A Toolchain for AI-Assisted Code Specification, Synthesis and Verification

Shaowei Lin[1], Evan Miyazono[2], Daniel Windham[2]

Executive Summary

Critical infrastructure is vulnerable to both malicious attacks and unintended failures, and these risks are expected to grow in the foreseeable future. Currently, deploying formal verification (FV) across critical cyber physical systems would dramatically improve safety and security, but has historically been too expensive and labor-intensive to use outside the smallest, most critical subsystems. Machine Learning tools that generate code and proofs from specifications could allow widespread use of FV within years, not decades, converting AI from a source of growing cyber risks into a technology with important defensive potential. To address these challenges, this document outlines a strategy for leveraging Machine Learning (ML) to scale FV, thereby enhancing the security and updating capabilities of legacy software systems.

Formally verifying software involves generating (a) the software itself, (b) a specification of how the software should behave, and (c) a mathematical proof that the software satisfies the specification. We could have ML tools that first help users generate specifications, and then automate most or all of the creation of the software and the proofs. This is clearly superior to current workflows, where ML systems are generating software that receives cursory reviews from human users, and has been shown to be less secure. Furthermore, as language models generate increasingly large and complicated libraries, we should expect that reviewing the specification of those libraries will be simpler than directly reviewing the code itself; as a result, tools for specifications will dramatically increase the accessibility of secure software generation.

Instead of attempting to fully automate the formal verification process, the toolchain envisioned here aims to augment existing formal verification workflows. This will involve incorporating human feedback in refining specifications, selecting implementations, and aiding in proofs. Focusing on addressing pain-points in existing workflows will ensure that the tools provide value prior to solving the biggest unsolved problems in fully-automated formalization.

This toolchain describes tools to integrate seamlessly into existing critical engineering workflows, which we have named *Formalize* (verifying existing unverified and partially verified

[1] Topos Institute. shaowei@topos.institute

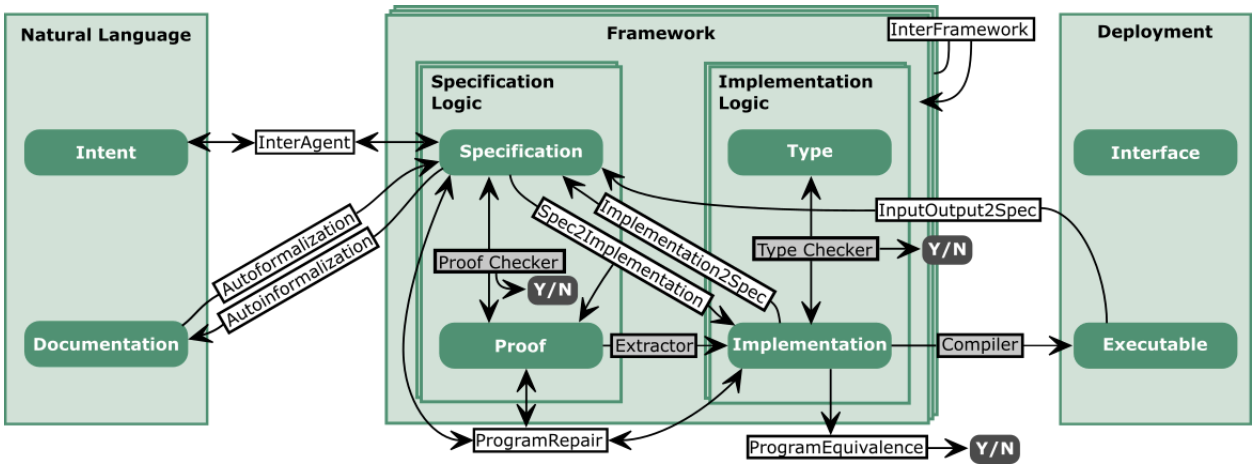
[2] Atlas Computing. {evan, daniel}@atlascomputing.org

Version 1.1 (2024-08-05)

code), *Construct* (generating code and proofs from a specification), and *Translate* (generating a specification as well as new code and proofs from legacy code in a different programming language). The 12 projects proposed and described in this toolchain are:

WorldModel	Case studies of logical frameworks and domain specific logics
LegacyCode	Case studies of legacy codes, documentations and executables
InterAgent	Collaboration between human and AI agents
InterFramework	Transpilation between logical frameworks
Autoformalization	Convert natural language to formal languages
Autoinformalization	Convert formal languages to natural languages
Implementation2Spec	Code implementations to formal specification
InputOutput2Spec	Input/output pairs from an executable to formal specifications
GenerateAndCheck	Generating implementations for autoverification in auto-active frameworks
CorrectByConstruction	Generating implementations and proofs jointly in expressive frameworks
ProgramRepair	Reconcile a divergence in program, proof, and specification
ProgramEquivalence	Determine if two programs have equivalent or divergent behavior.

The relationship between tools is shown in the following diagram (datasets projects not shown).



When completed, we expect these tools will considerably lower the cost of generating formally verified software. While our estimate for a bare minimum prototype of each tool in this entire toolchain would take roughly 21 work-years and cost just over $6M, most of the work could be performed in parallel, with the most uncertain tools requiring 2 years to demonstrate a polished, generally useful. Additionally, we also expect that one to three of these tools could deliver meaningful time and cost reductions to existing formal verification workflows in a matter of months while providing valuable training data to improve the tools themselves.

The GitHub repository github.com/atlas-computing-org/awesome-AIxFV will be a community tracker for projects that achieve the functionality of these tools.

Motivation

The Growing Risk of Unverified Code

The software running the modern world is vulnerable. This leads to serious risks for critical infrastructure, whether that infrastructure takes the form of software powering our apps, websites, databases, and networks, or software running critical hardware like pacemakers or power plants.

AI architectures, optimized for tasks like text completion, now answer programming challenges nearly as well as median software engineers,[3] which enables products like GitHub's Copilot. One experiment showed Copilot reduced the time needed to complete a software project by 55%[4], with one user quoted saying that, with Copilot, "I have to think less, and when I have to think it's the fun stuff." Unfortunately, ensuring security is rarely "the fun stuff." Furthermore, a 2022 study found that participants with access to an AI assistant wrote "significantly less secure code than those without access"[5], and it's not guaranteed this trend will reverse as assistants improve.

As research advances, we may see AI code generators suddenly able to patch vulnerabilities and generate exploits simultaneously, leading to an arms race that favors attackers, since patches take time to deploy. Even before that point, AI code generators will lower barriers to entry around new skill acquisition, changing the cybersecurity arms race to one limited, not by skill, but by willingness to quickly deploy new, untested AI capabilities.

However, these risks could be avoided with a different AI architecture. Rather than increase the capabilities of AI in the form of transformer-based language models, we should advance architectures that generate verifiable outputs, thereby providing us with the highest level of safety assurances for code, just like we do in other safety critical areas of engineering.

Formal Verification Is a Critical, Yet Costly Tool for Security

Formal verification (FV) allows programmers to mathematically prove properties of software for any possible inputs, and is generally considered the gold standard for security and robustness[6].

[3] https://paperswithcode.com/sota/code-generation-on-humaneval
[4] https://github.blog/2022-09-07-research-quantifying-github-copilots-impact-on-developer-productivity-and-happiness/
[5] https://ee.stanford.edu/dan-boneh-and-team-find-relying-ai-more-likely-make-your-code-buggier
[6] The First Tri-Lab Workshop on Formal Verification: Capabilities, Challenges, Research Opportunities, and Exemplars, 2024.

Formally verified software consists of software, specifications that list goals & constraints, and proofs that the software meets the specification. Examples include:

- Formal methods are key in validating high assurance systems, and are generally considered the de facto standard for demonstrating safety of flight-critical software systems certified by the FAA[7].
- The DARPA HACMS program yielded a formally verified system that maintained security of an autonomous aircraft while under attack from a red team with physical access. One "attacker" even stated "if you fully deployed HACMS technologies … I may not be able to imagine a way that I could even try to get in"[8].
- Formally verified microkernels (seL4), compilers (CompCert, CakeML), cryptographic tools (HACL*) and transport libraries (WireGuard, Project Everest), show consumer demand where reliability or efficiency justifies cost.

Verified code in one project was estimated to cost roughly twice that of an analogous unverified system[9]. Creating code, specifications, and proofs may seem inevitably costlier than code alone, but new tools may make this no longer true.

AI Could Be Part of the Solution

Instead of building AI-powered tools that reduce the time and thought needed to program, we should build tools that:

- …generate robust software specifications (and eventually, of general engineering systems) from natural language
- …help humans understand, compare, improve, or identify edge cases in various specifications
- …automatically synthesize programs from formal specifications with minimal or no human input required
- …provide objective proofs (or evidence) that the synthesized programs meet the specifications

These tools would enable systems to be designed, built, and audited with far less specialized expertise, and resulting systems would have verifiable guarantees. Additionally, when dependencies change, formally specified systems could be updated easily (or automatically) by generating updated software from the new dependencies and the old specifications.

This future requires shifting attention to AI architectures that carry more benefits with similar costs and lower risks, rather than advancing risky general capabilities. Ideally, early adopters

[7] Rushby, J. (1995). *Formal Methods and their Role in the Certification of Critical Systems* (SRI-CSL-95-1). Computer Science Laboratory, SRI International.
[8] https://www.youtube.com/watch?v=OyqNpn6IpBk and https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5597724/
[9] https://www.youtube.com/watch?v=lRndE7rSXiI

would find it to be cheap and easy to use formal methods to provide quantitative assurances about the safety of AI outputs, diverting R&D attention from risky general.

In practice, use of FV tooling is limited by the number of developers familiar with FV and ability for AI capabilities to support their workflows, the latter of which is itself likely limited by the small amount of training data available for LLM-based formalization. Thus, the first AI tools to dramatically scale FV will be narrowly scoped tools that are tightly integrated into existing workflows generating formally verified software. This will sidestep awareness/training bottlenecks, and initialize a virtuous cycle in which tool use creates more formally verified systems which can serve as training data to improve the tools.

