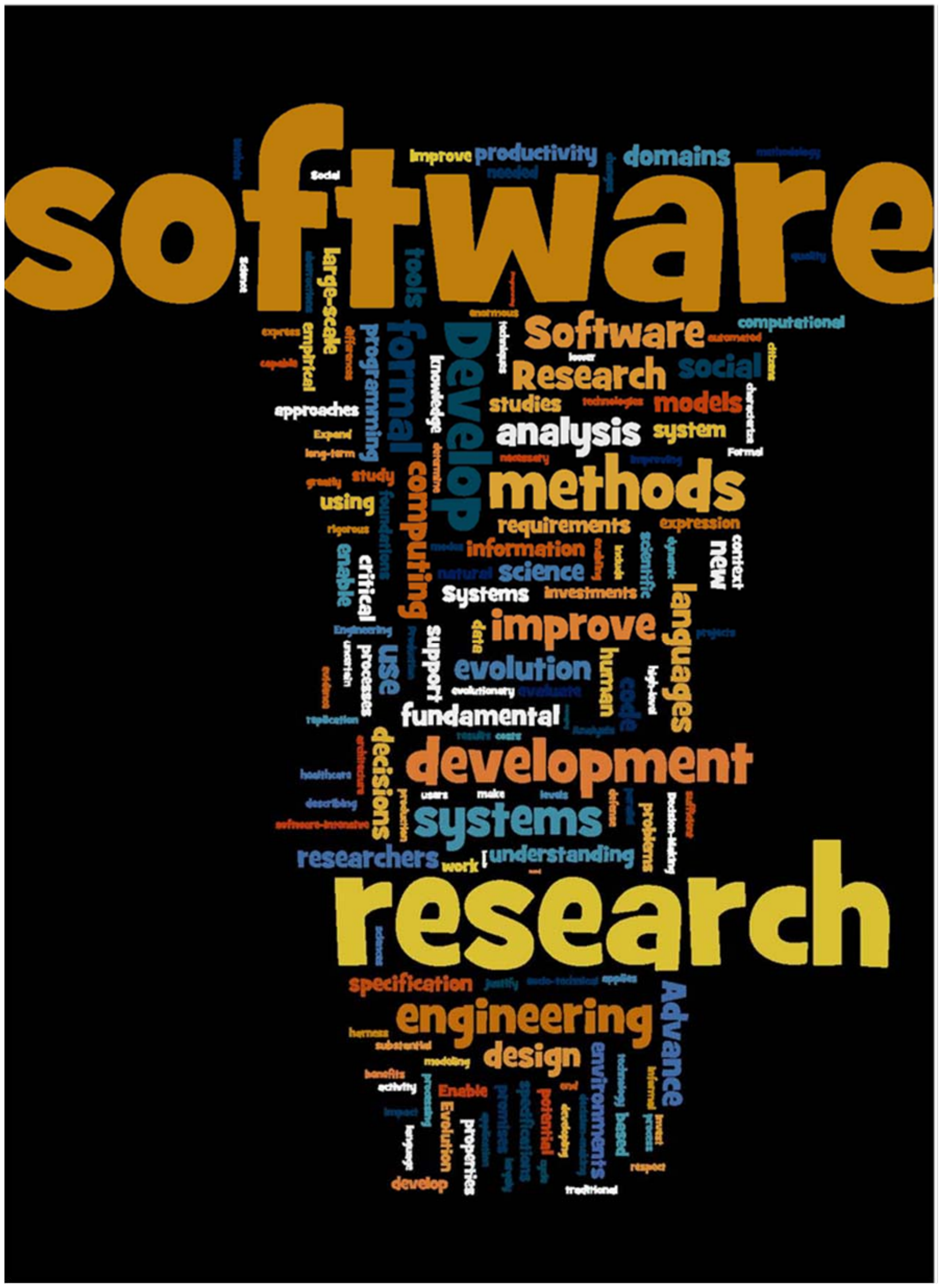
Future of Software Engineering Research



December 2011

The Networking and Information Technology Research and Development (NITRD) Program

The NITRD Program stems from the High-Performance Computing (HPC) Act of 1991 (Public Law 102-194) as amended by the Next Generation Internet Research Act of 1998 (Public Law 105-305). These laws authorize Federal agencies to set goals, prioritize their investments, and coordinate their activities in networking and information technology research and development.

The NITRD Program provides a framework in which many Federal agencies come together to coordinate their networking and information technology (IT) research and development (R&D) efforts. The Program operates under the aegis of the NITRD Subcommittee of the National Science and Technology Council's (NSTC) Committee on Technology. The Subcommittee, made up of representatives from each of NITRD's member agencies, provides overall coordination for NITRD activities.

Software Design and Productivity (SDP)

SDP is a NITRD Program Component Area (PCA) that conducts R&D that spans both the science and the technology of software creation and sustainment (e.g., development methods and environments, V&V technologies, component technologies, languages, and tools) and software project management in diverse domains. Complex software-based systems today power the Nation's most advanced defense, security, and economic capabilities. Such systems also play central roles in science and engineering discovery, and thus are essential in addressing this century's grand challenges (e.g., low-cost, carbon-neutral, and renewable energy; clean water; next-generation health care; extreme manufacturing; and space exploration.) A key goal of this science framework is to enable software engineers to maintain and evolve complex systems cost-effectively and correctly long after the original developers have departed.

About this Document

The 2010 Foundations of Software Engineering and Software Design and Productivity (FSE/SDP) Workshop on the Future of Software Engineering Research (FoSER) was part of the ACM SIGSOFT Eighteenth International Symposium on the Foundations of Software Engineering (FSE-18). FoSER provided the international software engineering research community - including academic, industrial, and government research personnel - with a unique opportunity to develop, discuss, refine, and disseminate consequential new ideas about future investments in software engineering research. The committee defined five major discussion themes based on the 89 position papers published in the workshop proceedings.[1] This report summarizes the results of these discussions.

Copyright Information

Publication of This Report

Electronic versions of NITRD documents are available on the NCO Web site: *http://www.nitrd.gov*.

