AFRL-RI-RS-TR-2023-042

MNEMOSYNE SOFTWARE DEVELOPMENT ASSISTANT

GRAMMATECH INC.

MARCH 2023

FINAL TECHNICAL REPORT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report was cleared for public release by the 88th ABW, Wright-Patterson AFB Public Affairs Office and is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2023-042 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE CHIEF ENGINEER:

/ S /
STEVEN L. DRAGER
Work Unit Manager

/ S /
GREGORY J. HADYNSKI
Assistant Technical Advisor
Computing & Communications Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings

REPORT DOCUMENTATION PAGE

PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ORGANIZATION.

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED	
		START DATE	END DATE
MARCH 2023	FINAL TECHNICAL REPORT	JUNE 2020	SEPTEMBER 2021

4. TITLE AND SUBTITLE

MNEMOSYNE SOFTWARE DEVELOPMENT ASSISTANT

5a. CONTRACT NUMBER	5b. GRANT NUMBER	5c. PROGRAM ELEMENT NUMBER
FA8750-20-C-0208	N/A	62303E

5d. PROJECT NUMBER	5e. TASK NUMBER	5f. WORK UNIT NUMBER
		R2Z7

6. AUTHOR(S)

Eric Schulte; Swarat Chaudhuri; Isil Dillig; Armando Solar-Lezama

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

GrammaTech Inc.
531 Esty Street
Ithaca NY 14850

8.PERFORMING ORGANIZATIONREPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

DARPA
675 N. Randolph St
Arlington VA 22203-2114

Air Force Research Laboratory/RITA
525 Brooks Road
Rome NY 13441-4505

10.SPONSOR/MONITOR'S ACRONYM(S)

AFRL/RI

11.SPONSOR/MONITOR'S REPORT NUMBER(S)

AFRL-RI-RS-TR-2023-042

12. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for Public Release; Distribution Unlimited. PA# AFRL-2023-1055
Date Cleared: 01 MARCH 2023

13. **SUPPLEMENTARY NOTES**

14. ABSTRACT

Under DARPA's Intent Defined Adaptive Software (IDAS) program, GrammaTech, with subcontractors UT Austin and MIT, developed Mnemosyne. Mnemosyne is an automated software development assistant which integrates with a software developers Integrated Development Environment (IDE) and works with them suggesting code, types, and tests as they type; and continually checks the consistency of documentation, types, tests, and implementation. Mnemosyne applies statistical machine learning, formal methods, and Search Based Software Engineering (SBSE) to enable developers to write higher quality, higher assurance, and more standards compliant code more quickly. Mnemosyne transcends Continuous Integration/Continuous Development (CI/CD) by making live substantive contributions to the software development process as the developer works. Mnemosyne integrates seamlessly with most popular text editors and IDEs. Mnemosyne is extensible, easily collaborating with varied program analysis, typing, testing, repair, and synthesis modules.

15. SUBJECT TERMS

Machine Programming, Program Synthesis, Automated Test Generation, Program Analysis, Software Development Assistant, Integrated Development Environment, Formal Methods, Machine Learning

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES
a. REPORT	b. ABSTRACT	C. THIS PAGE		
U	U	U	SAR	31

19a. NAME OF RESPONSIBLE PERSON	19b. PHONE NUMBER *(Include area code)*
STEVEN L. DRAGER	N/A

STANDARD FORM 298 (REV. 5/2020)

PREVIOUS EDITION IS OBSOLETE.

Prescribed by ANSI Std. Z39.18

Table of Contents

List of Figures

1 Summary

Under the Defense Advanced Research Projects Agency's (DARPA) Intent Defined Adaptive Software (IDAS) program, GrammaTech, in partnership with professors Işıl Dillig at the University of Texas at Austin (UT Austin), Swarat Chaudhuri at UT Austin, and Armando Solar-Lezama at Massachusetts Institute of Technology (MIT) have advanced both the academic front of program synthesis research and the practical front of automating software development practice.

On the research front, the team's academic members continue to lead the investigation into the use of combined formal and machine learning (ML) approaches to program synthesis. The neurosymbolic techniques being developed under this IDAS effort promise to scale formal program synthesis to real-world programming languages (PL), while retaining the formal guarantees of correctness and security lost by pure ML techniques.

On the practical front, the team has collaborated to field Mnemosyne,[1] an open-source system that brings software development automation research to bear in the traditional Integrated Development Environment (IDE). Mnemosyne automates the generation of tests, types, and code, helping developers to both express their intent and to then generate correct, secure, and assured implementations satisfying this intent. Mnemosyne integrates seamlessly into the modern IDE, providing suggestions to developers as they work, requiring no new or specialized knowledge on the part of the user. Mnemosyne automates the tricky parts of bringing PL research to practice, such as abstracting tests, types, and full programming languages used in industry into the simplified forms preferred by researchers, translating research results back to industry-standard formats, and mediating developer interaction. Mnemosyne also serves as a complement to other emerging ML driven software development automation such as GitHub's Copilot[2] and tabnine[3]. While these commercial tools focus on developer speed at the cost of correctness and security, Mnemosyne generates tests and types enforcing correctness and security providing guardrails for ML-accelerated software development.

In the short term, the Mnemosyne system is already providing utility to developers. The team is dogfooding Mnemosyne internally at GrammaTech and are in discussion with the LevelUp[4] team at the Air Force to field a pilot of Mnemosyne in their PlatformOne[5] environment to assist Department of Defense (DoD) software developers.

---

[1] https://grammatech.gitlab.io/Mnemosyne/docs/

[2] https://copilot.github.com

[3] https://www.tabnine.com

[4] https://software.af.mil/softwarefactory/platform-one-by-levelup/

[5] https://software.af.mil/dsop/services/

In the longer term, as the development assistants integrated under Mnemosyne collaborate to increase the effective skill of ML-driven software creation, and continually expand the pool of developers able to develop correct and secure software, it is hoped to cross a threshold where large software development steps can be automated with light supervision by a human software developer.

2 Introduction

Under DARPA's Intent Defined Adaptive Software program, GrammaTech in partnership with professors Işıl Dillig and Swarat Chaudhuri at the University of Texas at Austin, and Armando Solar-Lezama at the Massachusetts Institute of Technology have advanced both the academic front of program synthesis research and the practical front of automating software development practice.

