The First Tri-Lab Workshop on

Formal Verification

Capabilities, Challenges, Research Opportunities, and Exemplars

SAND2024-02142
Printed 26 February 2024

Issued by Sandia National Laboratories, operated for the United States Department of Energy by National Technology & Engineering Solutions of Sandia, LLC.

Available to DOE and DOE contractors from

U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@osti.gov
Online ordering: http://www.osti.gov/scitech

Available to the public from

U.S. Department of Commerce
National Technical Information Service
5301 Shawnee Road
Alexandria, VA 22312

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.gov
Online order: https://classic.ntis.gov/help/order-methods

NNSA
National Nuclear Security Administration

The First Tri-Lab Workshop on Formal Verification

Capabilities, Challenges, Research Opportunities, and Exemplars

Workshop Organizer

Samuel D. Pollard[1]

Report Authors

Samuel D. Pollard[1], Jon M. Aytac[1], Ariel Kellison[1,6], Ignacio Laguna[3], Srinivas Nedunuri[1], Sabrina Reis[3], Matthew J. Sottile[3], Heidi K. Thornquist[2]

Workshop Attendees

Jon M. Aytac[1], Lauren L. Beghini[1], Gregory Davis[5], Joseph Donato[4], Noah Evans[1], Sandra L. Frost[4], Thuc Hoang[7], Ignacio Laguna[3], Kirk T. Landin[2], Randy R. Lober[1], Jackson Mayo[1], Karla Vanessa Morris Wright[1], James Peltz[7] Alessandro Pinto[5], Tarun Prabhu[4], Blake C. Rawlings[1], Sabrina Reis[3], Daniel W. Shevitz[4], Sina Sontowski[4], Matthew J. Sottile[3] George Stelle[4], Heidi K. Thornquist[2]

[1] Sandia National Laboratories, Livermore, CA
[2] Sandia National Laboratories, Albuquerque, NM
[3] Lawrence Livermore National Laboratory, Livermore, CA
[4] Los Alamos National Laboratory, Los Alamos, NM
[5] NASA Jet Propulsion Laboratory, Pasadena, CA
[6] Cornell University, Ithaca, NY
[7] National Nuclear Security Administration, Washington, D.C.

SAND2024-02142

ABSTRACT

The First Tri-Lab Workshop on Formal Verification was held in Santa Fe, New Mexico, on December 5th, 2023. This workshop gathered staff from Sandia, Los Alamos, and Lawrence Livermore National Laboratories and NASA's Jet Propulsion Laboratory. This report summarizes and expands on the presentations given and discussion had at this workshop.

In this report, we describe the current capabilities and research needs related to formal methods at the NNSA labs. In particular, we identify medium-term and long-term research gaps in programming languages, formalization efforts of complex systems, embedded systems verification, hardware verification, cybersecurity, formal methods usability, workflows, numerical methods, the use of formal methods for artificial intelligence (and its converse, artificial intelligence for formal methods), and collaboration opportunities and considerations on these topics. We conclude with a small number of exemplar research problems related to these topics.

Acknowledgments

We acknowledge the Advanced Simulation and Computing program at the National Nuclear Security Administration for funding the workshop and the preparation of this report.

This report was written by the tri-lab workshop attendees based on notes, presentations, and discussions from the workshop as well as correspondence with workshop attendees before and after the workshop.

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA-0003525.

This page intentionally left blank.

CONTENTS

LIST OF FIGURES

Glossary of Terms and Acronyms

AI/ML Artificial intelligence/machine learning

API Application programming interface

ASC Advanced Simulation and Computing

AST Abstract syntax tree

CAN Bus Controller area network bus

CEA-List French Alternative Energies and Atomic Energy Commission - Laboratory for Integration of Systems and Technology

Coq/Rocq The Coq proof assistant [119]. Coq is planned to be renamed Rocq sometime in 2024.

CRADA Cooperative Research and Development Agreement

CUDA NVIDIA's parallel computing API and language; no longer an acronym

DARPA Defense Advanced Research Projects Agency

DOE Department of Energy

DSL Domain-specific language

GCC GNU Compiler Collection

GPU Graphics Processing Unit

HPC High-performance computing

ISA Instruction set architecture

I/O Input/output

IT Information technology

FFRDC Federally-funded research and development center

FM Formal methods

FPGA Field-programmable gate array

JOWOG Joint working group

JPL National Aeronautics and Space Administration Jet Propulsion Laboratory

LANL Los Alamos National Laboratory

LLM Large language models

LLVM Compiler and toolchain for multiple programming languages; no longer an acronym

LLNL Lawrence Livermore National Laboratory

MBSD Model-based system design

MPI Message Passing Interface

NNSA National Nuclear Security Administration

OpenMP Open Multiprocessing; an API for shared-memory parallel programming

OS Operating system

OT Operational technology

PPL Probabilistic programming language

PQC Post-quantum cryptography

PROVERS Pipelined Reasoning of Verifiers Enabling Robust Systems; DARPA project [81]

PSAAP Predictive Science Academic Alliance Program

RISC-V An open standard ISA

RTL Register transfer language

SMT Satisfiability modulo theories

SNL Sandia National Laboratories

TCB Trusted computing base

TCP/IP Transmission Control Protocol/Internet Protocol; also known as the Internet protocol suite

TLA Temporal Logic of Actions; TLA+ is its corresponding specification language

VST Verified Software Toolchain

This page intentionally left blank.

1. INTRODUCTION

This report describes the current state of formal methods research at the three National Nuclear Security Administration (NNSA) labs: Sandia National Laboratories, Lawrence Livermore National Laboratory, and Los Alamos National Laboratory, referred to as the *tri-labs*. In 2023, researchers at the tri-labs, with motivation from leadership at the Advanced Simulation and Computing (ASC) program office of the NNSA, identified the need to build a community around formal methods. In December, ASC funded a workshop where tri-labs researchers came together and shared their experiences at their respective labs, then discussed future research and collaboration opportunities. This report is a result of a compilation of the key ideas outlined in that workshop.

1.1. Background

Since this report is meant for a general formal methods audience that may not be familiar with the various government organizations, we provide some context. The tri-labs are Federally-Funded Research and Development Centers (FFRDCs) that are primarily funded by the NNSA. The NNSA is a semi-autonomous agency under the Department of Energy (DOE), which is responsible for the development and management of the United States nuclear weapons stockpile. What would later become the ASC program was founded in 1994 to address the need for nuclear stockpile stewardship after the United States' moratorium on live nuclear tests. 30 years later, ASC is responsible for many high-performance computing and predictive science capabilities and achieves this through modeling, simulation, verification & validation, computing infrastructure research, and industry and academic collaborations [36].

The position of this report is that formal methods can provide a multifaceted benefit for *predictive sciences*. Using formal methods, digital systems can—with proper rigor and effort—be *completely* predicted with respect to a formal specification. But even for systems whose scale is sufficiently large such that full specification and verification are not currently tractable, formal methods can improve developer productivity by eliminating classes of errors that can consume large amounts of developer time. We use the analogy of formal methods as "guard rails" to suggest that formal methods can add additional safety to software systems by preventing developers from making costly, albeit common, errors.

1.2. Overview of Formal Methods

Formal methods (also known as formal verification) as a field relates to the development of mathematical reasoning about computer systems. In general, formal methods can be partitioned into two broad categories: formal specification and formal verification. The first refers to generating precise,

unambiguous, and machine-checkable properties about systems and the second refers to verifying that these properties hold for a given system.

1.2.1. What Formal Methods Provide

Formal methods can provide the strongest possible mathematical guarantees about the behavior of a digital system. However, the caveats of using formal methods are important yet subtle. Properties proved about the behavior of a system are only with respect to a specification. Formal methods facilitate making claims in absolute terms such as "a digital system has been proven free of memory errors," but we stress that, like any mathematical proof, these claims only hold provided the required assumptions are met. For example, a claim about a digital system's correctness may not make any statement about timing, leaving systems vulnerable to race conditions. This was the case for the infamous Spectre and Meltdown vulnerabilities, which arose on CPU hardware that had deployed extensive formal verification efforts [46], and bugs in deployed hardware have motivated further industrial investments in formal methods [127]. We mention these caveats here to emphasize that formal methods can only effectively eliminate larger classes of errors if the systems they are modeling both accurately reflect reality and are designed such that proofs are tractable.

1.2.2. Formal Methods Development

To achieve these guarantees about digital systems, formal verification efforts generally require more up-front development. As practitioners and researchers in formal methods, it has been our experience that this up-front development effort pays dividends: formal methods can not only provide increased assurance of digital systems but can also provide guard-rails that allow developers to feel safe in making various design decisions knowing a given tool disallows certain classes of bugs. The cost of this up-front development is one of several reasons why formal methods have, historically, not been as widely adopted as other techniques for ensuring the correctness of computer software, such as testing, version control, and reproducibility through virtualization or containerization. Despite these limitations, applications where the cost of errors is dramatically high (e.g., mission-critical systems) have driven investments in formal methods, and as the field has advanced alongside computer science, its concepts have spread to other fields such as compilers and hardware design. There remain challenges and barriers to broader adoption, which we also discuss in this report.

1.2.3. Trusting Formal Methods

One natural question that arises when using computers to reason about computers themselves is: how can we trust a machine-checked proof? This issue is of great importance in the field of formal methods, and any good computer system for generating proofs will make an argument for its correctness. One solution is to have the core logic of a proof system be so simple it can be manually verified. This is called the *de Bruijn* criterion [47].

Formal methods tools also consider their own *trusted computing bases* (TCBs). In tools which make this distinction, bugs outside of the TCB do not affect the overall trust in system itself. However, developing the trusted/untrusted distinction is a design decision which not all formal methods tools develop. There is not consensus across the entire verification community regarding the acceptable size

of the trusted code base or trusted kernel of a prover, but trends appear to favor the smaller kernel approach as found in tools like Coq, Lean, and relatives. A detailed analysis of the degree to which formal methods techniques address the de Bruijn criterion or limit their TCBs is beyond the scope of this report but is discussed in detail by Pollack [96] and Ringer et al. [103].

1.2.4. Formal Methods at NNSA

The NNSA tri-labs are uniquely positioned to deploy formal methods in the design, development, and maintenance of digital systems. Sandia National Laboratories has two departments (Digital Foundations & Mathematics 1 and 2) specializing in formal methods, Lawrence Livermore National Lab (LLNL) has formal methods capabilities through its ROSE compiler team, and Los Alamos National Laboratory (LANL) has research scientists who specialize in compilers and cybersecurity who have exposure to formal methods techniques.

1.3. Structure of this Report

This report aims to outline the state of formal methods research within the tri-labs. We begin by broadly describing the current capabilities at each lab and the research areas unique to each lab. We then describe future research needs, potential research opportunities, and challenges we anticipate in these domains. The report concludes with a collection of exemplar problems and research topics that are either directly relevant to tri-labs needs or are prove problems that effectively model lab needs.

While this report focuses on the three NNSA labs (Sandia, LLNL, and LANL), the tri-labs collaborate with universities and other FFRDCs. The positions expressed here are those of the tri-labs, but may apply generally to formal methods researchers in other fields.

This page intentionally left blank.

2. CURRENT CAPABILITIES

2.1. Sandia National Laboratories

Sandia has two departments consisting of about 20 research scientists who work on formal methods [31] and contribute to other verification-focused projects throughout the labs. Sandia focuses on embedded system verification but also supports various verification efforts which we outline in this section.

Much of the formal methods effort at Sandia works towards the notion of "One Q.E.D." This refers to the goal of having a proof of correctness for each stage of the system design stack and that each proof feeds into the next level of fidelity. For example, a proof about full system correctness may consist of a proof of high-level properties, (such as safety or liveness, or a description of expected behavior), a proof that a C program implements those high-level properties, a proof that the compiled binary has the same behavior as the C program, and a proof that the hardware executing the binary correctly implements its assembly language. For this to become a reality, the reasoning techniques at each level of abstraction must link with each other.

While One Q.E.D. as a term was coined at Sandia, other researchers have worked towards this goal, notably the DeepSpec project [5]. Figure 2-1 shows some of the steps the tri-labs take towards full proofs of correctness.

2.1.1. Low-Level Systems Verification

To date, one of the largest-scale formal verification efforts at Sandia has been the full verification of a bootloader used in a high-consequence system. Verification of the bootloader demonstrated Sandia's proof engineering capabilities; the project involved developing verified compilation techniques, and used the Coq proof assistant [119] and the Verified Software Toolchain (VST) [3] to develop formal proofs of correctness of real C programs.

This bootloader demonstrates significant steps towards the goal of One Q.E.D. Beyond the actual proof development required for the verification, verifying the bootloader required advances in several areas; namely: cryptography, proof assistants, compilers, and low-level assembly verification, which we describe below.

2.1.2. Formally-Verified Secure Cryptographic Root-of-Trust

To fulfill the always/never requirements at the core of the NNSA mission, high-consequence systems must have some means of establishing the authenticity of their software and hardware; without such guarantees, any proofs about system behavior are meaningless. Cryptographic systems can implement such authentication, but may themselves be the source of vulnerabilities. To that end, Sandia has developed what is possibly the world's only secure boot with an end-to-end proof of security against

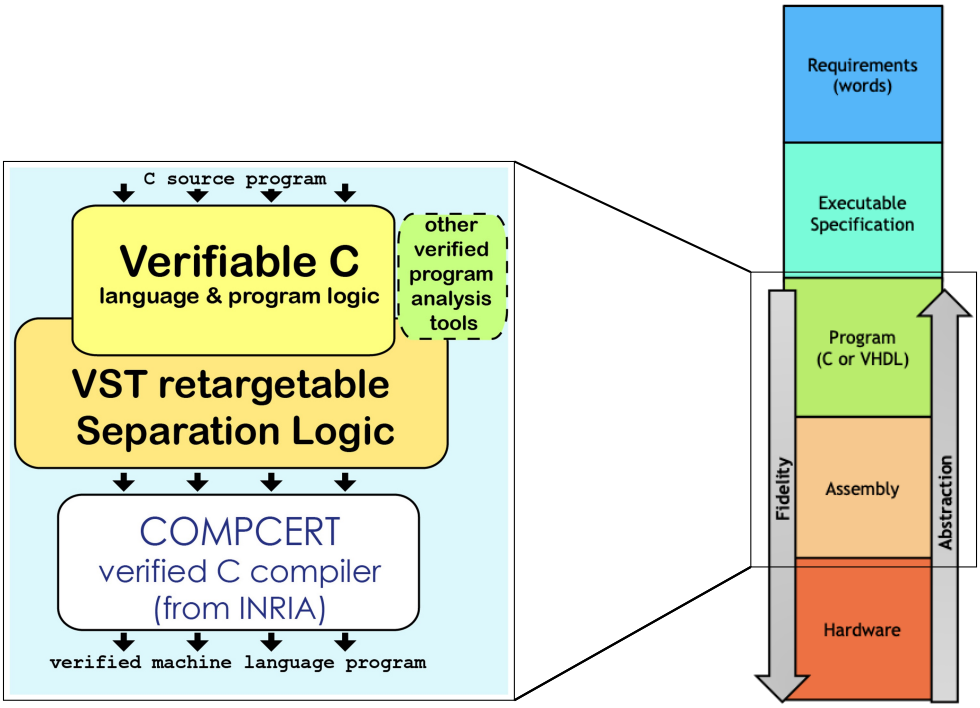

Figure 2-1.: Two steps of One Q.E.D.: from C code to assembly language. Left figure from Pierce [94].

adversaries equipped with probabilistic Turing machines. Here, by end-to-end, we mean the assembly is soundly approximated by a specification of the root of trust as a temporal protocol, including its elliptic curve operations, expressed in a formalization of elliptic curves over arbitrary fields developed in-house expressly for this purpose. However, security of these elliptic curve operations do not hold for sufficiently powerful quantum computers. We describe this future research direction further in Section 3.7.