Practicalities and Challenges of Scaling Formal Verification

Formal verification systems can be thought to have five main ingredients beyond those of a traditional software development project:

1. **Logical framework** - this is an underlying meta-language for defining, computing in, and reasoning in domain-specific languages or logics. The logical framework could be considered for formal verification what the programming language (e.g. C, Rust, JavaScript, Python) is for unverified software development. A particular logical framework may not be the universal best choice for verifying all programming projects, but will be generally reusable and is not project specific. In this toolchain, we will consider two kinds of logical frameworks:[10]
    - Auto-active lower-order frameworks, implemented by automated theorem provers or heuristics-based verification-aware languages (e.g. Dafny, Frama-C, Why3, Verus, Liquid Haskell)
    - Expressive higher-order frameworks, implemented by interactive theorem provers or proof assistants (e.g. Coq, Lean, Isabelle, HOL, F-star).
2. **Domain specific logic** - this is a collection of concepts, functions, relations and axioms for the application domain, defined within a logical framework. The domain specific logic could be considered for formal verification what the domain-specific software library (e.g. PyTorch for neural networks, Django for web applications) is in a programming framework (e.g. Python in the case of PyTorch and Django), and should be similarly flexible across projects without there being a universal optimum for all projects. (These software libraries are sometimes called “frameworks” in software engineering, but we avoid this terminology to prevent confusion with logical frameworks.) In a formal verification project, several domain

[10] Modal logics, which form a rich class of expressive frameworks, also play an important role in formal verification. They can be modeled in the K Framework or in dependently typed languages. Unfortunately, we will not have the bandwidth to explore them in this Toolchain.

specific logics may be employed, including a specification logic and an implementation logic, some of which may themselves be programming languages. For example, the CompCert project[11] involves at least eight intermediate languages such as Clight, Cminor and Mach. Here, the languages are embedded in a larger logical framework, namely Coq. Examples include the libraries fiat-crypto (Coq) and ValeCrypt (F*).

Fig. 1. An example of a specification, a proof and an implementation from the fiat-crypto library in Coq.

3. **Specifications** - this is the list of properties against which the software will be validated. The specifications must capture the required functionality, describing states of the program and/or properties of program outputs. For example, in Figure 1 above, the specification of the `base_25_5_mul` function asks that the output `fg` of the function satisfy some equation in modular arithmetic involving the inputs `f` and `g`.

4. **Proofs** - this is essentially a sound, formal mathematical argument mapping from assumptions and axioms to properties in the specification. A deterministic sometimes-small program (i.e. a proof checker) can check the proof to verify that the implementation satisfies the specification. For example, the above Figure 1 shows the script for a proof of the existence of the output of the `base_25_5_mul` function which was specified above. The script produces a proof term (not shown) which can be verified by a proof-checker.

5. **Implementations** - this is the software code itself. It is written in a domain specific language or logic that is dependent on the choice of logical framework, and is optimized for performance, while still satisfying the specifications. Often, the implementation has a type

[11] https://compcert.org/

signature that is a simplification of its specification, and a type checker verifies that the implementation has the right type. In practice, the implementation refers to the source code, rather than the compiled executable. For example, the above Figure 1 shows an implementation of the `base_25_5_mul` function of type `(tuple Z 10)->(tuple Z 10)->(tuple Z 10)` that was extracted from the proof term (not shown).

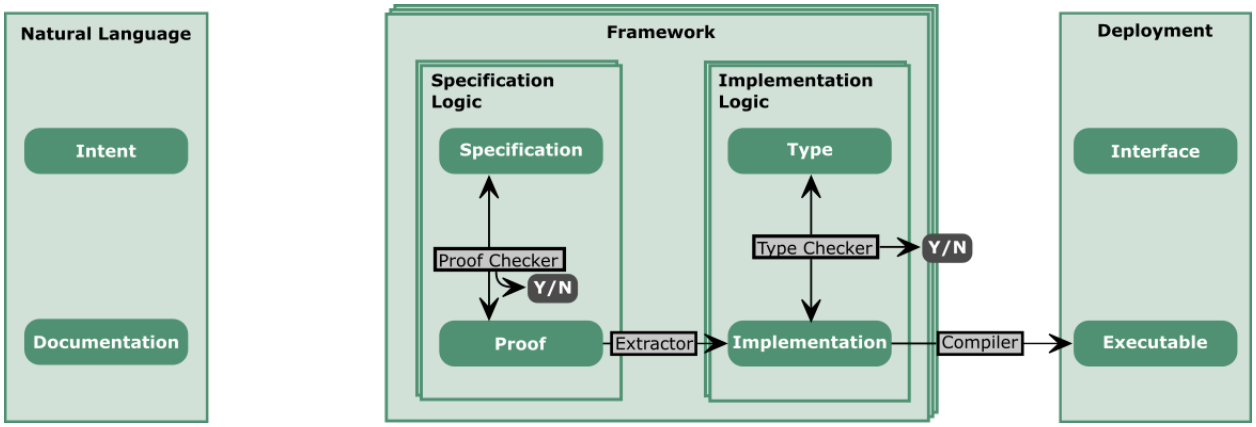


Fig 2. The components (dark green) of a formally verified system, as well as the choices of framework and logics (light green). Gray boxes show tools for transforming (→) or validating (↔) components.

Figure 2 illustrates the five ingredients for a formally verified system, as well as the relationships between them. The figure also shows other ingredients and steps common to traditional unverified software. First, software developers bring intent for what the software should do and may generate documentation, both expressed in natural language. The documentation often contains a high-level description of the software implementation. Second, software deployments involve compiling the implementation to an executable. The properties of the executable itself are described in an interface.

The naive solution to automate formal verification is to design a single system that takes in natural language and outputs a specification, implementation, and proof. However, this is unlikely to efficiently produce useful verified software, because *creation of the specification, implementation, and proof require a lot of human feedback*. Early on, human feedback will be necessary to support insufficient ML capabilities. However, even on longer timescales, human feedback may be necessary to follow evolving requirements and implementation trade-offs (e.g. memory vs bandwidth utilization).

Even incorporating human feedback is challenging as there is no "single universal workflow" followed by all formal verification projects. Instead, different projects will start with different initial artifacts: legacy code, existing codebases, or simply a desire to build and verify a new codebase.

As a result, tools must integrate into practical engineering workflows to achieve impact, and must be designed to do so. Software engineering rarely starts with an empty directory, and is instead more often bug fixes and feature additions to an existing codebase. Building a formally verified codebase similarly will have some features to be added and validated iteratively and incrementally. A successful tool must provide value inside this iterative step. As a result, we expect the best AI tools for FV will be highly modular and flexible enough to be used in any workflow, yet composable into common workflows.

Strategy

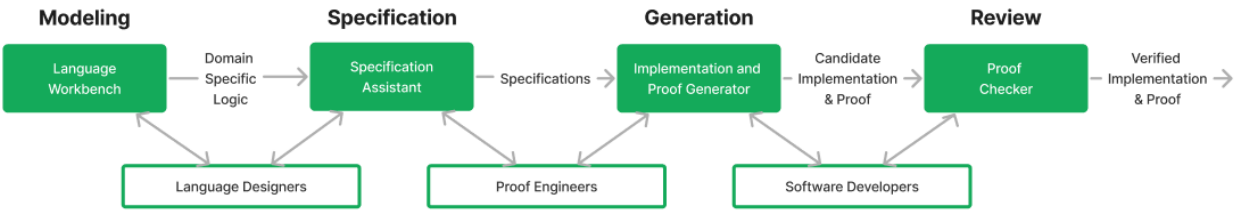
Overview of Application Workflow

Before identifying the specific tools, we identify five themes that will help organize the tools.

1. **Modeling.** Here assumptions framing the problem itself are encoded as part of the foundation of the formal specification. The world model provides a formal language for defining specifications, implementations and proofs. The model consists of
    a. A logical framework, and
    b. A domain specific logic (DSL), which includes a syntax for a specification logic

    Instead of using an existing framework and DSL, these could be generated via:
    c. Natural language documentation and prompts
    d. Legacy code bases, possibly in an outdated programming language
    e. An executable program that takes input, produces output, and can be reset.

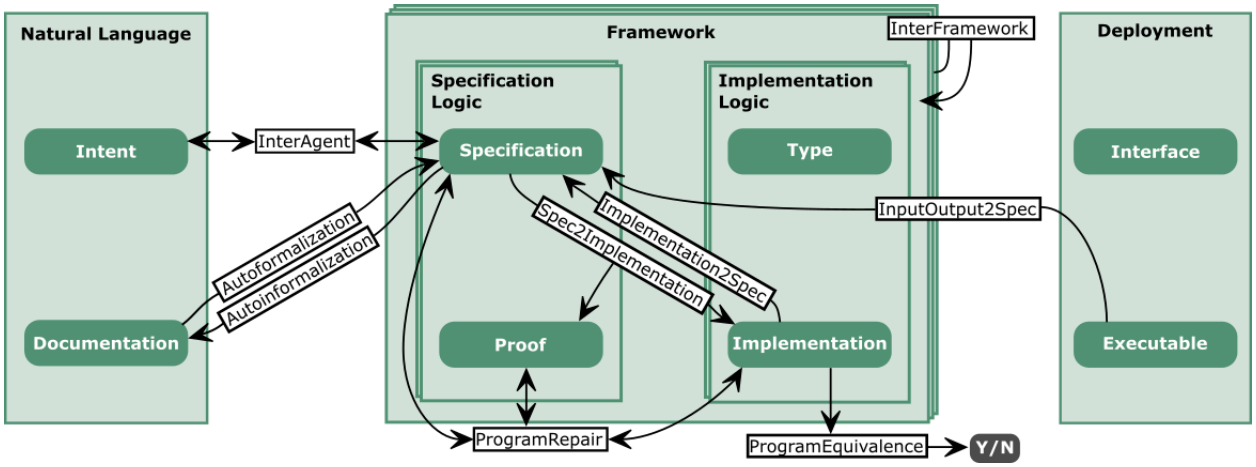
    While tools to generate a framework would be highly general, tools for building a DSL might be specific to the framework in which they are intended to operate
2. **Specification.** This theme includes tools for users to iteratively generate, better understand, and refine a specification.
3. **Generation.** This theme includes tools for users to iteratively construct, improve, and update verified programs and proofs for the specification.
4. **Review.** This toolchain lists no tools in this theme, because proofs can be easily checked by a small, highly general kernel for correctness.



Different groups of users, such as language designers, proof engineers, software developers and application users, will interact with the software platforms that facilitate this workflow.

Overview of Tools

In this toolchain, we delve into the **Modeling**, **Specification** and **Generation** themes of the application workflow named above, and identify specific tools to facilitate formalization workflows. We assume that the proof-checkers and system deployments are mature and fixed, and thus will not be exploring the **Review** theme of the workflow.



The figure above shows the different tools of the toolchain, organized by themes. We give a brief description of each tool below.

