program graph, it ignores the semantics in

[2]https://github.com/wala/WALA

the program text, like identifier names, etc. This information can be harnessed using transformer models to create embeddings, and use these appropriately while partitioning the program graph.
• *Automating Rewriting:* CARGO merely suggests partitions; it does not partition the program. We propose to use Large Language Models (LLMs) to do this rewriting automatically. This would entail multiple iterations of code generation, feedback, and bug fixing.

3.2 SpecTra - Code Translation [24]

3.2.1 Motivation. Transpilers generate code that is usually correct by construction. However, this code is often non-idiomatic, *i.e.,* very different from typical human-written code. For instance, it may have unusual control flow, use uninformative identifier names, call foreign functions from the original source language, and often contain same problems of the original legacy code. This can be hard to read and maintain, and often defeats the purpose of translating the code in the first place. On the other hand, LLMs generate idiomatic translations that are more readable, but offer fewer guarantees of correctness.

3.2.2 Proposed Approach. We propose SpecTra [24], a system to enhance the code translation capabilities of LLMs by integrating the principles of transpilers into the LLM framework. We use a novel self-consistency filter to generate specifications of the source code, in three modalities - static, dynamic and natural language. We expose these specifications within the prompt provided to the LLM while generating a translation. Thus, we guide the LLM to adhere to the original functional requirements, thus aiming to combine the readability and idiomatic quality of LLM-generated code with the correctness traditionally associated with transpilers. This method enhances the accuracy and utility of LLMs in code translation tasks, providing a robust solution that balances readability and correctness.

3.2.3 Initial Results. We evaluate SpecTra on three code translation tasks - converting **C** to **Rust**, **C** to **Go**, and **JavaScript** to **TypeScript**. We find that SpecTra is able to enhance the performance of 6 popular LLMs on these tasks by up to **26%** (relative) compared to a uni-modal baseline. This is shown in Table 1.

3.2.4 Future Plan. SpecTra considers only standalone programs that can entirely fit within the context window of an LLM. Real-world projects, on the other hand, are far more complex, involving multiple inter-dependent files with hundreds of lines of code each. We are currently working on a modular function-by-function approach to tackle this problem.

3.3 Yuga - Detecting Memory Safety Bugs in Unsafe Rust [23]

3.3.1 Motivation. Rust operates on a paradigm of ownership and borrows to ensure memory safety. The Rust compiler, specifically its *Borrow Checker*, ensures the validity of all borrows by tracking variable lifetimes. Often the checker can infer the lifetime of a variable without assistance. However, for certain cases related to inter-procedural reasoning, the compiler requires explicit *lifetime annotations* on function signatures to guide the Borrow Checker. For example, the type `&'a i32` denotes a borrow to an integer value, where the lifetime of the borrow is parameterized by the lifetime annotation `'a`. These annotations on the function signature provide valuable information to the compiler, enabling it to verify certain

Table 1: Comparing SpecTra to the baseline for C to Rust translations. The left half of the table shows the absolute number of correct translations out of 300 problems. The absolute number for SpecTra is highlighted in orange . Cells highlighted in blue show higher percentage of correct translations compared to the baseline, and cells highlighted in pink show a lower percentage. We see that SpecTra outperforms the baseline in most cases.

		C TO RUST					
		No. correct			improvement %		
		pass@1	pass@2	pass@3	pass@1	pass@2	pass@3
GPT4o	BASELINE	159	173	183	·	·	·
	SPECTRA	170	203	206	7%	17%	13%
Claude	BASELINE	141	151	165	·	·	·
	SPECTRA	142	167	177	1%	11%	7%
GPT3.5	BASELINE	103	120	129	·	·	·
	SPECTRA	102	134	143	-1%	12%	11%
Gemini	BASELINE	43	53	57	·	·	·
	SPECTRA	53	64	68	23%	21%	19%
Granite	BASELINE	54	68	78	·	·	·
	SPECTRA	50	71	80	-7%	4%	3%
Deepseek	BASELINE	37	53	62	·	·	·
	SPECTRA	42	65	73	14%	23%	18%

memory safety properties without even looking at the function body. For more background on Rust lifetimes, please refer to our paper [23].

One of the causes of vulnerabilities in unsafe Rust is incorrect lifetime annotations on function signatures. Deciding the correct lifetime annotations for a variable is non-trivial—it requires expert knowledge of how the value will propagate to subsequent borrows and pointers, often through deeply nested structures and function calls. Further, lifetimes in Rust are a non-intuitive concept even for experienced programmers, as there is no analogous concept in any other popular programming language (to the best of our knowledge).