2.1.3. Constructive Proofs of C Code Verification

Some programs are well-specified as pure (side-effect-free) functional programs. However, the programs embedded in high consequence systems are often reactive systems, specified in terms of their observable side-effects—specifically, their allowed I/O behavior. Existing tooling leaves a gap, either in soundness, expressivity, ergonomics, or all of the above, for such reactive systems.

One challenge for Sandia regarded specifications of the mathematics of cryptographic primitives for a high-consequence bootloader. Desired properties could not be expressed in the specification languages of automated program logics like Frama-C [61]. Moreover, Frama-C comes without any proof of soundness. For these reasons, Sandia used Appel's Verified Software Toolchain (VST) [3] to provide fully-constructive, sound guarantees of this bootloader.

VST implements a separation logic for C programs in Coq, and is proven sound with respect to the CompCert C semantics. However, separation logics are designed for proving the equivalence of imperative programs with respect to pure functional specifications.

To express specifications of the allowed I/O behaviors of embedded software controllers, SNL scientists wrote specifications in an extension of VST using recently-developed interaction trees [126], a Coq library for formalizing the trees of allowed observable behaviors of effectful programs. Here, SNL again faced limitations, both in expressivity of the specification language, in its accompanying proof theory, and in the ergonomics of proof engineering.

To give some context to this difficulty and how Sandia scientists overcame it, we first remark on the design process for high-consequence systems at Sandia. Because of the complexity of these systems, designers coordinate by producing and sharing *intermediate* specifications which deliberately include nondeterminism; this permits a separation of concerns between overall system-level design (such as how systems interact) and implementation details (such as hardware interaction and embedded system programming). However, interaction trees do not handle nondeterminism, nor does VST; to support this design paradigm while maintaining full-system verification, Sandia scientists modified existing utilities to match Sandia's design process, a process which was bespoke and required a deep understanding of both the systems being verified and the utilities themselves. The result of this effort was an end-to-end, fully-constructive proof of correctness of the high-consequence system, but this work also demonstrated the extensive effort currently required for such assurance.

2.1.4. Formally Verified Compilers

Sandia maintains an internal branch of CompCert, the formally verified compiler for C [72]. CompCert is written in Coq, and its proof of semantic preservation between source language (C) and target (several different architectures) is fully mechanized. Sandia develops formally-proven improvements to this internal version of CompCert and different back-ends to address mission needs. CompCert is structured as a collection of transformations between C and assembly, with proofs of correctness between each intermediate language in addition to proofs for every transformation or optimization. We show the overall architecture of CompCert and Sandia's extensions to it in Figure 2-2.

2.1.5. Q Framework: Formal MBSD, State Machine Verification, and Refinement of C Code Implementations

The Q framework is another large verification effort developed and maintained by formal methods researchers at Sandia. The Q Framework is an internal tool used by the labs for two main purposes: to provide model-based system design (MBSD) verification via model checking of temporal properties, and to prove that C implementations refine a given state machine model [98]. Q framework can be used to verify system designs at multiple levels of abstraction, and has been used to analyze systems with hundreds of states. An overview of the Q Framework is shown in Figure 2-3. Q Framework is used for several projects at Sandia and provides a language and compiler with which to carry out various temporal, state machine, and embedded system verification tasks.

Figure 2-4 illustrates a simple parallel composition of state machines along with the environment and properties of the overall system. For example, the statement in Linear Temporal Logic, $AG\ (= n\ m)$, asserts the two counters n and m are always equal.

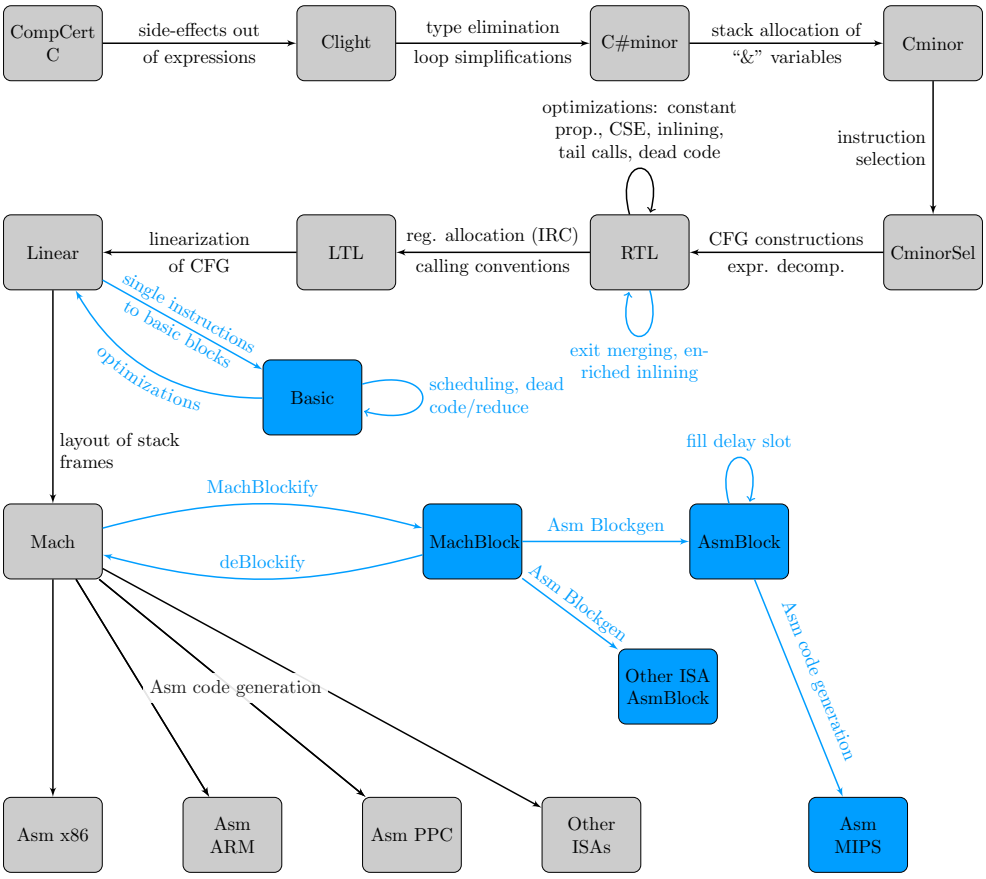

Figure 2-2.: Architecture of CompCert. The uncolored boxes represent existing transformations and the blue boxes indicate Sandia development and improvement. Each arrow consists of a transformation along with a proof of correctness in Coq.

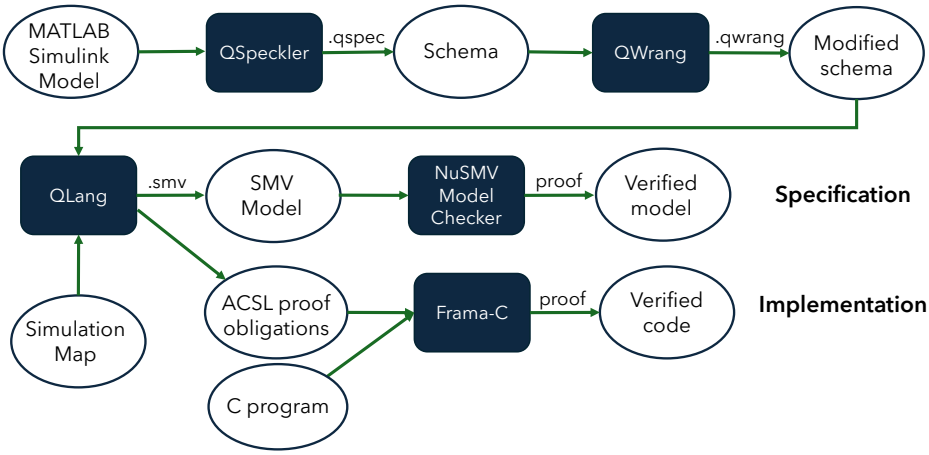

Figure 2-3.: Overview of Q Framework. More details available from Pollard et al [98].

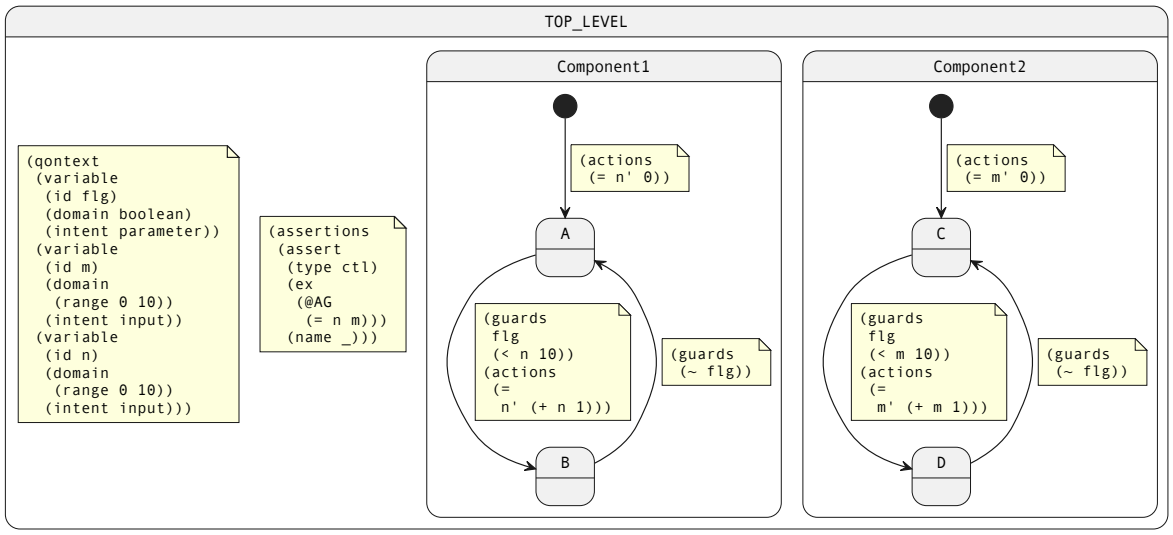

Figure 2-4.: Synchronous composition of two state machines. The leftmost note indicates the context of the system, the middle note indicates a temporal property, stated as an assertion to be checked in Linear Temporal Logic [13] and the rightmost boxes indicate the two state machines.

2.1.6. Probabilistic Programming Languages

Sandia also supports efforts toward probabilistic programming language research, which has applications in embedded system verification, AI/ML, threat modeling, and cybersecurity.

2.1.6.1. Embedded Inference

Inference of embedded systems is motivated by a need for extreme robustness in the face of uncertainty, for example from faults in a communication bus. Probabilistic programming languages (PPLs) serve as an intuitive language for specifying programs with random variables. Sandia scientists are investigating applications of PPLs to build better techniques for sampling and inference—which are at their core algorithmic processes—while providing useful abstractions for construction of stochastic models to domain experts.

Using PPLs, deployed systems with known uncertainties can be more completely modeled and queried. For example, in Figure 2-4, `flg` could be made a random variable. Sandia scientists are developing these classes of analysis, begun by first extending the PPL Dice [48] to work over arbitrary semirings. This extension allowed scientists to use Dice to perform a sensitivity analysis providing insight into those physical upsets to which our temporal surety properties are most sensitive.

Using an analogy between the drawing of fresh samples and the dynamic allocation of fresh mutable state, Sandia scientists and their academic partners have additionally developed first-order [77] and higher-order [78] modal separation logics for probabilistic programs over continuous random variables.

2.1.6.2. Game-based Security Proofs and Cryptography

By presidential mandate, future high consequence systems must be quantum resistant [120]. But quantum-resistant algorithms are still in flux; in the past years, new cryptographic attacks were discovered for every NIST-approved PQC algorithm family, dramatically reducing their security guarantees. Even the security proof for Sphincs+, whose security was thought to be well understood, was shown to suffer from flawed reasoning about conditional probability [51].

Security properties of quantum cryptosystems differ from traditional safety and liveness properties which can be proved using the techniques mentioned in the previous subsections. Instead,systems must have a game-theoretic property called indistinguishability under chosen plain-text attacks (written IND-CPA). Sandia scientists are researching how to verify such systems by formulating IND-CPA as an observational equivalence of parameterized probabilistic programs, which requires higher-order program logics for PPLs.

2.1.7. Concurrency and Distributed Systems

Reasoning about concurrency can be challenging: concurrent systems are difficult for humans to understand and verify and automated reasoning techniques such as symbolic execution have scalability limitations for concurrent systems. In general, the underlying reason is that every operation causes an additional potential interleaving between concurrent execution threads, making the state space of behaviors exponential in the number of operations. While this *state explosion problem* is not unique to concurrency, traditional techniques from formal methods that are used to manage the problem tend to not work well on concurrent systems. To this end, Sandia scientists are researching ways to formalize concurrency semantics and memory models to better analyze concurrent systems.

Distributed systems are different from concurrent systems in that they lack *synchronicity*: fewer assumptions can be made on the timeliness or integrity of messages sent between distributed systems (for example the property that m and n are always equal fails in Figure 2-4 with asynchronicity). At Sandia, tools such as Q Framework (Section 2.1.5) are being extended to better support more types of distributed systems and more classes of composition between interacting systems.

2.1.8. Program and Reactive Synthesis

Reactive synthesis [41, 95] is an approach to automatically construct transition system models that are guaranteed to meet formal requirements. As opposed to other verification approaches such as model checking, synthesis aims to produce an artifact that provably embodies required properties without the need for subsequent verification, thus eliminating the edit-check-debug cycle common in model checking approaches. Reactive synthesis shares similar aims to correct-by-construction, but instead of proving each step in a refinement chain or derivation, program synthesis relies on proofs of the refinement transformations [91], which need only be proven once. The correctness of the synthesis then follows from the chaining together of such transformations.

Several challenging problems have been synthesized at Sandia, such as flow control [112], multi-buffer flow control modeled as two-player games [115], synchronization and concurrency problems [88], and as well as nesting and concurrent composition, modeled as Q Framework nested state charts [88].

