Formal Methods:
Future Directions & Transition To Practice
Workshop Report

Ranjit Jhala[1], Rupak Majumdar[2], Rajeev Alur[3], Anupam Datta[4], Daniel Jackson[5], Shriram Krishnamurthi[6], John Regehr[7], Natarajan Shankar[8], and Cesare Tinelli[9]

[1] UC San Diego
[2] Max-Planck Institute For Software Systems
[3] University of Pennsylvania
[4] Carnegie Mellon University
[5] MIT
[6] Brown University
[7] University of Utah
[8] SRI International
[9] University of Iowa

1 Introduction

Recent years have seen many success stories where formal methods, or ideas influenced by formal methods, have transitioned successfully from the research lab to development tools. Yet, formal methods have remained the broccoli of the computing world: nutritious but not palatable, or, at any rate, an acquired taste. In domains where there is a catastrophic cost of failure (e.g., in hardware design, in safety critical embedded applications, or in security-critical domains), formal method techniques have become an indispensable part of the design and implementation process. But in the broader computing world, the potential for formal methods in the development process is not widely understood, and formal techniques are often treated with skepticism or dismissal. Thus, in December 2012 the National Science Foundation sponsored a workshop that brought together formal methods researchers and practitioners to take stock of the situtation, identify successes, the obstacles towards broader adoption, and to chart future directions for formal methods and its effective transition to practice. The workshop brought together researchers from academia, industrial research laboratories, and government organizations, as well as practitioners from formal methods based tool vendors.

What have we Achieved? In order to look forward, it is helpful to appreciate how far the field has travelled already. In 1996, the ACM organized a workshop with similar goals [10] . Thus, we decided to kick the workshop off by polling the participants about what they perceived to be the successes of the field, which could be used a lens to examine how the formal methods landscape had changed over the course of the last decade and half. The responses can be grouped into several categories:

Specification, Design & Modeling These include high-level modelling tools like StateCharts and Alloy which can be used to analyze and refine *designs*, domain-specific tools such as Simulink, Scade, and Stateflow in embedded systems, and others. Furthermore, the last decade saw a great deal of progress on standardizing extensible specification languages, such as JML, which has streamlined the development of various formal tools for *implementations* as well.

Language Technologies Perhaps the most pervasive formal methods are those that are not even viewed as formal methods anymore! These include the pervasive use of static type systems in mainstream languages like Java and C#, the use of synthesis tools like parser generators, the reliance on aggresive compiler transformations for program optimization, and liberation from manual memory management through garbage collection.

Hardware Analysis The last decade and half saw a huge adoption of methods in the design and implementation of hardware. These include VLSI synthesis and equivalence checking. It is routine for large EDA vendors like Cadence, Synopsys and Mentor Graphics to ship tools that use model checking for assertion-based verification. Companies like Intel and AMD have in-house teams that use interactive theorem provers like HOL or ACL2 to formally verify the correctness of various logic blocks.

Software Analysis Perhaps the least anticipated successes came in the realm of software, where remarkably effective tools based on model checking (e.g. SPIN), abstract interpretation (e.g. SLAM/SDV and ASTREE), or dataflow analysis (e.g. Coverity, KlocWork, Fortify) made static analysis mainstream, inducing influential software developers like John Carmack to write [11]

> "The most important thing I have done as a programmer in recent years is to aggressively pursue static code analysis. Even more valuable than the hundreds of serious bugs I have prevented with it is the change in mindset about the way I view software reliability and code quality."

[10] Clarke and Wing, "Formal methods: state of the art and future directions", ACM CSUR, Vol. 28, Iss. 4, Dec 1996

[11] `http://www.altdevblogaday.com/2011/12/24/static-code-analysis/`

Formal Testing In a related vein the last decade saw the adoption of a variety of formal approaches to software testing. Two prominent strands of work include, the development of *property based testing* pioneered by Quickcheck and now adopted across many modern languages, and thanks to the robustness of SMT solvers, the widespread use of symbolic and "concolic" execution for automatic testing, and due to the advances in SAT technology, the adoption of bounded model checking for hardware.

Proof Assistants While interactive proof assistants like ACL2, PVS, Isabelle, and HOL have been in development since the seventies and eighties, and adopted by hardware vendors like AMD and Intel, the last few years have seen a large upsurge of interest in the systems and languages communities thanks to several landmark projects such as then Sel4 formally verifed microkernel, and the CompCERT compiler.

Core Technologies Finally, many of the above successes rely crucially upon advances made in core algorithmic advances such as improvements with Binary Decision Diagrams (BDDs), propositional "SAT" solvers, and SMT solvers. Decades of sustained research on these problems came to fruition as we saw a revolution in the scalability of these tools, enabling their widespread use as generic constraint solving mechanisms in domains from dependency management to biology.

What has Changed? What is Changing? Thus, as is apparent from the litany of successes above, there has been a silent revolution in formal methods since the mid-1990s, as the stakes for reliable software and systems have grown vastly higher.

Application Areas Our news media remind us daily of security concerns about our infrastructure and the privacy concerns of individuals. The growing use of medical devices with significant computational features demands greater attention. The pervasive embedding of computing devices in all aspects of our lives, from smartphones to smart meters to self-driving cars, means these concerns will only grow. The deep reliance of our financial system on software adds to these pressures; and recently, attention has been drawn to the risks of bad computation adversely affecting policy decisions.

Tools The widespread adoption of safe languages makes it more likely that tools will be able soundly to establish the absence of certain classes of errors. The widespread popularity of non-trivial type systems in mainstream languages like Java and C# further strengthens this power. Moreover, we are now seeing the advent of a new generation of tools to help programmers—from partial verifiers to checkers for rich type systems—released by academic groups, industrial labs, and sometimes even commercial development groups. Finally, popular software engineering techniques like model-driven programming have the potential to be large-scale enablers of more advanced formal methods.

Resources A lack of resources of diverse sorts—in computing power, in training materials, in insights into the needs of users—has hindered the adotion of formal methods. In all these areas, change is appearing. Cloud services provide enormous computing power; large quantities of data, such as open-source program repositories, make it easier to study the patterns of errors programmers make and identify effective interventions; the Web enables large-scale dissemination (and sharing) of materials such as books and courses, as well as making it possible for expertise to be shared easily.

Attitudes Major corporations, which influence views about dependability and security in the public's mind, are paying more attention to security and safety risks; and the posture of the companies that are addressing security issues most aggressively will help change perceptions about the need for better development techniques, and for certification. At the same time, the

formal methods community has made a conscious effort to reach out to practitioners. In recent years, the formal methods community has focused on providing usable tools to programmers which, while they fall short of the ideal of complete functional correctness proofs, are nevertheless very useful in improving programmer productivity.

What did we find? In short, the pervasiveness and growing centrality of computing makes this a challenging and exciting time for formal methods. The goal of the workshop was to reflect upon the most effective ways for the formal methods community to influence the computing revolution. To this end, we structured the workshop along four (related) focus areas.

- **Infrastructure & Community Synergies(§ 2)** As the field develops, it is increasingly harder for researchers to make progress working in isolation. Building large scale verification tools requires a lot of effort in developing scalable infrastructure. Often, much of the effort is duplicated across projects. The goal of this area was to identify community synergies, for example, in defining standard intermediate formats.
- **Education & Outreach (§ 3)** A major stumbling block to the adoption of formal methods is often lack of expertise in the workforce. If formal methods are to be broadly adopted, they must be accessible to all engineers, not only specially trained Ph.D.s. The purpose of this direction is to recommend education and outreach activities that will lower the bar to the adoption of new techniques, as well as inform tool developers about the usability requirements of developers and quality assurance engineers.
- **Roadmap to Adoption (§ 4)** The goal of this area was to identify effective ways of transitioning research tools to industrial practice. Adoption requires better scalability and effectiveness of the tools, demonstrating effectiveness, and better usability of the tools.
- **Frontiers & New Challenge Domains(§ 5)** The rapid spread of computing machinery through a variety of critical domains has necessitated the development of specialized techniques that ensure the reliability and correctness of the software used in those domains. The purpose of this focus area was to identify the new domains where rigorous guarantees are required and which are ripe for current- or next-generation formal methods and tools.

In this report, we summarize the discussions from the four areas, in each case, identifying successes, obstacles and promising thrusts for future work. With a strong, co-ordinated and strategically chosen pushes we can significantly influence all these trends for the better, allowing society to continue to reap benefits from computing technology, while helping to build a foundation for a safer, more secure and more dependable future.

2 Infrastructure and Community Building

While formal methods technology has advanced significantly in recent years, the fruits of these advances are not yet readily available to the average software or hardware engineer. In this section, we outline the role of *community* and shared infrastructure in consolidating the success thus far, pinpoint a few concerns and prominent obstacles that must be overcome to accelerate the wider adoption of formal methods, and describe some promising approaches and future directions for achieving those goals.

Infrastructure refers to the set of resources that are used for performing research and delivering the fruits of this research to a broad audience. This broad definition spans a broad spectrum: from hardware resources such as server farms and compute clusters, to software resources like libraries, formal languages (including specification languages and domain-specific languages), extensible frameworks for model checking, theorem proving, program analysis, and verification.