The main artifact produced under this effort is the Mnemosyne automated software development assistant.[6] The goal of Mnemosyne is to bring modern automated strategies for software testing, type checking, and implementation to bear in the traditional software development environment. Mnemosyne integrates a developer's integrated development environment with multiple program analysis and program synthesis modules, called "Muses." (In Greek mythology Mnemosyne is the titan goddess of memory, inventor of languages, and mother of the Muses.) Mnemosyne facilitates the integration of Muses into the real-world software development in a number of ways:

1. Mnemosyne communicates with IDEs through the standard Language Server Protocol (LSP).[7] This ensures that the most popular IDEs are all able to communicate with Mnemosyne. Mnemosyne extends LSP with additional representations for "tests" (broadly construed as any *existential* requirement on software), "types" (broadly construed as any *universal* requirement on software) and "prose" (natural language specification and documentation). This LSP superset is called "Argot."

2. Mnemosyne provides "view Muses" to translate between the familiar representations of tests and types commonly used by software developers and abstracted forms provided by Argot. Each view Muse understands how to read and write a specific representation. For example, a "GoogleTest" view Muse is able to both read tests out of the popular C++ GoogleTest framework[8] for presentation to other Muses in an abstract form and is also able to take tests generated by Muses such as property based or fuzz testers and insert them back into a software project as tests in the GoogleTest framework.

3. Mnemosyne provides bidirectional functional lenses which translate between full-fledged real-world programming languages like C++ and Python and the simplified functional programming languages targeted by most program synthesis tools. This allows research tools to be applied to real-world software.

---

[6] https://grammatech.gitlab.io/Mnemosyne/docs/

[7] https://en.wikipedia.org/wiki/Language_Server_Protocol

[8] https://github.com/google/googletest

On the research front, the team's academic members continue to lead the investigation into the use of combined formal and machine learning approaches to program synthesis. The neurosymbolic techniques being developed under this IDAS effort promise to scale program synthesis to real-world programming languages while retaining the formal guarantees of correctness and security lost by pure ML techniques. By incorporating these emerging neurosymbolic techniques into Mnemosyne as Muses, it will provide developers with automatically synthesized implementations, which in some cases will come with formal proofs of correctness.

By synthesizing expressions of intent in terms of tests, types, and prose, Mnemosyne helps developers to more thoroughly and accurately both express and understand the implications of their intent. The resulting synthesized artifacts are useful both for subsequent code synthesis and for certification and software assurance. As the quality of automated intent expression improves and the scalability of program synthesis increases the hope is to cross a threshold beyond which collaborating Muses will automate significant portions of a typical software development workload.

By focusing on practical utility, interactivity, and transparency, Mnemosyne promises to bring together the software development community and the research community in a partnership with vision of incremental acceleration of software development in the short term, and radical automation of software development in the long term.

3 Methods, Assumptions, and Procedures

3.1 Methods

3.1.1 Mnemosyne architecture

Mnemosyne follows a hub-and-spokes architecture, as shown in Figure 1. The central hub is an LSP server (called the "*Argot Server*") which integrates a number of synthesis modules (called "*Muses*") and the developer's IDE.

The Muses communicate using Argot, an extension to LSP,[9] the Language Server Protocol (including extensions from the Build Server Protocol (BSP),[10]). Argot is documented in section 3.1.2.

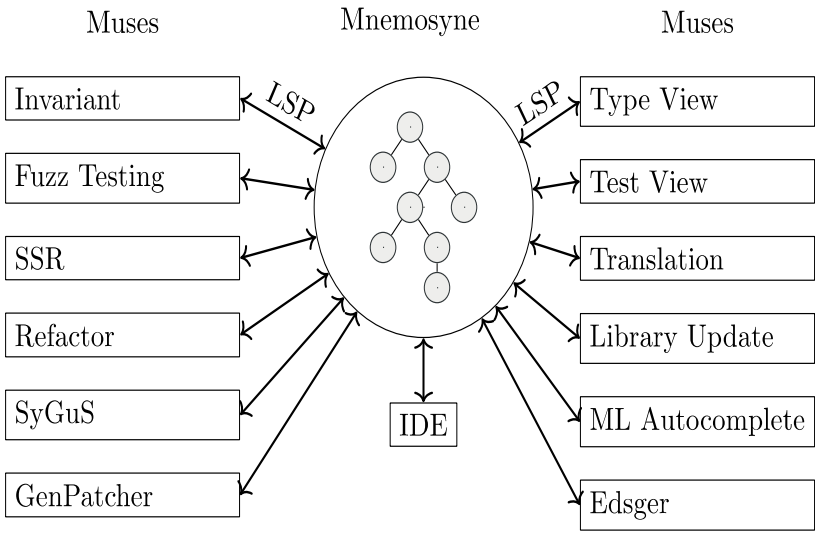

Figure 1: Mnemosyne high-level architecture.

3.1.2 Argot

As shown in Figure 2, Argot LSP supports explicitly represented (i) existential quantifiers such as examples and *tests*, (ii) universal quantifiers such as contracts and *types*, and (iii) natural language documentation and *prose*. These new *kinds* accompany subtrees of a software program.

Argot provides a medium of exchange between tools for program synthesis and analysis (termed "Muses") and the developer's IDE.