Modeling

- WorldModel - Connect specification to logical frameworks and domain specific logics
- LegacyCode - Connect legacy code to documentation and executables
- InterAgent - Collaborative information exchange between human and AI agents
- InterFramework - Transpilation between logical frameworks

Specification

- Autoformalization - Convert natural language to formal languages
- Autoinformalization - Convert formal languages to natural languages
- Implementation2Spec - Convert code implementations to formal specification
- InputOutput2Spec - Input/output pairs from an executable to formal specifications

Generation

- Spec2Implementation - Generate an implementation and proof from a specification. This can be separated down into two paths:

    - GenerateAndCheck - Generating implementations for autoverification in auto-active frameworks
    - CorrectByConstruction - Generating implementations and proofs jointly in expressive frameworks
- ProgramRepair - Repairing a program or proof that does not fit a given specification
- ProgramEquivalence - Determine if two programs have equivalent or divergent behavior.

Building Specific Workflows from Tools

The Formalize Workflow

In this workflow, we formalize a legacy codebase by adding specifications and proofs.

1. **Modeling.** User provides a software system to characterize based on one or more of:
    a. Legacy code, with optional documentation.
    b. An executable program that takes input, produces output, and can be reset. This optionally includes documentation. Source code is either provided or generated from a decompiler.
2. **Specification.** User generates and iteratively refines a spec via the following user-tool collaborations:
    a. `Autoformalization`, `Implementation2Spec`, and `InputOutput2Spec` automatically generate a formal specification (or update an existing spec) based on the resources provided during Modeling.
    b. `Autoinformalization` translates the formal spec into human readable format.
    c. The user evaluates and improves the spec's correctness and completeness, working interactively with tools.
3. **Generation.** User iteratively constructs a verified program for the spec. This involves the following user-tool collaborations:
    a. `Spec2Implementation` synthesizes a program from the spec, doing verified synthesis where possible, unverified synthesis as a fallback, and providing counterexamples where correct synthesis is not possible
    b. Existing legacy code is used as a prompt for guiding the synthesis process
    c. `ProgramRepair` fixes issues that occur during synthesis
    d. `ProgramEquivalence` checks if the synthesized program has the same behavior as the legacy code, and prompts the user to accept or reject any divergent behavior if the specifications are still met
    e. The user reviews and improves the performance of the synthesis results and revises the spec as needed, working interactively with tools.

    f. The user may switch frequently between Step 2 and Step 3
4. **Review.** The proof is checked by a small kernel for correctness. The verified implementations replace legacy implementations incrementally in production systems

The Construct Workflow

In this workflow, a formally-verified software system is constructed from just the specifications. Implementations and proofs are jointly built, possibly through refinement.

1. **Modeling.** The user provides an informal description of the system components and the specification of each component.
2. **Specification.** Formal specifications are generated from the user prompts.
    a. `Autoformalization` produces candidate formal specifications
    b. `Autoinformalization` translates the formal spec into a human readable format while `InterAgent` helps convey insights between the user and the tool.
    c. The user evaluates and improves the spec's correctness and completeness, working interactively with tools
3. **Generation.** User iteratively constructs a verified program for the spec. This involves the following user-tool collaborations:
    a. `Spec2Implementation` synthesizes a program from the spec, doing verified synthesis where possible, unverified synthesis as a fallback, and providing counterexamples where correct synthesis is not possible
    b. Existing legacy code is used as a prompt for guiding the synthesis process
    c. `ProgramRepair` fixes issues that occur during synthesis
    d. User reviews and improves the performance of the synthesis results and revises the spec as needed, working interactively with tools
    e. The user may switch frequently between Step 2 and Step 3
4. **Review.** The proof is checked by a small kernel for correctness. The verified implementations are deployed in production systems

The Translate Workflow

In this workflow, we start with legacy code in an outdated programming language, and want to construct a verified software system in a new logical framework and domain specific logic. We generate specifications, implementations and proofs by using the legacy codebase as a guide.

1. **Modeling.** User provides legacy codebase, documentation, and possibly executable programs, and choses the target logical framework and domain specific logic.
2. **Specification.** User iteratively refines a spec via the following user-tool collaborations:

    a. `Autoformalization`, `Implementation2Spec`, and `InputOutput2Spec` automatically generate a formal specification (or update an existing spec) based on the resources provided in Step 1
    b. `Autoinformalization` translates the formal spec into human readable format.Formal specifications are produced by autoformalization
    c. The user evaluates and improves the spec's correctness and completeness, working interactively with tools
3. **Generation.** User iteratively constructs a verified program for the spec.
    a. `Spec2Implementation` synthesizes a program from the spec, doing verified synthesis where possible, unverified synthesis as a fallback, and providing counterexamples where correct synthesis is not possible
    b. Existing legacy code is used as a prompt for guiding the synthesis process
    c. `ProgramRepair` fixes issues that occur during synthesis
    d. The user may use `ProgramEquivalence` to check if the synthesized program has the same behavior as the legacy code, and possibly accept any divergent behavior if the specifications are still met
    e. User reviews and improves the performance of the synthesis results and revises the spec as needed, working interactively with tools
    f. The user may switch frequently between Step 2 and Step 3
4. **Review.** The proof is checked by a small kernel for correctness. The verified implementations are deployed in production systems

Key to different types of projects:

Note: all tools start with a table of the following information. This information is reflected with additional cost breakdown in Roadmap Project Costs

Theme	Tool	Description	Category	MVP Cost ($k)	Personnel required	Duration (mo)

The category of each tool will be either

- Tech transfer: here the probability distribution for any of resources (e.g. cost, time, or talent) needed to complete the project is believed to be a narrow distribution
- Derisking: here the probability distribution of resources needed is somewhat broad
- Exploratory: here there is a long right tail on the distribution of resources needed

The other data points [Cost, Duration, and Personnel required] are the estimated median of each distribution in order to achieve a useful polished proof of concept.

Summary of Project Cost Estimates

Theme	Tool	Description	Category	MVP Cost ($k)	Personnel required	Duration (mo)
Modeling	WorldModel	Case studies of logical frameworks and domain specific logics	Transfer	250	1 Engineer	12
Modeling	LegacyCode	Case studies of legacy codes, documentations and executables	Transfer	250	1 Engineer	12
Modeling	InterAgent	Collaboration between human and AI agents	Transfer	600	2 Engineers	12
Modeling	InterFramework	Transpilation between logical frameworks	Transfer	300	1 Engineer	12
Specification	Autoformalization	Convert natural language to formal languages	Transfer	600	2 Engineers	12
Specification	Autoinformalization	Convert formal languages to natural languages	Transfer	300	1 Engineer	12
Specification	Implementation2Spec	Code implementations to formal specification	Derisk	300	1 Postdoc	12
Specification	InputOutput2Spec	Input/output pairs from an executable to formal specifications	Derisk	300	1 Postdoc	12
Generation	GenerateAndCheck	Generating implementations for autoverification in auto-active frameworks	Derisk	600	2 Postdocs	12
Generation	CorrectByConstruction	Generating implementations and proofs jointly in expressive frameworks	Explore	1200	2 Postdocs	24
Generation	ProgramRepair	Reconcile a divergence in program, proof, and specification	Transfer	300	1 Engineer	12
Generation	ProgramEquivalence	Determine if two programs have equivalent or divergent behavior.	Explore	1200	2 Postdocs	24
Total				6,200		

Note: some individual projects have a range of costs depending on the solution path; this lists the lower bound of each range to create the most minimum viable product (MVP). Here we define an MVP as a tool (likely with a polished API) that can be used to solve an arbitrarily challenging problem by decomposing it into smaller problems with more user interaction.

The linked Roadmap Project Costs document also lists the current Technology Readiness Level, the level we hope to reach by end-of-2024, and the level that we believe could be achieved for the MVP cost.

The **Modeling** Theme

Challenges

- There are many examples of formally verified software systems (e.g. CompCert, seL4), but most of them were developed from scratch rather than verifying an existing code base. One key reason is that it is more difficult to prove that an existing implementation is correct than to construct a new implementation whose structure makes verification easy. Additionally, legacy code is often written in outdated programming languages that do not have support for defining specifications. A different logical framework is therefore needed and the new verified codebase is then built from the bottom up.

- With AI-assistance for formal verification, we no longer need to start from nothing. The legacy code base provides both training data for unsupervised learning and prompts for zero-shot generation, and will improve the compatibility of suggestions by the AI model to the verification task at hand. The problem of collecting legacy codebases and their verified counterparts will be explored in the `LegacyCode` tool.

- A formal verification project begins with a suitable choice of logical framework and a description of the logic of the application domain in that framework. Domain specific logics let us work within the confines and logic of the domain (e.g. power grid security, sensor network security, air vehicle security, programming languages, hardware languages) without worrying about how they transpile to other programming languages or compile to executables in hardware languages. For example, in PyTorch, users can define neural networks as symbolic maps between tensors, without being concerned about how they eventually compile into efficient code for inference or training that is optimized on the available hardware.

- When querying AI models for formal versions of software specifications, we need to feed the model with descriptions of the world model, i.e. the logical framework and the domain specific logic. The problem of collecting frameworks and logics for training and for retrieval augmentation will be addressed in the `WorldModel` tool.

- The smooth exchange of contextual information between human agents and AI systems, expressed in formal and informal terms as needed, is a key to scalability of AI-assisted formal verification and code synthesis. The `InterAgent` tool looks into the problem of facilitating this exchange between humans, AI systems and proof assistants.

- There are many mathematical libraries and verified solutions in classical frameworks such as Coq and Isabelle. As newer and more powerful frameworks such as Lean are released, it helps to transpile existing libraries into the new framework. This problem will be explored in the `InterFramework` tool.
- Integrated development environments, such as VS Code and Lean, play an important role in facilitating the interactions and transpilations which are explored in the `InterAgent` and `InterFramework` tools.

Modeling | WorldModel

Case studies of logical frameworks and domain specific logics

Theme	Tool	Description	Category	MVP Cost ($k)	Personnel required	Duration (mo)
Modeling	WorldModel	Case studies of logical frameworks and domain specific logics	Transfer	250	1 Engineer	12

Problem

Logical frameworks and domain specific logics are essential for the modeling of ontologies, functions and relations, which are in turn used to define code specifications, implementations and proofs in an application domain. Collecting these frameworks and logics will assist language designers in developing new ones for different application domains. The information can also be used as training data or as reference documents for retrieval augmentation.