Community refers to a (possibly geographically distributed) group of researchers who share a common goal, notation, or framework. For example, we have communities around individual programming languages that share libraries and code. Similarly, the ACL2, Coq, Isabelle, or PVS communites are centered around a theorem prover and can share libraries of specifications, proof tactics, and theories. We also have a group of researchers centered around common representation languages like LLVM, SMT-LIB, JML, or Boogie so that different front-end and back-end tools can be reused.

2.1 Successes

We outline several success stories of community- and infrastructure-building within formal methods. Quite importantly, as our list below shows, successful infrastructure projects have come out of both academia and industry, and through various informal partnerships between academia and industry.

Languages and Compiler Frameworks Languages, exchange formats, and compiler infrastructure provide shared infrastructure on which one can develop verification technologies. By "languages" we mean not just programming languages but also compiler frameworks that parse and lower often complex front-ends and intermediate formats for verification artifacts (such as notations for specifications, formulas, and transition systems).

Several mainstream programming languages, such as Common Lisp, ML, OCaml, Haskell, and Scala, directly incorporate advances in formal methods technologies, most prominently, expressive type systems. Additionally, languages such as Haskell and Scala provide a sandbox for further language research through flexible mechanisms for language extensions and embeddings of domain-specific languages. For example, several recent projects explore software synthesis and programming by constraints as embedded DSLs within Scala.

The TPTP, DIMACS, SMT-LIB, AIGER, Boogie, and Why3 notations allow sharing formulas and benchmarks between verification projects, and have been instrumental in the rapid advances in solver technology. Additionally, common annotation languages, such as JML and ACSL, allow tool developers and verification engineers to share verification artifacts such as code specifications (pre- and post-conditions).

Finally, tool development in formal methods is helped by shared infrastructure that deals with the problem of reducing (complex) front-ends to manageable intermediate languages. Compiler infrastructures, such as LLVM and CIL, provide intermediate representations for software verification; the Boogie and Why3 languages provide clean intermediate representations for verification which form the back-end to many software analysis tools.

Core Libraries and Verification Tools A second direction is the development of robust components that can be reused across many different projects. These include SAT and SMT solvers (such as Chaff, MiniSAT, CVC4, MathSAT, Yices, and Z3), first-order theorem provers (such as E, SPASS, Vampire), core data structures (such as Apron, GMP, CUDD), model checkers (SMV, NuSMV, SAL, Spin, Java Pathfinder, ARMC, Blast, CBMC, CPAChecker), computer algebra systems (Maple and Mathematica), interactive theorem provers (ACL2, Coq, HOL, Isabelle, PVS), etc. More recently, verification frameworks such as Frama-C, KeY, Dafny, KIV, VCC, and Verifast integrate several of these components into a code verification tool. There are also attempts to help the integration of formal methods tools, such as the idea of an Evidential Toolbus.

Benchmark Suites Often, advances in tools and infrastructure are catalyzed by a well-chosen representative set of benchmarks. In several sub-disciplines in formal methods, most notably in the SAT and SMT communities, but more recently in hardware and software model checking,

annual "tool competitions" against a large set of benchmark problems have been crucial in improving the state of the art in tool development. Benchmark suites have been developed for various different verification tools, e.g., the SATLIB and SMT-LIB benchmarks, verification benchmarks arising out of the hardware and software model checking competitions, and logic synthesis and verification benchmarks in AIGER format.

Formal Methods as a Service More recently, there are several infrastructure projects offering "formal methods as a service." These include the SMT-Exec and StarExec execution services for benchmarking solvers as well as services on top of platform-as-a-service cloud providers for formal methods engines, such as the Cloud9 service for symbolic execution on the cloud.

2.2 Obstacles

While there are several local success stories of cooperative projects, the broader formal methods research community has not yet been mobilized around an overall shared pursuit. The formal methods research community can come together in a number of ways to systematically eliminate barriers to research and the wider adoption of formal techniques in mainstream computing. We now outline the main obstacles to "large-scale" research identified in the workshop.

Robust and Usable Infrastructure Developing robust and usable infrastructure for formal methods is a difficult task, but it is often a prerequisite for impactful research. For example, building front-end processors that map popular languages and notations to those used by formal tools is a significant software development project. While such infrastructure enables many research activities, the research community is often hesitant to embark on such a project, because there is no direct scientific benefit in building the infrastructure itself. Individual researchers, particularly students, therefore focus on specific research challenges that can be solved without the need for massive investments in software infrastructure.

Standard Formats and Benchmarks Standard formats enable community efforts by providing common platforms that can be shared across researchers, and common benchmarks enable researchers to directly compare different techniques and tools. While standard representations and benchmarks have been successful in some domains (e.g., SAT and SMT), the absence of standard formats across the board have been an obstacle in large-scale infrastructure development and cross-disciplinary collaboration.

Educational mnd Support Materials Finally, we consider the development of adequate educational materials as a prerequisite to transitioning formal methods from the lab to standard practice. Researchers enjoy solving technical challenges in developing their tools. However, without an adequate body of user manuals, tutorials, and technical documentation, it is hard for people outside the formal methods community to identify if a specific method is a fit for their problem, and to invest in deploying and learning a tool in the context of a real project. Similarly, new ideas in formal methods have to transition to the computer science curriculum, and this again requires significant investment in developing course materials (lecture notes, problem sets, projects and associated infrastructure).

Until there is a proven base of infrastructure, it will not be easy to embed formal methodologies into industrial design processes. At the same time, it is a mistake to require all scientific projects to be applicable to current industrial practice. Researchers often develop focused languages and models to study a scientific phenomenon in isolation, and there is sometimes a tendency to discount such research as "toy problems" irrelevant to the "real world." However, many significant experiments and explorations are best conducted using toy languages. This kind of isolation and separation of concerns has always been the hallmark of any science, and computing is no different.

The most promising ideas arising from these investigations can then be adapted to more realistic languages. Practical and popular programming languages are often the outcome of a number of design compromises that make them less than conducive for foundational experiments. Industrial-strength languages and related development tools raise a number of interesting challenges, but if we focus exclusively on the problems of these languages, we might miss opportunities for radical, paradigm-breaking research.

In other words, there is a need to strike a balance between foundational research focusing on the use of focused languages for developing and prototyping new ideas that can be adapted to more popular languages and applied research addressing the real-world challenges of scale and usability.

2.3 Future Directions

Formal methods technology is rapidly maturing, and this presents both opportunities and challenges. Technology based on formal modeling, analysis, and synthesis, is making an impact on hardware and software development, as well as on other fields such as systems biology. With the availability of mature tools and the emergence of standardized languages and interfaces, we now have the opportunity to make formal methods an integral part of the design lifecycle of complex hardware and software systems.

Thus, based on the identified obstacles, we recommend the following concrete and intertwined steps be taken with respect to infrastructure development in order to spur further advances and adoption of formal technologies.

Concrete Goals: Shared Standards and Platforms The first, and most readily achievable group of steps is to follow the example of efforts like SMTLIB and JML and develop *standard formats* for various formal technologies. Specifically:

- Adoption of standardized languages for specification, program annotation.
- Adoption of standardized intermediate languages and interchange formats for data structures and libraries.
- Development of unified tool interfaces and tool integration frameworks that can be used for rapid prototyping and experimentation.

The standardized formats for specification, invariants, intermediate languages and so on, can be used to develop significant *hardware and software platforms* that can act as *repositories*, as ways for users without heavy computing resources to use formal techniques "in the cloud", and ideally to help integrate and embed formal techniques within mainstream design tools and methodologies. To this end, we recommend:

- Creation of large-scale shared hardware infrastructure for experimental evaluation of FM technologies.
- Deployment of community portals such as SATLive, Starexec, and CSP-VO.
- Development and maintenance of public benchmark and problem suites.
- Implementation of shared suites of software utilities that can be used in developing formal tools and tool chains.

Building Communities A more complex future goal is to build research communities around specific formal methods problems. Research communities serve an important purpose in motivating good research and in both broadening and deepening the impact of the results. In the case of formal methods, community efforts can help kickstart ambitious formal verification projects and experiments that require large well-coordinated research teams. They can also help create a cadre of early

adopters, people who are willing to apply formal techniques in the development of new hardware and software designs. There is also a need to build bridges to other subdisciplines such as hardware design, systems software, programming languages, databases, and embedded systems.

However, many of the incentive mechanisms in academia are not directly aligned with promoting community-oriented research activity. Researchers often do not get a lot of academic credit for infrastructure development. Funding for such work is scarce. Infrastructure work does not directly yield very many publications, and there are not a lot of metrics for evaluating the impact of such work. Graduate students seeking academic positions are evaluated on publications rather than system development.

Still, the situation is not bleak if one looks at precedents in other fields. Disciplines other than computing do have a tradition of rewarding high-quality experimental work. Even within computing, the numerical analysis and systems communities have found a way to incentivize high-impact software development. For graduate students who go off to work in industry, experience in infrastructure development can be a strong qualification. Overall, there are research models that are more conducive to community-oriented research, and there is a significant payoff for this kind of research in terms of users, citations, impact, and funding.