---

[9] https://en.wikipedia.org/wiki/Language_Server_Protocol

[10] https://build-server-protocol.github.io/docs/specification.html

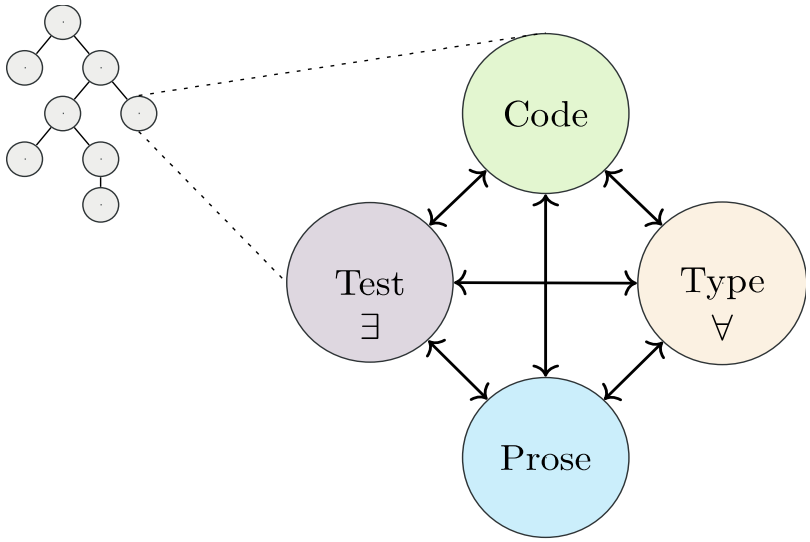

Figure 2: Argot extensions to LSP to represent tests, types, and prose.

3.1.3 View Muses

Views are responsible for non-synthesis management of software information, including collecting and writing code, types, tests, and prose from and to the software project. They provide baseline capabilities for the LSP superset for use by the IDE and downstream Muses. For example, a test View will register with Argot Server and provide endpoints to look up the available test cases and write new ones. When a downstream Muse generates new test cases, it may use this View Muse's registered test writing capabilities to add them to the project in a general way. In this way multiple downstream Muses can leverage the test wrangling functionality provided by a View Muse. This approach has three main advantages:

1. Synthesis Muses do not need to reimplement reading/writing operations.
2. The synthesis Muses (e.g., test or type generators) are not tied to the chosen testing framework; only the *View* Muse is. This allows the synthesis effort to work independent of the testing framework chosen for the project, relying on the View to handle encoding the results.
3. New language frameworks (e.g., alternative type checkers) can be easily added as secondary View Muses and used interchangeably.

3.1.4 Source Code Representation and Manipulation

The Argot Server at the hub of Mnemosyne, as well as many of the Muses (including all view Muses) are built on GrammaTech's Software Evolution Library (SEL)[11] — a general-purpose library for programmatically parsing, modifying, and evaluating software source code.

[11] https://github.com/gramamtech/sel

SEL builds on GitHub's tree-sitter[12] generated parsers to parse dozens of programming languages into parse trees, as shown in Figure 3. These parse trees are constructed using GrammaTech's functional-trees[13] applicative data structure, providing cheap copy and undo operations, efficient storage of large populations of program variants, and fast updates. Common Lisp's Object System (CLOS[14]) and generic function feature[15] power a programming model providing uniform, ergonomic traversal, analysis, and rewriting of program source code across programming languages. SEL's program rewriting produces developer-quality diffs which only change the mutated portion of the program text, and handle formatting, including indentation and the minutiae of `;`s and `,`s, for the developer.

SEL Application Programming Interfaces (APIs) are available in:

- *Common Lisp* documented in the SEL manual,[16], and
- *Python* available in the `asts` package at PyPi.org.[17]

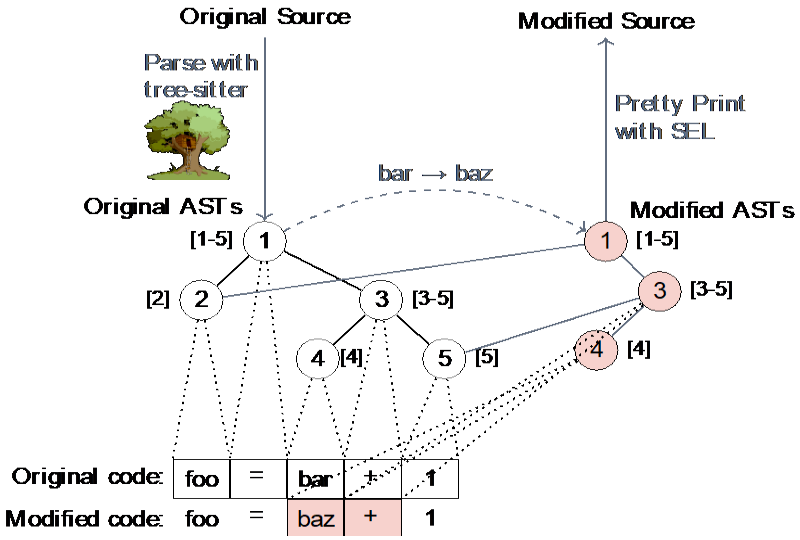

Figure 3: Modification of a functional tree and associated code.

[12] https://tree-sitter.github.io/tree-sitter/

[13] https://github.com/grammatech/functional-trees

[14] https://lispcookbook.github.io/cl-cookbook/clos.html

[15] https://gigamonkeys.com/book/object-reorientation-generic-functions.html

[16] https://grammatech.github.io/sel

[17] https://pypi.org/project/asts/

3.1.5 Bidirectional Lenses

Bidirectional translations allow a concrete value to be translated into an abstract value, manipulated in its abstract form, and then translated back into a concrete form without losing information.

The team has implemented general-purpose support for bidirectional lenses in the Fresnel library[18], as shown in Figure 4. This library provides support for bidirectional lenses which are specifically:

Discerning Lenses
Capable of passing opaque items from the concrete form through the abstract representation, and

Quotient Lenses
Capable of ignoring irrelevant details in the concrete form.

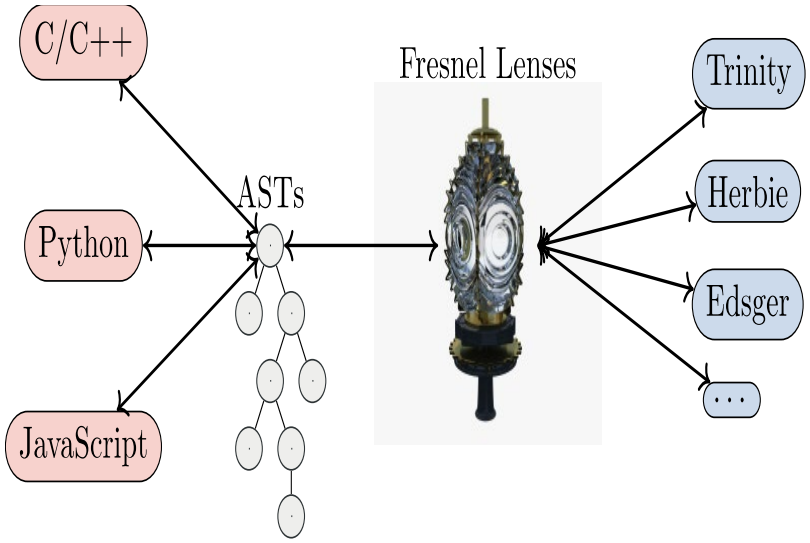

Figure 4: Bidirectional lenses translate between real and simplified programming languages.