[1] http://portal.acm.org/citation.cfm?id=1882362&coll=DL&dl=GUIDE&CFID=30687930&CFTOKEN=74002732

Table of Contents

1. FSE Preface 2

2. SDP Preface 3

3. Executive Summary 4

4. Key Findings 6

4.1 Help People Produce and Use Software 6
4.2 Build the Complex Systems of the Future 6
4.3 Create Dependable Software-Intensive Systems 7
4.4 Invest in Research to Improve Software Decision-Making, Evolution & Economics....... 8
4.5 Invest in Research to Improve Software Research Methodology 8

5. Helping People Produce and Use Software-Intensive Systems 10

5.1 Connecting Communities across the Software Lifecycle 10
5.2 Goals 10
5.3 Challenges 10
5.4 Promising Approaches 11
5.5 Potential Payoffs 12
5.6 Timeliness 13
5.7 Costs and Risks 13
5.8 Action plan, jump-start activities 14
5.9 Evaluation 14
5.10 Software Engineering for End-User Programmers 14
5.11 Human-Intensive Systems 15

6. Designing the Complex Systems of the Future 17

6.1 Technical Challenges Posed by Future Complex Systems 17
6.2 Opportunities Created by New Components and Platforms 18

7. Dependable Software-Intensive Systems 21

7.1 Automated Programming 21
7.2 Dependability Arguments 24
7.3 An Informal Approach to Automated Programming 27
7.4 Differential and Interactive Program Analysis 29
7.5 Defining Real Programs for the Masses 32
7.6 Evolution Group 33
7.7 Incentives 35

8. Improving Decisions, Evolutions, and Economics 37

8.1 Software Data Analysis 38

9. Advancing Our Discipline and Research Methodology 40

9.1 Challenge in Software Engineering Research 40
9.2 Motivation 40

9.3 SE Research with Impact 41
9.4 The Quest for building a Research Method Portfolio 41
9.5 Recommendations 42
10. The Future of Empirical SE Research 43
10.1 Asking the Right Questions and Providing Useful Answers 43
10.2 Replication 43
10.3 Data Sharing 44
10.4 Research in the Large 44
10.5 Too Much Focus on Generalizability and Positive Results 44
10.6 Costs and Risks 45
10.7 Evaluation 45
11. The Future of Formal Methods research 46
11.1 Goals 46
11.2 Challenges 47
11.3 Potential Payoffs 47
11.4 Timeliness 47
11.5 Evaluation 47
11.6 Progress through Research 48
12. The Future of Social Sciences in SE Research 49
12.1 Recommendations: 49
13. The NITRD Program 51
14. Appendix A - List of Attendees 52
15. Appendix B – Acknowledgements 60
16. Appendix C - Abbreviations and Acronyms 61

1. FSE Preface

Software is the underlying foundation, responsible for driving the technologies that society relies on to operate dependable processes for business, energy, healthcare, defense, business, engineering design, education, science, and entertainment, to name a few. Driven by the need for sustained, radical innovation to address major societal challenges and economic competitiveness, the demands for better, innovative and cost effective software are increasing exponentially, across all domains.

Notwithstanding the startling gains in software design and productivity achieved through software-related research in past decades, the rate of growth in the *demand* for improvements outstrips the growth in the *supply* of fundamental knowledge and engineering capacity to produce software, systems and services that rely on high performance, and cost effective software.

Addressing the economic challenges facing everyday-citizens demands accelerating advances in science, engineering, design, and increasing the productivity of software in all arenas. This requires significant and sustained investments in fundamental research in software engineering and related fields depending on the production of software-intensive systems, and in the growth of a software engineering research community driven to address the problems and opportunities that are paramount for the future.

The 2010 FoSER Workshop convened to provide the international software engineering research community with a valuable opportunity to develop, refine, and disseminate consequential ideas about major research problems and opportunities for the future. *Imagining the Future of Software Engineering Research* was the theme of this one-time international conference, which brought together top researchers and government research agency personnel from the U.S. and around the world to identify and develop major themes, problems, and opportunities for future software engineering research.

The goal was to attract a significant cross-sectional field of experts and, through two days of intensive discussions, identify some important future directions (without claiming to produce a comprehensive roadmap).

The workshop was divided into five themes based on an analysis of common threads gleaned from the submitted position papers:

- Help people produce and use software-intensive systems
- Design complex systems for the future
- Create dependable software-intensive systems
- Improve decision-making, evolutions, and economics
- Advancing our discipline and research methodology

Sessions on each theme resulted in the summaries contained in this report. The papers with abstracts from the FoSER workshop are on the ACM Digital Library (https://dl.acm.org/citation.cfm?id=1882362&picked=prox). Those who have access to the ACM Digital Library will be able to download the full text.

Attendance and energy at the FoSER workshop succeeded beyond expectations, attracting as many participants as the annual conference with which it was co-located – the Symposium on Foundations of Software Engineering - which is one of the international conferences in software engineering research. This was an unprecedented event for the field.

2. SDP Preface

"Leadership in software is important for our economy, our security, and our quality of life."[2] Software increasingly underlies the basic national cyber infrastructure and mission critical systems including communication, healthcare, transportation, the national power grid, weather forecasting, agriculture, finance, defense, and disaster response—as well as our scientific research infrastructure. Further, much of the economy depends upon computer software, whether for incorporating into products, for manufacturing products, or for designing competitive products. Consequently, the Federal Government has direct responsibility and a substantial interest in the U.S. "capacity to design, produce, assure, and evolve software-intensive systems in a predictable manner while effectively managing the risk, cost, schedule, and complexity"[3] associated with safety and mission critical systems. Unprecedented breakthroughs in software-intensive systems in the past have transformed the world and driven economic growth and job creation.

Future advances will depend on our ability to cost-effectively develop and sustain the transformative systems of tomorrow. In this budget-constrained era, advances in SDP are needed to enable the government to afford critical improvements in the nation's infrastructure, promote new missions, and foster the ongoing evolution of long-lived systems in civilian and defense agencies. In addition, improvements in SDP are critical for the Government's role as the overseer of systems impacting public safety - such as monitoring the development of technology to enable effective yet affordable standardized certification process.