2.1.9. Formalized Numerics

Floating-point computations on computers are ubiquitous, so much so that computers are measured by their FLOPS, or floating-point operations per second. Members of Sandia's formal methods team are researching ways to provide formal proofs bounding the floating-point round-off error of numerical programs used in high-consequence settings.

Scientists at Sandia have identified applications that require formally verified numerical kernels and are investigating techniques to automate the verification. One example program is a C implementation of a Kalman Filter, which requires verifying a diverse code base . This example is described in more detail in Section 4.5.

Recent work done in collaboration with research groups at Princeton and Cornell has resulted in advances in formal proofs of error bounds for numerical kernels [58, 59, 60]. Ongoing research involves integrating these formal proofs into verified system design stacks.

2.1.10. Hardware Design and Verification

SNL uses Yosys [125] and Kôika for its hardware verification tasks. Kôika is a hardware description language developed at MIT [19], which provides formal guarantees at the SystemVerilog level (similar to BlueSpec SystemVerilog [89]), while Yosys provides synthesis and model checking capabilities for RTL. These together bring the tri-labs one step closer to the goal of One Q.E.D.

Sandia also maintains Xyce [57], a large-scale analog circuit simulator that models hardware under various conditions. Xyce models are used at Sandia for hardware verification and validation. Further work has been done relating out-of-nominal hardware verification with abstraction, for example in cases with unexpected hardware bit flips [83].

2.1.11. HPC Correctness

Several of the formal methods researchers at Sandia started with the verification and validation of large-scale HPC codes, for example resiliency for large parallel systems [43, 64]. Recent work supports the correctness of HPC libraries such as Kokkos through automated test generation, symbolic execution using KLEE [21], and building formal models of parallel programming environments [54, 118].

2.1.12. Cybersecurity

Sandia scientists have developed formal theories for cryptographic operations along the path for verification of high-consequence systems, for example with root-of-trust. Sandia has also started using Cryptol [74] for the verification of cryptographic primitives. More cybersecurity-relevant work at Sandia was also mentioned in Section 2.1.6.2.

2.1.13. Static and Binary Analysis

Mission needs at SNL sometimes require the analysis of old hardware architectures not well-supported on other symbolic execution tools such as angr [110]. For example, systems using non-IEEE-754

floating-point arithmetic or systems without 8-bit bytes. Sandia has invested in capability for binary lifting and symbolic analysis for a wide variety of architectures, both with purpose-built capabilities and a more general tool called Quameleon [99]. Other work on symbolic execution with KLEE and Kokkos was previously mentioned in Section 2.1.11.

In cases where fully-formal models are infeasible (for example, large codebases or programming languages without fully-formal specifications), scientists at SNL have developed internal tools to perform static analysis at both the binary and source-level [29].

2.1.14. User-Centered Formal Methods

For formal methods to make a broad impact, there needs to be an additional focus on adoption. This requires consideration of and research on the usability and usefulness of these tools in the context of system designers instead of formal methods experts. Drawing from research in human factors, cognitive science, and user experience, Sandia is now pursuing tasks to investigate different users' needs, expectations, and limitations. These insights are being used to drive requirements and to design implementations of formal methods tooling. Additionally, telemetry-based metrics that improve design based on real-time usage data and can built into software tools are being developed by Sandia scientists.

2.2. Lawrence Livermore National Laboratory

Lawrence Livermore lab supports various formal methods research, focusing primarily on static analysis using the ROSE compiler [100] and HPC correctness.

2.2.1. The ROSE Compiler

The ROSE compiler is a source-to-source compiler written in C++ [100]. The main benefit of ROSE over traditional compiler analysis tools, such as LLVM-based tools, is that the abstract syntax tree (AST) is mutable. This allows more robust analysis and from whence the source-to-source characterization arises. This allows not only program analysis, but opportunities for optimization and parallelization. ROSE has been used to perform static and dynamic analysis on C++ codebases as well as other programming languages such as Fortran, OpenMP, Java, Python, and binary analysis.

2.2.2. HPC Correctness

Livermore Computing has some of the world's largest supercomputers and a large workforce devoted to the research, development, and effective use of these systems. Work by LLNL scientists focuses on various aspects of correctness for these large HPC codebases executing on these world-class supercomputers.

Two widely used programming paradigms in HPC are the Message Passing Interface (MPI) and OpenMP. LLNL scientists support a combination of static and dynamic analysis to analyze OpenMP programs [7]. Beyond the parallelism libraries, whole-code HPC debugging techniques have been investigated at LLNL [2].

HPC correctness is closely tied to numerics, and in particular, floating point. In theory, compilers should faithfully translate source semantics (e.g., C++) into target semantics (e.g., x86-64 executables), however, compilers can have bugs, or in the quest for ultimate performance, compilers may introduce numerical inconsistencies, especially for heterogeneous code such as those using both CPUs and GPUs. In particular, floating-point numerical exceptions and accuracy issues have also been investigated [85, 66], automated detection of sources for large numerical error [107], including for accelerator architectures like GPUs and heterogeneous architectures [79, 65].

2.2.3. Embedded Systems and Usability

LLNL researchers in have recently begun investigating embedded system verification, formal modeling, Automated Systems Understanding (ASU), and formal methods usability.

2.3. Los Alamos National Laboratory

Los Alamos National Laboratory scientists maintain a fork of the LLVM compiler called Kitsune [68] focused on optimization and code generation for large parallel codes. This, along with other compiler efforts require reasoning about software correctness and could effectively leverage formal methods techniques.

As developers and maintainers of large HPC codebases, LANL scientists research ways to improve scalability and code generation of HPC codes. Oftentimes, HPC codes require manual rewriting and porting to new architectures. Programming paradigms such as fork-join with OpenMP or task-based parallelism have the opportunity to improve performance while also making code more performance-portable Performance portable refers to programs to not just execute on next-generation hardware, but can make effective use of new hardware capabilities. However, there are challenges in getting these libraries to achieve the same performance and feature set of hand-written C++ and accelerator languages (such as OpenMP or CUDA). To address this, LANL scientists have developed LLVM Intermediate Representations (IRs) which can be linked with OpenMP and provide more opportunities for analysis and optimization [117]. LANL scientists have also worked towards developing implicit parallel task-based programming models [116] as well as better task-based parallel programming models [69]. These techniques could benefit from investment in more programming language research to more effectively mechanize and verify transformations and optimizations and multiple levels of abstraction.

LANL scientists also work in the cybersecurity space, with research into the formally verified microkernel, Sel4 [63], drivers, usability, and interoperability with other operating systems (OS) and devices, and how security guarantees are ensured across these interfaces.

2.4. NASA Jet Propulsion Laboratory

NASA mission applications undergo extensive testing and formal verification. In particular, the Autonomy Assurance program at JPL requires a higher level of safety and assurance compared to other autonomous systems such as automobiles.

JPL has historically developed state space abstraction, test generation, rule-based plan validation, model transformation, and test case generation [37]. Other research areas include fault diagnosis [90], System-Theoretic Process Analysis (STPA) [114], and assurance for AI/ML [104], and collaboration with Caltech and the University of Southern California in the areas of test synthesis, model-based design, and requirement modeling.

Overall, JPL scientists focus on full life-cycle assurance and towards developing high-assurance autonomous systems. JPL research, along with tri-labs, would benefit from improvements in FM tooling, in particular with requirements specification, usability, and scalability.

2.5. Other FFRDCs

This workshop focused on the tri-lab workshop and the interest of the NNSA in developing further connections amongst themselves. However, we note other Department of Energy labs, including Oak Ridge National Laboratory, Lawrence Berkeley National Laboratory and Pacific Northwest National Laboratory are investigating how to integrate formal methods into their work, such as applications of formal methods to ensure the cybersecurity of microcontroller and FPGA-based devices [122]. However, beyond these broad overviews, the work of other FFRDCs is beyond the scope of this report and we do not claim to know all other governmental efforts in formal verification.

3. FUTURE NEEDS AND RESEARCH OPPORTUNITIES

The NNSA is gathering information related to the deployment of formal methods at the tri-labs. This information includes research gaps and opportunities, with a particular emphasis on potential applications in the areas of artificial intelligence and machine learning. This chapter identifies research gaps and opportunities that scientists at the tri-labs believe will be important to NNSA mission needs in the next several years.

The tri-labs require diverse formal methods capabilities, including fully-formal proofs for the highest-consequence applications, deductive proofs for high-consequence but ancillary routines (e.g., for external libraries), and semi-formal methods that provide added assurance in cases where full formal verification is infeasible (e.g., in codebases that use languages with limited support for formal reasoning, as discussed in Section 3.4.1.3).

The credibility of tools used for formal verification is an important consideration. Put another way, if a proof is generated using a particular tool, trusting this proof in a high-consequence environment requires some thought. When using tools like the Coq proof assistant, trusting a proof can be reduced to trusting the implementation of the tool's proof checker[1] However, specifications may be incorrect or unrealistic (see Section 1.2.3), which limits the practicality of the proofs about them. The tri-labs are therefore interested in working towards enhancing the credibility of formal methods tools. This work includes counterexample generation, and developing explanations of formal results and specifications through principled documentation or publication.

The key challenge in getting formal methods adopted at the labs is usability; without consistent investment in documentation and usability, formal methods tools have the danger of remaining arcane tools used by only a few experts. This may be acceptable to meet mission assurance in some cases, but usability is critical for wider adoption throughout the labs and in the industry in general. Usability can be addressed with more involvement in human factors engineering, publications and documentation, and maintenance of software, as well as research into lightweight formal methods tools.

Tri-lab scientists are aware of many different research areas important to NNSA mission needs where formal methods can have a significant impact. While Sandia already has a robust formal methods department, LLNL and LANL are interested in expanding and collaborating in the formal methods space. We list some of the topics here and further expand on them later in this chapter.

Particularly relevant research areas are: provable cyber assurance, formal numerical methods, certified cryptography, static analysis for multiple languages, scalability of automated analysis, certification of formal methods results, challenges of formal specification and what language verification results are proved against, for example, how general-purpose specification languages such as Coq compare to domain-specific languages with respect to their usability and scalability, correctness error analysis of

[1] This is the de Bruijn criterion mentioned in Section 1.2.

numerical codes, especially in HPC systems, compiler verification, cyber-physical systems, security, operating systems, co-design of hardware accelerators and automatic verification, and improvements towards hardware verification tooling, AI/LLM code generation and correctness arguments for the code generated, programming languages tools and how the scale of these tools must be different for different codebases (e.g., how systems with thousands of lines of code versus systems with millions are managed), numerical reasoning and floating-point analysis, and in general lowering the barrier of entry to using formal methods tools.

3.1. Abstraction and Refinement

Many systems the NNSA labs are tasked with verifying are sufficiently complex so that verifying even a single component is intractable. Thus we rely on the notion of refinement. Using abstraction and refinement has the added benefit of mapping developer intent and compartmentalization of concerns into a formal model (put more concretely, engineers reason and design at various levels of abstraction, and verification analysts can leverage these structures to build better models of the systems). For example, Figure 3-1 shows a high-level overview of a model-based system design (MBSD) used for an example embedded controller.

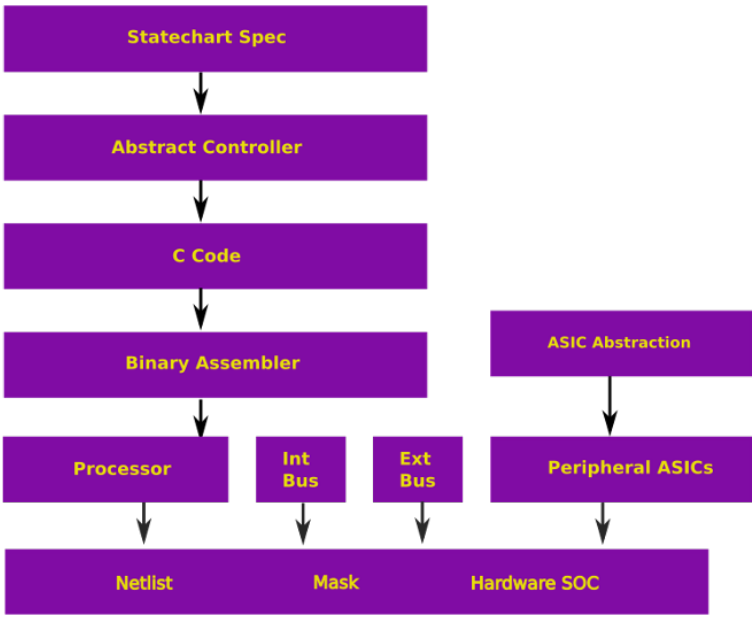

Figure 3-1.: Analysis stack for a MBSD embedded controller.

Between each layer, from statechart specification down to hardware netlist, a full verification workflow would build a refinement relation between each abstraction level. Note this figure looks similar to Figure 2-1, and is similar in spirit, however a refinement relation is slightly different; with refinement, properties proved about the abstract model hold in its refinement. Our current workflow (using Q Framework) accomplishes some of these refinement arguments, however the story is not complete.

Future work with abstraction/refinement would benefit from models of hardware, netlists, bus protocols, and processors, compatible with state chart models.

Beyond this, the notion of refinement is not a single concept: state charts have different notions of refinement and composition. Some models of composition do not generalize to all types of properties proved. In particular, we are interested in better formal models of contextual refinement, parallel asynchronous composition, and nested composition.

3.2. Concurrency and Distributed Systems

Verification of commonly-used protocols has the potential for high-impact because of their potentially wide-ranging use. Furthermore, distributed or concurrent systems are difficult to reason about; security bugs can remain latent for years in even well-tested and trusted protocols [32]. Example distributed systems (and their respective implementations) that would benefit from both high-level modeling as well as implementation verification include communication protocols such as SSH. State-machine based analysis has been used to analyze chat protocols such as WebRTC [111].

In addition, protocols used for embedded systems are of particular interest to the labs. Examples include Universal Asynchronous Receiver-Transmitter (UART), Inter-Integrated Circuit (I^2C), and the CAN bus protocol. We use the latter as an exemplar for formal modeling in Section 4.6.

In addition to communication protocols, in the field of operating systems (OSes) real-time operating systems (RTOS) schedulers are used in many mission-critical applications. Beyond this, device drivers and other network protocols can be difficult to analyze because of their complexity and temporal requirements.

3.3. High-Performance Computing

The tri-labs maintain HPC software libraries and codebases too numerous to list here. These codebases have diverse use cases and needs. One challenge with HPC is porting these codebases to new supercomputers. In particular, the shift towards accelerators like GPUs often require significant rewrites, so much so that re-framing problems in a data-parallel paradigm is its own area of research. While the labs maintain libraries such as Kokkos and RAJA for more easily writing performance-portable code, there remain correctness concerns when such large refactoring is done.

One example of this challenge are the subtle differences between memory models and synchronization primitives for different GPU architectures. These differences may not arise until code is ported. Not only this, but these bugs can cause nondeterministic errors and thus hard to detect and debug. One way to address this is to better formalize memory models for different GPU ISAs, such as OneAPI and PTX. Challenges with proprietary architectures could stymie progress for an individual architecture. However, even without vendor buy-in, the development of sufficiently weak memory models (e.g., for Kokkos, which targets many different parallel programming libraries) could result in better assurance for many different architectures. The drawback is potential unnecessary synchronization, but whether this causes significant performance degradation remains to be seen.