In a particular framework, we can curate a library of canonical definitions of domain specific logics as a foundation for developing more complex logics and for serving new application domains. Such a code library will be analogous to the mathematical library `mathlib` in the Lean programming language that has become a catalyst for modern theorem proving.

Plan

Here are some well-known examples of frameworks and logics for their application domains.

- DARPA High-Assurance Cyber Military Systems (HACMS)[12,13]
- ANSI/ISO C Specification Language (ACSL) for Frama-C[14]
- Project Everest: Provably Secure Communication Software[15]

Deliverables

A website or git repository containing examples of domain specific logics, grouped according to their underlying logical framework[16]. These should be sufficient to train any AI tools to output structured data in that framework (possibly assisted by, for example, syntax checkers).

[12] https://www.darpa.mil/program/high-assurance-cyber-military-systems
[13] https://loonwerks.com/publications/pdf/cofer2018computermag.pdf
[14] https://frama-c.com/html/acsl.html
[15] https://project-everest.github.io/
[16] The $250k estimated for this tool covers the construction of a specification library *speclib* for Lean that contains definitions of crucial safety and security specifications, the same way *mathlib* contains definitions of important graduate-level math concepts.

Modeling | LegacyCode

Case studies of legacy codes, documentations and executable programs

Theme	Tool	Description	Category	MVP Cost ($k)	Personnel required	Duration (mo)
Modeling	LegacyCode	Case studies of legacy codes, documentations and executables	Transfer	250	1 Engineer	12

Problem

The goal is to create datasets of legacy source code, documentation and executable programs for research. It will also be helpful to map some of these legacy datasets to formal specifications, implementations and proofs. Such a collection will provide language designers and proof engineers with user-defined reference problems as well as supply AI algorithms with training data for fine-tuning off-the-shelf models. As more legacy codes are verified with AI-assistance, they will be added as examples to this collection together with their verified counterparts.

Plan

Here are some examples of datasets and benchmarks that could be useful.

- IBM has a dataset CodeNet that seems potentially helpful and is referenced in this blogpost on Project Minerva for Modernization that targets microservices. It is a large dataset aimed at teaching AI to code, it consists of some 14M code samples and about 500M lines of code in more than 55 different programming languages, from modern ones like C++, Java, Python, and Go to legacy languages like COBOL, Pascal, and FORTRAN.
- This blogpost references a playground for refactoring a legacy web application that calculates the prices of ski lift passes, which could be a test for AI-assisted refactoring and formal verification. See also this collection of katas for practicing refactoring.
- miniCodeProps is a benchmark with 177 formal implementations and specifications in the proof assistant Lean which is aimed at the problem of finding proofs that the implementations satisfy their specifications.
- Project Everest may have a list of legacy codebases which need to be verified before we have a fully verified implementation of HTTPS.
- GitHub more generally may be a valuable source of unverified legacy source code if `Implementation2Spec` can be used to generate formal specifications to match the

source code and documentation. The value may likely be magnified by the project's interest and familiarity in formalization, as well as their willingness to participate and give feedback on the specification and verification process.

- Clover and DafnyBench include many formally specified and verified programs in Dafny, particularly intended for training ML systems.

Deliverables

A website or git repository containing examples of legacy codebases and possibly their verified counterparts. Each legacy function should preferably be paired with its verified twin. These datasets should be sufficient to train AI tools to convert between source code and documentation (possibly assisted by, for example, syntax- or type-checkers).

Modeling | InterAgent

Collaboration between human and AI agents

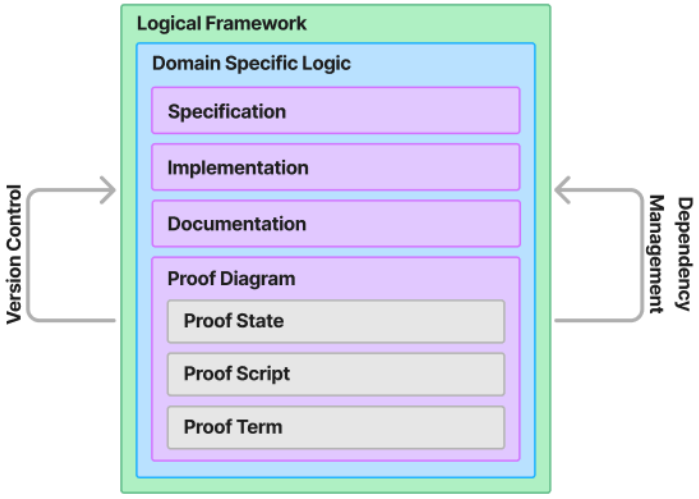
Theme	Tool	Description	Category	MVP Cost ($k)	Personnel required	Duration (mo)
Modeling	InterAgent	Collaboration between human and AI agents	Transfer	600	2 Engineers	12

Problem

The goal is to build an interface that facilitates interactions between human agents, AI systems and logical frameworks. The interface also allows specifications, implementations and proofs to evolve over time, and manages the different versions and the dependencies between them. At a later stage, human users should be able to explore implementations that are generated from the specifications. The interface should be usable by users who have little experience with formal verification or theorem proving, and have affordances for expert users at the same time.

This interface could be built as an extension of an existing IDE (integrated development environment, such as VS Code or Lean) with support for the following components and services:

1. **Specifications and Implementations.** They can be partial or full. By allowing partial specifications and implementations, humans can be ambiguous about parameters that they are unsure about, and let the AI system or logical framework fill in the blanks.
2. **Proof diagram.** For each full specification, we want to explore possible implementations and proofs. Ideally, we should store partial proofs as we construct them step by step in a directed acyclic graph which we call the proof diagram. The diagram will store:
    a. **Proof states** - subgoals generated by applying tactics to the original specification
    b. **Proof scripts** - lists of tactics used to transition between proof states
    c. **Proof terms** - the partial proof that will eventually be checked by the kernel
3. **Documentation.** This is a natural language version of the specifications and implementations that can be understood by humans.
4. **Version control.** Different versions of the specification, implementation and proofs, including unverified legacy code.
5. **Dependency management.** As we create libraries of definitions, theorems and tactics, the publishing and the dependencies between libraries are carefully managed.



Besides using AI-assistance for constructing specifications, implementations or proofs, we also ask that the interface translate the above formal constructs as well as error messages into human-readable sentences so that non-expert users can understand them.

Plan

The Open Agency Model recommends the separation of roles which could be fulfilled by human agents or AI systems in world modeling, problem specification, solution generation and solution verification, to create safer sociotechnical systems. We will be employing this strategy as far as possible in developing the Toolchain and an InterAgent interface.

Note that there isn't necessarily one solution interface that will solve all problems. We could have several potential interfaces depending on the use-cases, and it definitely isn't clear what these potential interfaces should look like at this point. For a start, the evaluation framework CodeEditorBench assesses the performance of LLMs in code editing tasks, including debugging, translating, polishing, and requirement switching. In the InterAgent project, we will explore the constraints and opportunities in this space, and more.

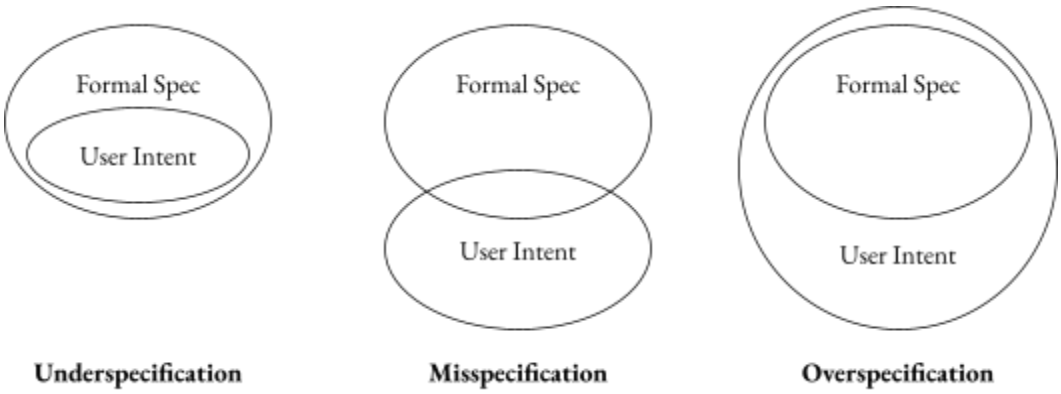
When working with verification-aware programming languages such as Dafny, Frama-C and Verus, it makes sense to extend the existing IDEs with AI assistance. However, the duplication of work needed to make this happen for different IDEs makes it hard to scale in the long run.

For more expressive logical frameworks such as dependent type theory, the Lean IDE is a promising base on top of which to build AI-enabled capabilities, because of the many existing features such as Copilot and LeanDojo and because of the ease of building extensions.

We could also extend a generic IDE, such as the structure editor Hazel[17], that makes it easy to work across different logical frameworks. AI-assisted code synthesis could in future reduce the effort required to write such extensions.

Translation between natural language and formal language will feature dominantly in the solution interface, to ease communications between human users and logical frameworks. The `Autoformalization` and `Autoinformalization` tools will play important roles here.

The formal specifications of a software system can be considered the cornerstone of the entire development process, potentially holding even greater importance than the implementations and proofs themselves. This is because these specifications serve as the guiding force behind the design and development of the other two components. Given this critical role, it's essential for the InterAgent tool to prioritize the accuracy and clarity of these formal specifications. The tool must ensure that the specifications precisely capture the user's intentions, especially when the user lacks technical expertise in formal methods.



The above diagram illustrates three different ways in which the formal specification might fail to accurately capture what the user truly wants from the software system: underspecification, misspecification, and overspecification. The InterAgent tool needs to effectively communicate any discrepancies between the formal specification and the user's original intent. One particularly powerful approach to highlighting these discrepancies is through the use of counterexamples. This method involves presenting the user with a concrete instance - be it an implementation, an input, or an output - that adheres to the formal specification as written, but clearly violates the user's intended functionality or behavior. This method of explanation not only helps in identifying specification errors but also aids in refining the specifications.

[17] https://hazel.org/

Deliverables

Extensions to well-known logical frameworks that support AI-assisted formal verification and code synthesis. Specifically, the extensions should allow views and edits of the specifications, implementations and proof diagrams, as well as their versions and dependencies. For instance, the Lean proof assistant directly visualizes most of them, except for implementations.