There already is a healthy amount of cooperation in formal methods, but there is room for plenty more. There are now a number of competitions for theorem provers, SAT and SMT solvers, hardware and software model checkers, and verified software. These are cooperative efforts that involve crafting benchmarks, problems, and notations that can be shared across the groups. One effective mechanism for advancing formal methods is by flagship "grand challenge" projects. In the past, projects such as seL4 kernel verification or the CompCert verified compiler have both advanced formal methods technology, fostered collaborative efforts, and marketed formal methods to the broader computing community. Another mechanism for cooperation is through the development and use of standardized languages, interfaces, and shared software components, as we outlined before.

We emphasize that infrastructure- and community-building has to be accompanied by incentive mechanisms that provide academic credit for community-oriented research and infrastructure building, for example, through research funding for long-term infrastructure development (beyond the lifespan of a single PhD dissertation) and through acknowledgement of value in the scientific evaluation process.

3 Education and Outreach

Ultimately, the adoption of formal methods will depend crucially on our ability to *educate* engineers in the importance and application of formal tools, techniques, and thinking. One can envision focusing education efforts at three levels: high-school, undergraduate, and graduate.

While we agree that one could target education at all levels, the main discussion at the workshop gravitated towards formal methods education in the undergraduate curriculum. The general sentiment was that graduate study is more specialized, and in many ways where formal methods has already been used extensively, at least in some contexts. At the other end, it was unclear that much can be done at the high-school and lower levels without massive investment of resources and radically new strategies for adoption. In contrast, focusing on the undergraduate level is likely to provide the greatest chance to influence not only future software developers but also others who will play a role in how software and hardware is created, procured, and evaluated.

3.1 Successes

As a whole, there is a great deal more optimism about the potential for formal methods today than there was even one or two decades ago.

A major catalyst for optimism is the widespread change in the commercial programming landscape. Whereas previously languages like C++ were dominant, today's programmers often gravitate towards safe languages (Java, C#, or Python, etc), that offer memory safety and automatic memory management. Furthermore, when programmers use statically typed languages (such as Java or C#), they are arguably *already using* formal methods, albeit in the most lightweight fashion. Language safety itself is a critical enabler for the successful, scalable application of richer methods, such as techniques that prove properties about programs. There is even some adoption in the wider programming community of research-driven languages such as OCaml, Haskell, and Scala, in part because of the influence of scripting languages (such as JavaScript and Ruby) that included features from functional programming.

Beyond standard programming, we are also seeing the use of formal methods across the curriculum. For instance:

- The large-scale use of formal methods to model programming infrastructure such as the JVM (in languages such as ACL2 and Isabelle, for example) have resulted in examples that are more readily explained to undergraduates than more domain-specific ones.
- Many research universities are starting to offer material on modeling and proving properties about languages using Coq at the undergraduate level (e.g., Brown, Maryland, Pennsylvania, Princeton), often using Pierce's *Software Foundations* materials. While experiences vary, the sentiment is generally positive. The fact that solid numbers of students are interested in taking such courses is in itself telling, and as learning and tool support improves, this material is likely to see wider adoption.
- Some universities (e.g., Northeastern, Oklahoma, etc.) have successfully used theorem provers (such as ACL2) to teach students how to prove properties not just about the languages they use but about the programs they write. Some of this material has been taught as early as the freshman year, thereby setting a tone for how students might approach the rest of their Computer Science education.
- Numerous universities use systems like Alloy to teach students formal modeling and lightweight analysis techniques. The focus here is usually on modeling the essence of a diverse range of real-world systems, thereby showing the broad applicability of formal methods. Moreover, the use of model finders (such as Alloy) and model checkers (such as SPIN or SMV) gives students immediate and concrete feedback, often in the form of counterexamples, which makes the experience of understanding formal models more engaging, less burdensome and far more concrete.

Of course, the majority of the above are special classes explicitly focused on formal methods, or more broadly, software engineering or programming languages. Looking ahead, it will important to introduce and disseminate these ideas throughout the curriculum, allowing their power and relevance to be demonstrated in context. For example, one could use formal methods in security courses (e.g., at the University of Maryland) or in classes focused on the design and implementation of embedded systems (e.g., at the University of Pennsylvania).

3.2 Obstacles

That formal methods can be successfully applied in diverse domains like security, embedded systems, and networking is an indication of the power of the tools currently available. Next, we consider some

of the limiting factors and identify some opportunities for improving the state of education and outreach for formal techniques and tools.

Teaching Materials for Non-Specialists As discussed above, it is critical that we integrate formal techniques within the computing curriculum, for example in classes on embedded systems, security and so on. One significant obstacle towards this goal is a dearth of suitable material to support teaching of formal methods content by non-specialists. As an analogy, due to the abundance of excellent textbooks, topics like data structures and algorithms, which may once have been solely taught by theoreticians, are now routinely taught by non-specialists. Similarly, key algorithmic ideas are taught in context, e.g. scheduling in an OS class, shortest paths in a networking class, by domain experts. Thus, one concrete goal for the formal methods community should be to develop materials that can be used by teachers who are not formal methods experts, but are instead specialists in the specific usage domains.

Antiquated Notions of Formal Methods Many attendees shared the concern that faculty members in other fields tend to have a somewhat old-fashioned view of formal methods. They are likely to be aware of the seminal ideas of program verification (such as Hoare logic, loop invariants, and weakest preconditions) but are less likely to be familiar with advances in automated theorem proving, model checking and lightweight approaches. When such faculty members make curricular decisions and advise students, this narrowness can have a significant negative impact. Fortunately, recent developments like the adoption of static analysis, the success of symbolic execution in discovering vulnerabilities, the use of formal techniques to model and analyze software defined networks, and large scale developments like the sel4 OS kernel are helping to dispel such antiquated notions.

Bug Hunting vs. Design The success and emphasis on bug-hunting and post-facto verification tools is a double-edged sword as it gets in the way of thinking clearly about design in the first place. As design is subjective and hard to define, we have a tendency to shy away from it and focus on issues, such as correctness, that are more easily evaluated (but which may actually be less important). However, long experience has shown us that design mistakes are often the hardest to recover from, and a formal mindset can be invaluable in clarifying design challenges and articulating design decisions. Looking ahead, we believe that more emphasis should therefore be placed on design methods and correctness by construction, and that today's single-minded focus on post-facto verification could be detrimental.

More generally speaking, perhaps the most important obstacle that arose in our discussion of education was the common tendency (sometimes referred to as "vitamins versus aspirin") to optimize for identifiable short-term gains at the expense of less tangible but more important long-term ones. Formal methods as a field seems to suffer from this problem.

3.3 Future Directions

Next, we recommend some concrete lines of work that should be pursued to overcome the obstacles identified above. Broadly speaking, we believe it is important to identify and consolidate and develop educational materials covering the "big ideas" in formal methods, to dispel antiquated notions about the field, to develop materials for teaching formal methods "in context", and to tap into the enormous potential that MOOCs have of reaching beyond traditional classrooms.

Big Ideas In Formal Methods Therefore, we thought it would be valuable to list some of the key ideas that typify formal methods thinking, and illustrate some of the breadth and depth of the field:

- Recursion and induction as two perspectives on the same idea
- Declarative specifications as a way of thinking about programming at a higher level
- Invariants, specification, and *under*specification
- New idioms for parallelism: MapReduce, functional programming, transactional memory, etc.
- Relational modeling of complex state
- The Curry-Howard isomorphism, relating types and proofs
- Safety, liveness, and other abstract properties of systems
- Tests as a bridge to more complex formal methods
- Advances in model checking: bounded model checking, partial-order reduction, interpolation, etc.

Formal Methods In Context There are many ways to motivate the consideration and adoption of formal methods.

One is through the application of formal methods in context. We have already mentioned embedded systems and security as two good contexts to consider. Other possibilities include:

- Building a secure and privacy-preserving Web browser
- Parallelism and concurrency
- Hardware verification and bug-finding
- The design of mobile, distributed applications
- Methods to verify cryptographic protocols

Many cautioned, however, that asking colleagues who teach these courses to incorporate formal methods is unlikely to succeed; the burden is on the formal methods community to provide notes, tools, and other components of a course module that would enable easy incorporation.

The other—orthogonal—approach is through the use of methods that make formal methods more accessible:

- Lightweight formal methods tools, such as Alloy
- Run-time assertions
- Testing oracles

Online Courses Many participants felt that the growing trend towards on-line courses offers huge potential. Having high-quality, on-line courses, and especially course modules, achieves several ends:

- Individual instructors can collaborate to create materials on their respective areas of expertise, which can then be re-mixed to create a variety of different courses.
- Instructors who do not have resources to develop full courses of their own (e.g., professors at liberal arts colleges who already have heavy teaching loads) can now incorporate material into their classes that they may not have had time to develop fully by themselves.
- Individual students who are sufficiently motivated can learn this material without having to rely on the offerings of their institutions, as can working professionals who no longer have an academic affiliation.
- Employers who are noticing a trend towards certification in their area can make taking such courses a requirement for their employees.