Beyond memory models, many parallel programming systems could benefit from better formalization. There have been efforts towards writing semantics of the Message Passing Interface (MPI) [75], but models of different parallel-programming paradigms [108] would also be useful to have formal models. One important caveat here is what we mean by *formalization*. For example, some formalization efforts

consist of building a model in a language such as TLA+ [67]. This can be used to reason about high-level properties such as the correctness of a distributed protocol. While this still permits implementation bugs (unlike, for example, a Coq proof of correctness, from which a verified implementation can be automatically extracted). However, a larger problem is the usability of high-level formalizations: it is not generally possible to modularize TLA+ models and use them for anything else. Therefore, any formalization efforts, especially for HPC codebases, should consider their ability to be connected with compilers and codebases directly. This is a challenging and unsolved problem.

For HPC codes, it is hard to overstate the importance of compilers. Tuning code to a particular architecture and compiler combination can sometimes result in orders of magnitude better performance, and other times compiler flags (in particular `fast-math` optimizations) can change the solution. Not only this, but modern compilers contain hundreds of optimizations, and the majority of compiler bugs reported relate to mis-optimization [131]. Not only do the labs need better optimization, but they also need better, provably correct optimization for parallel codes. There has been some work in this space [130, 80], but the labs would benefit from further research. Recent work with formalizing LLVM [129] in Coq using interaction trees, if used on tri-lab codebases, could permit much stronger correctness efforts while still allowing the flexibility of LLVM.

Another characteristic of HPC codebases at the tri-labs is their large size and long history: many codebases have been maintained and developed for decades. For these codebases, debugging can be challenging as they get ported to new architectures while maintaining rigorous performance and correctness criteria. Further investigation into root-cause analysis of numerical errors and exceptions, as mentioned in Section 2.2.2, could accelerate scientific code development while also increasing surety.

Formalized numerics and domain-specific languages are relevant to HPC applications; we describe them in more detail in Sections 3.10 and 3.5.2, respectively.

3.4. Programming Languages

Verification of software requires many connections between programming languages, their implementation, and formal reasoning tools. These include but are not limited to: formal definitions for the semantics of a language as defined by the language standard; formal definitions of the semantics as implemented by specific compilers; parsing and static analysis of program code; generation of models for code in formal reasoning tools; and compilation toolchains. Our observation is that the degree to which any of these are addressed varies widely by programming language both in terms of availability and robustness. In this section we discuss specific languages and our current assessment of their state with respect to verification activities.

3.4.1. Languages

3.4.1.1. Well-Supported Languages

The C programming language is the best supported in this context thanks in large part to the efforts in the CompCert [72] and Verified Software Toolchain (VST) [3, 4] projects. These projects focused on the needs of high assurance software development aimed at systems-level software, often in the context

of embedded safety critical systems. The key components that lead us to conclude that C is well supported are:

- The existence of a formal, machine-checkable language semantics encoded in a proof assistant.
- The existence of a formally verified compiler.
- The availability of supporting tooling and literature necessary to define and discharge proof obligations needed to establish safety and security verification conditions.

While the C language is one of the best supported with respect to verification methods, there remain gaps to fill to raise the assurance level even higher. We identified the following research directions to address this:

- Establishing assurance cases for the front- and back-ends of the compiler. Much of the current effort has focused on the core IR and algorithms within the compiler, leaving trusted but unverified components at the front-end (lexing and parsing) as well as the back-end (linking, semantics of generated machine code as executed by the CPU).
- Formal models of target hardware. Even with a formal definition of an assembly language we require a formal model of how the assembly is actually executed. Many aspects of processor design are not visible at the assembly level, but have a potentially large impact on the semantics of the program execution (such as out of order and speculative execution).

3.4.1.2. Moderately-Supported Languages

A larger set of languages we classify as moderately supported: some formalization efforts have occurred that can be applied to high assurance software development but there are significant gaps to connect them to a realistic application.

The Why3 tool along with its language for writing specifications and first-order logic reasoning is a key component in many verification processes [40]. The WhyML language can be used directly for specification and reasoning, with some mechanisms to extract executable code for languages like Ocaml. Why3 is used as an intermediate representation by verification tools for other languages like Frama-C [61], SPARK Ada, and others. In those cases, terms in the Why3 language are instantiated directly in the Why3 engine from specifications written in a language closer to the code to verify. For example, the ANSI-C Specification Language (ACSL) [11] is used by Frama-C to define properties of C programs that are then translated to Why3 for subsequent proofs by a back-end SMT solver or proof assistant.

The primary advantage of Why3 is that it provides a bridge between a specification language and different solvers. Recent work between Sandia and Princeton has built a formalization of the core Why3 language [23]. The largest gap that exists for making use of Why3 is connecting code written in a conventional language to Why3. Frama-C and SPARK provide this for C and Ada respectively, although in both cases users are limited to subsets of the language. Less mature but promising efforts are under way to provide a similar approach to Ocaml via Cameleer [92] and Rust via Creusot [30]: neither of these are ready for production application, but have matured beyond handling basic academic test cases.

There has also been work in industry building similar tools to Why3, the most well known being the Dafny [71] system atop the Boogie intermediate verification language [10]. Dafny presents a similar language for expressing specifications and proofs as Why3, but adopts a programming model closer to C# versus the ML-style language used by Why3. It also has more limited prover support, primarily targeting the Z3 SMT solver. Industrial use of Dafny has risen in the last decade: Dafny and Boogie originated as research tools at Microsoft Research, but have been recently seeing active development at Amazon in the Amazon Web Services (AWS) division with applications to production AWS systems.

3.4.1.3. Languages with limited support

Many other programming languages have much more limited support for verification activities. Few programming languages have no formalization efforts at all, but in most cases these efforts have been either academic exercises, restricted to a very specific part of the language, or are out-of-date and no longer track current language standards or implementations. Based on our understanding of use cases within the tri-labs and broader DOE complex, we identified the following languages as needing substantial work to raise the level of formalization and tooling for verification activities.

C++ : C++ is unique in that it is one of the most widely used languages yet has the least robust ecosystem of tools and techniques for formalization and analysis. This is in part due to the complexity of C++: it is a common misconception that C++ is simply a superset of C, where existing C work will map directly to C++. C++ presents a much more complex type system (especially with respect to templated code) and differences in everything from the memory model to how software is modularized and organized. To the knowledge of the authors of this report there does not exist a formalization of C++ that includes features common to production applications such as templates and classes. At best, there exist static analysis tools that are used to check code for properties known to be sources of bugs: but these tools often have not been shown to be sound nor complete. There is substantial work necessary across the board to raise the assurance level possible for C++. Fortunately, such work will have substantial impact due to the sheer volume of C++ that is core to the mission across the DOE/NNSA complex.

Rust : Rust is a recently invented language that aims to provide memory safety features at the language level that are compatible with the needs and constraints of systems level developers. Rust is appealing because the type checking performed by the compiler allows memory safety issues to be caught before compilation: in other languages these would often require runtime assertions or third-party static analysis tools to identify. While the type checking performed by the compiler has proven useful and has improved the quality of systems relative to C and C++ code, Rust lacks a stable language standard, formal language semantics, and other necessary details like a memory model definition [106]. These gaps limit the degree that we can make and prove formal statements about Rust programs. A number of efforts are under way to remedy this. Efforts have been undertaken to define a formal semantics of either subsets of the Rust language, such as RustBelt and Creusot [55, 30], or basic executable semantics, such as KRust [121]. The Ferrocene project [39], attempts to stabilize the Rust language to a particular version, but is not true formalization or standardization effort like ISO C.

Python : Outside the realm of embedded, systems, and high-performance computing, the Python language has a substantial level of adoption for data science, machine learning, and general application development. Python also lacks a robust formal specification that is accepted in the community. A number of academic efforts have occurred to provide a formal semantics for Python, but few have been adopted for any large-scale verification efforts. Recent changes to the language have moved closer to techniques amenable to formalization (such as the introduction of type hints and corresponding type checkers). Formalization for Python is not only necessary for asserting evidence of correctness for Python code, but is also necessary to validate that code transformations and just-in-time compilation schemes produce code that preserves the semantics of the original program. To the knowledge of the report authors, existing Python compilation schemes do not provide strong evidence to support any statements about the correspondence between the original Python and the resulting compiled code.

GPU programming : While not a single language, there are a handful of programming models (CUDA, OpenCL, etc.) that are used to program accelerators common in HPC platforms and increasingly in embedded applications. Little work has been performed in establishing a formalization of these systems. These systems are notable as they require a formalization of the parallel runtime model that the accelerators provide: correctness of a GPU-based program requires reasoning about shared memory concurrency, threading, and data parallel operations. Furthermore, these programming models often co-exist with traditional programming models for the host processor that the accelerator cooperates with. To make matters even worse, trends driven by the needs of machine learning are leading to accelerators that provide features like non-standard or reduced precision floating point, leading to a formalization gap when reasoning about numerical algorithms. Substantial work is required to raise the level of formalization with respect to accelerator programming models.

MATLAB : The final language in heavy use in science and engineering is MATLAB. A number of static analysis and formal reasoning projects have focused on MATLAB in the past, but they are often incomplete. This is in large part due to the opacity of the MATLAB implementation itself: MathWorks does not provide a formal semantics for MATLAB, so third-party tools that require a formal semantic model for MATLAB code are limited. Some tools (such as the Grackle symbolic execution engine from Galois [42]) exist to support verification activities but they are not widely used in the community at this time.

3.4.2. Compilers and language tooling

In addition to language-specific formalization, it is useful to consider the tooling that support compilation and static analysis. LLVM has become a standard tool across the computing community for languages work. Unfortunately, LLVM has limited existing formal semantics for the LLVM IR that all LLVM-based compilers, compiler optimizations, and back-ends must speak. Some efforts have been made to formalize optimizations (e.g., the Alive project [80]) but are currently limited with respect to proofs of correctness when considering composed optimizations and the overall compilation process. LLVM bitcode has been used in some formal verification contexts (e.g., the Galois Crucible symbolic execution engine [8]), so the level of production-level tooling is slowly rising. Additional research effort should be supported to provide a more holistic formalization of the entire LLVM project. This will allow stronger assurance statements to be made about compilers and tools based on LLVM.

3.4.3. Assembly and architectures

The lowest level of abstraction commonly used in computing before hardware is assembly languages. These encompass traditional assembly languages (X86, ARM, etc) as well as bytecode-based assemblies for virtual machines (JVM, .NET). Current CPUs in widespread use unfortunately provide limited (if any) formal semantics for the actual execution of assembly code. For example, while the semantics of an individual X86 instruction may be established for an abstract X86 processor, the actual semantics with respect to details like speculative execution, executed microcode instructions, interactions with caches and cache coherence protocols, and so on, are all opaque to end users. This has spurred significant interest in open architectures like RISC-V where the semantics can be defined all the way down to the hardware implementation. There has been promising work on building a machine interpretable formal semantics for RISC-V [45] (mentioned further in Section 4.1), but we still have a gap when it comes to semantics for other architectures in widespread use. There have been efforts to formalize x86 as early as 2004 with VeryPCC [123], as well as more recent work [28, 49, 38] in both the Coq and Isabelle/HOL proof assistants in support of proof-carrying code research, but these have limited usability in any production setting.

The microprocessor industry has a longer history than most areas of computing in the application of formal verification tools to hardware design. This was motivated by the extreme cost of hardware defects, the most commonly cited example being the floating point defect that shipped in the Pentium processor in the mid 1990s [34]. A substantial amount of formalization work has been performed by hardware vendors including AMD and Motorola using the ACL2 system [105]. This work in ACL2 has been applied to a number of hardware designs that have shipped in commercial processors by these vendors, many of which are acknowledged publicly by the vendors (even if the corresponding verification artifacts are not available). The largest issue with these formalizations is the style by which the formalization was performed. Approaches common in formal verification (such as those used by CompCert and VST) adopt a minimal kernel model (the de Bruijn criterion described in Section 1.2.3) to reduce the trusted code base and the scale of the logic that must be manually verified. ACL2 and related systems instead adopt a much larger kernel—in some cases, all of ANSI Common Lisp.

Interestingly, the bytecode-based languages have had some levels of formalization and verification. Bytecode verification is established as part of the Java language specification and is used to ensure that bytecode meets specific security and safety constraints prior to execution [18]. This is intended to capture defects or malicious intent in the compilation process or in the intervening path from compiler to execution environment. Much of this work has focuses on a limited degree of formalization of bytecode at the level of datatypes and stack usage. Given the prevalence of these bytecode-based languages in commercial software used across the DOE/NNSA complex, additional formalization is important for raising the assurance bar for this software.

3.5. New Programming Paradigms

In this section, we describe future research directions in the area of programming languages. Better assurance could be provided by research into subsets of existing modern programming languages, and also by the development of new domain-specific languages.

3.5.1. Model-Based System Design

While maintaining existing codebases is a large part of the tri-labs work, there is also a need for future-looking, more flexible programs. Especially in embedded applications, the MBSD artifact as an executable formal model (such as a state chart) could be *the* primary artifact, from which code and other properties are generated. Research into better MBSD-based approaches could introduce higher levels of abstraction, thus making implementation details such as the embedded programming language (such as C or Rust) mostly automated, akin to how hand-written assembly is rare in modern codebases.

3.5.2. Domain-Specific Languages

Domain-specific languages (DSLs) offer the potential to limit the scope of supported features, which can make a language both more expressive in its domain as well as simplify verification tasks. DSLs or embedded DSLs have seen success in some fields (e.g., Cryptol or Halide [101]), but have problems with expressivity and maintenance. The expressivity problem often arises when DSLs are used outside of their original intent. In particular, practical considerations such as foreign function interfaces or nonstandard data structures become either awkward or impossible to express in these DSLs; moreover, these problems tend to get worse as a project evolves unless scoping is deliberate and already well-established. For example, Q Framework uses MathWorks Simulink/Stateflow as its state chart front-end, but the limitations of Matlab's APIs and programming language semantics have caused extra developer time to be spent developing circuitous workarounds. Maintenance problems arise since DSLs typically do not have the same institutional investment as mature, general-purpose languages.

Potential research directions for DSLs include: development of annotation languages for verification (such as ANSI/ISO C Specification Language (ACSL) used by Frama-C, or Prusti [6] for Rust) and better formalizations for commonly-used DSLs such as OpenMP, CUDA, Kokkos, or RAJA [12].

3.6. Maturing Open-Source Hardware Design and Verification

Verification of hardware is a priority at the tri-labs; Sandia owns and operates its own semiconductor fabrication facility called MESA, but this does not mean the labs have perfect control over the hardware stack: often, hardware vendors have proprietary tooling or deliverables which are either closed-source or even purposely obfuscated. And so, the tri-labs leverage open-source hardware design and verification tooling.