The extensions should also integrate:

- The `Autoformalization` tool for converting natural language intents to formal specs
- The `Autoinformalization` tool for displaying codes and errors in natural language
- The `Implementation2Spec` tool for converting implementations to specs
- The `InputOutput2Spec` tool for refining specs using input-output pairs
- The `ProgramRepair` tool for fixing broken implementations and proofs
- The `ProgramEquivalence` tool for checking equivalence of implementations.

For auto-active frameworks, the extensions should additionally integrate

- The `GenerateAndCheck` tool for generating implementations from specifications.

For expressive frameworks, the extensions should additionally integrate

- The `CorrectByConstruction` tool for refinement-based code synthesis.

To enable the same tools and AI models to be used across different IDEs, the extensions should interface with the tools using one of these methods:

- An API server that is hosted either locally or globally
- An interoperable library such as OpenNMT's CTranslate2
- Tool-specific SDKs (software development kits) for different platforms

Modeling | InterFramework

Transpilation between logical frameworks

Theme	Tool	Description	Category	MVP Cost ($k)	Personnel required	Duration (mo)
Modeling	InterFramework	Transpilation between logical frameworks	Transfer	300	1 Engineer	12

Problem

We want to leverage on the rich mathematical and software verification libraries in mature logical frameworks such as Coq and Isabelle for newer logical frameworks such as Lean. The problem is to perform *idiomatic* translation of codes from one logical framework to another, i.e. translations which feel natural in the target language, such as employing commonly-used forms and importing widely-accepted libraries. The translation source may include specifications, implementations, proofs, and even tactics which are used for generating proofs. If successful, the tool will accelerate the development of libraries for newer frameworks, and facilitate knowledge exchange between existing ones.

Plan

Experiments show that GPT4o and Claude Sonnet do well in translating source codes from Coq to Lean, although the output tends to be in an older version of Lean. Building on this initial success, we may take a theory developed through a set of libraries in one logical framework, and convert that theory to another framework using one pass of an LLM. Any errors or gaps in the translation can then be patched with human feedback and some additional help from the LLM.

One challenge is that tactics in proof scripts often do not translate nicely, because a tactic in the source framework may not have a counterpart in the target framework. To solve this problem, we could perhaps translate just the proof term and not the proof script, or we could design new tactics in the target framework (with AI-assistance) to mirror those in the source framework.

Deliverables

An API that takes as inputs

- Signature of a source logical framework (or invocation of an established framework by name and version, or reference to an established syntax checker)

- Signature of a target logical framework (or invocation of an established framework by name and version, or reference to an established syntax checker)
- Source code (specification, implementation, or proof) in the source logical framework

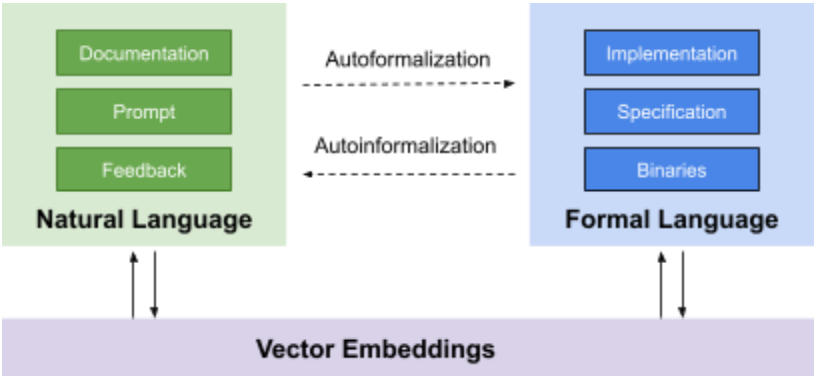
And generates the following outputs

- Translation of the input source code that satisfies the provided target signature.

The **Specification** Theme

Challenge

- At the heart of AI-assisted formal verification and code synthesis is the translation between natural languages and formal languages of everything from specifications and proofs to error messages and documentations.
- Besides converting between natural sentences and formal statements, we will also need tools for computing their vector embeddings. These embeddings will be used in other tools, such as CorrectByConstruction where we might ask how similar a new specification is to existing specifications so that we can choose an appropriate tactic for the next step in code synthesis.



- Given an implementation, deciding what its specification should be depends on the intent of its human author. The specification could be about its *behavior*, such as preconditions or assumptions about its inputs, and postconditions or assertions about its outputs. The specification could also be about its *properties*, e.g. the absence of certain side-effects such as changes to the database or calls to external APIs, or a bound on the largest value computed by the intermediate steps of an algorithm. Behavior-based specs are well-managed by auto-active frameworks, while property-based specs are better formulated in expressive frameworks.
- To train effective AI models for converting implementations or documentations to specifications, we need high quality datasets where the specifications are aligned with the implementations and documentations. Unsupervised learning on large corpuses of legacy code can help. As more examples are generated through AI-assistance and human curation, they can be added to the training datasets to improve future models.

State of the Art

- Autoformalization and autoinformalization can be approximated to some extent by existing large language models such as GPT4, Llama and Claude.
- Autoformalization, in the general form of an algorithm that automatically learns to read natural language content and turns it into abstract, machine verifiable formalization, is discussed in great depth in Szegedy's position paper. In this toolchain, we will focus however on simpler special cases, such as converting the natural language statement of a program's intent into a formal specification.
- The promise of autoformalization and autoinformalization for helping mathematicians explore strange new theories with confidence is discussed in Shulman's position paper[18].
- Fine-Tuning Language Models Using Formal Methods Feedback (2023) - Training language models to generate correct control policies. The language model is not required to produce formal outputs. Instead, the model is prompted to produce output that is aligned with a fixed vocabulary, and the output is then converted into the formal version.
- ARSENAL: Automatic Requirements Specification Extraction from Natural Language (2014) - Language models prior to GPT were already capable of specification extraction to some extent. The Stanford Typed Dependency Parser plays an important role.

[18] Strange new universes: proof assistants and synthetic foundations, Michael Shulman, 2024.

Specification | Autoformalization

From natural languages to formal languages

Theme	Tool	Description	Category	MVP Cost ($k)	Personnel required	Duration (mo)
Specification	Autoformalization	Convert natural language to formal languages	Transfer	600	2 Engineers	12

Problem

We need a tool that takes natural language descriptions of structures (e.g. specifications, implementations, proofs) and reliably generates versions of the structures in a specified formal language. The syntax of the formal language will be described by an appropriate description of the syntax (either explicitly in, for instance Backus Naur form, or by invoking an existing syntax, like "use Lean4"), and outputs of the tool should be checked against this signature.

For instance, we will be interested in natural language input in the form of documentation for an API or comments in the source code. The document could contain information about the structure of inputs and outputs to the API, and a description of what the API does. We may require the autoformalization tool to generate a formal specification of the API in some chosen domain specific logic.

Plan

Experiments have demonstrated success for simple examples (searching and sorting lists) using models like GPT-4o and Claude Sonnet.

- Microsoft TypeChat (2023) seems to be fairly reliable in using an LLM to produce JSON output that conforms to a TypeScript type or schema.
- MiniF2F (2021): a cross-system benchmark for formal Olympiad-level mathematics. Can we develop a similar benchmark for program synthesis?
- Autoformalization with large language models (2022) performs autoformalization on the MiniF2F benchmark via LLMs.
- Formalization of standards. Using transformers models, formalize 5G standards or clinical guidelines from natural language descriptions. This is unpublished work by Stéphane Graham-Lengrand (SRI).

More complex examples may be possible with additional attempts or fine-tuning.

- As a first attempt, use a language model to generate formal versions of the structure, and check the output against the syntax description, possibly by using a publicly available syntax checker. If the output fails the check, then have the LLM try multiple times and give up and announce failure after a maximum number of tries, providing the last failed output to the user so that the user can fix it herself.
- As a second attempt, syntax errors in the generated output could be fed back to the LLM, so that the LLM can attempt to fix the errors.
- Prompt engineering, few-shot learning, fine-tuning, or more sophisticated chain-of-thought strategies could likely better exploit capabilities of existing models.
- Develop a tactical metalanguage for generating and manipulating formal output, e.g. "create an empty list; append x to the head; append y to the head; append z to the tail". Train an LLM or reinforcement learner to generate formal outputs via tactics.
- Use Talia Ringer's work on proof repair to fix errors in the formal outputs.

Deliverables

An API that takes as inputs

- Signature of a formal language (or invocation of an established language by name and version, or reference to an established syntax checker)
- Natural language description of a structure to convert (e.g. specification, implementation, proof)

And generates the following outputs

- Formal version of the structure that satisfies the provided signature.

Challenges

When discussing AI for formal verification, a question is sometimes raised whether or not that humans cannot effectively provide sufficient detail for a specification in natural language. While we believe this is a hypothesis worth testing, the `InterAgent` tool is also intended to provide enhanced human-in-the-loop feedback when generating more sophisticated specifications.

Specification | Autoinformalization

From formal languages to natural languages

Theme	Tool	Description	Category	MVP Cost ($k)	Personnel required	Duration (mo)
Specification	Autoinformalization	Convert formal languages to natural languages	Transfer	300	1 Engineer	12

Problem

We need to translate formal specifications and implementations to a natural language representation that humans can understand and engage. We ask also that the tool be capable of answering formal queries in natural language, such as describing the difference between two formal specifications[19].

Plan

Initial experiments show that GPT4o and Claude Sonnet are fairly competent at deciphering what a formal specification or implementation is doing, especially if the underlying logical framework (e.g. Coq, Lean, Dafny) is well-known.

To check if the natural language output faithfully carries the full meaning of the formal input, we can convert the output to a formal version via `Autoformalization` and to compare this formal version with the original input. This consistency test was used in the Clover project[20].

Deliverables

An API that takes as inputs

- Formal version of a structure to convert (e.g. specification, implementation, proof)

And generates the following outputs

- Natural language description of the structure

[19] Sottile, Matt. Private communications, Lawrence-Livermore National Laboratories, Mar 2024.

[20] Clover: Closed-Loop Verifiable Code Generation (2023), Chuyue Sun, Ying Sheng, Oded Padon, Clark Barrett.

Specification | Implementation2Spec

Code implementations to formal specification

Theme	Tool	Description	Category	MVP Cost ($k)	Personnel required	Duration (mo)
Specification	Implementation 2Spec	Code implementations to formal specification	Derisk	300	1 Postdoc	12

Problem

Develop an *interactive* AI system that generates formal specifications for legacy code. The AI system should take a domain specific logic and a function in the legacy codebase as inputs, and suggest preconditions, postconditions and invariants for the function. The system should refine the specifications iteratively with feedback from human users. The generated specification should be verified against a signature or syntax checker for the domain logic.