There is therefore potentially great value in investing resources to make such modules available.

Looking Ahead We need especially to be aware of the fact that dramatic changes are possible, however unlikely they might seem. Twenty years ago, the widespread adoption of safe languages and garbage collection seemed unlikely; now this is barely commented on. Similarly, languages like Haskell that wallowed in academic obscurity now have strong and growing followings.

Therefore, we should be looking ahead to where the goalposts might be two decades from now and aiming in that direction, acknowledging that big changes take time but not despairing when they seem not to come in the short term.

4 Adoption and Technology Transfer

In order to transition formal methods into widespread practice, we must identify the key technical problems that must be addressed to develop usable methodologies and tools, but perhaps more importantly, determine the dissemination, incentive and reward mechanisms that will facilitate adoption among students and industry practitioners. That is, our goal is to elucidate the kinds of research that is needed to remove the obstacles and create the opportunities for transition to practice.

Implicitly, we recognize that not all FM researchers want to perform technology transfer, nor should they. Rather, the goal is worthwhile at the level of the entire community because it tends to create a culture rewarding the creation of robust and usable solutions to real problems, as opposed to clever point solutions with little relevance.

Although this section will mainly focus on the transfer of technology into industry, there are, of course, other targets for transfer. For example, transferring formal methods into the classroom may serve the goal of producing a new generation of developers who know more about formal techniques and who will be perhaps more likely to bring these tools with them into development positions.

4.1 Successes

"Success stories" for formal methods are numerous. Some of them, like parsing, code optimization, and type systems, have become so ingrained in the computing milieu that we do not think of them as nominal success stories. Formal methods tools have found significant adoption in other areas, such as formal verification of hardware (equivalence checking, model checking, semi-formal methods), model-based design of embedded systems, and (more recently) in the analysis and testing of software. Additionally, formal methods ideas have deeply influenced computing practice, for example, in programming language design, systems design, security and privacy, etc.

We list these successes not merely to highlight them and hold them up as exemplars, but to use them to pinpoint the set and sequence of activities required to transition basic research results into practical use. We focus the discussion on *direct* adoption of formal methods technology, such as new tools or methodologies.

Sometimes, formal methods researchers push an idea all the way from basic research to a production-grade tool with a substantial user base. For example, the Astree teams performed the entire range of activities shown in Figure 1. The achievements of these teams were enabled by factors such as (a) A relatively large team with diverse expertise (b) Close access to developers who have verification problems and who are motivated to use formal tools (c) Stable funding over a period of at least 10 years

While the SLAM and Astree projects are exemplary, it is not realistic to expect or even desire that all (formal methods) researchers— even the best ones —to do research with such a broad scope. Individual researchers may simply prefer to operate at the scope of one or more arrows shown in Figure 2 and also they may be forced to generate results on a shorter time frame. For

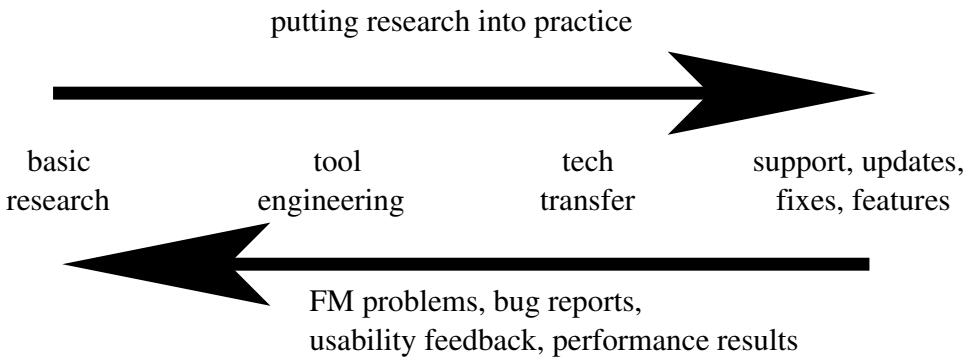

Fig. 1. Transition by A Single Group: In one style of formal methods research, a single team performs all of the activities required to move basic research results into practical use

example, the typical durations of a PhD, a tenure clock, and an NSF grant are all well under a decade. Also, the machine learning concerns (for assigning priorities to software defects) and HCI (human-computer interface) concerns involved in creating effective methods for allowing formal tools to convey information to users, may simply be beyond the scope of expertise of a typical research group.

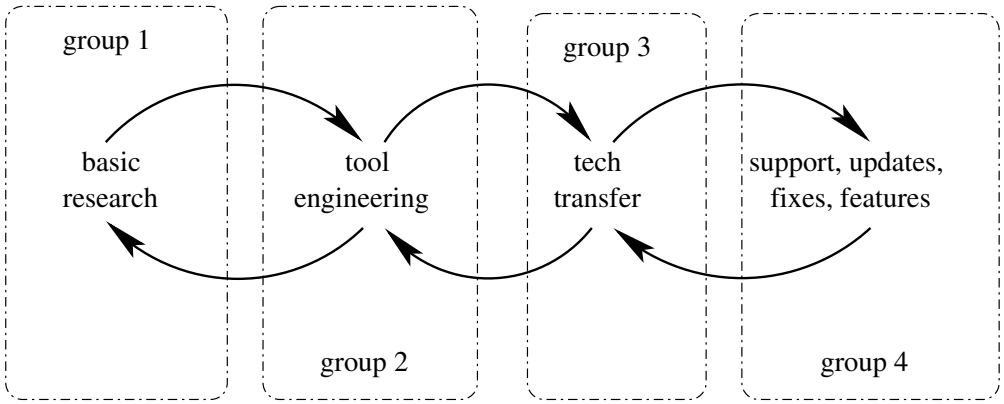

Fig. 2. Transition by A Community: More commonly, different groups of researchers collaborate directly and indirectly—and often, over a long period of time—to bring research ideas into production. This style of research can be supported by papers that include code and data, open source experimental artifacts, shared benchmarks, etc.

4.2 Obstacles

The two *modes* for successful transitions to practice indicate several obstacles that must be addressed in order to increase adoption. Broadly speaking the obstacles can be grouped into the purely *technical:*: *e.g.,* developing faster, more scalable algorithms and tools and making it easier to connect tools developed by different groups, to *user-centric: e.g.,* understanding ways in which users interact with formal tools, and helping them find and evaluate appropriate methods, to the *social:* finding ways to build communities of consumers and producers of formal methods and connecting the two, and identifying incentives for using formal techniques, and for infrastructure development and technology transfer.

Tools: Scalability & Integration A formal tool will only be used by practitioners if it *scales* to large, real-world designs and code bases, and if it is seamlessly *integrated* into the development workflow.

Many formal methods rely on some form of combinatorial search and hence are limited to small systems. The first obstacle is one familiar to most researchers – that of building formal tools that scale up to large designs and codebases. Thus, for successful adoption, it is essential that we continue to develop techniques to improve the scalability of various formal techniques. Of course, every million line program is fashioned from many thousand or ten thousand line components. Thus, it is equally important to find ways to identify the components, develop sophisticated mechanisms for inferring how they interact and thereby, divide and conquer the scalability problem.

Similarly, one of the underappreciated aspects of adoption is the importance (and difficulty) of insinuating formal tools within the development workflow so that using these tools becomes a natural part of the engineering process (e.g. akin to hitting `make` after editing a source file) instead of an elaborate foreign process. Indeed, according to the biggest challenge in getting the Coverity static analyses commercially adopted was not so much the program analysis bottlenecks as getting the tools inside the development environment.

Composable Tools One significant challenge to building tools that can be readily integrated into developers' environments, is that, as indicated in Figure 2, such tools are pipelines fashioned as from many small(er) but sophisticated pieces. The *semantic gap* between these pieces – *e.g.,* the output produced by one and the input required by the next – is a significant obstacle to building widely usable tools. While large organizations may have enough resources to develop everything from scratch in a monolithic fashion, clearly this is infeasible for smaller groups, and yet, an essential obstacle to adoption.

For example, in the domain of hardware verification, one of the places where formal methods has a lot of success, there are several excellent model checkers each of which require as input transition systems expressed in a certain format, but there is a dearth of parsers that convert Verilog into the input format, which makes the model checkers hard to use. Similarly, when analyzing software there are several alias analyses that characterize the shape of the heap while ignoring data values and there are Hoare-logic based verifiers that can precisely reason about data values, but it is very difficult to get the two to interoperate because there is no common language that the two speak.

A related obstacle, is the multitude of incompatible and incomparable formalisms and languages supported by different formal tools. One panelist reported that a common complain from potential users interested in formal methods was that they look out there and theres a vast sea of languages, tools, and techniques, and we have no idea what to pick. Worse, users fear picking the wrong tool, writing specifications, hints and annotations for that tool, and then being "locked into" their poor choice forever. That is, potential users are wary of picking a specification technology and putting in all the work of writing the specifications and requirements as they are afraid of having to redo all that work when a better verification engine becomes available. (On the positive side, we note that this particular problem is a happy one to have – ten or fifteen years ago, no one would have complained about the *abundance* of formal tools!)