One of the main research gaps with respect to verified hardware is the maturity of its tools: in recent years, (spurred perhaps by the development of the open architecture RISC-V), there has been more development of open hardware verification tooling. The main issue is that these tools are not yet at the software maturity necessary for the scale the labs needs; they need to be developed beyond a research-level software into an industrial focus.

With respect to hardware architectures, RISC-V, the open-source instruction set architecture, is the only reasonable path for full-stack verification. From a verification perspective, the tri-labs are interested in developing open-source and formal languages into more mature tools. The tri-labs already use Kôika, but desire improved scalability, both with the number of gates handled. For example, we can support about 100,000 gates, but have systems of interest larger than that. Beyond this, SMT solvers are

typically optimized for hardware-motivated problems and so often scale poorly. YICES [33] is the only SMT solver well-suited for hardware. Other considerations include developing better support of out of nominal behavior. Furthermore, developing Kôika to support clock domains, buses, and other modern features could help bring its feature set on par with proprietary solutions.

Hardware design presents challenging issues with trace-and-route. One open-source project which begins to solve this problem is Triton Route [56]. The labs are interested in developing into a more mature environment.

Another challenging issues in hardware verification is the hardware provenance and supply chain issues: physical hardware is less inspectable, but may still be possible to tear down. NNSA labs are thus interested in chain of trust for fabrications, hardware masks, bills of materials, and other supply-chain concerns. One of the later steps of hardware fabrication converting from a netlist into a mask, which are still made using commercial tools. One potential solution to this issue which would still allow the use of commercial tools would be development of proof-carrying code with netlists.

Beyond the facilities at MESA, Skywater [44] is an open-source process design kit and has been used with Xyce [57]. Further research into the Skywater workflow could establish better chain-of-trust for all stages of hardware development.

Lastly, there may be situations where there is no feasible path other than untrusted hardware. Research into verifiable computing (e.g., a trusted co-processor with an untrusted component) could help improve assurance.

3.7. Cybersecurity and Cryptography

Developing cryptography that is secure even in the presence of potential quantum computers, called Post Quantum Cryptography (PQC), is an active area of research [1, 9, 25, 26]. But just because a PQC algorithm is secure against quantum computers in *theory*, it may not be secure against classical computers. Proving absence of these types of vulnerabilities (without even counting implementation bugs) is also rapidly developing [93, 22, 82, 14], motivated in part by the National Institute of Standards and Technology (NIST) PQC standardization efforts, which began in 2016. As of early 2024, the competition is in its third (of four) rounds of algorithm selection [87], wherein algorithms are being carefully evaluated for their robustness and correctness. All the proposed PQC algorithms are probabilistic in nature, and so investment into probabilistic programming languages could improve reasoning techniques.

Another interest of the tri-labs is security of Operational Technology (OT). Many traditional cybersecurity contexts assume some properties about a computer system such as their ability to be updated, or are networked with modern (i.e., TCP/IP) protocols. Even if these do not hold for many systems in use today, OT still often lags behind other computer systems by decades.

In addition to OT, other infrastructure-related devices such as protocol gateways, routers, and switches, are increasingly relied upon for critical infrastructure systems. These have similar concerns with OT in that they have high reliability requirements and may not be easily to replace or update. Incorporating formal methods, and in particular cybersecurity requirements, early in the development lifecycle would benefit OT assurance and cybersecurity.

Two themes of infrastructure and other cybersecurity-related concerns are of early-adoption of formal methods techniques and improved modeling. Because these infrastructure-related systems are not easily updated, the cost for errors is much higher. One of the key benefits that formal methods has established in the tri-labs portfolio is its ability to catch more errors earlier-on in the development lifecycle. While adversarial behaviors may not be reducible to a simple mathematical model, better modeling (such as threat models or informed algorithmic input generation) can further produce guidelines of where to focus cybersecurity testing and hardening efforts [35].

These concerns cannot be mentioned without issues of scoping and responsibility: the sheer amount of IT/OT systems in deployment for the tri-labs (and the United States in general) produces a challenging problem of how to identify the needs and responsibilities of interested parties. Thus, more effort at the policy and procurement level could be effective, for example, ensuring standardizations for Software Bills of Material (SBOMs) in procurement, semantic matching of software packages (e.g., between trusted source code and binary distributions), and secure root-of-trust.

Other research gaps in the cybersecurity space are those of formal threat modeling. For example, cybersecurity properties may look different from traditional safety or liveness properties. Work in the areas of hyperproperties, observational equivalences between protocols, and composable proofs of security have been investigated, but these techniques do not have adequate tooling investment to be used in situations other than bespoke, labor-intensive solutions. Because of this, proprietary cybersecurity solutions are often advertised without any assurance guarantees, but instead charge for services akin to software linters, which prove merely syntactic guarantees about software which do not correlate with a lack of cybersecurity vulnerabilities.

Some potential strategies these gaps in cybersecurity threat modeling could be addressed are with automata learning (such as building state machine-based adversarial models), sound state compression to handle scalability limitations of threat modeling, and abstract interpretation from binaries.

3.8. Collaborations

The NNSA labs already have established university connections, such as university partnership programs and the Predictive Science Academic Alliance Program, but we have identified several opportunities to improve these collaborations.

One consideration which consistently requires revisiting is selection of research topics. For example, as mentioned in Section 3.4.1.3, better formalization of Rust could include a formally-verified Rust compiler. However, this would require a huge undertaking with many person-years of development. While some of this is relevant to universities and could result in publications or dissertations, much of it requires development work not well-suited towards PhDs. However, funding this work through NNSA-lab scientist time may be prohibitively expensive. Investigating the best paths forward, or pursuing other avenues for progress would be valuable to the community as a whole.

Other scientists have appreciated transfer of ideas, for example through regular discussions with vendors, such as US-based technology companies or international institutions such as CEA-List in France, are valuable and help connect the tri-labs with the broader formal methods community. Likewise, potential partnerships with European agencies are incredibly valuable because of the large formal methods community in Europe (In particular, French organizations maintain many FM and

FM-adjacent tools, including Why3, Coq, Frama-C, CompCert, and OCaml, just to name a few. The United States has Cooperative Research and Development Agreement (CRADAs) with both the United Kingdom and France; US and UK host Joint Working Groups (JOWOGs) to share information across the DOE and Atomic Weapons Establishment. We believe the already-established CRADAs with France and the UK could be better-utilized in the formal methods space.

Beyond existing CRADAs with governmental organizations, the tri-labs sees opportunities for better open university collaborations, which could facilitate work with with US citizens and foreign nationals in US universities, as well as foreign universities. At times the open/closed information delineation requires extra effort up-front to ensure all materials can be publicly released; this overhead can stymie good science. Motivations to assuage this process are twofold: for one, academics are increasingly expected to make their results publicly accessible, and so a sensitive/nonsensitive distinction is already being required even for domestic students, and second, programs where this has been done, such as ASC's Predictive Science Academic Alliance Program (PSAAP), have demonstrated the value of this up-front effort. Another benefit of these university collaborations is to encourage a robust recruitment program, especially considering companies such as Amazon have a large demand for formal methods researchers.

Beyond well-established allies, modern open-source software comes from worldwide sources. Tri-labs are interested in establishing better protocols for software provenance. We have identified challenges with using software whose maintainers have emails ending in `.ru`, for example, and processes for sharing code between labs, within labs, and externally could be simplified while improving security.

Open examples and benchmark suites are common in HPC codebases [24] and formal methods could benefit from similar investment. For example, many binary analysis problems at the scale which the labs must deal with are not tackled by academics because of their complexity, and the challenge of publishing these results (which can be seen as "experience reports" or "engineering effort" and not publishable research), especially for C or C++ codebases.

Concrete steps to achieve these collaborations include building open surrogate models and benchmarks, cutting red tape for university collaborations, and using CRADAs which exist in theory, but are underused in practice.

3.9. Tool Usability

The usability of formal methods tools underpins every research opportunity mentioned above. Other sections reference specific usability concerns; here, we highlight general considerations, including practical concerns, tool choice, interpretability, and scalability. Addressing barriers to tool usability is essential to the tri-lab mission of enabling further formal methods research.

3.9.1. Usability as a Formal Methods Issue

The problem of tool usability may at first seem to fall squarely within the domain of software development. While not altogether incorrect, this distinction belies the relevance of usability to formal methods. The usability of tools is a complex problem that must take into account various aspects of

formal methods, including programming language design, human-computer interaction, logical constraints like the decidability of a proof obligation, and more.

Due to these factors, the formal methods tools that tri-lab scientists use are often tailored to a specific software system or problem domain. One example at Sandia is Q Framework (Section 2.1.5), which uses Frama-C to specify and verify properties of programs. To fulfill mission deliverables, Sandia scientists have restricted their focus to C programs that closely match common MBSD design paradigms, namely the event loop design pattern. Their need to restrict the problem domain to match the capabilities of the tools at hand illustrates the critical relationship between usability and formal methods research. However, some restriction is always necessary: one theorem from the foundations of computing, Rice's Theorem, states that in general it is undecidable to prove "nontrivial[2]" properties about a given computer program [102].

While usability is a formal methods problem, there is no universal solution that fits all needs at the tri-labs. Some codebases are best suited for lightweight formal methods to detect common classes of mistakes, others can leverage domain-specific tooling, and the most high-consequence applications demand more rigorous, in-depth verification, such as full proofs of correctness in Coq. Ultimately, the usability of formal methods tools will be highly dependent on the context in which they are applied.

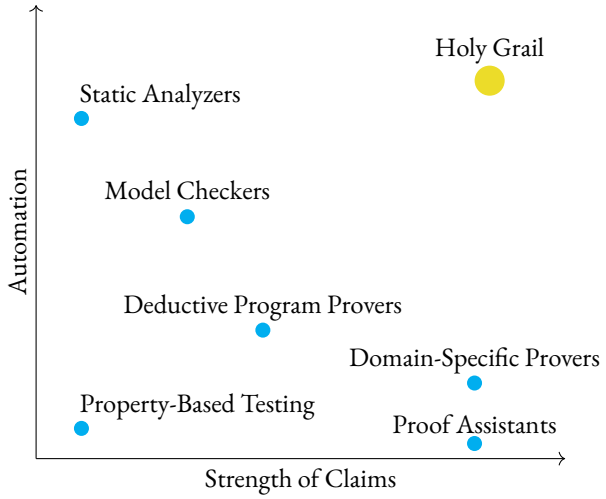

Figure 3-2.: The landscape of formal verification, as described by Leroy [73] (recreated in Pollard [97]). The Holy Grail point represents verification that takes no manual effort without decreasing the strength of claims, which is desirable but unattainable.

3.9.2. Practical Concerns

Formal verification efforts often take upwards of fifteen times the person-hours as the original development [62]. The tri-labs is interested in vastly decreasing the time expended on verification. One possible way of achieving this outcome is integrating verification into an earlier part of the development

[2]Put slightly more technically, Rice's theorem is *extensional*. Properties such as "What is the size of the computer program" or "does a given program typecheck" can be decided; what precisely makes a property *nontrivial* is beyond the scope of this report.

lifecycle through model-based system design, more powerful type systems, design-by-contract, better specification generation, and lightweight formal methods tools like Alloy [53]. In addition to integration, the verification feedback loop must be made more "pleasant" so that formal verification tools get more use. This shift will likely require increasing the interpretability of tool output, mentioned below.

The changes proposed above would prove helpful to both formal methods experts and non-experts. For non-experts, a critical barrier is lack of familiarity with formal methods-specific tools. A workaround may be domain-specific languages (DSLs), discussed in Section 3.5.2.

3.9.3. Tool Choice

A key aspect of usability is choosing the right tool for the right problem. One primary way that verification tools differ is in their degree of automation. Automation enables a tool to automatically check programs for the preservation of certain (potentially automatically-discovered) properties with little guidance from the user. This enhanced usability typically comes at a cost of decreased expressivity of guarantees that can be made about the program. In comparison, manual approaches require users to not just specify properties but write proofs that the properties hold. This is the approach taken by VST, for example, [3] and manual approaches allow for developers to use the full power of mathematics and expert reasoning. The variety of projects at the tri-labs necessitates the use of verification tools with varying degrees of automation and expressivity.

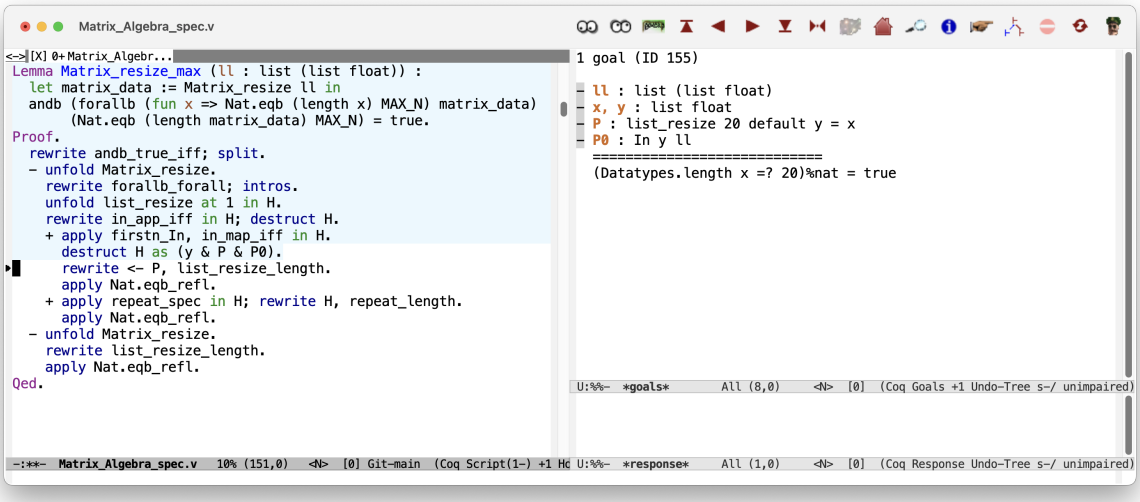

Figure 3-3.: An Emacs interface to The Coq proof assistant. The interface is similar to a debugger, with the proof script on the left and the current proof state after the highlighted section on the right. This particular theorem is an auxiliary lemma about verifying a C code implementation of matrix algebra.

Proof assistants are a commonly used verification tool that typically have few automated capabilities but allow for high degrees of expressivity. For example, Coq is known as an interactive theorem prover; its

[3] VST has some limited automation but generally requires a large amount of developer time and expertise.

interface is shown in Figure 3-3. This particular tradeoff introduces the challenge of proof repair: small changes in the implementation often require large rewrites to the structure of proofs [103]. Better proof automation and search could help this, as well as using AI/ML to have more powerful proof search and heuristics. Provided the trusted computing base is small, [4] sophisticated proof search could be safely performed without affecting the trust of the system.