3.2 Assumptions

The goal of Mnemosyne is to automate software development, meaning the writing of *code*, *tests*, and *types*. *Tests* are broadly interpreted as any *existential* program requirement and *types* broadly as any *universal* program requirement — including simple and refined types, assertions, and logical constraints. As the lion's share of the work on a typical project is *adaptation* to new requirements and bug-reports, this is also the focus of Mnemosyne. The following themes have guided development:

[18] https://gitlab.com/GrammaTech/Mnemosyne/fresnel

Familiar.
Existing software development tools including IDEs, programming languages and compilers, test infrastructure, static analyzers, linters, and documentation generators representing decades of advancement in the practice of software engineering are to be *extended*, not *replaced*. Mnemosyne works directly with existing languages, tools, and techniques integrating into this ecosystem.

Holistic.
Software developers spend time writing tests, types, and documentation as well as code. For adaptation the descriptions of the intent of the software are often more important than the concrete implementation of the software. Mnemosyne works to synthesize *all* aspects of a software project — not just code.

Interactive and Transparent.
Most existing synthesis techniques tend to exhibit "all-or-nothing" behavior: they either successfully synthesize a program or fail completely. As a result, code synthesis tools often seem opaque and brittle to users. Successful synthesis-aided software development requires transparency, exposing partial results and incremental refinements to the user. This transparency also provides graceful degradation by enabling the user to step in when full automation is not feasible. In particular, the team expects that the user will largely handle higher-level structural decomposition and Mnemosyne will help by propagating tests and types down this decomposition and by synthesizing code towards the program leaves.

3.3 Procedures

The hope is to accomplish these goals — achieving an unprecedented practical application of program synthesis — through the following mechanisms and phenomena.

Hierarchical problem decomposition and incremental refinement.
Mnemosyne will support interactive, incremental synthesis in which the top-level goal is *refined* into increasingly more concrete sub-problems through continual interaction between the software designer and automated synthesis tools. At each step in the refinement process, the user can either (i) refine the problem specification or software design, (ii) prompt Mnemosyne to automatically synthesize code or (iii) infer additional specifications of sub-problems in the form of tests, types, or documentation. This hierarchical decomposition will enable development of large-scale systems by allowing developers to control the problem's decomposition into modules, without requiring the developer to prematurely commit to low-level implementation choices.

Retention of alternate implementations.
While Mnemosyne will be useful for designing and implementing new software, it will particularly shine for easy and efficient software adaptation. Mnemosyne-enabled software development proceeds in rounds of interactive refinement steps involving both the developer and automated tools. Mnemosyne can leverage this rich refinement history when performing adaptation. Specifically, Mnemosyne will maintain both probability distributions over refinements and populations of diverse concrete

implementations. These saved distributions and populations will jump-start subsequent probabilistic and search-based Software Engineering (SBSE) techniques.

Rich multi-modal intent specifications.
Mnemosyne will allow users to communicate their requirements and preferences using multiple artifact modalities, including test, examples, types, logical constraints, high-level code structure, and natural language. Because different modalities are more convenient for expressing different types of intent, users may communicate their intent using the optimal mechanism for the problem at hand. The synthesis and adaptation engines underlying Mnemosyne will leverage these synergistic modalities when performing synthesis.

Mutually reinforcing synthesis modalities.
Mnemosyne's Muses will synthesize mutually reinforcing modalities of program specification and implementation. For example, a type-inference technique will yield richer type information which may in turn enable a test-generation technique to generate more suitable inputs, which may in turn enable a program-synthesis technique to generate code. In this way the products of Muses will enable other Muses in a virtuous cycle.

Combined probabilistic and logical synthesis.
Mnemosyne will leverage a combination of *logical* and *probabilistic* techniques to make correct and efficient code synthesis practical. While logical reasoning is needed to ensure satisfaction of semantic requirements, purely logical reasoning (e.g., based on satisfiability modulo theories (SMT) solvers) is unlikely to scale to real-world software development. Mnemosyne will refine partially realized specifications using code refinements selected from a learned probability distribution of refinements. Besides making synthesis more efficient, probabilistic techniques will also allow Mnemosyne to leverage the latent domain knowledge embedded in existing code by automatically learning prior probability distributions.

Combined probabilistic and SBSE adaptation.
To support efficient adaptation to changing requirements, Mnemosyne will use combined probabilistic and search-based techniques. SBSE techniques have been shown to be applicable to real-world software and are entering real-world use for restricted development tasks. Diversity in software systems is able to *jump-start* SBSE processes (this same dynamic is familiar from biological systems). By pairing probabilistic refinement and diverse populations — re-weighting probability distributions against concrete population members and evolving populations to fit modified distributions — Mnemosyne will remain poised to quickly and effectively adapt existing implementations to new requirements.

4 Results and Discussion

4.1 Research Results

4.1.1 Neural Program Generation Modulo Static Analysis

The research developed a novel framework for Neural Generation Modulo Static Analysis. State-of-the-art neural models of source code tend to be evaluated on the generation of individual expressions and lines of code, and commonly fail on long-horizon tasks such as the generation of entire method bodies. The neurosymbolic method addressed this deficiency using weak supervision from a static program analyzer. Specifically, the method allows a deep generative model to symbolically compute, using calls to a static analysis tool, long-distance semantic relationships in the code that it has already generated. During training, the model observes these relationships and learns to generate programs conditioned on them. The team applied this approach to the problem of generating entire Java methods given the remainder of the class that contains the method. Experiments have shown that the approach substantially outperforms a state-of-the-art transformer and a model that explicitly tries to learn program semantics on this task, both in terms of producing programs free of basic semantic errors and in terms of syntactically matching the ground truth. A paper on this topic has been submitted to the Annual Conference on Neural Information Processing Systems (NeurIPS'21)[1].