Composable Models As mentioned above, every large system is built from smaller components. To scale, we have to build tools that analyze particular components. One significant obstacle towards this goal is that in addition to the tool, a great deal of effort is required to *build models* that suitably characterize the *environment* within which the component executes. Several panelists point out that modeling the environments with enough accuracy to be useful turns out to be very expensive.

For example, working our way up through the "computing stack" when checking hardware or device drivers, one often requires precise models of the Operating System or the effects of system calls; similarly, when analyzing low-level machine code, one requires a precise semantics for ISA (e.g. x86), microprocessor memory models, or of the semantics of special operations for specialized embedded platforms. Further up the stack, software applications make heavy use of libraries or frameworks like Boost or STL (C++) or JQuery (JavaScript) to name but two. These libraries often offer complex, extremely domain-specific abstractions that cannot be recovered directly by analyzing their code bases.

Typically, the environment model is developed in an an ad-hoc, tool-specific manner which prevents reusing the model across different analyses and groups. Furthermore they must be rebuilt when the implementations change. (In some cases, *e.g.,* the exact semantics of the x86 instruction set are trade secrets and must be painstakingly "discovered" by academics, like natural science artifacts!)

Thus, to facilitate adoption, it is essential that we find ways to *reduce the cost of building models*, to build *general purpose* models that can be reused across different kinds of verification tasks and groups, thereby amortizing the cost of model building. Some panelists pointed out that to maximize reuse, researchers should focus on developing appropriate models for open-source components, not merely because they are readily accessible, but because over the last decade they have become essential to commercial software development.

Incentives From the above, it is clear that it would be extremely worthwhile to encourage the creation or more, high-quality pieces of shared and reusable formal methods infrastructure that will make it easier for researchers operating at the scope of an individual arrow in Figure 2 to have impact. Similarly, researchers can have practical impact without the (significant) difficulty of building a tool with a broad user base if their methods are integrated into existing open-source tools and frameworks.

It is plausible that formal methods research would advance at a faster pace if more kinds of infrastructure were available. For example, while embedded software offers an attractive target for formal techniques, at present there is a notable lack of open-source embedded systems including artifacts such as full-system simulators, models, formal specifications [12], test vectors, documentation, and other elements that formal methods researchers could exploit.

Indeed, most instances of applied formal methods research could not be performed at all without significantly leveraging existing infrastructure. Perhaps the most striking example has been the factoring out of domain-independent SAT and SMT solvers from domain-specific formal methods tools. Theorem provers such as ACL2, Isabelle, and PVS and their large (and growing) theory libraries have demonstrated similar benefits to researchers. Most recently, the CIL, LLVM, WALA, frameworks have drastically lowered the bar for developing industrial-strength analyses for C and Java.

Several panelists felt that in each of these cases, the incentive mechanisms for researchers – e.g. publications and grant funding – are not commensurate with and hence, not suited to encourage the significant effort required to build and maintain crucial pieces of shared FM infrastructure. For example, one of the panelists has recently gotten a relatively small amount of dynamic error checking code, which corresponds to the implementation part of a research paper, added to LLVM. The exercise of interacting with the LLVM community, fitting the changes into the existing architecture in a maintainable way, etc., multiplied the total effort of this research project by several times.

[12] First steps in this direction have been taken, as in the Pacemaker Formal Methods Challenge: `http://sqrl.mcmaster.ca/pacemaker.htm`

Similarly, George Necula, the author of the CIL library wrote: "When I started to write CIL I thought it was going to take two weeks. Exactly a year has passed since then and I am still fixing bugs in it. This gross underestimate was due to the fact that I thought parsing and making sense of C is simple."

In fact, currently, much of the shared infrastructure is developed either in industry or by researchers in their spare time – as J. Moore, put it: "You do it because that's what *moves* you". Perhaps progress could be significantly accelerated if as a community, we could identify ways of rewarding, recognizing and hence, incentivizing infrastructure building efforts.

Cognitive Gaps Between Tools and Users Perhaps an even bigger *gap* is that between formal tools and the *humans* that must ultimately use these tools. Of course, in theory we want tools that offer "push-button automation". In practice, however, the user must possess a clear understanding of how to express the *properties* they are trying to check. Next, they must often have a reasonable understanding of how the tool works, at the very least, to know what knowledge about the system, and its interfaces are required by the tool and where the tool may *lose information*. Finally, they must be able to *interpret* the results returned by the tool to fix errors or modify the design or implementation.

As a concrete example, let us examine the above issues in the setting of type systems, the most widely used formal method in existence today. The first problem – expressing properties is relatively well understood by programmers. (Of course, advanced users will be able to design their programs to capture more properties within types, but that is another matter.) However, the more advanced the type system, the more the user must know of its inner workings. For example, it may be difficult to use Hindley-Milner based systems without at least some familiarity with how unification works as otherwise it is difficult to understand why the type checker rejects certain programs.

Paradoxically, the more advanced the formal method, and the greater its automation, the harder this problem becomes. The user must know *more* about the inner workings of the method. It is perhaps for this reason that Java's more verbose and explicit type system is viewed as more user-friendly than ML-style systems which support aggressive inference.

Several panelists noted that model checking based tools have the great advantage of producing counterexamples that allow the user to step through a concrete *execution*. It was noted that in general, tools that help users map errors concretely back to source positions, thereby shortening the cycle of find-and-fix, saw much greater adoption than tools that actually found more bugs but produced results that were harder to understand.

Finally, a significant consequence of the semantic gap is that potential users find it difficult to *choose* a tool, that is, to identify which formal technique may be best suited to their particular problem domain. One part of the solution here is better education and outreach – for example, if one is looking to verify assertions in Java then one ought to know about loop invariants, Floyd-Hoare logic, JML and the related suite of tools. However, another part is developing tools, methods and ultimately, abstractions, that don't require users to have deep knowledge about their inner workings in order to be effectively used.

Understanding Users Perhaps the broader issue that has not been addressed head on by the formal methods community is that psychology and technology are very closely intertwined in software and hardware engineering. For some users (and use-cases and tools), formal methods are perceived as tedious while others view them as addictive like "video games". How can we identify the features that make a technique more like the latter than the former?

So far, there is relatively limited (quantitative) understanding of how formal methods are *used* by practitioners, even for pervasive methods like type systems. Do users use a particular method for

high-level design, diagnosis and debugging? Or continuously while developing implementations? Or for post-facto quality assurance? Are the users also the developers or designers? What features make a particular tool more or less attractive to a user? What differentiates a "novice" users interaction with a tool from that of a "expert"? How can this information be used to assist the conversion of the former into the latter?

These questions are somewhat outside the realm of traditional formal methods inquiry, and yet are likely to be critical in spurring adoption. Fortunately, they are squarely within the ken of machine learning and HCI research, indeed, one may argue that formal methods is, at its core, a human-computer interaction problem. Thus, it may be worth exploring ways to encourage formal methods researchers to form collaborations with machine learning and HCI groups.

Adoption as a Social Process Technology transfer is primarily about people and relationships rather than tools and technology. Assuming the existence of a solid formal methods tool, moving it into practical use by a product group requires a collection of circumstances:

- The tool's customers must be in significant pain. Lacking a strong incentive to use a tool, they will simply continue developing software as they have in the past. From the point of view of a product group, changing the software development process adds risk and cost to an already risky and costly activity.
- The tool must have a strong champion on the product group side.
- There must be a long-term commitment on both the research side and the product group side.
- Researchers must be responsive to feedback about the tool.
- It must be possible to estimate the ROI (return on investment) due to using the tool.
- Tool technology must be robust, predictable, and simple to use.
- The vocabulary offered by the tool should be in terms that a product group already understands and uses. For example, developers understand test cases and debuggers, but probably not loop invariants or terms in separation logic. Tools that emulate compiler warnings or debugging and testing activities may fit into the workflow better than tools that appear to do completely new things—regardless of what is going on behind the scenes.
- Tools should offer a good value proposition in the short term as well as the long run. It has been observed that if the first error reported by a FM tool is not just bogus, but also "looks stupid" from the point of view of a developer, it is unlikely that that developer will continue using the tool.

Clearly, it can be difficult to make these circumstances come together. Therefore, it is important that researchers have other ways of making their work into a meaningful part of the broad, community-wide effort to improve software quality using formal methods.

Measuring Success Just as there is a dearth of quantitative data on how formal tools are *used*, there is a lack of metrics and data on the *benefits* offered by these approaches, which in turn, is a major obstacle towards persuading engineers and managers that development processes should be shifted to accomadate formal methods.

Both informally and formally (in the sense that corporations attempt to maximize shareholder value), the true impact of formal methods is economic. The engineering community already understands how to create good software when cost is no issue. The challenge is to produce an artifact of specified quality at a specified time for a specified cost. Formal tools that make it easier or less risky to achieve these goals are highly valuable, and will be readily adopted. Thus, we should think of formal methods as a process of optimization. However, since quality has proven to be difficult to measure, it is difficult to optimize for.