On the other side of the automation spectrum are program logics like Frama-C, which achieve greater automation through powerful and interoperable abstract interpretation capabilities but do not offer the same assurance and soundness arguments provided by more manual tools like VST. Future investment in developing better soundness arguments for automated tools such as Frama-C, or better automation capabilities for tools like VST, would vastly improve developer productivity while also making proofs more robust to change.

3.9.4. Interpretability

A common obstacle to working with formal methods tools is the interpretability of their output, which often consists of low-level logical statements or errors that are difficult to connect to their corresponding high-level specification language or code. The lack of interpretability leads to increased time and effort invested in understanding and debugging these tools, diminishing usability. We provide an example with Frama-C, but this challenge remains in many other tools.

Frama-C takes C code and a specification as input. These properties get translated into an intermediate language, optimized to achieve better scalability, and dispatched to external solvers as *verification conditions*. These solvers can return one of three verdicts: unsatisfiable (which, because of the way propositions are formulated, typically represents valid conditions), satisfiable (with a counterexample), or a timeout. Regardless of the verdict reached by Frama-C, the default output offers little insight into how or why it reached a certain conclusion, limiting the tool's interpretability. Frama C's opaque nature is especially challenging when proof goals fail, as the user has to check both the specification and the implementation for a bug. A possible solution that could increase interpretability is counterexample generation, which analysts could understand more easily than the output of complex FM tools.

3.9.5. Scalability

As mentioned in Section 2.1.2, the challenges encountered in large-scale proof efforts push the limits of existing proof tools, which increases the level of effort required to achieve scalability. As an example, consider the limitations encountered by the Sandia team when verifying the cryptographic root-of-trust in the proof theory of interaction trees. Interaction trees are *coinductive* specifications of possibly infinite streams of behaviors. Using Coq and the Verified Software Toolchain (VST), the Sandia team demonstrated that the interaction trees produced by their embedded software was observably equivalent to the acceptable, specified collection of interaction trees. To prove this equivalence, researchers had to substantially extend existing proof rules for interaction trees and custom-build libraries for VST and Coq, illustrating the scalability limitations of existing verification tools. Developing more robust proof tactics and modular libraries could decrease the effort needed to make existing verification approaches scalable.

[4] It is generally accepted in the proof assistant community that Coq and HOL4 satisfy the de Bruijn criterion [113].

3.9.6. DARPA PROVERS

The concerns mentioned above provide support for the idea of validation workflows, wherein the usability, scalability, and trustworthiness of tools are evaluated together. DARPA PROVERS is a project to address the scalability concerns with formal methods [81]. Sandia is involved with scaling formal methods tools, developing metrics, and evaluating new approaches for formal methods. This project is currently in its first year but aims to improve upon existing formal methods tools as well as develop new tools. The success of this project would produce tools that are widely used and well-maintained while also lowering the barrier to entry for formal methods.

3.10. Formalized Numerics and Floating-Point

Researchers at Sandia and LLNL have identified the need for tools that verify the numerical behavior of floating-point computations in several domains. As mentioned in Sections 2.1.9 and 2.1.11, developing better tools for formally verifying numerical programs could have a large impact on the trustworthiness of both embedded systems and large HPC codebases. Furthermore, the increasing heterogeneity of hardware accelerators and FPGAs has created an increased need for tools that provide *parameterized, generic* analyses (i.e., analyses that do not assume a single floating-point representation or numeric library interface). Extending existing formalizations of floating-point arithmetic [16, 17, 59, 60] for increased scalability, usability, and support for heterogeneous computing would have broad impact at the labs, both in embedded system verification and in assurance for HPC modeling and simulation.

The primary formalization of the IEEE-754 standard in the Coq proof assistant is the Flocq library [17]. While this library has been used successfully by experts in numerical analysis [86, 15, 20, 70], it can be hard to use for non-experts. And, while the design of Flocq is sufficiently generic to apply to a wide variety of floating-point formats beyond those in the IEEE-754 standard, work should be done to develop particular formalizations for nonstandard formats, such as those used on GPUs and low-precision (16-bit or 8-bit) floats used for AI/ML applications. Beyond this, developing tools or proof libraries that interface with Flocq and can guide non-expert users would encourage further development.

Formally verifying numerical programs requires the presence of suitable specifications, which don't always exist in practice. One example that could benefit from improved specification is Intel's OneAPI; by default, it improves performance by using imprecise options for certain floating-point operations. If a program requires precise models, porting to OneAPI may change the result of a computation, and it can be time-consuming to determine the provenance of the discrepancy. Automated analysis, or at least tools for better error tracking, could improve the reliability of porting efforts in the presence of improved specifications.

Finally, the main challenge of floating-point arithmetic is that the mathematical abstraction—infinitely precise real number arithmetic—is mapped into a finite precision approximation. There is no good language support for managing this abstraction, so the onus lies on programmers to understand how error propagates. The tri-labs would benefit both from automated static analysis of numerical codes which could identify potentially high-error operations, as well as more feature-rich tools that could, for example, track real-number semantics alongside floating-point approximations. While there has been some work in this space related to precision tuning [84, 109], we emphasize the need for tools that

generate rigorous guarantees and proof certificates.

3.11. Artificial Intelligence

With the explosion of AI—especially Large Language Models (LLM)—in the past 7 years, we again emphasize the analogy of formal methods providing "guard rails" for AI. The idea of guard rails around AI is simple to conceptualize, but difficult to implement. Difficulties with formal reasoning around modern AI techniques occur because of the size of the models, their complex, high-dimensional input data, and their problem domains. For example, it is not clear how to formally specify to a autonomous vehicle when it is safe to continue through an intersection. We claim formal verification around these topics is worth investment.

There are two ways to think of combining FM and AI: *FM for AI* and *AI for FM*. We discuss both in this section. Successful research in the first direction would result in increasing safety and trust of AI systems. For example, developing formal models for the input and output space of a large language model using programming language or FM techniques could ensure classes of output are not generated. The simplest example of this would be to assign a type checker to the output of an LLM, but more sophisticated models of output could be developed.

For the second (AI for FM), we note a complementary nature between the rigorous and less flexible (formal methods) versus the less rigorous and flexible (AI). Our position is that combining AI and FM can be gestalt, provided the roles are clearly demarcated. In particular, if powerful AI techniques can be used in situations where users do not need to know from where the solution arose, but only that it is correct. In particular, situations (such as *NP*-complete problems) where the solution can be checked quickly using a trusted verifier provide the best opportunities. One example relevant to the FM community is proof search [128]: formal proof requires a large development effort that can benefit from better tools.

Another example of using AI for FM are in the heuristics for SMT solvers. To give some context, modern SMT solvers work in practice for large-scale problems because many heuristics have been implemented which can drastically reduce a problem search space. As a counter-point, machine-learned models can produce shockingly bad models for Xyce [57] circuit simulations, which cannot meet requirements. And so, there remains future research to identify when ML models can be effective and when they cannot.

Another area for improvement for generative AI are the lack of good specifications. AI/ML systems are often tasked with generating input based on poor specifications, and even when specifications are clarified (using in-depth prompt generation or pre-trained models tuned to particular domains), these are probabilistic in nature and provide no guarantees. Addressing this challenge could improve the accuracy of models for domains where there are good specifications (i.e., where notions of "correctness" are well-posed).

Because the tri-labs has a responsibility to safeguard national security, researchers here are concerned with adversarial issues related to AI. Modern AI systems often have limitations in accurately reporting their capabilities (for example, adversarial input generation can trick algorithms into providing incorrect answers they predict with high probability). We propose research into smaller examples that can address these challenges, such as developing polyhedral abstract domains that can constrain output spaces.

There remains challenges to scale these up to larger problems, but research in this area could be a start towards verifying larger and more complex AI systems.

We mention here the *hallucination problem* of AI, and in particular, LLMs. This refers to the tendency of LLMs to generate incorrect or nonsense outputs without any understanding of whether these statements are correct. A primary issue of the hallucination problem is LLMs, unlike a human, cannot provide a chain of reasoning to explain an output. In some applications, constraining input to a particular domain at first may seem like a potential solution. For example, suppose an LLM is constrained to only generate valid Rust programs, and this is confirmed with a compiler. However, this constraint is merely syntactical: semantic (behavioral) correctness is a much a harder—in fact, it is an undecidable—problem. It is not possible to solve this problem in the general case, but research into particular domains could be valuable.

Other solutions to this hallucination problem could be sandboxing, wherein the "sandbox" refers to constraining output into a known, formally-verified domain. This is used in autonomous control systems (a list of projects is maintained by DARPA [27]) and could be extended to other domains.

Other areas that LLMs have historically shown to be effective are in knowledge acquisition search [76], such as compiling and formalizing the results of a large quantity of technical documentation. These classes of problems are more limited in scope, but there remains research in both requirement specification and modeling, as well as usability. In the space of knowledge acquisition, it remains to be seen whether automated tools actually accelerate developer effort, or if the hallucination problem of AI systems could have the opposite effect of the guard rails we have previously mentioned. To be more specific, it is unknown whether solutions provided by LLMs in a technical space may cast more doubt on an analysts' understanding, and if this doubt is productive (in the case that the analyst had a misunderstanding of the system under scrutiny) or a waste of time (in the case the LLM is wrong).

Commonly-used datasets for model training could also be better-scrutinized. Because these are so common in AI/ML systems, they could potentially be exploited for adversarial uses. Because AI/ML systems cannot produce anything new but only model data from training sets, analysis of these training sets could provide insight into potential gaps or underrepresented samples.

4. EXEMPLARS

While much of the work the tri-labs performs cannot be released to the public for national security reasons, we outline a few exemplar problems that capture some key features of tri-labs interests. We note that many of these problems are open-ended to motivate several potential research directions for medium and long-term projects relevant to tri-lab needs. Compared with Chapter 3, which outlines broad, high-impact research areas, these exemplars provide more specific problems collaborators can refer to.

4.1. Further Formalization of C Compiler Toolchains

As mentioned in Section 3.4.1.1, we have identified several opportunities for research in formalizing C Compilers:

- CompCert's semantic preservation theorem is not modular. In the presence of libraries compiled with compilers other than CompCert, and the semantics of linkers is difficult. Extending and modularizing CompCert's theorems would be valuable to scale semantic preservation to larger codebases.
- CompCert has limited optimizations. Further work on CompCert and its model of memory cells could allow large classes of verified optimizations and, within this formally-verified environment, permit aggressive optimization search.
- Sail [45] has been used to specify instruction set architectures (ISAs) and is the canonical representation for RISC-V as well as newer versions of ARM. While Sail can be used to generate Coq, the code is not usable with existing tools (such as CompCert) without further development. Linking Sail semantics of ISAs could provide better models of ISAs to CompCert. However, CompCert's assembly model is based off an infinite memory model, which means there is no refinement to a real, finite-memory machine.

4.2. Formal Programming Language Specifications

As mentioned in Section 3.4, machine-checkable language semantics can be useful for improving trust in computer systems. However, beyond just rubber-stamping "formalization," useful language specification efforts require interoperability with existing tools. Therefore, when we mention "formal language semantics" integration, we refer the reader to our discussion in Section 3.4.1.3 about the differences between the use-cases and challenges of C versus C++ from a verification perspective.

In any case, these are programming languages and models used at the labs where formalization efforts would have high-impact:

- Rust (see discussion about this in Section 3.4.1.3)
- Python
- .NET
- Java
- High-performance computing (HPC) libraries such as Kokkos, OpenMP, CUDA.

4.3. Numerical Analysis on Next-Generation Accelerators

As HPC demands in modeling, simulation, and AI/ML progress, the tri-labs need faster, more efficient computational capabilities. Extending assurance arguments for HPC applications into the next decade requires support for the increasingly heterogeneous hardware of future architectures, such as domain-specific accelerators, GPUs, neuromorphic architectures, and Field-programmable gate arrays (FPGAs).

The IEEE-754 standard for floating-point arithmetic [50] has vastly improved reproducibility of numerical computations since its original release in 1985. However, hardware is moving ahead of the standard, in particular with GPUs using non-conforming IEEE 754 arithmetic and datatypes.

Applications of formalized numerics could include:

- automation of formal proofs of numerical software
- numerical precision and numerics-aware compilers
- exception detection for massively-parallel applications.

4.4. Formally-Verified Compiler Optimizations

Work with equality saturation or tools such as Halide and Exo [101, 52] can make writing optimizations easier and simpler. While both have found success in their application-specific domains, (image processing and accelerator programming, respectively) neither are designed to handle the massive feature set required by C++. One research direction would be to develop a limited formal semantics, not (at first) of the entirety of C++, but relevant subsets such as high-performance or parallel libraries. Once these semantics have been written down, semantics-preserving translations could more easily be discovered using new data structures such as equality saturation [124] and verified using formal techniques. This could open an avenue of "superoptimizers" which can make more aggressive optimizations that are not obvious without a deep knowledge of the subtleties of a language and target architecture's semantics.

Moreover, three lab-relevant optimization directions for research include:

1. Runtime.
2. Binary size.

3. Ease of analysis. Techniques such as aggressive constant folding and function inlining may result in larger or slower programs but make binary analysis more tractable.

Direction 2 and 3 typically see less focus in compiler research compared to runtime, but are more important for embedded or resource-constrained high-consequence applications.

4.5. Formal Verification of a Kalman Filter in C

A Kalman filter is an algorithm that produces and updates estimations of the state of a physical system using a statistical model of the underlying system including noise and dynamics. Kalman filters have wide-ranging applications in sensing, navigation, and signal processing. Kalman filters are optimal estimators, provided their assumptions on the distributions of noise are correct. However, they require detailed knowledge of the underlying physical system (for example, an understanding how wind resistance on a moving object has a known physical effect).

Implemented in C, a Kalman filter exemplifies many challenges of verification:

- Linking real-world properties and specifications (such as properties of sensors, noise, and physical systems) to their C code implementation;
- Formal specification of real-number properties, and their floating-point approximation;
- Mixing of computational kernels within larger control software.

Developing a formally-verified Kalman filter, from its physical representation and the techniques learned throughout, would benefit the labs in many facets.

4.6. CAN Bus

A CAN, or Controller Area Network, is a vehicle bus standard which allows microcontrollers to communicate with each other. In particular, there are several interesting research directions regarding a CAN bus:

- CAN bus as a distributed system; it is important to verify safety and liveness properties of the many interacting components, as well as ensure noninterference between sub-components.
- How should a CAN bus be formally specified? One can build models of systems using, e.g., TLA+, however, it is necessary to model both the components interacting on the CAN bus, the hardware interface, and the computer programs executing on these components. Permitting theorems to be stated and mechanically or manually proved for multiple levels of abstraction is challenging.

This page intentionally left blank.

5. CONCLUSION

In this report, we have outlined the formal methods research done at SNL, LLNL, LANL, and NASA JPL, and identified many research areas and some of the challenges. We then provided a list of example research questions that are highly-relevant to lab interests. Broadly speaking, the research interests at the labs related to a better formal understanding of digital systems in the areas of hardware, compilers, cybersecurity, AI/ML, HPC, programming languages, embedded system verifications, and other formal methods techniques.

