Verifying Systems at Scale
(Advanced Topics in Program Verification)

Suresh Jagannathan

Tu Th 1:30 - 2:45
Phys 110

PURDUE UNIVERSITY
An Ideal …
In Reality ....

Scalability, complexity, and precision challenges abound!

- Libraries
- Models
- Services
- Kernels
- Middleware
- Databases
- Compilers and Runtimes
- Protocols

Continuous Assurance
In Reality ...

- Proofs / Invariants
- Programming Systems

HARD!

Verifier

- program complexity
- model complexity
- specification complexity

Properties

Ok

Bug
But, lots of success stories
In this course …

- Examine techniques and methodologies to address these challenges:
  - mechanized proof assistants
  - type systems and abstract interpreters
  - theorem provers and model checkers

- Study the application of these methods in different domains:
  - Compilers
  - Programming Languages
  - Shared Memory Concurrency
  - Distributed systems
  - File systems and operating system kernels
  - Neural networks

Structure

Paper reading and discussion
Semester project - tool building
Abstract
This paper reports on the development and formal verification (proof of semantic preservation) of CompCert, a compiler from Clight (a large subset of the C programming language) to PowerPC assembly code, using the Coq proof assistant to mechanize the compiler and for proving its correctness. Such a verified compiler is useful in the context of critical software and its formal verification: the verification of the compiler guarantees that the safety properties proved on the source code hold for the executable compiled code as well.

1. INTRODUCTION
Can you trust your compiler? Compilers are generally assumed to be semantically transparent: the compiled code should behaves as prescribed by the semantics of the source program. Yet, compilers—and especially optimizing compilers—are complex programs that perform complicated symbolic transformations. Despite intensive testing, bugs in compilers do occur, causing the compilers to crash at compile-time or—much worse—to silently generate an incorrect executable for a correct source program.

For low-assurance software, validated only by testing, the impact of compiler bugs is low: what is tested is the correctness of the compiler; the source program is assumed to be semantically transparent: the compiled code as well.

For high-assurance software, like the software that controls an aircraft or a nuclear reactor, the impact of compiler bugs is high: what is tested is the correctness of the safety-critical high-assurance software. Here, validation by testing reaches its limits and needs to be complemented or even replaced by the use of formal methods, such as proof of semantic preservation.

The compilation of arithmetic expressions down to stack machine code was mechanically verified in 1972. Since then, many other proofs have been conducted, ranging from single-pass compilers for toy languages to sophisticated compilers for realistic, high-assurance software. This implies a multipass compiler that features good register allocation and some basic optimizations. Despite intensive testing, bugs in compilers do occur, causing the compilers to crash at compile-time or—much worse—to silently generate an incorrect executable for a correct source program.

In the CompCert experiment, we carry this line of work all the way to end-to-end verification of a complete compilation chain from a structured imperative program to machine code. This implies that the compiler must generate code that is efficient enough and compact enough to fit the requirements of critical embedded systems.

A formal verification of the compiler guarantees that the safety properties proved on the source code hold for the executable compiled code as well.

2. STATEMENT
The remainder of this paper is organized as follows. Section 2 compares and formalizes several approaches to establishing trust in the results of compilation. Section 3 presents the semantics of the source programs. For the last 5 years, we have been working on the development of a realistic, verified compiler called CompCert. By realistic, we mean a compiler that is accompanied by a machine-checked proof of a semantic preservation property: the generated machine code behaves as prescribed by the semantics of the source program. By realistic, we mean a compiler that could realistically be used in the context of production of critical software.

In the CompCert experiment, we formally verified a complete assembly code, from C programs to PowerPC assembly code, using the Coq proof assistant. The complete source code of the Coq development, extensively commented, is available on the Web.

3. RELATED WORK
This paper gives a high-level overview of the CompCert compiler and its mechanized verification, which uses the Coq proof assistant. This compiler, classically, consists of two parts: a front-end translating the Clight subset of C to a low-level, structured intermediate language called Cminor, and a tightly optimizing back-end generating PowerPC assembly code from Cminor. A detailed description of Clight can be found in Blazy and Leroy3; of the compiler front-end in Blazy et al.4,5; and of the compiler back-end in Leroy.6

The complete source code of the Coq development, extensively commented, is available on the Web.