With further prompts from users, it may also suggest additional properties that the function should satisfy, e.g. bounds on the value computed by intermediate steps in the function. The AI system may suggest new definitions to be added to the domain specific logic, so that the specifications can be stated more clearly.

Plan

Initial experiments show that LLMs such as GPT4 are reliable at recognizing simple functions (e.g. maximum of an array, binary search) and generating preconditions, postconditions and invariants for the function in Dafny, possibly because these functions and their specifications are included in its training data. The same LLMs are also capable of generating higher-order formal specifications (e.g. the functorial function is always positive) in Lean, but prompts from human users (e.g. “show that the functorial function is positive”) are needed.

The next step is to use lightweight fine-tuning techniques, such as QLoRa, for updating off-the-shelf language models with changes in a core domain specific logic (e.g. a security library) or in the logical framework (e.g. Lean version 3 to version 4). We need to explore when retrieval augmentation is not enough and fine-tuning is needed to improve reliability.

Matt Sottile (LLNL) has suggested that this tool be tested on large-scale codebases, such as the compiler for the LLVM IR (Intermediate Representation), and not just small-scale ones. Samuel Pollard (Sandia) also gave ideas for codebases for testing, such as CubeSAT, CAN (Control Area Network) bus protocols, HFODD, Xyce, and Rust libraries.

Deliverables

An interactive extension to an existing logical framework (e.g. Lean, Verus) that takes as inputs

- Implementation of a function from a legacy codebase
- Domain spec logic in the form of a library
- (Optional) Natural language requirements or feedback
- (Optional) Prior versions of a formal specification

And generates the following outputs

- Suggested formal specifications for the function implementation. Proof that the implementation satisfies the specification is not required.

Specification | InputOutput2Spec

Input/output pairs from an executable to formal specifications

Theme	Tool	Description	Category	MVP Cost ($k)	Personnel required	Duration (mo)
Specification	InputOutput2Spec	Input/output pairs from an executable to formal specifications	Derisk	300	1 Postdoc	12

Problem

Refine a given specification using a compiled executable that satisfies the specification, by experimenting with inputs and their observed outputs. For instance, the executable could be a function that organizes some data values into a packet, and we could have a partial specification for the executable that does not spell out the packet protocol. The goal then is to discover the packet protocol, with some hints on its format. This tool should also be capable of generalizing unit tests (given as a list of input-output requirements) to formal specifications. The tool is important for situations where specifications are hard to infer from implementations alone.

Plan

When converting, say, a list of data values into a tabular format, LLMs have shown that they are capable of generalizing what needs to be done from just a few input-output examples. We want to show that they are equally capable of writing down the formal input-output relationships.

The next step is to enable the tool to choose inputs and to observe the outputs through its own experiments. The tool may use reinforcement learning to improve at choosing the right inputs. This experimentation should involve humans in the loop, where the humans provide oversight on the process. Automating the experiments in this manner will save the human user time.

Deliverables

In the short run, the tool will take as inputs

- A partial specification for some function
- Input-output pairs for the function

And generate the following output

- A refinement of the specification that is consistent with the input-output pairs.

In the longer run, the tool will take an executable instead of input-put pairs. It will call the executable in order to extract its own list of input-output pairs.

The **Generation** Theme

Challenges

- The process of generating proofs and implementations from specifications is heavily influenced by the logical framework being used. On one end of the spectrum, we have auto-active frameworks, such as Hoare logic and separation logic. These frameworks contain specialized fragments of first-order logic that offer a significant advantage: automatic determination of whether an implementation meets a given specification. In these auto-active frameworks, algorithms are employed to verify compliance, eliminating the need for manual proof construction. As a result, when working within such frameworks, the focus shifts primarily to generating the implementation itself, as the proof of its correctness is produced automatically by the system.

- On the other end of the spectrum, we have expressive frameworks, exemplified by dependent type theories, which offer a more sophisticated approach to logical reasoning and specification. These frameworks are built upon higher-order logics, which allow for a greater degree of complexity and abstraction in their formulations. Unlike simpler logical systems, these expressive frameworks permit predicates—essentially boolean functions—to take not just elements of sets as inputs, but also entire sets and functions themselves. This increased flexibility enables the expression of more intricate mathematical relationships and allows for the creation of more nuanced and comprehensive code specifications.

- However, this expressiveness comes with a trade-off. In these more complex frameworks, the process of proving properties of implementations often cannot be fully automated. Instead, proofs frequently require manual construction via metaprogramming actions called *tactics*, demanding a deeper understanding and more hands-on approach from the developer or mathematician. Interestingly, the interconnected nature of proofs and implementations in these expressive frameworks sometimes leads to an unexpected workflow: in certain cases, it may be more efficient and effective to develop the proof and the implementation simultaneously, rather than sequentially. This joint construction process, also known as the *correct-by-correction* strategy in deductive synthesis, can lead to a more holistic understanding of the problem and solution, potentially resulting in more robust and well-verified implementations.

- There are varying approaches to representing code implementations within a logical framework. One method, known as shallow embedding, involves using the framework's

built-in programming constructs such as loops, pattern matching, and arithmetic operations to directly construct the implementation. In contrast, deep embedding takes a more indirect approach. This method first defines the implementation language as an abstract data type within the logical framework. Once this abstract representation is established, the actual implementations are then expressed as terms or instances of this custom data type. These two techniques offer different trade-offs in terms of simplicity, expressiveness, and the ease of reasoning about the implemented code.

- In the last two decades, proof assistants have garnered significant attention for their role in validating intricate mathematical propositions, including the Four Color Theorem and the Odd Order Theorem. These tools have also demonstrated their value in facilitating large-scale collaborative efforts like the Liquid Tensor Experiment. The remarkable achievements of proof assistants in the realm of theorem proving hint at their potential for revolutionizing verified software development. This potential extends across both auto-active and expressive logical frameworks, and encompasses techniques involving shallow and deep embeddings of code. Moreover, recent advancements have seen the integration of artificial intelligence into the field of theorem proving, yielding encouraging outcomes[21,22,23,24]. This success suggests that similar AI-driven approaches could be effectively applied to the domains of formal verification and code synthesis.

- Talia Ringer's thesis introduced the concept of *Proof Engineering*, drawing a parallel with Software Engineering. Just as software engineering encompasses the creation and ongoing maintenance of software, proof engineering extends these principles to the realm of implementations and their associated proofs. This paradigm acknowledges that proofs and implementations, like software, require not just initial development but also continuous maintenance and refinement. In this Toolchain, we will examine various tools that support both the development and maintenance aspects. On the development side, we'll explore tools like `GenerateAndCheck` and `CorrectByConstruction`. For maintenance, we have tools such as `ProgramRepair` and `ProgramEquivalence`. These tools represent just the beginning of what's needed in this field. As proof engineering continues to evolve and gain prominence, it's anticipated that an even wider array of specialized tools will become necessary. These future tools will likely address the unique challenges posed by verified software systems, supporting their development, verification, and long-term maintenance in increasingly sophisticated ways.

[21] Tactictoe: Learning to prove with tactics, Gauthier et al., 2021.
[22] Hypertree proof search for neural theorem proving, Lample et al., 2022.
[23] Dt-solver: Automated theorem proving with dynamic-tree sampling guided by proof-level value function, Wang et al., 2023.
[24] A Survey on Deep Learning for Theorem Proving, Li, Z., Sun, J., Murphy, L., Su, Q., Li, Z., Zhang, X., Yang, K. and Si, X., 2024.

- Legacy codebases serve a crucial function in the evolution of AI-assisted verification and development tools. These legacy systems provide valuable insights and patterns to AI models, much like how they guide human architects in designing new, formally verified software systems. Specifically, these codebases serve a dual purpose: they act as training datasets and reference documents for AI models tasked with generating implementations and proofs. However, the proprietary nature of many of these legacy systems presents a unique challenge. To address this, there's a need to develop AI tools that can leverage the knowledge embedded in these codebases while simultaneously safeguarding their confidentiality, possibly through the use of privately hosted models.

State of the Art

- Auto-active frameworks are often implemented as verification-aware programming languages such as Dafny, Frama-C, Verus and Liquid Haskell. Under the hood, auto-active frameworks employ SMT (Satisfiability Modulo Theories) automation to solve complex verification problems. Expressive frameworks are often implemented as proof assistants such as Lean, Coq, Agda, Isabelle, HOL, Why3 and F*. Some expressive frameworks, such as Why3 and F*, contain auto-active fragments where verification of implementations against preconditions, postconditions and invariants is automated. Despite being cast in the expressive framework Coq, MIT's Bedrock performs highly automated verification of low-level programs by using techniques from separation logic. The CakeML project employs a similar approach, providing end-to-end verification via the HOL4 proof assistant. HOL functions are compiled to CakeML AST with proof that behaviors are preserved in the process. A verified compiler then translates this generated AST to machine code. Meanwhile, Charguéraud's CFML verification framework has also been adapted to CakeML for Hoare-style verification of low-level CakeML programs.

- In recent years, the field of program synthesis has seen remarkable advancements, particularly in the area of generating code from natural language prompts. A prominent example of this technology is GitHub Copilot, which leverages large language models to interpret human-written prompts and generate corresponding code implementations. However, it's important to note that while these systems can produce functional code, they typically do not include built-in verification mechanisms. Besides LLMs, other AI techniques such as reinforcement-learning-guided tree search have also been employed for program synthesis to improve the quality and relevance of generated code. These developments represent a significant step forward in automating aspects of software development, but they also highlight the ongoing challenges in ensuring the reliability and correctness of AI-generated code.

- Deductive synthesis, a method for creating efficient implementations by transforming simpler code forms, has its roots in the 1977 work of Burstall and Darrington. This approach has evolved over time, integrating formal methods to give rise to the correct-by-construction methodology. Notable examples of this evolution include the Kestrel Institute's Specware system, and the Spiral project, which focuses on synthesizing digital signal processing algorithms and numerical kernels.

- A significant advancement came in 2015 with the introduction of the Fiat system. The system develops domain-specific logics (via refinement types) within the expressive Coq framework, and leverage Coq's tactics language (via refinement rules) to largely automate the joint creation of implementations and their proofs. This approach offers a high-level language for specifying formal requirements while also enabling low-level optimizations for generating efficient codes. The Fiat system has successfully synthesized formally verified database systems and cryptographic algorithms. Pit-Claudel's thesis further advanced the field by introducing the relational compilation framework. This work reframed program extraction as a proof-search process, aiming to derive correct efficient implementations from shallowly-embedded functional programs. Runge et al. have contributed to the field by introducing block-based and trait-based strategies, which offer greater flexibility in correct-by-construction software development.