There has been a great deal of research on measuring reliability, typically using notions of mean time between failures or the number of errors observed in a stochastic environment. However, these metrics are quite insufficient in the presence of *adversaries* or attackers that are actively looking for, and hence greatly amplifying the probability of hitting errors.

Similarly, while there are studies and data on defect rates, bugs and vulnerabilities within particular components, languages, or projects, one can view these as *symptoms* of a disease or problem, but there is little insight into what sorts of development practices *prevent* such issues in the first place. That is, how can we quantify the notion of "technical debt", by quantifying the potential long-term harmful effects of particular short term design or implementation choices.

Basic distinctions, such as whether a formal methods effort is aimed at finding bugs or verifying their absence, have a strong effect on the flavor of the tools that are eventually produced. It is worth observing that since real software development efforts almost never have a shortage of known bugs that are waiting to be fixed, simply providing developers with additional bugs to fix may not be helpful. The bugs that developers really need to know about are the ones that cannot be found using testing, but that will cause negative consequences after deployment. Thus, a better metric for a formal methods tool than "bugs found" would be "bugs found that were considered important enough to fix" or "bugs labeled as critical by developers." Similarly, a better metric for an formal methods tool than "programs verified" would be "days of testing time saved because developers trusted verification results."

4.3 Future Directions

The obstacles identified identified previously naturally point the way towards many important directions for future research.

Incremental Verification One way to address the problems of scalability and integration with the development workflow would be to devise new theories and tools targeted towards *incremental verification*. As observed earlier, we, as a community, already "know" how to engineer large, verified artifacts when cost is not a constraint. The key obstacle is how to do it cheaply with "real programmers" (who may not have Ph.Ds in formal methods.) However, new artifacts – hardware or software – are not developed from scratch. They are typically modifications that build upon and retain much of the behavior of older versions. Thus, the key research question is, suppose we have already paid a high cost – compute power or human expertise – to analyze an original version, say from five years ago, *can we quickly and cheaply analyze the changes to verify the version from one week or one hour ago?* That is, with incremental verification, the cost of the initial (potentially herculean) verification effort becomes less relevant; what matters is what it costs to *keep verifying* the software. Similarly, if the goal is defect detection, then the same principle leads to the notion of a "semantic baseline" that characterizes the bugs or warnings that, for one reason or another, will not be fixed. The research questions are *how do we characterize bugs so that they are not detected or reported again?*

Tool Interoperability Recall that the problem of tool interoperability manifests itself in two vexing ways. First, we have the *external* problem that each formal tool has its own specification language, which makes commercial users reluctant to pay the cost of writing specifications as they fear being "locked into" that tool. Second, we have the *internal* problem of the semantic gaps between the information produced and consumed by different formal techniques, which makes it difficult to compose them to build more powerful user-facing tools.

The *external* interoperability problem may prove easier to solve, through the development of suitable standards. For example, in the late 1990s, there were many specification languages for Java, but now they have been standardized as JML [13]. Similarly, recent efforts have lead to common standards for C, namely the ANSI/ISO C Specification Langage (ACSL) [14].

Some cases of the *internal* problem, for example connecting front-end parsers and back-end solvers, interoperability boils down to a matter of engineering the right interchange formats, such as SMTLIB. More generally, however, for example connecting different kinds of analyses – like alias- and value- abstractions – we may require new abstractions that allow the analyses to interoperate. Consequently, several panelists cautioned against looking for a general, silver bullet solution to the *internal* interoperability problem, noting that such efforts, to develop general frameworks proved fruitless in the EDA domain. Instead, it is likely that solutions will only emerge in a domain-specific fashion. A notable success here is the Nelson-Oppen framework for building co-operating decision procedures and the *Satisfiability Modulo Theories*(SMT) architecture for composing theory solvers with propositional logic (SAT) engines. More recent directions include the use of datalog, set constraints, and Horn-clauses to specify program analysis problems, which could prove to be an effective way of decoupling language specific "front-ends" which generate the program analysis *queries* from "back-end" solvers, and also of dividing-and-conquering large queries using multiple cooperating solvers.

Incentives via Gamification & Competitions Many of the obstacles discussed earlier can be crossed by identifying *incentives* that reward *researchers* who engage in long term infrastructure development, and *users* who spend the time to familiarize themselves with and gain expertise with formal methods and tools.

The various tool *competitions* like the SAT, SMT model checking and verification competitions and challenge benchmarks have proven to be an extremely effective catalyst for infrastructure building for their particular domains. There are several intertwined benefits that such competitions can offer. First, a means of building a cohesive research community for the particular challenging problem domain. Second, they provide a stimulus for standardizing the various input and output formats for the tools in that domain, and hence, for rigorously evaluating the relative merits of different algorithms and tools. Third, and perhaps most importantly, they provide a concrete incentive to constantly find new improvements to the tools, as placing well in the competitions provides means of recognition other than the traditional publication, which is often difficult given the inherently so-called "incremental" nature that many improvements that are essential to adoption. Thus, going forward, we should try to identify newer domains that are ripe for similar competitions and challenges.

Of course, competitions are only possible when there is enough "critical mass" in a particular problem domain that one can find participants. Another way of recognizing and rewarding infrastructure building would be to foster a pervasive culture of releasing *code* alongside every applicable research paper? Indeed, several conferences have started a formal process of *artifact evaluation*[15] one of whose goals is to recognize and reward infrastructure building efforts.

At the same time, we need to develop new ways to incentivize users to pay the cost of learning and using formal tools (in addition to lowering those costs, of course.) One immediate direction is to identify concrete quantitative metrics that argue the case. Ultimately however, formal tools

[13] `http://www.eecs.ucf.edu/~leavens/JML/index.shtml`
[14] `http://frama-c.com/acsl.html`
[15] `http://cs.brown.edu/~sk/Memos/Conference-Artifact-Evaluation/`

will only be used if they are *fun* to use. Thus, it could be worthwhile to explore ways in which the principles of could be used to spur adoption.

In particular, we might develop tools with explicitly marked levels of expertise and reward users with "badges" as they gained more expertise. We could look for ways of making the somewhat solitary activity of interacting with a formal tool more social by finding ways in which multiple users could either compete or collaborate with each other to find defects or proofs. Similarly, instead of just demonstrating, dryly that a particular methodology had "proven" some system to be safe, we might create a game where players are rewarded if they can "break" a verified system, or can insert errors that slip past the system. These games would attract potential users not familiar with the innards of the formal technique, and create tangible evidence and incentives, in the form of attackers who were unable to penetrate the system.

The security community has already embraced these approaches. For example, the malware economy is a giant version of gamification, researchers can earn tens of thousands of dollars in bug bounties from Mozilla and Google by finding errors via fuzz testing. Similarly, the security community has adoped *Capture The Flag*(CTF) competitions where teams compete to break into and defend a given computing platform, thereby gaining experience with real-world attacks and how to secure machines against them. These competitions all rely heavily upon various kinds of reverse-engineering, including code and protocol analysis, and hence could be ripe for showcasing the virtues of formal methods.

Formal Methods on The Web The time is more than ripe for formal methods researchers to explore creative ways of using "the cloud" to address a variety of obstacles like like scalability, ease-of-deployment, understanding users, as well as, more generally, building communities of expert users that can help others choose the right formal tool for a given task.

Many of the obstacles to adoption have to do with *understanding users* and use-cases, which in turn, rely on easy dissemination of tools and building a community of users in the first place. Simply *packaging* and *installing* tools so that they work across different platforms (OS, Libraries, Compilers) is challenging enough to deter most researchers. This greatly limits the number of potential users and hence prevents the virtuous feedback loop where usability concerns can be understood and addressed. These problems could be addressed if researchers made their tools web-based (i.e. available via a browser client "front end" that invokes the tool remotely on a server "back-end").

Web-based tools would establish a two-way communication with collections of users who would otherwise have been difficult to find. Users would have access to a variety of different tools, and researchers would be able to get insight into the kinds of problems the tools were being used for, precise usage logs, and particular usability concerns that may not be apparent *a priori*. For example, by gathering and exploiting statistics on usage, researchers may be able to craft better diagnostic messages, error metrics, or even heuristics for inferring properties. As a side benefit, the increased accessibility would make it much easier to integrate the tools in classes, so engineers enter industry with experience and familiarity with formal tools. Making tools available on the web could also help with issues like scalability since particularly heavy analyses – e.g. model checking large state spaces, analyzing an exploding number of paths – could be run on more powerful machines (or clusters) than individual users have access to.

Once tools are easily deployed over the web, researchers should undertake the goal of cultivating communities of users who can help each other use the tools, and can, ideally help with adding and improving the tool in the first place. While such collaboration has always happened, the growth of the internet, and sites like `github` make it trivial to share source code and accept contributions

from third parties. For example, the widely used LLVM infrastructure has grown to its current prominence, as it supports a wide gamut of analyses contributed by a large community of developers. Of course, this was only made possible as it was carefully architected to support the composition of analyses. These communities can be fostered through the use of blogs, social media, and sites like `stackoverflow` which help connect experts with new users by getting the former to answer questions posed by the latter. Similarly, we might envision the creation of an "app" store for formal method tools, which, in addition to hosting different tools, would help to share experience and expertise. Users could write reviews or provide feedback on which tools work best in which situations, which would both help improve tools, and help new users find the right tool for their particular task. Through judicious use of these sites, we could arrive at an inflection point where we can "democratize" formal methods broadly in the computing community.