4.1.2 Bottom-up Synthesis of Recursive Functional Programs using Angelic Execution

The team developed a novel bottom-up method for the synthesis of functional recursive programs. While bottom-up synthesis techniques can work better than top-down methods in certain settings, there is no prior technique for synthesizing recursive programs from logical specifications in a purely bottom-up fashion. The main challenge is that effective bottom-up methods need to execute sub-expressions of the code being synthesized, but it is impossible to execute a recursive subexpression of a program that has not been fully constructed yet. In this work, the team addressed this challenge using the concept of *angelic semantics*.

Specifically, this method finds a program that satisfies the specification under angelic semantics, analyzes the assumptions made during its angelic execution, uses this analysis to strengthen the specification, and finally reattempts synthesis with the strengthened specification. The proposed angelic synthesis algorithm is based on version space learning and therefore deals effectively with many incremental synthesis calls made during the overall algorithm. The team has implemented this approach in a prototype called Bottom-up Synthesis of Recursive Functional Programs (BURST) and shown that it outperforms prior approaches to synthesis of functional recursive programs. A paper on this topic has been submitted to the Association of Computing Machinery (ACM) Symposium on Principles of Programming Languages (POPL'22)[2].

4.1.3 Program Adaptation via Data Type Refactoring and Refinement

A common type of program adaptation during software evolution is to change the underlying data structures in the program. For example, developers might add new auxiliary data structures to improve performance or change the data layout to improve memory usage. While such changes are quite common, they often require significant changes to the implementation; as a result, they can be time-consuming as well as error prone. In the team's recent work supported by IDAS, two techniques were explored that can be used to automatically adapt programs in the presence of such changes.

Specifically, in one of the research paths, a program adaptation technique was investigated that can be used to automatically add new auxiliary data structures to the program. Specifically given the original program P and an integrity constraint C relating an existing data structure D to a new data structure D', the technique automatically inserts code that correctly updates and maintains D'. Thus, the technique allows programmers to conveniently insert new data structures without having to worry about introducing bugs. The team implemented the technique in a tool called Volt and evaluated it on several real-world program adaptation scenarios found on Github. Evaluation shows that Volt can automate the overwhelming majority of such changes. A paper on this technique has been published in the ACM Special Interest Group on Programming Languages (SIGPLAN) International Conference on Programming Language Design and Implementation (PLDI'21)[3].

In a related research path, another program adaptation technique was investigated for modifying data layout (e.g., how primitives are organized into classes/structs). Specifically, a convenient domain-specific language (DSL) was proposed for specifying such data type refactorings. Given a type-level refactoring in this DSL (e.g., split a struct into two), the method can re-implement the code so that the new program is equivalent to the original version but uses the refactored types specified by the programmer.

4.1.4 Program Modernization for Changing APIs

In a different project, program modernization techniques were investigated that can be used to adapt programs to changing APIs and language features. In the work, this problem was viewed as an instance of transpilation (or source-to-source translation) but leverage several salient features of the problem domain. In particular, it was observed that the modernized version of the program shares many syntactic connections to the original program (particularly at the level of low-level expressions); furthermore, these shared syntactic expressions take the same values in corresponding program executions. Based on these observations, a new practical and general source-to-source translation approach was developed that uses a combination of search, state-of-the-art deep learning models (to guide search) and concolic execution (to prune the search space). This technique has been used to modernize Java applications to use the functional Stream API and make old-fashioned Python code more "Pythonic". This approach has been implemented in a tool called Neural Guided Source to Source Translation (NGST2) and a paper has been submitted to POPL'22[4].

4.2 Muses

An up-to-date list of Muses with demo videos is maintained in the "Muses" page of Mnemosyne's online documentation.[19]

4.2.1 TypeWriter

The TypeWriter[20] technique was integrated for the inference of types in Python. The TypeWriter technique is implemented with two main components, the first leverages statistical ML techniques to identify a distribution of probability types for every variable in the program, and the second unifies these type probability distributions into the most probable typing that successfully type checks. Each of these portions has been implemented independently, enabling easy incorporation of improved statistical ML techniques with the existing type checking.

4.2.2 LambdaNet

LambdaNet[21] ML technique was integrated for the inference of types in JavaScript. LambdaNet was developed by co-PI Isil Dillig's group at UT Austin.

4.2.3 Hypothesis

Type-driven automated test generation was integrated using the popular hypothesis library[22] for Python. This Muse integrates a popular property-based testing library for Python. The Muse automates the application of the library to functions encouraging developers to thoroughly test all of their functions. Better types improve the tests generated by hypothesis, so this Muse is made more effective by the TypeWriter Muse.

4.2.4 ML Auto-complete

The ML-driven Galois Autocompleter[23] was integrated as a Muse. The Galois Autocompleter provides large-scale auto-completion of as much as whole lines of code in a single completion. This Muse is used not only by the developer directly, but may also be leveraged by other Muses seeking high probability code at a given location in a program. For example, the GenPatcher automated program repair Muse is able to invoke the Galois Autocompleter when performing program repair.

---

[19] https://grammatech.gitlab.io/Mnemosyne/docs/muses/

[20] https://arxiv.org/pdf/1912.03768.pdf

[21] https://arxiv.org/abs/2005.02161

[22] https://hypothesis.readthedocs.io/en/latest/

[23] https://github.com/galois-autocompleter/galois-autocompleter

4.2.5 DIG

Mnemosyne integrates with dynamic invariant generation (DIG)[24] to collect likely program invariants from test executions. Mnemosyne makes DIG significantly more useful to developers by automating the two difficult parts of using DIG. First, Mnemosyne automates the collection of traces that drive DIG by instrumenting the program to emit trace data for every variable in scope and by automatically collecting and executing test using Mnemosyne's test view Muses. Second, Mnemosyne takes likely invariants output by DIG and inserts them directly into the software source code as assertions at the location of the invariant.