We identified areas where there are already strong verification efforts which can be improved even further, such as using the VST and CompCert, as well as areas where there remain significant barriers making verification more difficult, in particular with large HPC codebases and C++. We also identified areas where AI/ML can help FM, or that FM can help AI/ML, and potential pitfalls for relying on newer AI systems such as large language models.

Throughout, a running theme of this report is the challenge with *tooling and usability* of formal methods. Partly by design, partly by resource constraints, formal methods tools are often sophisticated and difficult to use. By design, because the intricacies of digital systems results in many complex cases to consider. By resource constraints, because it is time consuming to encode the large amount of human expertise and intuition used to reason about software. We hypothesize that investment in tooling and usability will have large impacts in software assurance, both from proving useful correctness statements about software, but also in programmer productivity: formal methods can provide "guard rails" to eliminate large classes of errors and free up developer effort towards solving higher-level problems. We hope this report provides a roadmap for future researchers in government, academia, and industry to develop correct, robust, secure software.

This page intentionally left blank.

BIBLIOGRAPHY

[1] Aguirre, A., Barthe, G., Gaboardi, M., Garg, D., Katsumata, S.-y., and Sato, T. Higher-order probabilistic adversarial computations: categorical semantics and program logics. *Proceedings of the ACM on Programming Languages 5*, ICFP (2021), 1–30.

[2] Ahn, D. H., Baker, A. H., Bentley, M., Briggs, I., Gopalakrishnan, G., Hammerling, D. M., Laguna, I., Lee, G. L., Milroy, D. J., and Vertenstein, M. Keeping science on keel when software moves. *Communications of the ACM 64*, 2 (2021), 66–74.

[3] Appel, A. W. Verified software toolchain. In *Programming Languages and Systems* (Berlin, Heidelberg, 2011), G. Barthe, Ed., Springer Berlin Heidelberg, pp. 1–17.

[4] Appel, A. W. Verified software toolchain: (invited talk). In *European Symposium on Programming* (2011), Springer, pp. 1–17.

[5] Appel, A. W., Beringer, L., Chlipala, A., Pierce, B. C., Shao, Z., Weirich, S., and Zdancewic, S. Position paper: The science of deep specification. *Philosophical Transactions of the Royal Society A* (2017).

[6] Astrauskas, V., Bílý, A., Fiala, J., Grannan, Z., Matheja, C., Müller, P., Poli, F., and Summers, A. J. The prusti project: Formal verification for rust. In *NASA Formal Methods: 14th International Symposium* (Berlin, Heidelberg, May 2022), NFM, Springer-Verlag, p. 88–108.

[7] Atzeni, S., Gopalakrishnan, G., Rakamaric, Z., Ahn, D. H., Laguna, I., Schulz, M., Lee, G. L., Protze, J., and Müller, M. S. ARCHER: Effectively spotting data races in large OpenMP applications. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (2016), pp. 53–62.

[8] Bakst, A., and Dodds, M. Building a concurrency verifier using crucible. Tech. rep., Galois, Inc., June 2021.
https://galois.com/blog/2021/06/building-a-concurrency-verifier-using-crucible/.

[9] Barbosa, M., Barthe, G., Fan, X., Grégoire, B., Hung, S.-H., Katz, J., Strub, P.-Y., Wu, X., and Zhou, L. Easypqc: Verifying post-quantum cryptography. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* (2021), pp. 2564–2586.

[10] Barnett, M., Chang, B.-Y. E., DeLine, R., Jacobs, B., and Leino, R. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO 2005* (Nov. 2005), Springer Berlin Heidelberg.

[11] Baudin, P., Filliâtre, J.-C., Marché, C., Monate, B., Moy, Y., and Prevosto, V. ACSL: ANSI/ISO C specification. Tech. rep., CEA-List, Laboratory for Integration of Systems and Technology, 2021.

[12] Beckingsale, D., Hornung, R., Scogland, T., and Vargas, A. Performance portable c++ programming with RAJA. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2019), PPoPP '19, Association for Computing Machinery, p. 455–456.

[13] Ben-Ari, M., Pnueli, A., and Manna, Z. The temporal logic of branching time. *Acta Informatica 20* (1983), 207–226.

[14] Beullens, W. Breaking rainbow takes a weekend on a laptop. In *Annual International Cryptology Conference* (2022), Springer, pp. 464–479.

[15] Boldo, S., Joldes, M., Muller, J.-M., and Popescu, V. Formal verification of a floating-point expansion renormalization algorithm. In *Interactive Theorem Proving: 8th International Conference, ITP 2017, Brasília, Brazil, September 26–29, 2017, Proceedings 8* (2017), Springer, pp. 98–113.

[16] Boldo, S., Jourdan, J.-H., Leroy, X., and Melquiond, G. Verified compilation of floating-point computations. *Journal of Automated Reasoning 54* (2015), 135–163.

[17] Boldo, S., and Melquiond, G. *Computer Arithmetic and Formal Proofs: Verifying Floating-Point Algorithms with the Coq System*, 1st ed. ISTE Press - Elsevier, United Kingdom, Nov. 2017.

[18] Börger, E., and Schulte, W. *A Programmer Friendly Modular Definition of the Semantics of Java*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999, pp. 353–404.

[19] Bourgeat, T., Pit-Claudel, C., Chlipala, A., and Arvind. The essence of bluespec: A core language for rule-based hardware design. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2020), PLDI '20, Association for Computing Machinery, p. 243–257.

[20] Brisebarre, N., Joldeş, M., Martin-Dorel, É., Mayero, M., Muller, J.-M., Paşca, I., Rideau, L., and Théry, L. Rigorous polynomial approximation using taylor models in co q. In *NASA Formal Methods: 4th International Symposium, NFM 2012, Norfolk, VA, USA, April 3-5, 2012. Proceedings 4* (2012), Springer, pp. 85–99.

[21] Cadar, C., Dunbar, D., and Engler, D. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, 2008), OSDI'08, USENIX Association, p. 209–224.

[22] Castryck, W., and Decru, T. An efficient key recovery attack on sidh. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques* (2023), Springer, pp. 423–447.

[23] Cohen, J. M., and Johnson-Freyd, P. A formalization of core why3 in coq. *Proceedings of the ACM on Programming Languages 8*, POPL (Jan. 2024).

[24] Cook, J., Aaziz, O., Alexeev, Y., Balakrishnan, R., Fletcher, G., Junghans, C., Kim, Y., Liber, N., Liu, G., Lund, A., Mayagoitia, A., McCorquodale, P., Pavel, R., Ramakrishnaiah, V., Vaughan, C., and The ECP Proxy App Team. Fy22 proxy app

suite release: Report for ecp proxy app project milestone adcd504-14. Tech. rep., Department of Energy, Dec. 2022.
https://proxyapps.exascaleproject.org/wp-content/uploads/2023/03/ADCD504-14.pdf.

[25] Cremers, C., Fontaine, C., and Jacomme, C. A logic and an interactive prover for the computational post-quantum security of protocols. In *2022 IEEE Symposium on Security and Privacy (SP)* (2022), IEEE, pp. 125–141.

[26] Cremers, C., Jacomme, C., and Lukert, P. Subterm-based proof techniques for improving the automation and scope of security protocol analysis. In *2023 IEEE 36th Computer Security Foundations Symposium (CSF)* (2023), IEEE, pp. 200–213.

[27] DARPA Assured Autonomy Program. Assured autonomy tools portal continual assurance of learning-enabled, cyber-physical systems (le-cps), 2024. https://assured-autonomy.org/tools.

[28] Dasgupta, S., Park, D., Kasampalis, T., Adve, V. S., and Roşu, G. A complete formal semantics of x86-64 user-level instruction set architecture. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2019), PLDI 2019, Association for Computing Machinery, p. 1133–1148.

[29] Davis, N. A., Berger, T. E., McDonald, A., Ingram, J. B., Foster, J. D., and Sanchez, K. Software verification toolkit (svt): Survey on available software verification tools and future direction. Tech. rep., Sandia National Laboratories, United States, Sept. 2022.

[30] Denis, X., Jourdan, J.-H., and Marché, C. Creusot: A foundry for the deductive verification of rust programs. In *Formal Methods and Software Engineering* (Cham, 2022), A. Riesco and M. Zhang, Eds., ICFEM, Springer International Publishing, pp. 90–105.

[31] Digital Foundations & Mathematics. https://proof.sandia.gov. Accessed 22 Dec 2023.

[32] Durumeric, Z., Li, F., Kasten, J., Amann, J., Beekman, J., Payer, M., Weaver, N., Adrian, D., Paxson, V., Bailey, M., and Halderman, J. A. The matter of heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference* (New York, NY, USA, 2014), IMC '14, Association for Computing Machinery, p. 475–488.

[33] Dutertre, B. Yices 2.2. In *Computer Aided Verification* (Cham, 2014), A. Biere and R. Bloem, Eds., CAV, Springer International Publishing, pp. 737–744.

[34] Edelman, A. The mathematics of the pentium division bug. *SIAM Review 39*, 1 (1997), 54–67.

[35] Epifanovskaya, L., Meeson, R., McCormack, C., Lee, J. R., Armstrong, R. C., and Mayo, J. R. Algorithmic input generation for more effective software testing. In *IEEE 46th Annual Computers, Software, and Applications Conference* (2022), COMPSAC, pp. 1708–1715.

[36] Etim, D. N. Introduction to PSAAP IV.
https://psaap.llnl.gov/sites/psaap/files/2023-09/01_etim_introduction_to_psaap_iv.pdf, 2023.

[37] Feather, M. S., and Pinto, A. Assurance for autonomy – JPL's past research, lessons learned, and future directions. https://arxiv.org/abs/2305.11902, 2023.

[38] Fernandez, M., Ed. *Workshop on Instruction Set Architecture Specification* (Sept. 2019), SpISA. https://www.cl.cam.ac.uk/~jrh13/spisa19.html.

[39] Ferrous Systems. Ferrocene: An open source qualified rust compiler for functional safety. https://github.com/ferrocene, 2024.

[40] Filliâtre, J.-C., and Paskevich, A. Why3 – where programs meet provers. In *European Symmposium on Programming* (Berlin, Heidelberg, 2013), M. Felleisen and P. Gardner, Eds., ESOP, Springer, pp. 125–128.

[41] Finkbeiner, B. Synthesis of reactive systems. In *Dependable Software Systems Engineering*, J. Esparza, O. Grumberg, and S. Sickert, Eds., vol. 45 of *NATO Science for Peace and Security Series - D: Information and Communication Security*. IOS Press, 2016, pp. 72–98.

[42] Galois, Inc. Grackle: A symbolic simulator for engineering code. https://grackle.galois.com.

[43] Gamell, M., Teranishi, K., Heroux, M. A., Mayo, J., Kolla, H., Chen, J., and Parashar, M. Local recovery and failure masking for stencil-based applications at extreme scales. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (New York, NY, USA, 2015), SC '15, Association for Computing Machinery.

[44] Google and Skywater Technology. Skywater sky130 process design kit documentation. Tech. rep., 2021. https://skywater-pdk.readthedocs.io/en/main/.

[45] Gray, K. E., Kerneis, G., Mulligan, D. P., Pulte, C., Sarkar, S., and Sewell, P. An integrated concurrency and core-ISA architectural envelope definition, and test oracle, for IBM POWER multiprocessors. In *Proceedings of the 48th International Symposium on Microarchitecture (Waikiki)* (Dec. 2015), pp. 635–646.

[46] Harrison, J. Formal verification at intel. In *18th Annual IEEE Symposium of Logic in Computer Science. Proceedings.* (June 2003), LICS, pp. 45–54.

[47] Henk, B., and Freek, W. The challenge of computer mathematics. *Philophical Transactions of the Royal Scoiety A 363*, 1835 (2005).

[48] Holtzen, S., Van den Broeck, G., and Millstein, T. Scaling exact inference for discrete probabilistic programs. *Proceedings of the ACM on Programming Languages 4*, OOPSLA (Nov. 2020).

[49] Hóu, Z., Sanan, D., Tiu, A., Liu, Y., Hoa, K. C., and Dong, J. S. An isabelle/HOL formalisation of the SPARC instruction set architecture and the TSO memory model. *Journal of Automated Reasoning 65* (2021), 569–598.

[50] Hough, D. G. The IEEE standard 754: One for the history books. *Computer 52*, 12 (Dec. 2019), 109–112.

[51] H ulsing, A., and Kudinov, M. Security of WOTS-TW scheme with a weak adversary, July 2023. Official comments at: https://csrc.nist.gov/csrc/media/Projects/post-quantum-cryptography/documents/round-3/official-comments/Sphincs-Plus-round3-official-comment.pdf.

[52] Ikarashi, Y., Bernstein, G. L., Reinking, A., Genc, H., and Ragan-Kelley, J. Exocompilation for productive programming of hardware accelerators. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (New York, NY, USA, 2022), PLDI '22, Association for Computing Machinery, p. 703–718.

[53] Jackson, D. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, Cambridge, MA, 2012.

[54] Jin, F., Jacobson, J., Pollard, S. D., and Sarkar, V. Minikokkos: A calculus of portable parallelism. In *2022 IEEE/ACM Sixth International Workshop on Software Correctness for HPC Applications (Correctness)* (2022), pp. 37–44.

[55] Jung, R., Jourdan, J.-H., Krebbers, R., and Dreyer, D. RustBelt: securing the foundations of the rust programming language. *Proceedings of the ACM on Programming Languages 2*, POPL (Dec. 2017), 66:1–66:34.

[56] Kahng, A. B., Wang, L., and Xu, B. Tritonroute: The open-source detailed router. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 40*, 3 (2021), 547–559.

[57] Keiter, E. R., Verley, J. C., and Thornquist, H. K. Xyce: Open source simulation for large-scale circuits. Tech. rep., Sandia National Laboratories, 2019.

[58] Kellison, A., Tekriwal, M., Jeannin, J.-B., and Hulette, G. Towards verified rounding error analysis for stationary iterative methods. In *2022 IEEE/ACM Sixth International Workshop on Software Correctness for HPC Applications (Correctness)* (2022), IEEE, pp. 10–17.

[59] Kellison, A. E., and Appel, A. W. Vcfloat2: Floating-point error analysis in coq. In *Proceedings of the 13th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2024)* (Jan. 2024). To appear.

[60] Kellison, A. E., Appel, A. W., Tekriwal, M., and Bindel, D. LAProof: a library of formal proofs of accuracy and correctness for linear algebra programs. In *30th IEEE International Symposium on Computer Arithmetic (ARITH)* (Sept. 2023).

[61] Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., and Yakobowski, B. Frama-c: A software analysis perspective. *Formal aspects of computing 27* (2015), 573–609.

[62] Klein, G., Andronick, J., Elphinstone, K., Heiser, G., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., and Winwood, S. Sel4: Formal verification of an operating-system kernel. *Communications of the ACM 53*, 6 (June 2010), 107–115.

[63] Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., and Winwood, S. SeL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (New York, NY, USA, 2009), SOSP '09, Association for Computing Machinery, p. 207–220.