Understanding Users: Psychology & HCI At the root of many of the above lines of work is a need to properly understand the target audience for formal methods: the developer. How do they think about design? How do they create architectures? How do they actually write code? For example, we may find that developers learn a relatively fixed repertoire of techniques (building blocks); they break a given problem down via an incomplete case enumeration, and then address each case by suitably assembling the blocks. At any rate, a clear understanding of developers' mental processes is key to identifying the shortcomings and limitations of *informal* methods during the design, implementation and validation of computing systems, which can then be profitably complemented by formal approaches.

Thus, the ultimate question when evaluating a formal tool is not how fast or scalable it is, but how *effective it is in the hands of practitioners.* While we are all familiar with the basic scientific approach of creating test and control groups of users, designing experiments to measure the effectiveness of particular fine-grained aspects is somewhat outside the traditional expertise of the formal methods community. These experiments might address, for example, questions like: What makes a tool fun to use? What are the optimal psychological feedback mechanisms? How do we split large analyses into smaller tasks that can be competitively or collaboratively solved by different groups of users? How does the tool generate feedback – be it requests for hints, or bug reports – that is most easily comprehended and acted upon by the user?

Thus, in order to spur adoption, we believe it is essential that the formal methods community complements its lack of expertise in these areas by engaging with researchers who have expertise in the study of *humans*, for example, human-computer interaction (HCI) experts, psychologists and even sociologists who have started to devise creative ways to use the web to dramatically increase the scale and fidelity of their experiments, and hence, could prove to be invaluable collaborators in the design and evaluation of various gamification approaches. While this line of research may appear nebulous and far fetched, several panelists felt that it would take a few pioneers to design and execute the experiments and share their knowledge to popularize this route within the rest of the community.

Funding agencies like the NSF could encourage the building of these bridges. Future proposals can look at previous work for guidance in how to perform these experiments (at the very least, in adhering to the strict legal requirements when dealing with human subjects.) Furthermore, agencies could require or create incentives for formal methods proposals that explicitly have a component that looks to study how people will use the proposed method or tool.

5 New Frontiers

What is next for formal methods? Does it have the potential to transform existing domains and to catalyze new domains? For long-term viability of the field, it is important to take a step back and identify problem domains where formal methods can play a crucial role. We hope new problem domains will provide focus to formal methods research and drive foundational advances, and conversely, these advances will be validated via application to real problems in these domains.

As we list new frontiers and opportunities for applications of formal methods, we want to first emphasize the criticality of the ongoing research in the core technologies in methodology and tools for specification and verification. We do not believe that formal methods, at its core, is a solved problem and only waiting to be adopted in different application domains. Additionally, we point out several application domains in which formal methods has made inroads, and others where formal methods may make a difference. Accordingly, we organize this section along three axes: *foundations*, *inroads*, and *targets of opportunity*. We recognize the difficulty of predicting the future; these lists are not meant to be exhaustive, but only suggestive.

5.1 Foundations

Core research in formal methods spans two directions: methodology, the development of specification languages and verification techniques, and tools, the development of computational engines and tuning them for enhanced scalability. We assert that these problems are still far from being solved; moreover, as applications change, they lead to new sets of core problems in methodology and tools. We enumerate a set of foundational challenges in formal methods which have all seen partial progress, some adoption, but still require more research. Our list is not meant to be exhaustive, but a sampling of topics that have seen breakthroughs in recent years but which still consume the attention of the community.

Constraint-based analysis High-efficiency constraint solvers (aka SAT and SMT solvers) have been a huge success story for formal methods, with applications ranging from, among others, program verification to test generation to compiler optimizations for hardware and software. Research in core constraint-solving technologies (more expressive logics, domain-specific theories) and their interfaces (e.g., direct support and optimizations for specific domains) continue to be important. For example, we are seeing first generation solvers for non-linear arithmetic (useful in verifying control systems) and heap analysis, direct support for software verification using Horn clauses, and interfaces to diverse applications such as OS package management. SMT solvers are increasingly seeing integration into programming languages and interactive environments, and we expect this trend to continue.

Synthesis In current software development practice, programming and verification are distinct phases, typically executed by separate teams. Improved verification technology allows detection of subtle bugs, but this only means a costly reiteration of the design and programming phases, and not necessarily a quality improvement. A plausible alternate is *synthesis*, which starts with a higher level specification of the desired behavior and produces implementations which are correct by construction. Synthesis has found different incarnations over the years, such as the synthesis of finite-state reactive controllers from temporal specifications, synthesis of functional programs from declarative logic-based specifications, and semantics-preserving rewriting of inner loops in imperative programs. It has also provided many computational challenges, such as developing efficient

game-theoretic algorithms, decision procedures for quantified constraints, iterative schemes for abstraction and refinement, and data-driven inductive learning. All these remain vibrant areas with active research and potential applications.

An emerging vision is to make synthesis interactive, by allowing the programmer to express the desired functionality via different artifacts —not necessarily logical formulas— described using different styles. Such artifacts can include programs (that may not yet be complete), declarative specifications of high-level requirements, positive and negative examples of desired behaviors, and optimization criteria for selecting among alternative implementations. This diversity is aimed at allowing a programmer to express her insights in a way natural to the domain, leading to a more intuitive and less error-prone way of programming. The synthesis tool composes these different views into a unified, concrete implementation using computational engines and programmer interaction. An early example of such a system is the Sketch programming environment. The potential for synthesis in transforming system design is immense, and recent advances in many diverse areas —from hardware design to mobile application design to massive online education— are extremely encouraging.

Design space exploration An ambitious goal is to use modeling and analysis for both correctness and performance optimization. For example, the design of a processor in a mobile device requires logical correctness, but minimizing energy consumption is also of critical importance; in design of a wireless control network consisting of sensors, actuators, and computing elements, the implementation needs to ensure stability of the control system, while minimizing response time and communication costs. Formal design for safety and performance can be formalized using parameterized models. The goal of the analysis tool is to explore the space of parameters to find values that ensure that the resulting model meets all the correctness requirements, and are optimal with respect to certain performance metrics. There is a lot of exciting theoretical work incorporating quantitative models into verification —a shift from "model checking" to "model measuring" and "model optimization"— we expect to see the development of efficient analyses and integration into the system-design cycle.

5.2 Inroads

Security The importance of developing scientific foundations for security has received wide recognition in government, academia, and industry. These initiatives recognize that foundational advances in security science is needed to develop systems that can resist attacks by sophisticated adversaries. Formal methods have an essential role to play in the development of this science.

Formal methods has had significant impact and successes in security, with many successful transfers of technology from research to industry. Formal verification of cryptographic protocols formalized finding subtle bugs in many of them). Language-based security improved our understanding of languages and type systems to reason about security and secure systems. Among other successes, it provides the basis for reasoning about information flow in complex systems. Static and dynamic analyses are routinely applied to detect common classes of security problems such as buffer overruns, integer overflows, cross-site scripting attacks, identifying malware, etc. Test case generation for security flaws (based on symbolic execution) is routinely used in Microsoft.

Formal methods will continue to play a foundational role in security. One open direction is reasoning *compositionally* about security. Another is the application of adaptive defense mechanisms to systems that are partly software, but also interact with the physical world (cyber-physical systems), such as energy and communication grids and medical systems, which underlie critical infrastructure. Modeling suitable adversaries, formalizing security, and understanding security of physical systems are open problems.

Systems Biology Computational modeling of biological processes and their automated analysis are becoming increasingly common. Techniques from formal methods have been applied to modeling biological processes (e.g., as state machines, as hybrid systems, as stochastic systems), to provide an executable model for the process that can be analyzed through exhaustive model checking techniques. In several instances, such models have complemented wet-lab experiments and have been used to predict the behavior of biological systems that were then confirmed in the wet lab. Conversely, biological models have given rise to foundational problems in formal methods, for example, the algorithmic analysis of stochastic systems and the use of statistical techniques in model checking.

The interaction between formal methods and biology are still in the early phases, and there are many challenges, both in modeling and in analysis. In modeling, a challenge is to identify appropriate abstractions of biological systems, and be able to "step through" different abstraction levels. These abstractions have to be complemented by experimental techniques in biology that can confirm whether the abstraction is meaningful. In analysis, a challenge is to scale to large systems that incorporate non-linear dynamics, time, and probabilities.

Safety-critical cyber-physical systems Cyber-physical systems, such as controllers in automotive, medical, and avionic systems, consist of a collection of interacting software modules reacting to a continuously evolving environment. Many of these systems operate in safety-critical environments, and the cost-effective, but high assurance, development of cyber-physical systems is a challenge. Formal methods have made significant inroads in this domain: from model-based design of large scale systems, to verification and analysis tools for *hybrid systems* (formal models that integrate both discrete and continuous behavior), to simulating different models of computation, to platform-based design and design space exploration.