- Chlipala et al. clarified their hypothesis that domain specific languages should be replaced by libraries in proof assistants that mix theorem proving and programming, allowing for modularization of functionality away from performance. In their position paper, Appel et al. argued that in order to routinely build large verified systems out of smaller reusable verified components, we need to design specifications which are deep, i.e. rich, two-sided, formal and live.

- Correct-by-construction program synthesis shows great promise in enhancing software reliability and bridging the divide between high-level specifications and efficient, practical implementations. This approach is particularly valuable for developing critical software systems where formal correctness guarantees are crucial. However, the widespread adoption and application of these techniques face several hurdles. The process requires advanced proof techniques, in-depth domain-specific knowledge, and sophisticated automation to be practical in real-world scenarios. Further research is necessary, such as scaling the techniques to handle larger and more complex systems, and making the approach more accessible to developers who may not have extensive backgrounds in formal methods.

Generation | (Spec2Implementation) GenerateAndCheck

Generating implementations for autoverification in auto-active frameworks

Theme	Tool	Description	Category	MVP Cost ($k)	Personnel required	Duration (mo)
Generation	GenerateAndCheck	Generating implementations for autoverification in auto-active frameworks	Derisk	600	2 Postdocs	12

Problem

The goal is to develop an AI system that generates code for a given specification in a verification-aware programming language (e.g. Dafny). Through auto-active verification, this code can often be checked by the logic engine without human direction. If this project succeeds, we will have verified software systems for simple well-specified applications, e.g. accounting systems, database lookups.

Verifier. We assume that we have a verifier V that takes in a state string S and outputs a pair V(S) = (Z,E) where Z is the status with values in {`fail`, `incomplete`, `success`} and E is any error or state message. A `fail` means that the state string S has failed verification and appending additional text to the string will not help. The error message is provided in E. A `success` means that the state string S has passed verification. The error message E will be empty. An `incomplete` means that the state string S has failed verification but appending additional text to the string may help. In other words, the state string S is incomplete. The parser state is provided in E.

Interface. We want to design an AI assistant that can facilitate program synthesis via one of the interactive platforms described in the InterAgent tool. The simplest interface is a REPL (Read-Eval-Print Loop).

1. The interface reads a state string S from the human.
2. The verifier will evaluate the state string V(S) = (Z, E).
3. The interface will output both the status Z and any error message E to the human.
4. Repeat by returning to Step 1.

Tree Search. The dynamics of this interface unwraps into a tree of possible interactions. We want to train an AI system that can navigate this interaction tree successfully and arrive at an implementation that passes the check by the verifier.

Alternative Interfaces. If the chosen interface from the InterAgent tool is not a simple REPL, we should expect a different space of interactions and we may need to design a different AI model to navigate that space efficiently.

Plan

The Clover project[25] and the VerMCTS project[26] provides a good initial foundation for this plan. In VerMCTS, the authors experimented with multi-step LLM-based code generation and verification in Dafny, Coq and Lean with positive results. Similar generate-and-check ideas were also proposed by the NuSCI research group in the context of automated planning with LLMs. The GenerateAndCheck tool will explore having formal specifications as inputs rather than natural language inputs, as well as the ability to do repair using error messages from the verifier.

Language Model. We incorporate a large language model L into the AI system for generating possible action strings A from the state string S. We assume for simplicity that the string concatenation S+A is a proposal for a new state string S'. We may also assume that a standard prompt string P is used with the state string S as inputs to the LLM to get action strings as outputs, i.e. A = L(P+S).

Reinforcement Learner. We train a reinforcement learner on the states S and the actions A generated by the LLM. Given that the space of interactions for REPL is a tree, we can use Monte Carlo Tree Search for reinforcement learning. If computing resources permit it, we could also fine-tune the LLM using a strategy like DPO (direct preference optimization) to generate better actions from a given state S over time.

Error Messages. If there are errors E, we will then use the LLM to generate a new corrected state S' from S and E. This repair step could make direct application of MCTS tricky. We could use the ProgramRepair tool to explore alternative ways of addressing this problem.

Use-cases. Apply this plan to software components in Project Everest which are written in the F* framework. We could train GenerateAndCheck on the batch of existing verified components, and use the tool in developing and repairing new components. Instead of batch-training, we could explore online learning for GenerateAndCheck, so it progressively gets better at assisting the human in new application domains.

[25] Clover: Closed-Loop Verifiable Code Generation, C. Sun, Y. Sheng, O. Padon, C. Barrett, 2023.
[26] VerMCTS: Synthesizing Multi-Step Programs using a Verifier, a Large Language Model, and Tree Search, D. Brandfonbrener, S. Henniger, S. Raja, T. Prasad, C. Loughridge, F. Cassano, S. R. Hu, J. Yang, W. E. Byrd, R. Zinkov, N. Amin, 2024.

Deliverables

Monthly milestones are as follows.

- M2: GenerateAndCheck generates actions from formal specs. Error messages ignored.
- M4: GenerateAndCheck incorporates error messages into choice of actions.
- M8: GenerateAndCheck performs online learning for developing new components.
- M12: Integrating GenerateAndCheck into InterAgent - the human can choose to run GenerateAndCheck step by step, or let GenerateAndCheck pilot automatically.

At the end of the project, we should have an API that takes as inputs

- The formal specification of a desired low-level function in an auto-active framework
- A verifier for the auto-active framework

And generates as outputs

- An verified implementation that satisfies the specification.

Generation | (Spec2Implementation) CorrectByConstruction

Generating implementations and proofs jointly in expressive frameworks

Theme	Tool	Description	Category	MVP Cost ($k)	Personnel required	Duration (mo)
Generation	CorrectBy Construction	Generating implementations and proofs jointly in expressive frameworks	Explore	1200	2 Postdocs	24

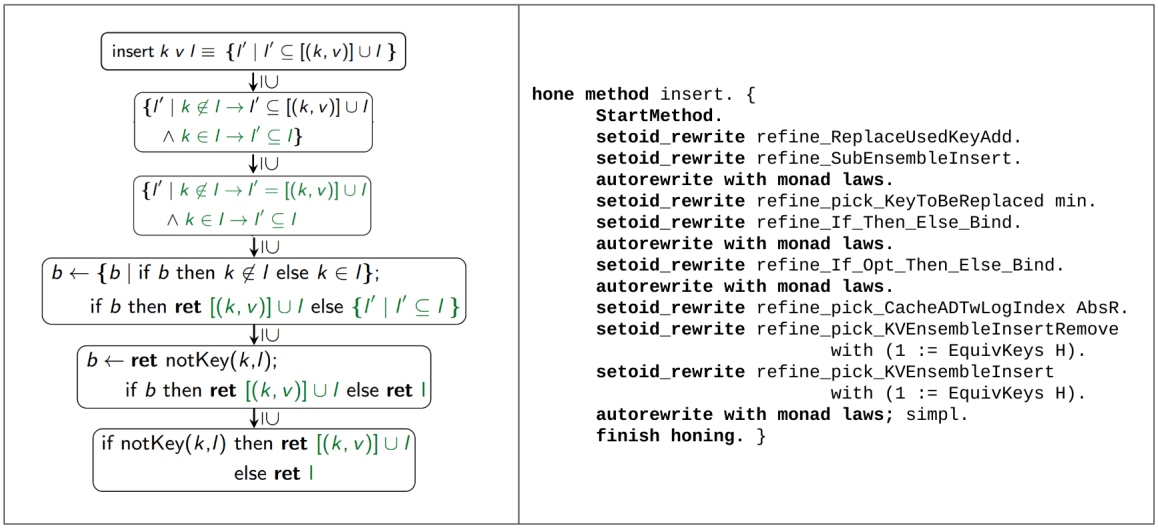
Problem

We develop an AI system for semi-automating refinement-based synthesis in expressive logical frameworks. The implementation is jointly generated with its proof and therefore correct by construction. If this project succeeds, we will have a new programming paradigm for building verified software systems with high-level architectural specifications, e.g. cryptosystems, knowledge bases.

Trusted code base. In traditional refinement-based program synthesis, e.g. the Specware project, the synthesis steps are chosen from a trusted database of correct constructions. As this database grows over time, the burden of proof of its correctness grows as well. To solve this problem, we ask that synthesis produces proofs in addition to implementations. These proofs should be verifiable by a proof checker at the end of refinement, so only a small trusted code base is needed, namely that of the proof checker.

Expressive frameworks and refinement-based synthesis. As we build more complex software systems, we need to use more expressive frameworks, such as dependent type theory or higher-order logics, for defining our specifications. Unlike verification-aware programming languages (e.g. Dafny, Frama-C, Verus) which are auto-active frameworks with highly automated proof generators, expressive programming languages (e.g. Lean, Coq) typically need more human feedback in constructing proofs, and this feedback is facilitated by proof assistants. We ask that the implementation be constructed jointly with its proof through a process of *refinement*.

General proof development. We ask that the refinement steps proceed by metaprogramming actions called *tactics*. Tactics decompose the desired specification or goals into subgoals, and compose proofs and implementations of the subgoals into those of the original goal. This tactical approach puts program synthesis in the same general proof development environment as that of mathematical theorem proving, thus allowing mathematical libraries to be applied.



For example, the above figure shows Fiat's refinement-based code synthesis. On the left, we have refinement types for synthesizing the insertion of a key-value pair into a cache. On the right, we have the honing tactics that produce the refinements on the left.

Functionality and Performance. This correct-by-construction strategy has made it easier to synthesize high-level code that is both *functional* and *performant*, such as the fiat-crypto library for cryptographic arithmetic[27]. It is estimated that over 95% of HTTPS connections by browsers run the generated algorithm. We ask that the proposed solution demonstrates the efficiency of synthesized implementations.

Learning-based automation. While many simple program synthesis goals can be automated by powerful search-based tactics, deciding which tactic to use and deciding the sequence of tactics required for more complex synthesis goals can be challenging for non-experts. We ask that an AI assistant be trained to provide help in the choice of tactics, performing tree-search in the space of tactic sequences, if necessary, to find a solution.

Human-AI integration. Integrate AI assistance into the InterAgent tool, because human feedback is much more important for this tool than, say, for the GenerateAndCheck tool.