[64] Kolla, H., Mayo, J. R., Teranishi, K., and Armstrong, R. C. Improving scalability of silent-error resilience for message-passing solvers via local recovery and asynchrony. In *IEEE/ACM 10th Workshop on Fault Tolerance for HPC at eXtreme Scale (FTXS)* (2020), pp. 1–10.

[65] Laguna, I. Fpchecker: Detecting floating-point exceptions in gpu applications. In *34th IEEE/ACM International Conference on Automated Software Engineering* (2019), ASE, IEEE, pp. 1126–1129.

[66] Laguna, I., Tran, A., and Gopalakrishnan, G. Finding inputs that trigger floating-point exceptions in heterogeneous computing via bayesian optimization. *Parallel Computing 117* (2023), 103042.

[67] Lamport, L. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[68] Lang, M. LANL NNSA software technology. Tech. rep., Los Alamos National Laboratory, Nov. 2019.

[69] Lee, W., Stelle, G., McCormick, P., and Aiken, A. Correctness of dynamic dependence analysis for implicitly parallel tasking systems. In *2018 IEEE/ACM 2nd International Workshop on Software Correctness for HPC Applications (Correctness)* (2018), IEEE, pp. 17–24.

[70] Lefèvre, V., Louvet, N., Muller, J.-M., Picot, J., and Rideau, L. Accurate calculation of euclidean norms using double-word arithmetic. *ACM Transactions on Mathematical Software 49*, 1 (2023), 1–34.

[71] Leino, R. M. Dafny: An automatic program verifier for functional correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers* (2010), E. M. Clarke and A. Voronkov, Eds., vol. 6355 of *Lecture Notes in Computer Science*, Springer, pp. 348–370.

[72] Leroy, X. Formal verification of a realistic compiler. *Communications of the ACM 52*, 7 (July 2009), 107–115.

[73] Leroy, X. In search of software perfection. https://youtu.be/lAU5hx_3xRc, Nov. 2016.

[74] Lewis, J. R., and Martin, B. Cryptol: high assurance, retargetable crypto development and validation. In *IEEE Military Communications Conference, 2003. MILCOM 2003.* (2003), vol. 2, pp. 820–825.

[75] Li, G., Palmer, R., DeLisi, M., Gopalakrishnan, G., and Kirby, R. M. Formal specification of mpi 2.0: Case study in specifying a practical concurrent programming api. *Science of Computer Programming 76*, 2 (2011), 65–81. Selected papers from the workshops on Formal Methods for Industrial Critical Systems.

[76] Li, H., Su, Y., Cai, D., Wang, Y., and Liu, L. A survey on retrieval-augmented text generation, 2022. https://arxiv.org/abs/2202.01110.

[77] Li, J. M., Ahmed, A., and Holtzen, S. Lilac: a modal separation logic for conditional probability. *Proceedings of the ACM on Programming Languages 7*, PLDI (2023), 148–171.

[78] Li, J. M., Aytac, J., Johnson-Freyd, P., Holtzen, S., and Ahmed, A. Towards a categorical model of the lilac separation logic. Presented at LAFI, POPL 2024, London, UK, 2024. To appear.

[79] Li, X., Laguna, I., Fang, B., Swirydowicz, K., Li, A., and Gopalakrishnan, G. Design and evaluation of gpu-fpx: A low-overhead tool for floating-point exception detection in nvidia gpus. In *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing* (2023), HPDC, pp. 59–71.

[80] Lopes, N. P., Lee, J., Hur, C.-K., Liu, Z., and Regehr, J. Alive2: bounded translation validation for llvm. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (New York, NY, USA, 2021), PLDI '21, Association for Computing Machinery, p. 65–79.

[81] Martin, W. Broad Agency Announcement: Pipelined Reasoning Of Verifiers Enabling Robust Systems (PROVERS), Mar. 2023. https://sam.gov/api/prod/opps/v3/opportunities/resources/files/f54d93b9d6c54f16adc4561a454b0465/download.

[82] MATZOV. Report on the security of lwe: Improved dual lattice attack, Apr. 2022.

[83] Mayo, J. R., Armstrong, R. C., and Hulette, G. C. Leveraging abstraction to establish out-of-nominal safety properties. Tech. rep., Sandia National Laboratories, 2015. https://www.osti.gov/servlets/purl/1331958.

[84] Menon, H., Lam, M. O., Osei-Kuffuor, D., Schordan, M., Lloyd, S., Mohror, K., and Hittinger, J. Adapt: Algorithmic differentiation applied to floating-point precision tuning. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis* (2018), pp. 614–626.

[85] Miao, D., Laguna, I., and Rubio-González, C. Expression isolation of compiler-induced numerical inconsistencies in heterogeneous code. In *International Conference on High Performance Computing* (2023), Springer, pp. 381–401.

[86] Muller, J.-M., and Rideau, L. Formalization of double-word arithmetic, and comments on "tight and rigorous error bounds for basic building blocks of double-word arithmetic". *ACM Transactions on Mathematical Software 48*, 1 (Feb. 2022), 9:1–9:24.

[87] National Institue of Standards and Technology. Post-quantum cryptography. http://web.archive.org/web/20240121220052/https://csrc.nist.gov/projects/post-quantum-cryptography, Jan. 2024.

[88] Nedunuri, S., and Smith, D. R. Synthesis of correct digital controller models from specifications by model transformation. Tech. Rep. SAND2023-10239, Sandia National Laboratories, 2023.

[89] Nikhil, R. S., and Czeck, K. R. BSV by example: The next-generation language for electronic system design. Tech. rep., Bluespec, Inc., 2010.

[90] Nikora, A., Srivastava, P., Fesq, L., Chung, S., and Kolcio, K. Assurance of model-based fault diagnosis. In *IEEE Aerospace Conference* (Mar. 2018), pp. 1–14.

[91] Pavlovic, D., and Smith, D. R. Software development by refinement. In *Formal Methods at the Crossroads. From Panacea to Foundational Support, 10th Anniversary Colloquium of UNU/IIST, the International Institute for Software Technology of The United Nations University, Lisbon, Portugal, March 18-20, 2002, Revised Papers* (2002), B. K. Aichernig and T. S. E. Maibaum, Eds., vol. 2757, Springer, pp. 267–286.

[92] Pereira, M., and Ravara, A. Cameleer: A deductive verification tool for ocaml. In *Computer Aided Verification: 33rd International Conference, CAV 2021, Virtual Event, July 20–23, 2021, Proceedings, Part II* (Berlin, Heidelberg, 2021), Springer-Verlag, p. 677–689.

[93] Perlner, R., Kelsey, J., and Cooper, D. Breaking category five sphincs+ with sha-256. In *International Conference on Post-Quantum Cryptography* (2022), Springer, pp. 501–522.

[94] Pierce, B. C. The science of deep specification. http://www.cis.upenn.edu/~bcpierce/papers/DeepSpec-workshop-2019-Intro.pdf, June 2019. Opening talk at *DeepSpec* workshop.

[95] Pnueli, A., and Rosner, R. On the synthesis of a reactive module. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 1989), POPL '89, Association for Computing Machinery, p. 179–190.

[96] Pollack, R. How to believe a machine-checked proof. *Twenty Five Years of Constructive Type Theory 36* (1998), 205.

[97] Pollard, S. D. *When Does a Bit Matter? Techniques for Verifying the Correctness of Assembly Languages and Floating-Point Programs*. PhD thesis, University of Oregon, Eugene, OR, USA, June 2021.

[98] Pollard, S. D., Armstrong, R. C., Bender, J., Hulette, G. C., Mahmood, R. S., Morris, K., Rawlings, B. C., and Aytac, J. M. Q: A sound verification framework for statecharts and their implementations. In *8th International Workshop on Formal Techniques for Safety-Critical Systems (FTSCS)* (New York, USA, Dec. 2022), ACM, pp. 16–26.

[99] Pollard, S. D., Johnson-Freyd, P., Aytac, J., Duckworth, T., Carson, M. J., Hulette, G. C., and Harrison, C. B. Quameleon: A lifter and intermediate language for binary analysis. In *Workshop on Instruction Set Architecture Specification* (Portland, OR, USA, Sept. 2019), SpISA '19, pp. 1–4.

[100] Quinlan, D., and Liao, C. The ROSE source-to-source compiler infrastructure. In *Cetus users and compiler infrastructure workshop, in conjunction with Parallel Architectures and Compilation Techniques* (Oct. 2011), PACT, Citeseer, pp. 1–3.

[101] Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F., and Amarasinghe, S. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2013), PLDI '13, Association for Computing Machinery, p. 519–530.

[102] Rice, H. G. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society 74* (1953).

[103] Ringer, T., Palmskog, K., Sergey, I., Gligoric, M., and Tatlock, Z. Qed at large: A survey of engineering of formally verified software. *Foundations and Trends in Programming Languages 5*, 2–3 (Sept. 2019), 102–281.

[104] Rothrock, B., Kennedy, R., Cunningham, C., Papon, J., Heverly, M., and Ono, M. Spoc: Deep learning-based terrain classification for mars rover missions. In *AIAA SPACE 2016* (Sept. 2016).

[105] Russinoff, D. M. Formal verification of floating-point hardware design. *Springer, doi 10* (2022), 978–3.

[106] Rust Foundation. Memory model. https://web.archive.org/web/20231205185143/https://doc.rust-lang.org/reference/memory-model.html, Dec. 2023.

[107] Sawaya, G., Bentley, M., Briggs, I., Gopalakrishnan, G., and Ahn, D. H. Flit: Cross-platform floating-point result-consistency tester and workload. In *IEEE International Symposium on Workload Characterization* (2017), IISWC, pp. 229–238.

[108] Schardl, T. B., Moses, W. S., and Leiserson, C. E. Tapir: Embedding recursive fork-join parallelism into llvm's intermediate representation. *ACM Transactions on Parallel Computing (TOPC) 6*, 4 (2019), 1–33.

[109] Schkufza, E., Sharma, R., and Aiken, A. Stochastic optimization of floating-point programs with tunable precision. *ACM SIGPLAN Notices 49*, 6 (2014), 53–64.

[110] Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Kruegel, C., and Vigna, G. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy (SP)* (May 2016), pp. 138–157.

[111] Silvanovich, N. The state of state machines. Tech. rep., Google Project Zero, Jan. 2021.

[112] Slanina, M., Sankaranarayanan, S., Sipma, H., and Manna, Z. Controller synthesis of discrete linear plants using polyhedra. *REACT Technical Report (Stanford University) 1* (Jan. 2007).

[113] Slind, K., and Norrish, M. A brief overview of hol4. In *Theorem Proving in Higher Order Logics* (Berlin, Heidelberg, 2008), TPHOLs, Springer Berlin Heidelberg, pp. 28–32.

[114] Smith, B., Feather, M. S., Huntsberger, T., and Bocchino, R. Software assurance of autonomous spacecraft control. In *2020 Annual Reliability and Maintainability Symposium (RAMS)* (Jan. 2020), pp. 1–7.

[115] Smith, D. R., and Nedunuri, S. Model refinement. In *16th NASA Formal Methods Symposium* (2024), NFM. To appear. Sand No. SAND2021-12895C.

[116] Soi, R., Bauer, M., Treichler, S., Papadakis, M., Lee, W., McCormick, P., Aiken, A., and Slaughter, E. Index launches: Scalable, flexible representation of parallel task groups. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (New York, NY, USA, 2021), SC '21, Association for Computing Machinery.

[117] Stelle, G., Moses, W. S., Olivier, S. L., and McCormick, P. Openmpir: Implementing openmp tasks with tapir. In *Proceedings of the Fourth Workshop on the LLVM Compiler Infrastructure in HPC* (2017), pp. 1–12.

[118] Teranishi, K., Mukherjee, S., Rutledge, R., Pollard, S. D., Morales, N., Evans, N., Orso, A., and Sarkar, V. Toward automatic test synthesis for performance portable programs. In *3rd International KLEE Workshop on Symbolic Execution* (Sept. 2022).

[119] The Coq Development Team. The Coq reference manual – release 8.19.0. https://coq.inria.fr/doc/V8.18.0/refman, Jan. 2024.

[120] The White House. National security memorandum on promoting united states leadership in quantum computing while mitigating risks to vulnerable cryptographic systems (nsm-10). https://www.govinfo.gov/app/details/DCPD-202200355, May 2022.

[121] Wang, F., Song, F., Zhang, M., Zhu, X., and Zhang, J. Krust: A formal executable semantics of rust. In *2018 International Symposium on Theoretical Aspects of Software Engineering* (2018), TASE, pp. 44–51.

[122] Whyatt, M., Feb. 2024. Pacific Northwest National Laboratory, private communication.

[123] Wildmoser, M., Nipkow, T., Klein, G., and Nanz, S. Prototyping proof carrying code. In *Exploring New Frontiers of Theoretical Informatics, IFIP 18th World Computer Congress, TC1, 3rd International Conference on Theoretical Computer Science* (Aug. 2004), J.-J. Levy, E. W. Mayr, and J. C. Mitchell, Eds., TCS, Kluwer Academic Publishers, pp. 333–347.

[124] Willsey, M., Nandi, C., Wang, Y. R., Flatt, O., Tatlock, Z., and Panchekha, P. egg: Fast and extensible equality saturation. *Proceedings of the ACM on Programming Languages 5*, POPL (Jan. 2021).

[125] Wolf, C., Glaser, J., and Kepler, J. Yosys-a free verilog synthesis suite. In *21st Austrian Workshop on Microelectronics* (2013). http://yosyshq.net/yosys/files/yosys-austrochip2013.pdf.

[126] Xia, L.-y., Zakowski, Y., He, P., Hur, C.-K., Malecha, G., Pierce, B. C., and Zdancewic, S. Interaction trees. *Proceedings of the ACM on Programming Languages 4* (2020).

[127] Yang, J., Yang, Z., J., C., and Ray, S. Correct-by-construction design of custom accelerator microarchitectures. *IEEE Transactions on Computers 73*, 01 (Jan. 2024), 278–291.

[128] Yang, K., Swope, A., Gu, A., Chalamala, R., Song, P., Yu, S., Godil, S., Prenger, R., and Anandkumar, A. LeanDojo: Theorem proving with retrieval-augmented language models. In *Neural Information Processing Systems* (2023), NeurIPS.

[129] Zakowski, Y., Beck, C., Yoon, I., Zaichuk, I., Zaliva, V., and Zdancewic, S. Modular, compositional, and executable formal semantics for llvm ir. *Proceedings of the ACM on Programming Languages 5*, ICFP (2021), 1–30.

[130] Zhao, J., Nagarakatte, S., Martin, M. M., and Zdancewic, S. Formal verification of ssa-based optimizations for llvm. *SIGPLAN Not. 48*, 6 (June 2013), 175–186.

[131] Zhou, Z., Ren, Z., Gao, G., and Jiang, H. An empirical study of optimization bugs in GCC and LLVM. *Journal of Systems and Software 174* (2021), 110884:1–110884:13.