Design and verification of cyber-physical systems is a vibrant research area, and we have seen significant progress in past years — from more efficient reachability algorithms to novel combinations of control-theoretic methods and formal methods. For example, there is a clear trend toward model-based design in industry, formal methods are integrated into the design process of companies like Airbus, and we are seeing progress into formal modeling and analysis of medical devices. An additional challenge specific to the domain concerns certification: not only the analysis tool be able to check correctness of the design, but it should also produce evidence of correctness that can be independently checked and certified by a regulatory agency. Indeed, formal techniques may be required to obtain certification in certain domains.

High-performance computing High performance computing (HPC) is often regarded as "the third pillar of science" and is central to new scientific discoveries and engineering solutions. In HPC, there is ever growing need to simulate systems at increasing levels of resolutions. The massive scale of computation required in these simulations necessitates constant adaptation of HPC computational frameworks to the latest CPU/GPU/accelerator components (which change every few years) and the use of heterogeneous models of concurrency, in order to minimize power consumption and consequently infrastructure costs. The use of legacy software that has been tuned over decades is also a reality of how software can evolve.

Formal methods are essential to manage the intellectual complexity of large-scale designs, to develop focused testing methods to find defects early, and to ensure that long-lived APIs are well described to ensure uniformly understandable descriptions and portable implementations. Formal methods have made inroads in designing programming languages for HPC, and in the obvious application of bug-hunting techniques to this domain. In addition, the HPC domain has additional problems where formal methods can help. For example, can one bound the errors introduced by un-

certainties in the model and due to floating point arithmetic? How can one design long-running concurrent simulations that are also deterministic, and can one isolate the sources of non-determinism? Building error monitors is an opportunity, given that rare bugs often surface quite predictability at scale, but without sufficient bug root-causing support. System resilience is also an issue, given the rapidly diminishing probability of all the components of a multi-million core system operating without errors over months, and the increasing cost of checkpointing. Can formal methods help design HPC systems for verifiability?

5.3 Targets of Opportunity

Clean slate system design Traditionally, the role of formal methods has been to prove or disprove properties of system components (such as device drivers, communication protocols). The maturing of formal tools has resulted in recent years to functional verification of full-fledged applications. Examples of these include seL4, a fully verified operating system microkernel that is representative of the critical core component of modern embedded system architectures, and the CompCert project, a realistic verified C compiler. Success of these projects leads us to the next challenge for formal verification: can we design a complete system (ideally all the way from applications such as a web-browser, through language-implementation tools such as a compiler and an operating system, down to the hardware execution platform) with an accompanying proof of correctness? Such a clean-slate approach to system design requires long-term investment that partners researchers from formal methods with different subdisciplines such as operating systems, compilers, and computer architecture. However, the potential payoff is high, as it would likely lead to systems with high assurance and dramatically reduced security vulnerabilities. The US Department of Defense is also greatly interested in such a radical system redesign. The key challenge, and a target of opportunity, is to find a sweet spot between cost of a fully verified design versus the higher payoff due to higher assurance.

End-user programming More people have access to programmable devices than ever before. Yet, programming remains the job of an expert. Can formal tools and techniques help novice programmers to construct systems? Can novice programmers design working programs in an interactive environment by presenting example behaviors? This opportunity is best illustrated by the success of the recent interactive add-in FlashFill for Microsoft Excel spreadsheet software. In this system, the user demonstrates the desired transformation task (for example, rewriting all names to a uniform format such as the first name followed by a single space, followed by the last name) using a few examples, and the synthesis tool automatically generates the "best" program (in the form an Excel macro) that is consistent with all the examples. The synthesized program is then used to transform all the entries in the spreadsheet. If the synthesized transformation does not match the users intent, the user can guide the synthesizer by highlighting the incorrect transformations, thus providing negative examples for the subsequent iteration.

Use of formal methods to fully automate simple programming tasks has the potential of enormous impact by facilitating intuitive programming by millions of end users. Some domains where such techniques can be useful include programming of robots (for example, a robotic waiter can be programmed by the restaurant owner using commands in natural language) and development of applications of a mobile platform (for example, a person with training in programming can build a new app, as a composition of existing apps, simply by demonstrating the intended behavior).

To realize this potential, research along the following directions is needed. First, efficient algorithms for infering programs that meet users intent are needed. Such algorithms can potentially combine logical reasoning methods common-place in software analysis with concepts from statistical inference methods from learning theory. Second, benefits of such techniques crucially depend on

how the tool interacts with the user. Human-computer interaction issues have been traditionally not investigated by the formal methods community, and needs exploration.

Cloud Computing Recent years has seen the phenomenon of computing as a service, popularly called *cloud computing*, where companies provide computing and storage platforms that can be shared across many different users. Cloud computing brings in new problems for the formal methods community in terms of new programming models and verification opportunities for datacenter-scale computing systems, as well as new problems in design space exploration (e.g., how should computation be scheduled to minimize energy? how should failure be handled at the programming level?), security (e.g., how can we trust computation performed on the cloud? how can we ensure data is not leaked to outsiders?), and end-user programming. We believe there is ample opportunity for research problems in this area.

Specifically, one way that the formal methods community can engage in software systems development is by developing and maintaining precise specifications, models, and standards for widely used systems components. The open source development model has been extremely successful in building large-scale, robust, software systems through the time and commitment of a large number of volunteer contributors. Can the open source model be adopted for more formal artifacts such as specifications and interface standards for systems components and protocols? Such an endeavor can have several positive outcomes. First, the cost of developing such artifacts can be amortized across the community, and regained when many different tools can reuse such specifications. The challenge is to keep specifications up to date as components evolve, but at least for relatively stable interfaces —such as kernel interfaces for drivers, or standard libraries, or components such as caches, load balancers, etc. shared across many infrastructure projects— such specification and model artifacts could be developed and reused.

Privacy and Trust Privacy has become a significant concern in modern society as personal information about individuals is increasingly collected, used, and shared, often using digital technologies, by a wide range of organizations. Certain information handling practices of organizations that monitor individuals' activities on the Web, data aggregation companies that compile massive databases of personal information, cell phone companies that collect and use location data about individuals, online social networks, and search engines (while enabling useful services) have aroused much indignation and protest in the name of privacy. Similarly, as healthcare organizations are embracing electronic health record systems and patient portals to enable patients, employees, and business affiliates more efficient access to personal health information, there is trepidation that the privacy of patients may not be adequately protected if information handling practices are not carefully designed and enforced.

Given this state of affairs, there is a strong need for systematic development of foundations of privacy, including formalizing privacy concepts and developing methods and tools for enforcing privacy properties. Formal methods can play an important role in this area, in particular, in addressing the following representative problems.

First, there is much work to be done in formal specification of privacy policies and logical methods for their enforcement. Second, the importance of private data analysis is being increasingly recognized for many applications, e.g., releasing census statistics, privacy-preserving recommendation systems. In addition to work on algorithms for private data analysis, there is an important role for formal methods to ensure that implemented systems for private data analysis provide rigorous privacy guarantees. We expect to see formal methods techniques applied to privacy and trust in online interactions in the future.

Public Policy In addition to the topics above, there was widespread agreement that formal methods can also be an effective tool in many matters of public policy. Formal methods provide the tools to clearly state and analyze policies, and to design enforcement and audit mechanisms. Already there is some progress in formaling legal requirements such as HIPAA in the medical domain and privacy and utility in business processes. These are first steps, but we expect similar projects to continue. Public policy is often drawn up without fully comprehending what is or is not feasible. The formal methods community can and should have a role in forming public opinion on these issues.

6 Conclusions

We believe formal methods has had a lot of impact on computing practice. Part of the success has been the ability of the community to transform from being focused on methematical rigor on academic examples to seeking out practical problems, in investing in developing infrastructure and tools, and aiming for good-but-not-perfect products that can help practitioners. Formal methods remains relevant, both as a scientific discipline and as a conduit for technology transfer to different areas of computing. We have summarized some of the big challenges facing the community today, as we see it, and some recommendations for the field. The future will tell whether our concerns, recommendations, and challenges for the community are justified.

7 Acknowledgements

We thank the National Science Foundation for sponsoring the workshop and in particular, Nina Amla, Sol Greenspan, and John Reppy for advice and discussions. We thank the participants of the workshop, Aarti Gupta, Alan Hu, Allen Emerson, Andrew Appel, Andreas Kuehlmann, Ashish Tiwari, Benjamin Monate, Corina Pasareanu, Cormac Flanagan, Dan Grossman, Daniel Kroening, Edmund Clarke, Fabio Somenzi, Ganesh Gopalakrishnan, J Moore, Jasmin Fisher, Jens Palsberg, Jim Grundy, Ken McMillan, Lenore Zuck, Nikolaj Bjorner, Pamela Zave Ras Bodik, Rustan Leino, Somesh Jha, Sorin Lerner, Todd Millstein, Tom Ball, Vigyan Singhal, Vikram Adve, and YY Zhou, without whose active participation this report would not be possible.