The responsibility for coordinating U.S. federal software research funding falls under the auspices of the National Coordinating Office (NCO) for Network and Information Technology Research and Development (NITRD). To assess the state of affairs in software research and obtain intensive input from the software research community, the NITRD Software Design and Productivity (SDP) Coordinating Group collaborated with the Association of Computing Machinery (ACM) Foundations of Software Engineering community to organize the Future of Software Engineering Research (FoSER) workshop. The enthusiasm and responsiveness of the community resulted in an important community-building event, a rigorous activity to identify challenges and directions. The collective input produced a volume of thoughtful, articulate, imaginative position papers, which is available separately[4], while the conclusions and recommendations are documented in this report. The results of the SDP/ACM workshop contribute to a dynamic process in which policy makers, stakeholders, visionaries, and R&D leaders continue to formulate and pursue the software design and productivity research agenda. Fundamental advances in software science and engineering are presently poised for potential breakthroughs in SDP. The time is right to assess what can be accomplished through new R&D investments.

[2] "Designing a Digital Future: Federally Funded Research and Development in Networking and Information Technology", PCAST Report, December 2010.

[3] Definition of "software producibility" from "Critical Code: Software Producibility for Defense," National Research Council Report, 2010

[4] Gruia-Catalin Roman, Kevin J. Sullivan, November 7-11, 2010, Proceedings of the Workshop on Future of Software Engineering Research, FoSER 2010, at the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, Santa Fe, NM, USA

3. Executive Summary

The 2010 Report of the President's Council of Advisors on Science and Technology (PCAST), entitled *"Designing a Digital Future: Federally Funded Research and Development in Networking and Information Technology,"* documents the transformation of our society driven by advances in networking and information technology, catalyzed by our nation's past investments in research. "… [O]ur world today relies to an astonishing degree on systems, tools, and services that belong to a vast and still growing domain known as Networking and Information Technology (NIT). NIT underpins our national prosperity, health, and security. In recent decades, NIT has boosted U.S. labor productivity more than any other set of forces.... [NIT] a key driver of economic competitiveness.... [is] crucial to achieving our major national and global priorities in energy and transportation, educations and life-long learning, healthcare, and national homeland security.... [NIT] accelerate[s] the pace of discovery in nearly all other fields.... [and is] essential to achieving the goals of open government."

The NIT revolution has been powered by unprecedented sustained advances in two broad areas: digital devices - primarily general-purpose processors, network and storage systems, displays and sensors, for example - and special-purpose software, i.e., application-specific. A software system expresses a desired computational/machine behavior in a number of different forms both to drive general-purpose devices to solve specific problems and to support human design of and reasoning about such computational behaviors. Software enables general-purpose devices to perform such specialized tasks as searching for information, transmitting movies into homes, managing the flight controls of modern aircraft, and operating the payment systems at the core of the modern economy. Today we also see a growing need for specialized devices, but the computational logic that drives these devices is still software, in the form of circuits embodying desired information processing procedures.

Past advances in the science and technology of software and its design and production provided the foundations for today's enormously valuable software-driven industries. Past advances include modern programming mechanisms; software development methods; mathematical techniques for verifying the conformance of software behavior to software specifications and for finding faults in programming; approaches to testing software for faulty behavior and for securing it against misuse; methods for structuring software artifacts sometimes comprising tens of millions of lines of programming code for human understanding and maintainability; and tools to enable networks of software engineers to work 24/7 globally to create many millions of lines of debugged and documented code every day for a plethora of diverse software products used in industry, academia and by the every-day citizen.

The past investments in software and software engineering research have greatly enriched our society and improved our quality of life. American industry, in particular, has translated knowledge produced by research into enormously profitable and innovative products with incredible effectiveness. Google now dwarfs all of the greatest libraries in human history, with the volume of information indexed. More than one in ten people of the entire human population is registered with Facebook. Modern military aircraft are flown by (and essentially are flying) computers. Banking, finance, and commerce rely profoundly on NIT: today most money is stored not in tangible currency but in the digital record-keeping systems of banks and other institutions. Past investments in research have provided us with the benefits of a knowledge and technology base that has truly changed the world.

The goal of the 2010 FoSER Workshop is to promote and accelerate significant, government investments in fundamental, use-directed software engineering research. Complicated unsolved problems in software engineering remain while entirely new opportunities for research are emerging, driven by ongoing advances in information technology and changing societal needs. Fundamental, use-directed software engineering research continues to promise enormous dividends for industry and society for the foreseeable future.

Today, for example, we cannot express computations in naturally understandable language, so the computational power of software remains inaccessible to most citizens. We cannot adequately verify that the software that runs our banks, medical devices, and defense systems is sufficiently safe, reliable and secure; as both software complexity and our reliance on software grow, so do risks to our security, health, and prosperity. Our record of producing software for major societal infrastructure systems for health, defense, and in other such areas, remains unacceptably poor. We do not yet fully understand how to design software for affordable sustainability over decades-long lifetimes of major infrastructure systems. At the same time, stunning advances in computing are creating new opportunities for fundamental research in software design and productivity, social networking systems, massive data gathering and analysis capabilities, and verification algorithms, etc.

The contributions of past software engineering research to our society have been just stunning. Today the software engineering research community and the computer science and engineering research community are ever increasingly rich and vibrant. The need for, and the promise of, future research is compelling. Yet clouds are now on the horizon. The very success of the software industry, and of software in all industries, has led some to hold that the major open problems in software design and productivity are solved, and that private industry can now lead future advancements. Growing fiscal pressures on many governments around the world, including the U.S. also threaten government support for research.

Such misunderstandings, and shortsighted strategies, must be confronted. Industry will seek, recognize, and exploit opportunities with strong short-term profit potential and low technical risk. However, investments in fundamental, use-directed research are simply not in this category: they generally involve high technical risks requiring diversification at scales not feasible for individual companies; they involve long time scales that are hard or impossible to justify against the short-term profit demands on industry; and the breakthroughs they produce are often pre-competitive in nature and accrue to the benefit of industry and society in general, but are not easily appropriated by individual companies.