The remainder of this paper is organized as follows. Section 2 compares and formalizes several approaches to establishing trust in the results of compilation. Section 3 presents the semantics of the source programs. For the last 5 years, we have been working on the development of a realistic, verified compiler called CompCert. By realistic, we mean a compiler that is accompanied by a machine-checked proof of a semantic preservation property: the generated machine code behaves as prescribed by the semantics of the source program. By realistic, we mean a compiler that could realistically be used in the context of production of critical software.

In the CompCert experiment, we formally verified a complete assembly code, from C programs to PowerPC assembly code, using the Coq proof assistant. The complete source code of the Coq development, extensively commented, is available on the Web.

The remainder of this paper is organized as follows. Section 2 compares and formalizes several approaches to establishing trust in the results of compilation. Section 3 presents the semantics of the source programs. For the last 5 years, we have been working on the development of a realistic, verified compiler called CompCert. By realistic, we mean a compiler that is accompanied by a machine-checked proof of a semantic preservation property: the generated machine code behaves as prescribed by the semantics of the source program. By realistic, we mean a compiler that could realistically be used in the context of production of critical software.

In the CompCert experiment, we formally verified a complete assembly code, from C programs to PowerPC assembly code, using the Coq proof assistant. The complete source code of the Coq development, extensively commented, is available on the Web.

The remainder of this paper is organized as follows. Section 2 compares and formalizes several approaches to establishing trust in the results of compilation. Section 3 presents the semantics of the source programs. For the last 5 years, we have been working on the development of a realistic, verified compiler called CompCert. By realistic, we mean a compiler that is accompanied by a machine-checked proof of a semantic preservation property: the generated machine code behaves as prescribed by the semantics of the source program. By realistic, we mean a compiler that could realistically be used in the context of production of critical software.

In the CompCert experiment, we formally verified a complete assembly code, from C programs to PowerPC assembly code, using the Coq proof assistant. The complete source code of the Coq development, extensively commented, is available on the Web."
Verifying Higher-order Programs with the Dijkstra Monad

Nikhil Swamy1  Joel Weinberger2  Cole Schlesinger3  Juan Chen1  Benjamin Livshits2
Microsoft Research1  UC Berkeley2  Princeton University3

{swamy, juanchen, livshits}@microsoft.com jww@cs.berkeley.edu cschlesi@princeton.edu

Abstract
Modern programming languages, ranging from Haskell and ML to JavaScript, C# and Java, all make extensive use of higher-order state. This paper presents a verification methodology for higher-order stateful programs, based on a new monad of predicate transformers called the Dijkstra monad.

Using the Dijkstra monad has a number of benefits. First, the monad naturally captures a weak pre-condition calculus. Second, the computed specifications are structured similarly in several ways, e.g., single-state post-conditions are sufficient (rather than the more complex two-state post-conditions). Finally, the monad can easily be turned to handle features like exceptions and heap inventories, while retaining the same type inference algorithm.

We implement the Dijkstra monad and its type inference algorithm for the F
classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation class. To copy otherwise, to republish, to post on servers or to redistribute for profit or commercial advantage and that copies bear this notice and the full citation class.

Keywords
Verification
Categories and Subject Descriptors
D.2.4 [Software/ Program Verification]: Validation

General Terms
Verification

In a language like JavaScript, even to reason about simple function calls one needs to reason precisely about higher-order state. This paper advocates a new verification methodology for modern programming languages, ranging from Haskell and ML to JavaScript, C# and Java, all making extensive use of higher-order state.

Our main technical contribution is a new way of structuring specifications for higher-order stateful programs, and of inferring and automatically solving verification conditions (VCs) for these programs using an SMT solver.

Projects

F*
Prusti
Liquid Haskell
Iris from the ground up
A modular foundation for higher-order concurrent separation logic

RALF JUNG
MPI-SWS, Germany
(e-mail: jung@mpi-sws.org)

ROBBERT KREBBERS
Delft University of Technology, The Netherlands
(e-mail: mail@robbertkrebbers.nl)

JACQUES-HENRI JOURDAN
MPI-SWS, Germany
(e-mail: jjourdan@mpi-sws.org)

ALEŠ BIZJAK
Aarhus University, Denmark
(e-mail: abizjak@cs.au.dk)

LARS BIRKEDAL
Aarhus University, Denmark
(e-mail: birkedal@cs.au.dk)

DEREK DREYER
MPI-SWS, Germany
(e-mail: dreyper@mpi-sws.org)

Abstract

Iris is a framework for higher-order concurrent separation logic, which has been implemented in the Coq proof assistant and deployed very effectively in a wide variety of verification projects. Iris was designed with the express goal of simplifying and consolidating the foundations of modern separation logics, but it has evolved over time, and the design and semantic foundations of Iris itself have yet to be fully written down and explained together properly in one place. Here, we attempt to fill this gap, presenting a reasonably complete picture of the latest version of Iris (version 3.1), from first principles and in a coherent narrative.