This invariant assertion exemplifies a number of the themes guiding the approach:

- Translated mathematical expressions output by DIG into program code in the language used by a project which is readily understood by software developers.
- *Transparent* presentation of tool results directly to developers in their IDE providing the ability to edit or modify the tool output. This is in contrast to many synthesis or model-based design tools which take an all-or-nothing approach in which only the final result is presented to the developer as an opaque executable leaving the developer a single accept or reject decision and causing many 90% solutions to have 0% utility.
- *Integration* into the existing workflow, where invariants are not stored separately from a software project but rather are inserted directly into the project source code ensuring they're implicitly run by any downstream process and are accepted and stored as any other software artifact (e.g., by being committed to version control).

4.2.6 Herbie

As shown in Figure 5, Herbie is a tool for fixing floating point errors in source code.[25] Herbie is developed by researchers unaffiliated with this IDAS team. Herbie operates over a simplified representation of software. Using Mnemosyne and the bidirectional lenses this team was able to integrate Herbie into the Mnemosyne system in a matter of days. Using this integration Mnemosyne translates between C/C++ and Python and Herbie's simplified representation. This makes it trivial for developers to invoke Herbie to automatically fix floating point errors in place in their code base. After this integration, it is believed, Mnemosyne is now one of the most practical and easiest ways to use Herbie in application to real software.

[24] https://github.com/unsat/dig

[25] https://herbie.uwplse.org

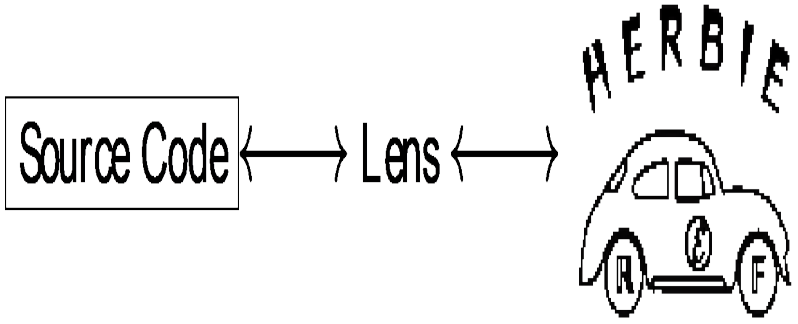

Figure 5: Herbie fixes floating point problems in source code.

4.2.7 Software Search and Replace

Software Search and Replace (SSR) is a software transformation and exploration tool that operates on abstract syntax trees (ASTs), such as those produced by parsers for C, C++, Javascript, and other languages. Rules specify which ASTs to match and what actions to take against matches: for example, users can specify code rewriting actions to implement sophisticated refactoring. Rule application order, including simultaneous application, is fully under user control; multiple SSR passes can be used to iterate to a stable, fully rewritten result.

Developers working in Integrated Development Environments have traditionally used textual search and replace (perhaps augmented by regular expressions) to perform global transformations such as renaming, simple refactoring, or adaptation to changed APIs. Mnemosyne, by integrating SSR, provides a safe, syntax-aware alternative to textual search-and-replace that is usable from within the IDE, without any loss in convenience.

4.2.8 Refactoring

"*Refactoring* is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior."[26] The refactoring Muse implements the refactoring's described in the book *Refactoring: Improving the Design of Existing Code*. These are both useful tools to provide to developers in their own right, and useful building blocks for other Muses. For example, the GenPatcher automated program repair Muse is now able to leverage the refactoring Muse to prepare a program for subsequent repair.

[26] https://refactoring.com

4.2.9 GenPatcher

As shown in Figure 6, GenPatcher uses a suite of automatic program modifications to modify a program so that it passes a provided test suite, which the original program could not pass. GenPatcher uses these automatic modifications to build a population of program variants and an iterative search process mimicking natural selection to improve the population until a version of the program is found which passes all tests in the test suite. Automated program repair performed in this manner has been shown able to fix a wide range of real-world bugs, and is currently used in production by large software companies.

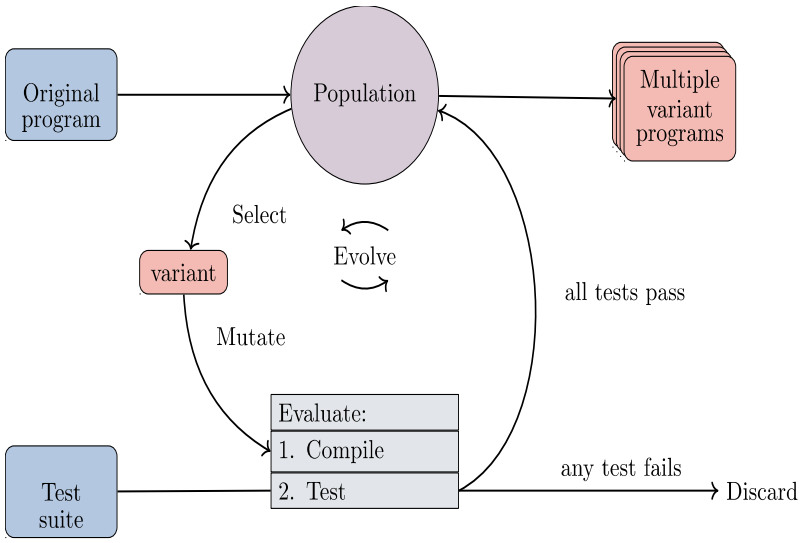

Figure 6: GenPatcher automated program repair.

In the context of Mnemosyne, GenPatcher is augmented with access to multiple other Muses which it is able to invoke in the same way as a developer. In effect, GenPatcher can be thought of as a very junior developer who is limited to copy/paste coding and heavy reliance on a test suite to evaluate code modifications. By improving the quality of software development "guard rail" like tests, types, and invariant assertions; and by providing program synthesis aids, such as Herbie, Edsger, and Trinity, the Mnemosyne environment enhances the effectiveness of both junior developers and automated program repair tools like GenPatcher.

4.2.10 Edsger

Edsger is in development by Swarat Chaudhuri's lab at UT Austin. Edsger is an interactive program synthesis engine which operates in a manner similar to interactive theorem provers, in which the user advances the synthesis by invoking tactics from a set tactic repository. Edsger leverages statistical ML techniques to learn which tactics are appropriate in which settings. This should allow Edsger to increasingly automate the tactic application process over time.