Notwithstanding the excellent research laboratories run by a few near-monopolistic companies, industry cannot and will not make the kinds of investments needed to drive fundamental research at the required scale. The onus falls on governments to provide for the general welfare of current and future citizens by investing in research. Safety, security, and prosperity in our society depend on it. Particularly in times of tight fiscal constraints, it is essential that governments and citizens summon the will to enhance research: the seed of future economic and security harvests.

The question, then, is not whether government-funded, fundamental use-directed research must continue. Without significant and sustained investments, we face a dimmer future, of diminishing innovation, prosperity, safety, and security. Rather, the question is what critical problems and opportunities should such research target going forward? This is the question that the FoSER Workshop asked the community to address. It is not enough simply to continue on the path that yielded today's vital information technologies. The question is, where should we go from here? This report presents the findings and recommendations regarding five overarching themes.

4. Key Findings

4.1 Help People Produce and Use Software

4.1.1 Develop Social Network Technology for Software Engineering

- Revolutionize software design, productivity, and quality through research investments in social computing technologies to connect software engineers and other stakeholders in software and software-intensive system development projects and processes.
- Advance fundamental knowledge of design as a socio-technical activity.

4.1.2 Broaden Participation in the Production and Use of Software

- Democratize the production, use, and benefits of software by enabling citizens ("end users") in all domains to easily, quickly, and conveniently create and customize high-quality software for their own sophisticated purposes.
- Advance our understanding of end-user motivations, skills, interests, and domain-specific abstractions; of the needs of the socio-technical environments in which end users need to develop and use software; and of means by which all citizens can be empowered to harness the enormous power of custom computing systems.

4.1.3 Develop the Science of Cyber-Social Computing Systems

- Develop the science and engineering of cyber-social computing systems, in which systems comprising computational machinery and human elements (i.e., People as components) work synergistically to carry out complex information processing tasks.
- Expand our understanding of computing, programming, software and the design and engineering of software.
- Potential to vastly improve information handling in human-intensive domains, such as defense and healthcare.

4.2 Build the Complex Systems of the Future

4.2.1 Address Societal Grand-Challenge Problems

- Our society's most pressing problems demand information systems of unprecedented complexity. Working within demanding application domains, such as healthcare, energy, transportation and defense; researchers should work first to characterize the fundamental information processing problems in these domains, and then develop the science and technology necessary to enable the development of large-scale software-intensive systems to achieve transformative improvements in these domains.

4.2.2 Enable Effective Certification of Societal-Scale Information Systems

- Develop the science, tools and methods needed to enable rigorous engineering certification of critical properties, such as safety, in large-scale systems that are characterized by such complexities as humans-in-the-loop, very high availability requirements, highly distributed independent subsystems, substantial autonomy, and active and capable adversaries.

4.2.3 Learn to Exploit Rich Emerging Platform Opportunities

- Produce the fundamental knowledge, e.g., in programming abstractions, architecture, and verification needed to harness the potential of a vast new diversity of computing devices, from the microscopic, to the hand-held and mobile, to massively parallel hardware, to the large-scale cloud-based infrastructures.

4.3 Create Dependable Software-Intensive Systems

4.3.1 Enable User-Friendly Programming

- Enable people to program computers by expressing desired computational behaviors using informal languages and other natural modes of expression rather than arcane, formal programming languages.
- Develop new approaches to computer-assisted, iterative refinement of informal natural language specifications into computer programs with increasingly constrained semantics, resulting in executable code, precise specifications, test sets, and so forth.
- Develop the foundations to process natural language and other human modes of expression, linked to automated software synthesis that would broaden participation in computing while improving software design and productivity.

4.3.2 Automate Software Evolution

- Significantly improve software evolution - the most costly and largest component of the system life cycle - by developing technologies for machine-assisted modification of requirements and specifications and designing automated re-derivation of implementations by replaying derivations up through high-level design.
- Advance our understanding of how to express computational intent, how to transform such expressions into useful software, and how evolutionary changes at one level of expression translate to updates at lower levels of derivation.
- Unify scientific knowledge across the largely disjointed fields of requirements, specification, architecture, formal methods, and programming languages, to name a few.

4.3.3 Design for Dependability

- Develop the science and technology needed to support parallel development of software and corresponding assurance cases based on partial evidence for software dependability, so that decisions to use software in critical environments can be based on rigorous analysis sufficient for effective risk management.
- Strengthen the scientific foundations of software validation while enabling policy makers to make informed, dependable decisions to approve the use of software and software-based systems in critical environments.

4.3.4 Employ Differential and Interactive Program Analysis

- Combine static analysis with dynamic, real-time feedback to determine the impact of code changes. Early detection in the development cycle, will improve code quality and developer productivity as the source code evolves.
- Verify a code version with respect to previous versions using differential analysis.
- Engage developers in interactive analysis dialogs to discuss key assumptions that justify their coding decisions.

4.3.5 Enhance Usability

- Make formal specification languages more accessible and usable to the human. Develop the capacity to express critical system properties beyond the reach of contemporary software specification languages.

- Add to the existing substantial momentum in the development and implementation of high-level, often mathematically formal, software specification languages.

4.4 Invest in Research to Improve Software Decision-Making, Evolution & Economics

4.4.1 Advance the Planning and Management of System Evolution

- Develop and evaluate evolution-aware software processes and practices, models of software evolution to include software cost models that adequately model evolution-related costs, and software tools and representations capable of describing and modeling software and requirements evolution. Such research would help characterize fundamental issues in design evolution, and would greatly aid practitioners in devising sustainable software systems.

4.4.2 Improve Data Collection and Analysis Software Decision-Making Methods

- Conduct fundamental research to improve cognitive support for software development decision-makers and decision-making.
- Develop approaches that combine traditional software analysis with techniques used in data analytics, business intelligence, data mining, prediction models, empirical studies, and economics. This research promises to yield improved software development processes and a new generation of software tools, improving software productivity by eliminating the necessity of rework due to poor decisions early in the developmental cycle, by better adapting system designs to suit their environments.

