Trends and Challenges in Algorithmic Software Verification

Rajeev Alur

Department of Computer and Information Science
University of Pennsylvania
Email: alur@cis.upenn.edu

Recent years have witnessed remarkable progress in principles and tools for automated software verification. In this position paper, I briefly discuss the relevant projects in my group, and outline some near-term challenges for the community as concrete milestones for measuring progress.

1 Research Directions

In this section, I will briefly describe some directions we are currently pursuing that can enhance the scope and scalability of software verification tools. More information about these projects can be obtained from my homepage `http://www.cis.upenn.edu/~alur`.

1.1 Model Checking Structured Programs

Classical program verification focussed on correctness of structured procedural programs, while typical model checkers are aimed at concurrent finite-state reactive systems. Recent progress in software model checking allows checking temporal logic requirements of code via abstraction and symbolic state-space exploration. Standard temporal logics such as LTL and CTL employed by current model checkers, however, can specify only regular properties, and properties such as correctness of procedures with respect to pre and post conditions, that require matching of calls and returns, are not regular. Recently, we have introduced a *temporal logic of calls and returns* (CaRet) for specification and algorithmic verification of correctness requirements of structured programs. CaRet can specify a variety of non-regular properties such as partial and total correctness of program blocks and access control properties that involve inspection of the call-stack. Even though verifying context-free properties of pushdown systems is undecidable, we show that model checking CaRet formulas against a pushdown model is decidable. This result allows us to combine the classical Hoare-style reasoning about structured programs, Pnueli-style temporal specifications of reactive programs, and automated reasoning as in model checking. The decidability of CaRet against pushdown automata is not ad-hoc, and we have developed a theory of *visibly pushdown languages* that is rich enough to model interprocedural program analysis questions and yet is tractable and robust like the class of regular languages. Current efforts include extending these results to branching-time logics and tree automata.

1.2 Games and Interfaces

While a typical software component has a clearly specified (static) interface in terms of the methods it supports, the information about the correct sequencing of method calls is usually undocumented. For example, for a file system, the method *open* should be invoked before the method *read*, without an intervening call to *close*. While such interfaces can be made precise using, for instance, regular expressions as types, these kinds of precise specifications are typically missing. Such dynamic interfaces for components can help applications programmers, and can possibly be used by program analysis tools to check automatically whether the component is being correctly invoked. In the JIST project, we are developing a novel solution for automatically extracting such temporal specifications for Java classes. Given a Java class, and a safety property such as "the exception E should not be raised", the corresponding *(dynamic) interface* is the most general way of invoking the methods so that the safety property is not violated. Our synthesis method first constructs a symbolic representation of the finite state-transition system obtained from the class using *predicate abstraction*. Constructing the interface then corresponds to solving a *partial-information two-player game* on this symbolic graph. We have developed a sound approach to solve this computationally-hard problem approximately using algorithms for learning finite automata and symbolic model checking for branching-time logics. A preliminary implementation of the proposed techniques has succeeded in constructing interfaces, accurately and efficiently, for sample Java2SDK library classes.

1.3 Real-time and Hybrid Systems

Embedded systems, such as controllers in automotive, medical, and avionic systems, consist of a collection of interacting software modules reacting to a continuously evolving environment. Despite the proliferation of embedded devices in almost every engineered product, development of embedded software remains a low level, time consuming and error prone process. This is due to the fact that modern programming languages abstract away from time and platform constraints, while correctness of embedded software relies crucially on hard deadlines. The CHARON project at Penn aims at developing novel model-based design and implementation methodology for synthesizing reliable embedded software using the foundations of *hybrid systems*. Hybrid systems models allow mixing state-machine based discrete control with differential-equation based continuous dynamics. In the past, we have developed algorithms and tools for model checking of timed and hybrid systems. More recently, we are developing a programming environment using hybrid models with constructs such as hierarchy, concurrency with synchronous continuous interaction, and preemption. A key technical challenge in this work is bridging the gap between the platform-independent and timed semantics of the hybrid models and the executable software generated from it. This is crucial to be able to infer properties of software from properties of models. We are also exploring ways of integrating generation of control tasks with scheduling.

2 Challenge Projects

Research in formal methods has typically focussed on questions such as "what is the most expressive temporal logic that can be algorithmically verified," "what is the most effective way of pruning the search for a satisfying assignment for a propositional formula in clausal form," and "what are all the errors that can be found in existing code using this highly optimized tool for pointer analysis?" Such questions have led to improved understanding and technology, and are clearly essential for progress. At the other extreme, one can speculate about "grand" challenges such as developing a verifying compiler, or certified software, or new design paradigms. Instead, I have tried to articulate some concrete suggestions regarding some sample projects that are feasible within the next five years. I believe that they all share the following characteristics. First, each project is beyond the ability of individual researchers as well as current technology. Second, with sustained commitment and collaboration among a group of researchers, the project seems feasible. Third, the progress on each challenge can be measured and evaluated, and the goal of the project is clearly articulated. Finally, success in these projects will have notable impact in terms of addressing the skepticism among computer scientists concerning viability of formal methods, and in education and dissemination of our tools.