Plan

Fiat. Using the `InterFramework` tool, translate the Fiat synthesis library from Coq to Lean, because we will be using Lean extensions for AI-assisted theorem proving.

Data. We will experiment with examples in the Fiat library. The fiat-crypto library also provides many examples for testing. The problem is deciding how to split the examples into data for

[27] Simple High-Level Code For Cryptographic Arithmetic – With Proofs, Without Compromises Erbsen, A., Philipoom, J., Gross, J., Sloan, R. and Chlipala, A., 2020.

training or inputs for prompts, and test data. For each domain, it seems appropriate to use the earlier examples for training or prompts and to keep the later examples for testing.

VerMCTS. We will use the VerMCTS (verified Monte Carlo tree search) platform as a first attempt at building an AI system for tactic selection for refinement-based synthesis. We can prompt the language model to suggest refinement tactics from Fiat by feeding sample proof scripts to the large language model as an input file.

LeanDojo. Alternatively, install LeanDojo and Lean Copilot. The `suggest_tactics` and `search_proof` commands work well in VS Code to suggest a single tactic or a chain of tactics for the goal at hand. You can "bring your own model" - configure Lean Copilot to use a custom AI model, such as the VerMCTS. It is not clear from the documentation if you can feed an input file to the underlying LLM, so it may be necessary to develop this feature.

Typeclasses. Moreover, we believe that greater automation can be achieved by combining AI assistance with typeclass-based strategies for type inference, an approach that has led to successful formalized proof of complex math theorems such as the Four Color Theorem and Odd Order Theorem; see Ch 6 of Mathematical Components for an elaboration. A similar attempt to solve this problem involves using traits[28] to chain complex reasoning steps (e.g. induction/recursion) . We can replicate this strategy with typeclasses, so as to exploit the same general proof development engine as that for theorem proving.

Generic Tactics. The eqType typeclass, which consists of types with a comparison function, allows the equality operator and the rewriting tactic in Coq to be overloaded. Type inference involves guessing the relevant instance of the typeclass from context when it is not explicitly given with the operator or tactic. Through overloading, generic refinement-based tactics can be designed for all types in a given typeclass, and AI models can be trained to use these generic tactics across a variety of problems with similar structures.

SMT Solvers. Some automated solving can be achieved with refinement reflection[29]. These SMT solvers can be integrated into the tool as powerful tactics, and the CorrectByConstruction tool should recommend such tactics whenever they are applicable.

More details of the above plan is laid out in the Topos, Stanford and Atlas' NSF AIMing proposal *AI-Assisted Refinement-Based Code Synthesis via Generic Typeclass-Based Tactics*.

Deliverables

An interactive tool that initially takes as inputs

[28] Flexible Correct-by-Construction Programming, Runge, T., Bordis, T., Potanin, A., Thüm, T. and Schaefer, I., 2023.
[29] Refinement reflection: complete verification with SMT, Vazou, N., Tondwalkar, A., Choudhury, V., Scott, R.G., Newton, R.R., Wadler, P. and Jhala, R., 2017.

- A higher-order formal specification of the desired function as a starting goal
- A domain specific logic that describes permissible refinement types
- A set of tactics that act as refinement rules

And generates as outputs

- An implementation that satisfies the specification
- A refinement-based proof that the implementation is a solution

By taking as interactive feedback

- Generic typeclass-based tactics that decompose intermediate subgoals

And by providing as interactive responses

- Suggested tactics for the next step
- Recommended proof scripts to synthesize the desired implementation and proof.

Generation | ProgramRepair

Reconcile a divergence in program, proof, and specification

Theme	Tool	Description	Category	MVP Cost ($k)	Personnel required	Duration (mo)
Generation	Program Repair	Reconcile a divergence in program, proof, and specification	Transfer	300	1 Engineer	12

Problem

Proof repair is needed when code specifications are updated or when there are errors in a proof script that is proposed by a human or AI agent. By providing AI assistance for proof repair, we reduce the friction faced by humans doing code synthesis, increase the success rate of proof automation by AI agents, and improve proof maintenance of large complex verified systems.

In its full generality, proof repair concerns the fixing of proof scripts in proof assistants. In the context of formal verification, a proof script may fail because of changes in the specification, implementation or imported libraries, or even just a change in the script itself. Information about the change which caused that failure will be extremely useful for deriving a fix for the proof script. Other information such as the context, documentation or error messages from the proof assistant can also help.

Plan

Many existing tools and libraries for proof repair are written for the proof assistant Coq. We want to translate those tools to Lean, so that they can be used in conjunction with other plugins for AI-assisted code synthesis. For example, Talia Ringer's lab has some Coq-based solutions, such as Pumpkin Pi and Baldur, for proof repair.

Pumpkin Pi. This library is an impressive tool for performing proof repair across type equivalences, and it involves configuration, transport and decompilation. Translating each step of the tool to work in Lean could be challenging. We could add the methods in the tool as new tactics (e.g. `transport`, `repair`) to Lean Copilot's current offering (`suggest_tactics`, `search_proof`, `select_premises`).

Baldur. Another notable Coq tool for LLM-assisted proof generation and repair is Baldur. Instead of formally deriving the required repair from the specification and proof term, Baldur queries a large language model for a new proof script by supplying the specification, original proof term and the generated error message. The new script is then checked by the proof

assistant for correctness. Baldur's machine learning approach may be more suited for complex repair problems which are not easily handled by the formal strategies of Pumpkin Pi.

As a start, it will be easier to implement Baldur in Lean, than to implement Pumpkin Pi in Lean, because Baldur only requires queries to a language model but Pumpkin Pi requires careful design of the decompiler from the native proof language of Lean to its tactic language.

Drawing from the computational setup of Lean Copilot, we can expose the large language model or AI system to a proof repair plugin for Lean via CTranslate2 or an API call to a remote server.

We add a new tactic `repair` to Lean Copilot that takes the current target specification, proof script and latest error messages, and proposes edits to the proof script. The `repair` tactic can then be used in an alternative `search_proof` tactic that iterates between full proofs and their repairs until a working proof is obtained, in the spirit of Baldur.

Deliverables

Here are the quarterly milestones.

- Q1: Set up Lean Copilot, and write a Lean tactic that is able to make a call to a large language model via an API call to a remote server.
- Q2: Write the `repair` tactic to get suggested fixes from the language model.
- Q3: Provide avenues to finetune the language model on new domains and tactics.
- Q4: Write a new `search_proof` tactic that iteratively generates full proofs and fixes.

At the end of the project, we should have an interactive tool that initially takes as inputs

- The original formal specification and a verified proof script
- The replacement specification
- The current proposed replacement proof script
- Error messages generated by the proof script

And generates as outputs

- Proposed corrections to the proof script.

Generation | ProgramEquivalence

Determine if two programs have equivalent or divergent behavior

Theme	Tool	Description	Category	MVP Cost ($k)	Personnel required	Duration (mo)
Generation	Program Equivalence	Determine if two programs have equivalent or divergent behavior.	Explore	1200	2 Postdocs	24

Problem

Given two implementations of a function with inputs and outputs, such as a legacy implementation and a new verified implementation, formally check if they have the same behavior, i.e. the same output for all inputs. If the behaviors are different, provide an example of the divergent behavior or counterexample, i.e. an input to the function that generates an undesirable output. In other words, we want a reproducible bug that can help humans in debugging the implementation. Solving this problem will improve productivity in both code synthesis and code specification.

In its full generality, this is a very difficult problem. For example, if the function returns `True` if the Collatz algorithm (where we iteratively compute 3n+1 or n/2 depending on whether n is odd or even) terminates but `False` otherwise, proving that the function is equivalent to the constant function that returns true is the content of the infamous Collatz conjecture.

We focus on special cases of program equivalence checking. In many cases, we want to use equivalence checking as a way of finding out if the specification for a new verified implementation covers all important properties of the legacy implementation. If there is a difference in program behavior, this difference is either a bug in the legacy implementation, or a hole in the new specification.

In a verification-aware programming language such as Dafny or Frama-C, it is desirable to have a plugin that finds a counterexample to a failed verification of the current implementation.

Plan

For auto-active frameworks, we can often use SMT techniques to prove program equivalence or to find a counterexample. Dafny currently prints a counterexample if you click on the red dot next to a failed assertion, but the counterexample is often cryptic and difficult to understand. The goal is to produce smaller examples[30], or print the examples in a way that is easier to read[31].

[30] Improving counterexample quality from failed program verification, Huang, L., Meyer, B. and Oriol, M., 2022.
[31] Better counterexamples for Dafny, Chakarov, A., Fedchin, A., Rakamarić, Z. and Rungta, N., 2022.

For more sophisticated problems, there are alignment techniques such as semantic program alignment[32] or algebraic methods[33]. We can also use test cases and memory traces[34] to find divergence in behaviors. Note that this does not give us a formal proof, but in many cases, this may be good enough.

Deliverables

An API that takes as inputs

- Two implementations with the same input and output types.

And generates as outputs, either

- A formal proof that the two implementations have the same behaviors; or
- A counterexample, i.e. an input value where the implementations have different outputs.

[32] Semantic program alignment for equivalence checking, Churchill, B., Padon, O., Sharma, R. and Aiken, A., 2019, June.
[33] An algebra of alignment for relational verification, Antonopoulos, T., Koskinen, E., Le, T.C., Nagasamudram, R., Naumann, D.A. and Ngo, M., 2023.
[34] Practical, low-effort equivalence verification of real code, Ramos, D.A. and Engler, D.R., 2011.

Call to action

A product plan is incomplete unless it clarifies how an engineer uses the product to deliver an incremental change to an existing codebase that has a mix of verified and unverified code in specific languages/etc.

We are creating and will maintain the GitHub repository atlas-computing-org/awesome-AIxFV as a community tracker for projects that achieve the goals of these tools.

Acknowledgments

Many thanks to Nada Amin, Nora Ammann, Clark Barrett, David 'davidad' Dalrymple, Mike Dodds, Stephane Graham-Lengrand, Justin Gottschlich, Syed Jafri, Anders Jagd, Ramana Kumar, Mishaal Lakhani, Sam Douglas Pollard, Dawn Song, Bogdan Stanciu, Agustín Martinez Suñé and Adam Vandervorst for their valuable feedback and suggestions in early versions of this Toolchain.

This work is a collaboration between the Topos Institute and Atlas Computing, with Topos providing expertise and writing and Atlas providing funding, direction, and structure. This work was made possible by the generous support of Protocol Labs and the Survival and Flourishing Fund.