Although there are multiple existing static and dynamic approaches to detect memory safety bugs in Rust, none of them can detect bugs arising due to incorrect lifetime annotations. This is because such bugs are rare and typically only surface during execution in specific edge cases. Thus, detecting them through dynamic analysis is non-trivial. Additionally, they tend to be deeply embedded within a program, and analyzing them requires accurate modeling of program memory. Thus, it becomes challenging to study using standard static analyzers or symbolic execution tools.

3.3.2 Proposed Approach. We have developed and implemented a novel static analysis tool called Yuga, capable of detecting incorrect lifetime annotation bugs. We employ a multi-phase analysis approach that starts with a pattern-matching algorithm to quickly identify the potential buggy components. Subsequently, we perform

Table 2: Results of running Yuga on real-world Rust projects

Category	Sub-category	#
Exploitable bugs		**3**
Code smells	Freed ptr but no deref	14
	User-implemented ref counting	30
	Memory copy, not alias	21
	Different field of struct	16
	Two lifetime annotations	1
Total		**85**

a flow and field-sensitive alias analysis, focusing only on the potential buggy components to confirm the bugs. By adopting this multi-phase analysis, we effectively address the well-known scalability challenges associated with alias analysis, while still achieving reasonably high precision.

3.3.3 Initial Results. We evaluated Yuga using a dataset of known security vulnerabilities, and a synthesized vulnerability dataset. We found that Yuga can detect lifetime annotation bugs with 87.5% precision on this dataset of known vulnerabilities. None of the existing static analysis tools we evaluated could detect *any* of these vulnerabilities.

We also ran Yuga on 372 Rust libraries, and Yuga generated 85 reports. Of these, 3 were confirmed to be exploitable vulnerabilities, and the rest were "code smells" that are not exploitable but indicate potential for a vulnerability to arise in future development. These results are summarized in Table 2.

3.3.4 Future Plan. Yuga is based on static analysis, and thus, suffers from lack of precision and completeness. This is due to the inherent limitations of static analysis. Further, Yuga is a purely intraprocedural approach and has to make guesses or overapproximations about the behavior of function calls. Several of our false negatives or false positives are due to incomplete inter-procedural knowledge. Incorporating interprocedural knowledge could work in theory, but would make the tool challenging to scale to real-world repositories.

Our proposed solution to the above problem is to use large language models to reason about the inter-procedural properties of code. When our analysis encounters a function call, it can use an LLM to derive properties of the called function that it can use in its analysis.

References

[1] [n. d.]. Upgrading GitHub from Rails 3.2 to 5.2. https://github.blog/2018-09-28-upgrading-github-from-rails-3-2-to-5-2/. Accessed: 2022-05-06.

[2] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2022. Summarize and generate to back-translate: Unsupervised translation of programming languages. *arXiv preprint arXiv:2205.11116* (2022).

[3] Yechan Bae, Youngsuk Kim, Ammar Askar, Jungwon Lim, and Taesoo Kim. 2021. RUDRA: finding memory safety bugs in Rust at the ecosystem scale. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 84–99.

[4] Marek Baranowski, Shaobo He, and Zvonimir Rakamarić. 2018. Verifying Rust programs with SMACK. In *International Symposium on Automated Technology for Verification and Analysis*. Springer, 528–535.

[5] Caleb Chamberlain. 2023. Southwest Airlines software meltdown a warning for manufacturers. *The Fabricator* (March 2023). https://www.thefabricator.com/thefabricator/article/shopmanagement/southwest-airlines-software-meltdown-a-warning-for-manufacturers

[6] Clippy Contributors. 2022. Rust-clippy. https://github.com/rust-lang/rust-clippy.

[7] Mohan Cui, Chengjun Chen, Hui Xu, and Yangfan Zhou. 2021. SafeDrop: Detecting memory deallocation bugs of rust programs via static data-flow analysis. *arXiv preprint arXiv:2103.15420* (2021).

[8] Utkarsh Desai, Sambaran Bandyopadhyay, and Srikanth Tamilselvam. 2021. Graph Neural Network to Dilute Outliers for Refactoring Monolith Application. In *Proceedings of Association for the Advancement of Artificial Intelligence*. AAAI, virtual, 72–80.

[9] Stuart I Feldman. 1990. A Fortran to C converter. In *ACM SIGPLAN Fortran Forum*, Vol. 9. ACM New York, NY, USA, 21–22.