2.1 Automatic Graders for Classical Programming Assignments

Typical programming assignments in undergraduate courses include writing recursive programs such as *Quicksort*, and writing multi-threaded programs with synchronization such as *Dining Philosophers*. Typically, the submitted code is checked in an automated manner by executing it on sample tests. The goal of this challenge is to develop automatic grading programs that can *verify* the submitted code.

While formal methods tools have reported amazing successes, the heuristics underlying the decision procedures employed by the tools tend to be very fragile, and using these tools is always a frustrating experience for the novice users. To develop automatic grading programs, we need to focus on the *robustness* and *usability* of the tool rather than performance measures such as the number of lines of code analyzed by the tool or the time taken to analyze published benchmarks. Consequently, developing such automatic graders is an interesting challenge for the verification technology. Software model checking is not yet a scalable technology, but I believe with some effort, it is possible to develop a robust grading tool for analyzing the submitted code by exploiting the knowledge of the problem being solved during the abstraction phase.

The criteria for evaluating progress and success in this project are clear: with some training, the students who have correct solutions should be able to get their code verified by the tool, and the instructor should find the tool valuable enough so that (s)he recommends the tool to other instructors for use. If successful, this will make educators around the world, and the undergraduate students, aware of the promise and utility of formal methods and program analysis.

2.2 Conformance Checkers for Network Protocols

Network protocols has been a fruitful domain for application of formal methods. Existing efforts in formal analysis of network protocols are usually aimed at formalizing the specification of the protocol and proving that it has desirable properties. There have been some efforts in constructing executable code from such rigorous specifications. A complementary effort that can serve as a challenge to automated tools will be to establish that existing code for a protocol such as `tcp` or `dhcp` conforms to the published RFC specification. The RFC specification is partially in the form of state machines, and can possibly be extracted using automated tools. To demonstrate that C code implementing such a protocol in, say Linux, is a refinement of the RFC specification in a formal sense, is a non-trivial task that would require combination of several abstraction and analysis technologies. Again, I feel that this is something that is beyond the abilities of the existing tools, but can be achieved with domain-specific focus and efforts.

The success in this project should eventually lead to a scenario in which regulatory agencies will publish new RFCs in a form which can be analyzed by the tool, and the industries will be compelled to publish evidence of the conformance of their implementations with respect to the RFC specifications. Such success will result in improved reliability of widely used network protocols, and also adoption of formal methods by the industry.

2.3 Code Generator for Simulink

Contemporary industrial control design already relies heavily on tools for mathematical modeling and simulation. The most popular of such tools is Simulink developed by Mathworks. While commercial tools for compiling Simulink models into executable software exist, they do not offer any verification guarantees for the generated code. Programming languages community has largely ignored this domain, but as embedded devices become more ubiquitous, the importance of embedded systems programming will increase. When embedded software is employed in safety-critical applications such as automobiles and autonomous medical devices, the need for high assurance is obvious. Consequently, the current state of poorly understood relationship between the models and the code is not satisfactory. This creates an opportunity and a challenge problem for the formal methods community. Applying formal methods to introduce rigor into the code generation step is an interesting research challenge that can have a significant impact on reliability of control software. The semantic gap between the model and the code is large: the semantics of the model is typically defined as a solution to differential or difference equations, while the software consists of periodic tasks. This gap makes the code generation problem particularly challenging. Research in synchronous languages, time-triggered architectures, real-time scheduling, control theory, and hybrid systems offers insights into this problem, and can be exploited to build a semantics-preserving code generator for Simulink-like models.

2.4 Stateful Specifications for Java Libraries

As explained in Section 1.2, *behavioral interfaces* can capture temporal constraints such as the method `initSign` must be invoked before calling the method `sign`, without an intervening call to the method `initVerify`. Rigorously specified behavioral interfaces can play a critical role in documentation, maintenance, testing, verification and ultimately, in ensuring provable security properties of the system. Researchers in programming languages and software engineering have made a variety of proposals for notations for specifying interfaces (a prominent effort in this direction is the Java Modeling Language), automatic extraction of such interfaces, and for enforcement or verification of conformance of usage. A unified effort to annotate all the classes in public Java libraries and open source applications with stateful interfaces in machine readable and formal notation will be a worthy project.

To make this project a reality, first we need to agree on the notion of a stateful interface and a formal notation for expressing it. Second, we need to develop tools that can either extract such interfaces automatically from existing libraries, or find ways that will facilitate people to add these annotations. Finally, the interfaces will be useless unless we have effective analysis tools that will check client code against the interfaces or check conformance of newer versions of libraries with respect to interfaces for older versions. The combination of these activities make this an interesting project requiring collaboration and commitment from a team of researchers. Given the wide-spread use of Java, and the enthusiasm for open source projects, I believe that such interfaces will find expected as well as unexpected use.