Edsger has illuminated one fruitful opportunity of continued work on Mnemosyne, which is that as the logs saved by the system are collected, they can be used to train ML processes,

which can then improve the degree of automation in the system moving forward. As Mnemosyne learns what developers like and don't like in any given setting, it will increasingly be able to anticipate developers' needs and proxy for developers' inputs to specific Muses.

4.2.11 Trinity

Trinity is a program synthesis tool developer by Işıl Dillig's lab at UT Austin.[27] Like Herbie, Trinity operates over a simplified representation of source code. Trinity is focused on providing program synthesis that is easily *customized* to any given domain, *efficient*, and *usable*. In the Mnemosyne context, bidirectional lenses are leveraged to translate between regions of real-world programming languages and domain-specific Trinity representations for specific synthesis tasks. The existing Trinity integration is focused purely on mathematical code, but the open ended design of both Trinity and Mnemosyne should allow easy extension to other domains as needed.

4.3 Transitions

4.3.1 GrammaTech Dogfooding

The team at GrammaTech is sufficiently convinced of the near-term practical utility of Mnemosyne that they are beginning to dogfood, as shown in Figure 7, the tool internally. GrammaTech's Information Technology (IT) department has configured an internal Kubernetes (K8) environment with sufficient resources to run a large-scale instance of Mnemosyne capable of serving all of GrammaTech's developers. Mnemosyne and the Muses have been refactored to run in individual docker images. These images may be deployed to a single K8 environment.

[27] https://github.com/fredfeng/Trinity

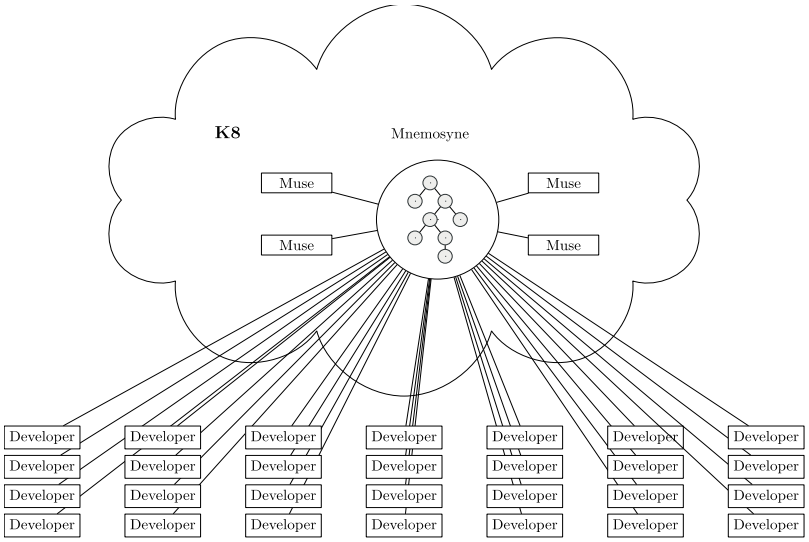

Figure 7: Dog-Fooding.

Through this process it is hoped to quickly identify which portions of the Mnemosyne system are most beneficial, and most distracting, in day-to-day development. This will drive short-term improvements in the practical usability of the system. It is also planned to collect detailed logs of the usage of the system. This will drive longer-term larger scale automation of inter-Muse interactions.

4.3.2 Platform One Outreach

There have been multiple meetings with the Air Force's LevelUp team[28]. LevelUp are the creators of PlatformOne[29], the flagship software factory in the DoD. The team is working with the LevelUp team to identify an appropriate development group in their organization with which to run a pilot of the Mnemosyne system.

[28] https://software.af.mil/softwarefactory/platform-one-by-levelup/

[29] https://software.af.mil/dsop/services/

5 Conclusions

5.1 Mnemosyne and GitHub Copilot

In the last months of this effort, GitHub released their Copilot system[30]. Copilot is an "Artificial Intelligence (AI) pair programmer", which on the surface sounds identical to Mnemosyne. Figure 8 shows Mnemosyne running alongside Github Copilot. The differences lie in *what* (§5.1.1) developer tasks each supports and in *how* (§5.1.2) each works. The top-level takeaway is that given their different focuses Copilot and Mnemosyne can work productively together. Think of Copilot as a very productive but junior coder prone to making correctness[31] and security[32] mistakes, and Mnemosyne as an automated defensive coder following behind adding tests, assertions, and type annotations, and discovering and fixing bugs and stability issues.

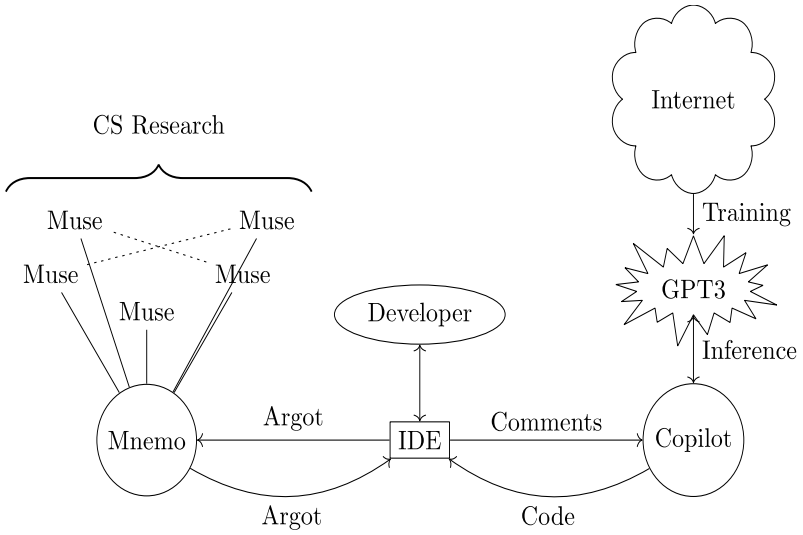

Figure 8: Mnemosyne running along GitHub's Copilot.