[10] Martin Fowler. 2004. Strangler Fig Application. https://martinfowler.com/bliki/StranglerFigApplication.html

[11] Martin Fowler. 2006. Inversion of Control Containers and Dependency Injection pattern. *http://www. martinfowler. com/articles/injection. html* (2006).

[12] Galois. 2018. C2Rust. https://galois.com/blog/2018/08/c2rust/

[13] Google. 2021. Rust in the Android platform. https://security.googleblog.com/2021/04/rust-in-android-platform.html

[14] W. Jin, T. Liu, Y. Cai, R. Kazman, R. Mo, and Q. Zheng. 2019. Service Candidate Identification from Monolithic Systems based on Execution Traces. *IEEE Transactions on Software Engineering* 47, 5 (Apr 2019), 1–21.

[15] Anup K Kalia, Jin Xiao, Rahul Krishna, Saurabh Sinha, Maja Vukovic, and Debasish Banerjee. 2021. Mono2Micro: a practical and effective tool for decomposing monolithic Java applications to microservices. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1214–1224.

[16] Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. 2011. The Soot framework for Java program analysis: a retrospective. In *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, Vol. 15.

[17] Zhuohua Li, Jincheng Wang, Mingshen Sun, and John CS Lui. 2021. MirChecker: detecting bugs in Rust programs via static analysis. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 2183–2196.

[18] Yusuke Matsushita, Xavier Denis, Jacques-Henri Jourdan, and Derek Dreyer. 2022. RustHornBelt: a semantic foundation for functional verification of Rust programs with unsafe code. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 841–856.

[19] Yusuke Matsushita, Takeshi Tsukada, and Naoki Kobayashi. 2021. RustHorn: CHC-based verification for Rust programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 43, 4 (2021), 1–54.

[20] Christopher Mims. 2024. The Invisible $1.52 Trillion Problem: Clunky Old Software. *The Wall Street Journal* (March 2024). https://www.wsj.com/tech/personal-tech/the-invisible-1-52-trillion-problem-clunky-old-software-f5cbba27

[21] Miri contributors. 2022. Miri. https://github.com/rust-lang/miri.

[22] Vikram Nitin, Shubhi Asthana, Baishakhi Ray, and Rahul Krishna. 2022. Cargo: Ai-guided dependency analysis for migrating monolithic applications to microservices architecture. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–12.

[23] Vikram Nitin, Anne Mulhern, Sanjay Arora, and Baishakhi Ray. 2023. Yuga: Automatically Detecting Lifetime Annotation Bugs in the Rust Language. *arXiv preprint arXiv:2310.08507* (2023).

[24] Vikram Nitin and Baishakhi Ray. 2024. SpecTra: Enhancing the Code Translation Ability of Language Models by Generating Multi-Modal Specifications. *arXiv preprint arXiv:2405.18574* (2024).

[25] Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pouguem Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbarvand. 2024. Lost in translation: A study of bugs introduced by large language models while translating code. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.

[26] Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised translation of programming languages. *Advances in neural information processing systems* 33 (2020), 20601–20611.

[27] Baptiste Roziere, Jie M Zhang, Francois Charton, Mark Harman, Gabriel Synnaeve, and Guillaume Lample. 2021. Leveraging automated unit tests for unsupervised code translation. *arXiv preprint arXiv:2110.06773* (2021).

[28] Rust developers. 2022. Rust Security Advisory Database. https://rustsec.org/advisories/

[29] Rust team. 2022. cargo fuzz. https://github.com/rust-fuzz/cargo-fuzz.

[30] Johannes Thönes. 2015. Microservices. *IEEE software* 32, 1 (2015), 116–116.

[31] Steven Vaughan-Nichols. 2022. Linus Torvalds: Rust will go into Linux 6.1. https://www.zdnet.com/article/linus-torvalds-rust-will-go-into-linux-6-1/

[32] Dan Wallach. 2024. Translating All C to Rust (TRACTOR). *DARPA* (July 2024). https://www.darpa.mil/program/translating-all-c-to-rust

[33] Hui Xu, Zhuangbin Chen, Mingshen Sun, and Yangfan Zhou. 2020. Memory-safety challenge considered solved? an empirical study with all rust cves. *arXiv preprint arXiv:2003.03296* (2020).

[34] Zhen Yang, Fang Liu, Zhongxing Yu, Jacky Wai Keung, Jia Li, Shuo Liu, Yifan Hong, Xiaoxue Ma, Zhi Jin, and Ge Li. 2024. Exploring and Unleashing the Power of Large Language Models in Automated Code Translation. *arXiv preprint arXiv:2404.14646* (2024).