4.4.3 Predictive Models for Software Production

- Develop and evaluate new approaches to modeling and managing software evolution as a decentralized process in a dynamic and uncertain environment. This research would integrate concepts from traditional software engineering with areas ranging from finance and evolutionary biology and morphogenesis to the study of software. Such research promises to improve predictive models for software development and evolution, and new normative models to inform decision-making at multiple levels of granularity. In practice, such work could greatly improve our ability to manage tradeoffs between short-term costs and benefits as well as uncertain but much larger long-term returns in software development.

4.5 Invest in Research to Improve Software Research Methodology

Advance the scientific foundations of software engineering research by developing and documenting taxonomy of research methods, ensuring their applicability to research dissimilar problems, describing criteria for validating results, and identifying opportunities for improvement. Such methodological research promises to enable researchers to select appropriate research methods for projects based on their characteristics, to compare research methods, to improve evaluation of research and the education of researchers, and to make better-informed decisions about future research methodology investments. This initiative has the potential to improve the nature and application of formal, empirical, and social science-based research methods in software engineering.

4.5.1 Advance the Relevance of Empirical Software Engineering Research

- Improvements in software engineering are dependent upon research advancement that focuses on comprehensible and actionable answers to relevant research questions by explicitly discussing the context of the research. Additionally, study replication can extend the context and the study's relevance.
- Sharing data and tools can both facilitate replication and lower barriers to performing empirical research.

To develop compelling evidence of trends in software production requires financial support that is sufficient in both scale and time to perform large-scale, long-term baseline studies.

4.5.2 Expand the Breadth and Scope of Formal Methods Research

Formal methods research applies logical and mathematical analysis to determine properties of software systems. The current blossoming of formal methods research applies to a wide variety of analyses. Ease of use and transparency are primary factors when transitioning to mainstream software development. The goal should be to migrate towards routine use of formal methods including invisible analysis embedded in tools. The potential payoffs for a broad portfolio of formal methods research are enormous. Such research investments should include interoperability and infrastructure development, to support the synergistic collaboration of researchers that should enhance experimentation, and enable the understanding of alternative techniques.

4.5.3 Utilize Social Science Research

Because software engineering is largely a human activity and the social sciences are more mature with respect to studies involving humans, software engineering research would benefit from exploring social science research methodology. For example, researchers should consider using qualitative methods, a staple of the social sciences, to address the impact of individual differences on research results. Research should embrace, justify, and explain the context of the study. The similarities and differences between individuals should be investigated and reviewers should have the necessary resources to evaluate studies using social research methods.

5. Helping People Produce and Use Software-Intensive Systems

5.1 Connecting Communities across the Software Lifecycle

Computer-mediated communication technologies, such as email, audio/video conferencing, social networking, blogging, micro blogging, online discussion and question and answer forums, have transformed the way that communities connect online. Software development communities on many scales, from individuals, to teams, to organizations, and ecosystems of organizations, are able to take advantage of the ubiquity of these technologies to facilitate communication about, collaboration in, and coordination of shared work.

For example, structured communication protocols can be encoded in online dispute resolution systems to enable more flexible, cheaper ways to work through negotiations than by meeting face-to-face. Free and Open Source Software (FOSS) development practices are conducted entirely online, using tools like mailing lists and open software repositories, to enable collaboration among diverse communities of constituents across distance, time, cultures, and even across projects in an ecosystem of related software (e.g., any one of the 20,000 projects in a typical GNU/Linux distribution).

These free, open, online collections of software projects have indeed reached critical mass, enabling people from many different communities to innovate exponentially and create software products with little self-developed or maintained infrastructure support. Seth Priebatsch, the current CEO and Chief Ninja of SCVGNR.com, created his first startup company, Giftopedia.com, in 2001, offering an Internet service to help people decide what gifts to buy for their loved ones by revealing what others were buying. Not knowing how to program, he found Russian and Indian programmers through the Internet to build the site for him. After communicating his vision, design specifications, and work ethic using his blog, Skype text chat, and email conversations, he was able to bring his site live and use social media to virally market it amongst his friends' online social communities. *Seth created this site when he was 12 years old.*

Connecting software engineers together has the potential to hire more employees in software companies; to increase the sharing of timely, concise knowledge about current, attention-worthy development activities and events among communities of developers, testers, managers, those with non-engineering job roles, and consumers and customers of the software products; and to revitalize legacy software ecosystems whose component technologies have been made obsolete and whose participants have long ago moved on to newer endeavors.

5.2 Goals

The goal is to enable an ever-increasing number of communities to easily, quickly, and economically create, maintain, and adapt software by understanding and improving how these communities of software engineers, supporters, and their users communicate, collaborate, and coordinate using various kinds of computer-mediated communication.

5.3 Challenges

This research area spans a wide spectrum of producing and consuming communities, including:

- Individual software developers
- Teams of engineers (a scrum team of five developers, four testers, two requirements engineers, and one manager)

- Independent software vendor and domain-specific application organizations made up of many cross-functional teams (e.g., shrink-wrapped, mobile, and Web service vendors as well as banks, hospitals, scientific research organizations, defense and aerospace companies, and government services)
- Ecologically balanced ecosystems of software organizations (i.e., software supply networks that link software developers to integrators and consumers)
- Operating system-specific products along with third-party vendors and consultants
- Application-specific software stacks [such as LAMP, Oracle, and SAP]
- Mobile phone platforms [such as iPhone and Android]
- Open source development farms [such as SourceForge, Github, and MSDN] where the predominant form of development is integration rather than feature creation

In addition, the communities range from collocated to globally distributed; from individuals to open source projects to industrial corporations; from developer-centric communities to multi-role communities including testing, specifications, management, marketing, and sales; from mainstream Internet-connected communities to those on the margins; and from currently active communities to those that are essentially defunct.

Each community has its own set of stakeholders, problems, requirements for appropriate solutions, development tools, and varied means to deploy and enact solutions, requiring investigation, analysis, and realization approaches customized to each problem under consideration.