Iris
Topics and Syllabus

Oct. 19 - Oct. 28: Distributed Programs

- Ivy: Safety Verification by Interactive Generalization
- Verdi: A Framework for Implementing and Formally Verifying Distributed Systems
- Programming and Proving with Distributed Protocols
- Specifying Systems
- Compositional Programming and Testing of Distributed Systems
- IronFleet: Proving Practical Distributed Systems Correct

Projects

IronClad
TLA

Ivy: Safety Verification by Interactive Generalization

Oded Padon
Tel Aviv University, Israel
opedp@mail.tau.ac.il

Kenneth L. McMillan
Microsoft Research, USA
kenmcmil@microsoft.com

Aurojit Panda
UC Berkeley, USA
apanda@cs.berkeley.edu

Mooly Sagiv
Tel Aviv University, Israel
msagiv@post.tau.ac.il

Sharon Shoham
Tel Aviv University, Israel
sharon.shoham@gmail.com

Abstract

Despite several decades of research, the problem of formal verification of infinite-state systems has resisted effective automation. We describe a system — Ivy — for interactively verifying safety of infinite-state systems. Ivy’s key principle is that whenever verification fails, Ivy graphically displays a concrete counterexample to induction. The user then interactively guides generalization from this counterexample. This process continues until an inductive invariant is found. Ivy searches for universally quantified invariants, and uses a restricted modeling language. This ensures that all verification conditions can be checked algorithmically. All user interactions are performed using graphical models, easing the user’s task. We describe our initial experience with verifying several distributed protocols.

Categories and Subject Descriptors D.2.4 [Software/Program Verification]: Formal methods. F.3.1 [Specifications and Verifying and Reasoning about Programs]: Invariants.

Keywords safety verification, invariant inference, counterexamples to induction, distributed systems

1. Introduction

Despite several decades of research, the problem of formal verification of systems with unboundedly many states has resisted effective automation. We describe a system — Ivy — for interactively verifying safety of infinite-state systems. Ivy’s key principle is that whenever verification fails, Ivy graphically displays a concrete counterexample to induction. The user then interactively guides generalization from this counterexample. This process continues until an inductive invariant is found. Ivy searches for universally quantified invariants, and uses a restricted modeling language. This ensures that all verification conditions can be checked algorithmically. All user interactions are performed using graphical models, easing the user’s task. We describe our initial experience with verifying several distributed protocols.
IN FEBRUARY 2017, a helicopter took off from a Boeing facility in Mesa, AZ, on a routine mission around nearby hills. It flew its course fully autonomously, and the safety pilot, required by the Federal Aviation Administration, did not touch any controls during the flight. This was not the first autonomous flight of the AH-6, dubbed the Unmanned Little Bird (ULB); it had been doing them for years. This time, however, the aircraft was subjected to mid-flight cyber attacks. The central mission computer was attacked by rogue camera software, as well as by a virus delivered through a compromised USB stick that had been inserted during maintenance. The attack compromised some subsystems but could not affect the safe operation of the aircraft.

This article explains that change and the technology that enabled it. Specifically, it is about technology developed under the HACMS program, aiming to ensure the safe operation of critical real-world systems in a hostile cyber environment—multiple autonomous vehicles in this case. The technology is based on formally verified software, or software with machine-checked mathematical proofs it behaves according to its specification. While this article is not about the formal methods themselves, it explains how the verified artifacts can be used to secure practical systems. The most impressive outcome of HACMS is arguably that the technology could be retrofitted onto existing real-world systems, dramatically improving their cyber resilience, a process called “seismic security retrofit” in analogy to, say, the seismic retrofit of buildings. Moreover, most of the re-engineering

Formally Verified Software in the Real World

One might think surviving such an attack is not a big deal, certainly that military aircraft would be robust against cyber attacks. In reality, a “red team” of professional penetration testers hired by the Defense Advanced Research Projects Agency (DARPA) under its High-Assurance Cyber Military Systems (HACMS) program in 2013 compromised the baseline version of the ULB, designed for safety rather than security, to the point where it could have crashed or diverged to any location of its choice. In this light, risking an in-flight attack with a human on board indicates that something had changed dramatically.