[30] https://copilot.github.com

[31] https://www.itprotoday.com/development-techniques-and-management/github-unveils-ai-tool-speed-development-beware-insecure-code

[32] https://portswigger.net/daily-swig/devsecai-github-copilot-prone-to-writing-security-flaws

5.1.1 What Mnemosyne and Copilot do

Copilot

As shown in their very impressive home page demos[33] and YouTube videos[34], GitHub's Copilot and the underlying OpenAI Codex model do a great job of generating large amounts of running code very quickly. These tools go from code *comments* or *documentation* to running code.

Mnemosyne

As shown in the demo video[35], Mnemosyne focuses on adding test, types, and assertions of invariants to existing code. It also updates existing code to fix minor flaws, fix failing tests, and improve other properties such as floating-point stability. Mnemosyne operates by taking and returning Argot[36] (the extension to LSP which communicates code, tests, types, and prose). Note, in some cases Mnemosyne Muses *do* synthesize code, e.g., the Trinity[37] Muse can synthesize mathematical code against provided input/output sets.[38]

A developer has to perform many disparate tasks in their daily work. Producing code is certainly an important one, but writing the structures that document, constrain, test, and defend that code are also essential elements.

5.1.2 How Mnemosyne and Copilot work

Copilot

GitHub Copilot is a front end to OpenAI Codex, which builds on their Generative Pre-trained Transformer third generation (GPT3) model. GPT3 is a massive machine learning model supporting a generic "text in, text out" interface. In the case of Copilot, the text in is code comments and the text out is running code.

Mnemosyne

Mnemosyne is a system that coordinates many developer assistants, rather than a single assistant. The actual assistants currently incorporated into Mnemosyne (i.e., Muses) vary widely in their method of implementation using techniques including formal methods, machine learning, evolutionary computation, and simple mechanical refactoring.

---

[33] https://copilot.github.com

[34] https://www.youtube.com/watch?v=SGUCcjHTmGY

[35] https://grammatech.gitlab.io/Mnemosyne/docs/video/integrated-demo-june.mp4

[36] https://grammatech.gitlab.io/Mnemosyne/docs/architecture/#argot

[37] https://github.com/fredfeng/Trinity

[38] https://grammatech.gitlab.io/Mnemosyne/docs/video/trinity-demo

5.2 Concluding Summary

This report details the research, design decisions, and software development work that have resulted in Mnemosyne — a practical tool for the significant automation of modern software development. There are a wealth of developer tools with significant potential which remain locked up in Academia[39]. Mnemosyne demonstrates the practical utility of these tools by generically addressing the underlying factors that limit their use; factors including providing a developer User Interface (UI) (Mnemosyne extends LSP), dealing with real-world programming languages (Mnemosyne provides bi-directional translation), dealing with real-world test and type representations (Mnemosyne provides type- and test-view Muses), and deployment (Mnemosyne provides containerization and packaging).

In the short term, the Mnemosyne system is already providing utility to developers. The team is dogfooding Mnemosyne internally at GrammaTech and are in discussion with the LevelUp[40] team at the Air Force to field a pilot of Mnemosyne in their PlatformOne[41] environment to assist Department of Defense software developers.

In the longer term, as the development assistants integrated under Mnemosyne collaborate to increase the effective skill of ML-driven software creation, and continually expand the pool of developers able to develop correct and secure software, it is hoped to cross a threshold where large software development steps can be automated with light supervision by a human software developer.

[39] https://www.pathsensitive.com/2021/03/developer-tools-can-be-magic-instead.html

[40] https://software.af.mil/softwarefactory/platform-one-by-levelup/

[41] https://software.af.mil/dsop/services/

6 Recommendations

It is recommended that DARPA continue to fund research integrating test generation, type inference, and formal and neurosymbolic methods of program synthesis into modern development environments. Investment by DARPA is essential to ensure that the coming automation of the software development process results in increasingly *correct* and *secure* software development as required by the DoD, rather than just resulting in the *faster* software development motivated by the market.

7 References

[1] Rohan Mukherjee, Yeming Wen, Dipak Chaudhari, Thomas W. Reps, Swarat Chaudhuri, and Chris Jermaine. Neural Program Generation Modulo Static Analysis. Neural Information Processing Systems (NeurIPS), 2021. (Spotlight paper)

[2] Anders Miltner, Adrian Trejo Nuñez, Ana Brendel, Swarat Chaudhuri, and Isil Dillig. Bottom-Up Synthesis of Recursive Functional Programs using Angelic Execution. Principles of Programming Languages (POPL), 2022. (Distinguished paper)

[3] Shankara Pailoor, Yuepeng Wang, Xinyu Wang, Isil Dillig. Synthesizing Data Structure Refinements from Integrity Constraints. Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI), 2021.

[4] Ben Mariano, Yanju Chen, Yu Feng, Greg Durrett, Isil Dillig. Automated Transpilation of Imperative to Functional Code using Neural-Guided Program Synthesis. Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA), 2022.

List of Symbols, Abbreviations and Acronyms

ACM	Association of Computing Machinery
AI	Artificial Intelligence
API	Application Programming Interface
AST	Abstract Syntax Tree
BSP	Build Server Protocol
BURST	Bottom-up Synthesis of Recursive Functional Programs
CLOS	Common LISP's Object System
DARPA	Defense Advanced Research Projects Agency
DIG	Dynamic Invariant Generation
DOD	Department of Defense
DSL	Domain Specific Language
GPT3	Generative Pre-trained Transformer third generation
IDAS	Intent Defined Adaptive Software
IDE	Integrated Development Environment
IT	Information Technology
K8	Kubernetes
LSP	Language Server Protocol
MIT	Massachusetts Institute of Technology
ML	Machine Learning
NeurIPS	Neural Information Processing Systems
NGST2	Neural Guided Source to Source Translation
PL	Programming Languages
PLDI	Programming Language Design and Implementation
POPL	Principles of Programming Languages
SBSE	Search-based Software Engineering
SEL	Software Evolution Library
SIGPLAN	Special Interest Group on Programming Languages

SMT	Satisfiability Modulo Theories
SSR	Software Search and Replace
UI	User Interface
UT Austin	University of Texas at Austin