5.4 Promising Approaches

The main research method consists of three steps. First, the researchers must understand the social context of the community under study. What development processes (e.g., Waterfall, Agile, XP, ad hoc), work practices, communication modes (e.g., discussion forums, software repositories, online chat, social networking sites, blogs and micro blogs), and collaboration and coordination norms do community members employ? Who are their stakeholders? What are their roles, whether formal or emergent? Who are the developers and the users? What is their culture? What are their values? What are their motivations (e.g., for profit, for the public good, for freedom, to help their own community)? With whom do they interact to create, maintain, and adapt their software (e.g., administrators, management, marketing, sales, customers, lawyers, etc.)?

Next, researchers must gain access to, gather, analyze, and validate software process information related to the software project. These steps should address the people as well as the software artifacts (the code, bugs, tests, builds, documentation, specifications, schedules, designs, run-time profiles, and operations logs). In addition, development activities related to these artifacts must be collected and codified. Finally, the project-related communications among the members of the community need to be compiled, including status meeting notes, bug triage records, code reviews, specification reviews, tags, annotations, blogs, public emails, online chat, microblogs, discussion forums, check-in messages, bug descriptions, and any other communication mode and forum that can be electronically recorded. Software development in the cloud makes it easier to collect and analyze these data sources.

Should these voluminous amounts of data be saved in raw form or should only a pre-analyzed filtered form be saved? The decision must be evaluated from a research perspective. Most traditional database technologies cannot mine or store information on such a large scale (space and time), nor can current analysis techniques scale to process it, as required to support both search and browsing interfaces and to correctly associate and enforce license, legal, privacy, and confidentiality rules that may be customized for each project.

Next, researchers need to perform multiple types of analysis to identify the critical connections between artifacts, activities, and communications. For example, the people in charge of a particular technology must be distinguished from groups of people working together as a team, even if these groups are not identified explicitly by the software team. Changes to the code can be explained by exploring linked feature specifications, code, bug reports, and customer and operations complaints. Communication logs can be used to understand how software engineers who are no longer working with a project conducted their development work. Organizational hierarchy charts can help explain the structure of the software architecture (i.e., Conway's Law). Distributed software developers can be connected to one another, their dependencies (easy for them to know) and anti-dependencies (usually very difficult to know) identified, as well as which customers rely on specific features. Using multiple data sources and analyses, results can be triangulated to provide higher confidence answers as well as transparency to enable consumers of the analyses to cross-correlate answers and increase their confidence in the results.

Many of these analyses require further research to define optimal data structures for storage and retrieval of linked structural metadata; for displaying personalized, relevant demographics (e.g., the results of a customized repository search for key worker data necessary to establish narratives for software processes); and for identifying relational subgroups of people, artifacts, and communications.

These analyses are codified into tools designed to deliver relevant, concise, verifiable information to the appropriate audience, when needed. This can facilitate necessary and useful communication that can enhance collaboration and coordination among community members on various scales. For example, filtered newsfeeds about code can help a developer determine what happened to code he wrote and its dependencies while he was on vacation. An engineer, tagging a bug related to a function in a kernel driver, can be notified of other related, open bugs involving the same function. An individual can use social networking sites to find potential collaborators who are sympathetic to his personal project management manifesto. A team, fixing bugs in one variant of the PHP language[5], can learn of code changes made by the Apache team designed to circumvent these bugs, enabling the two teams to collaboratively fix the bug for both projects. A software "anthropologist" can explore a bank's legacy code base to help new hires understand design decisions made 30 years ago.

5.5 Potential Payoffs

Finally, research must address how to measure the impact that tools, including communication tools, have on the communities they serve. These tools will enable communication of innovative ideas among related software teams, as well as regeneration of tacit design knowledge lost to the ravages of time. The tools will enable people to create new scientific, technological, and engineering knowledge for building complex software systems and enable the creation of even larger systems of systems (e.g., the entire GNU/Linux ecosystem). New businesses will replace the traditional corporate worker infrastructure with people and services in the cloud connected by open communication networks. Companies in the software industry are already taking advantage of a critical mass of open source projects to quickly and easily build increasingly more compatible and more complex software products for the commercial market.

[5] PHP is a general-purpose server-describing scripting language.

The impact of the communication tools could possibly be assessed by determining whether they offer a productivity boost to teams – such as by increasing knowledge sharing, improving learning, decreasing interruptions, unblocking information needs, and facilitating negotiations between groups of people. Other measures could determine whether there is a decrease in the cost of coordinating a team to develop, test, produce, market, and sell a system, collect customer requirements, issue change requests, fix bugs, provide feedback, and respond with more agility, thereby lowering maintenance costs.

5.6 Timeliness

The communications and data availability environment have changed dramatically in the last few years facilitating people to produce and use software-intensive systems. Web 2.0 and other large-scale communication networks have become dominant only in the last five years; Facebook alone has 750 million users (more than one-tenth of the world population)! In addition, the scale of the communities and the data required for analysis is enormous, on the order of terabytes and petabytes for large individual projects and even small ecosystems of projects. Today, a critical mass of publicly available, open source targeted components has been reached, enabling even lone developers to create entire software projects, requiring minimal additional design and programming.

5.7 Costs and Risks

One key challenge in designing systems that measure social behavior is balancing openness and privacy. The data analyses in these systems are only as good as the data sources and analytical tools. A system that reveals information about its participants may influence their reputations in a biased way, especially if inadvertent biases are designed into information visualizations. A person being evaluated by the system will certainly attempt to influence the metrics to ameliorate the outcome. Attribution of ownership and responsibility may be presented in a misleading way, potentially influencing employee performance evaluations. Could these visualizations or communications paint a negative picture of an employee, so as to warrant their dismissal? Would that violate employment law in their jurisdiction?

The European Union (especially Germany) has stricter privacy laws governing the gathering and use of data about employees, even with their consent. How does one support an opt-in/out mechanism when the data mined is a byproduct of the normal organizational output and is necessarily available in raw form? If a team consists of one member who opts out and the rest opt in, the manager incorrectly may assume that the opted-out member has done nothing of note.

Finally, corporations that value their intellectual property may not share knowledge of confidential data, even inside their own organizations.