This article explains that change and the technology that enabled it. Specifically, it is about technology developed under the HACMS program, aiming to ensure the safe operation of critical real-world systems in a hostile cyber environment—multiple autonomous vehicles in this case. The technology is based on formally verified software, or software with machine-checked mathematical proofs it behaves according to its specification. While this article is not about the formal methods themselves, it explains how the verified artifacts can be used to secure practical systems. The most impressive outcome of HACMS is arguably that the technology could be retrofitted onto existing real-world systems, dramatically improving their cyber resilience, a process called “seismic security retrofit” in analogy to, say, the seismic retrofit of buildings. Moreover, most of the re-engineering

Formally Verified Software in the Real World

One might think surviving such an attack is not a big deal, certainly that military aircraft would be robust against cyber attacks. In reality, a “red team” of professional penetration testers hired by the Defense Advanced Research Projects Agency (DARPA) under its High-Assurance Cyber Military Systems (HACMS) program in 2013 compromised the baseline version of the ULB, designed for safety rather than security, to the point where it could have crashed or diverged to any location of its choice. In this light, risking an in-flight attack with a human on board indicates that something had changed dramatically.

This article explains that change and the technology that enabled it. Specifically, it is about technology developed under the HACMS program, aiming to ensure the safe operation of critical real-world systems in a hostile cyber environment—multiple autonomous vehicles in this case. The technology is based on formally verified software, or software with machine-checked mathematical proofs it behaves according to its specification. While this article is not about the formal methods themselves, it explains how the verified artifacts can be used to secure practical systems. The most impressive outcome of HACMS is arguably that the technology could be retrofitted onto existing real-world systems, dramatically improving their cyber resilience, a process called “seismic security retrofit” in analogy to, say, the seismic retrofit of buildings.
Topcs and Syllabus

Nov. 16 - Nov. 30: Neural Networks

- Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks
- AI2: Safety and Robustness Certification of Neural Networks with Abstract Interpretation
- An Abstract Domain for Certifying Neural Networks
- An Inductive Synthesis Framework for Verifiable Reinforcement Learning

Projects

SafeAI
Stanford Center for AI Safety

Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks*
Guy Katz, Clark Barrett, David Dill, Kyle Julian and Mykel Kochenderfer
Stanford University, USA
{guyk, clarkbarrett, dill, kjulian3, mykel}@stanford.edu

Abstract. Deep neural networks have emerged as a widely used and effective means for tackling complex, real-world problems. However, a major obstacle in applying them to safety-critical systems is the great difficulty in providing formal guarantees about their behavior. We present a novel, scalable, and efficient technique for verifying properties of deep neural networks (or providing counter-examples). The technique is based on the simplex method, extended to handle the non-convex Rectified Linear Unit (ReLU) activation function, which is a crucial ingredient in many modern neural networks. We evaluated our technique on a prototype deep neural network implementation of the next-generation airborne collision avoidance system for unmanned aircraft (ACAS Xu). Results show that our technique can successfully prove properties of networks that are an order of magnitude larger than the largest networks certified using existing methods.

1 Introduction

Artificial neural networks [7, 31] have emerged as a promising approach for creating scalable and robust systems. Applications include speech recognition [9], image classification [20], game playing [32], and many others. It is now clear that software that may be extremely difficult for humans to implement can instead be created by training deep neural networks (DNNs), and that the performance of these DNNs is often comparable to, or even surpasses, the performance of manually crafted software. DNNs are becoming widespread, and this trend is likely to continue and intensify.

Great effort is now being put into using DNNs as controllers for safety-critical systems such as autonomous vehicles [4] and airframe collision avoidance systems for unmanned aircraft (ACAS Xu) [13]. DNNs are trained over a finite set of inputs and outputs and are expected to generalize, i.e. to behave correctly for previously-unseen inputs. However, it has been observed that DNNs can react in unexpected and incorrect ways to even slight perturbations of their inputs [33]. This unexpected behavior of DNNs is likely to result in unsafe systems, or restrict the usage of DNNs in safety-critical applications. Hence, there is an urgent

* This is the extended version of a paper with the same title that appeared at CAV 2017.