This research must attain a critical mass of users for any new, experimental socially oriented communication system. Because communities cover a widely distributed area, enlisting an adequate minimum number of real-life users may be quite difficult. For open source software, it may be possible to piggyback experimental tools on top of existing, widely-used blogging and social networking platforms. For communities with privacy concerns (such as in industry) or those without widespread Internet access (such as Third World communities), achieving critical mass to make tools useful may not succeed without significant determination and effort by researchers and community influencers.

5.8 Action plan, jump-start activities

Many researchers are already working to understand and develop tools for various communities of developers. These activities include open source projects, software development corporations, scientists, medical organizations, defense contractors, school systems, and ecosystems of development projects and game platforms. The key next steps are to describe the relevant work practices of desired target communities, to identify their most relevant communication problems, and to design and build tools to address these problems.

5.9 Evaluation

Assessments are needed to determine whether the utility of the tools, the accessibility of the information, the engineering practicalities involved in the storage and analysis of data, and the means of using the results to facilitate communication and coordination among the people, are improving performance. In addition, evaluation metrics must be developed to demonstrate a return on investment (ROI) to prove that the tools (e.g., for the social network of the community, for the code, for their communications) directly improve the summative measures, such as team productivity, product quality, project cost, and project agility.

5.10 Software Engineering for End-User Programmers

More than 10% of the world population is now using Facebook. These users engage in an elementary form of programming by specifying their privacy settings, namely "rule-based programming." This is indeed a form of programming, because it is a way of instructing the computer what action to take when data arrives.

In the particular case of Facebook, any unintended effects of users' "programs" affect not only their own privacy, but also the privacy of everyone connected to them.

Clearly, end-user programming (EUP) is an important phenomenon, and many people other than those who develop software as their profession also engage in this activity. Examples include healthcare professionals, engineers, accountants, teachers, and even children. Software development activities include creating new formulas in spreadsheets, customizing software with preference settings, "mash up" or combining Web services by dragging them together. Yet, the software engineering community has taken little notice of the enormous body of software these users are producing, and even less notice of how to help end-user programmers monitor, assess, and correct errors in the software they create or customize.

The overall goal of software engineering for end-user programmers is to introduce the software engineering research community to the end-user practices and environments to enable this new class of producer-consumers of software to more easily, quickly, and conveniently create, maintain, and/or adapt software with quality suitable for the software's purpose.

Significant work is needed to achieve this goal, and we do not pretend to be able to enumerate all of the challenges. However, it is clear that any solutions must include at least these attributes:

- Support for the motivations, skills, interests, and abstractions of this audience
- Seamless integration in the socio-technical environment in which they perform software development activities

Examples include support for testing, validating, customizing, debugging, and reusing their software in ways that smoothly integrate "coding" and execution.

In essence, the goal is to empower end-user programmers to transparently improve the quality of their software without requiring training, or even taking an interest in traditional software engineering methods, but instead using a holistic approach, with no separated tasks, modes, or tools.

5.11 Human-Intensive Systems

The “Helping People Produce and Use Software-Intensive Systems” discussions concentrated on software engineering challenges in developing and improving what might be called "human-intensive systems." These are systems in which people are participants in the execution of the system, not just as so-called users considered to be external to the system, but rather as integrated components of the system in much the same way as hardware and software components are related. Such systems are increasingly central to a large variety of domains. Examples are:

- Healthcare processes, whereby healthcare professionals in conjunction with medical devices deliver critical and everyday care
- Automobile traffic management systems in which drivers, their cars, optimization software, and a range of sensors serve to improve traffic flow in congested areas
- Social networks in which configuration options, imposed by others, may have substantial impact on an individual's privacy

Such systems are relatively new but are growing rapidly in complexity, number, and impact. As technological advances enable new kinds of communication and an explosion of new devices and computational power, such systems become increasingly feasible. As these systems penetrate new domains, they introduce new kinds of interactions between their human and non-human participants.

These systems-of-systems offer tremendous opportunities to improve social organization, economic efficiency, and the daily lives of people everywhere, but they are extremely difficult to understand, develop, and maintain. The challenges are numerous. Human beings are varied and often unpredictable; they may respond to incentives, but cannot be "programmed." It is therefore hard to predict the behavior of a system in which people are critical components. Compounding this difficulty is the fact that these systems are typically large and complex, making them difficult to scale and test. Because of their size, complexity, and intimate incorporation of the behaviors of people, such systems exhibit very broad ranges of behavior encompassing deviations from the norm requiring constant monitoring, correction, and recovery algorithms.

Because the interaction between human and non-human participants is pervasive and critical, understanding such a system requires understanding the humans' mental models of the system. Research methods must include results from human factors research, industrial engineering, cognitive science, and other relevant fields. Developers must take into account participants with diverse levels of expertise. Similarly, context of various kinds is critical for both the human and non-human components. Some of this context can be derived from execution history, but some depends on other factors such as participants' training and background, and the availability of resources.

Approaches from software engineering, enriched and enhanced by methods from a variety of human sciences, will have a substantial impact on the understanding, development, and maintenance of these systems. It is clear that any approach must take the hardware, software, and human participants into account, starting at the beginning stages of operational concept formulation. The 2007 National Research Council (NRC) report, “Human-System Integration in the System Development Process: A New Look," makes a number of relevant recommendations.

The most significant approaches will depend on the development and analysis of rich models of the entire system, including the behavior of human components and their interactions with the hardware and software components. (This is in contrast to the emphasis on human factors research, which focuses on the human participant and his/her interface with the cyber-physical components.) Much is known about modeling the behavior of the hardware and software components, but key advances are necessary concerning modeling and analyzing the behavior of the human participants and their interactions with the hardware and software. The most promising approaches involve process modeling, so-called systems engineering methods, and problem decomposition techniques. Analysis methods will need to include standard software engineering approaches such as testing and model checking, as well as safety analyses, e.g., fault tree analysis and failure modes and affects analysis, including various kinds of simulation. Information gained from monitoring a system, for instance, to detect deviations and provide guidance to human participants, can be used to support longitudinal studies and evidence-based improvements of the system. Historical information from monitoring could, for example, provide information about the (possibly highly conditional) probabilities of occurrence of certain events, enabling corrective changes to an existing system.

These systems are important to society and increasingly pervasive. Software engineers must recognize that a failure to help develop improved methods will result in costly and underperforming systems unable to take advantage of technological advances, and possibly catastrophic failures in systems that are critical for the well-being of our economy and society as a whole.

Given the range of such systems, the initial research efforts should focus on well-selected case studies with different perspectives, attempting to model and analyze varied types of human-intensive systems. As the recommendations of the NRC report noted, how to measure the successful integration of human and hardware/software components is a critical yet underexplored area. Encouragement and support for this venture will be essential from the software engineering community, including conference program committees, journals, and funding agencies.

Improving our ability to understand, develop, and maintain such human-intensive systems will require software engineers to collaborate with researchers in such areas as human factors, industrial engineering, cognitive science, and the social sciences.

6. Designing the Complex Systems of the Future

Our ability to address societal grand challenge problems in such areas as health, energy, transportation and national security is limited to a significant degree today by a lack of fundamental knowledge of and experience with information systems at the scale and level of complexity and function needed to transform societal-scale systems. For example, we still do not know how to build a cyber-infrastructure supporting life-long, multi-source health records for citizens in a country with as complex a medical system as the U.S.

The proliferation of new information technology components, at scales ranging from the microscopic to all-encompassing global expansion presents startling and vitally important new opportunities for the exploitation of information and information processing in these areas of societal importance. Today we lack the knowledge needed to harness these novel capabilities effectively and sustainably within broader societal contexts. In healthcare, for example, mobile and sensor-intensive devices are creating dramatic new opportunities for personal acquisition and exploitation of health-related information, but fundamental questions remain such as determining the national-scale cyber-infrastructure required to collect, process and socially integrate data necessary for a healthy society.

Discussions regarding the design of the complex systems of the future focused on three main points: a segment of the software engineering research community must be involved with societal grand challenge problems in specific (non-software) domains, such as health; they must be attune to the fundamental systems-level complexities and challenges that future information systems will present; and they must be cognizant of the challenges and opportunities presented by new component and platform technologies.

6.1 Technical Challenges Posed by Future Complex Systems

Technical components and market incentives exist today to drive the development of a broad range of complex new systems in numerous domains of societal importance. New systems are being envisioned, and in some cases being implemented. In the U.S., for example, the Federal Government is spending tens of billions of dollars in an attempt to create a national system for the exchange of health information. Without the knowledge produced by future research, however, such systems are unlikely to be built. These systems will either profoundly benefit society if build well or have a negative impact if inadequate methods of design, development, and deployment are used. There is a compelling need to fund research for the technical challenges posed by systems at this scale of complexity and criticality.

These systems have numerous characteristics that challenge the current state of the art and current knowledge in software-intensive systems engineering. These challenges include people in the loop of information processing. We are just beginning to understand how the intellectual frameworks of computer science and software engineering can work synergistically to analyze and design the processing of human information. Such systems also generally have very high availability requirements and often have demanding requirements for data consistency, as well. The ability to provide for both properties simultaneously, in networked systems, subject to partitioning, remains an open challenge.

These systems are often constructed by the integration of a broad range of independently developed, evolving, and governed subsystems. That is, they are systems of systems, with varying degrees of centralized authority and control and varying degrees of architectural consistency. They will increasingly include autonomous computational elements, as well, posing profound new challenges in such areas as validation and verification.

Systems of this kind are also deeply embedded into the fabric of society, and they can and will touch millions of individuals. Designing systems to support the people who are users will require significant attention to issues involving the human factor and social interaction. Concurrently such systems will be accessible to real-time adversarial agents and thus open-source problems in software engineering demands ongoing attention to ensure the security, and privacy, necessary for system dependability.

Some of the specific technical challenges that we anticipate during the construction of future complex systems include integrating multiple system models, including models of organization, governance, individual human users, and societal concerns; designing novel forms of feedback control and system adaptation at scale; incorporating uncertainty and probabilistic reasoning into computing; and online characterization of system execution and health at scale.

6.2 Opportunities Created by New Components and Platforms

We continue to see a flourishing of novel information processing technologies. In some cases, novelty is driven by accelerating advances in key areas, e.g., in miniaturization of sensors; and in other cases, by exhaustion of historically rich veins, e.g., the diminishing potential to increase processor clock speeds by the further miniaturization of transistors on silicon chips. Advances in sensors are producing whole new categories of computing devices, such as sensor-enabled, hand-held mobile computers (cell phones), while the looming exhaustion of Moore's law has driven the processor industry to place enormous bets on the emergence of programming models and technologies for massively parallel multi-core processors. At a much larger scale, we are obviously now seeing the growth of globally important information processing services, such as search and logistics, based in large part on the development of football-field-sized utility computing centers.

This rich proliferation of new technologies is vastly expanding the design space for future systems, but in ways that we are not yet adequately equipped to exploit in an efficient and effective manner. How can we weave together technologies across numerous types of scales to produce next generation systems of high value to society and individuals? Major challenges arise from the characteristics of such systems: distribution; the interconnection of components without centralized control; resource management and online adaptation to hard-to-predict variations in resource availability; device, data format, semantic, and software architectural heterogeneity; continuous execution of long-lived applications (on the order of decades of continuous service); and the need in some cases to use regulation, planning, monitoring, enforcement and incentive systems in place of centralized control to maintain system integrity.

Just a few of the novel research approaches that might be pursued in a context of diverse technology components and platforms include the following: computational speculation on potential future system states to recognize and exploit opportunities or to avoid poor outcomes; running multiple versions of systems simultaneously to improve and expedite the results, or for automated exploration of a system design space in support of self-evolution; and the development of new categories of computational abstraction, such as introspection over execution histories and meta-abstraction for reflection and self-adaptation.

6.2.1 Creating Multi-Disciplinary Communities

How can societal “grand challenge” problems—Climate Change, Energy, Safety and Security, Transportation, Health and Healthcare, and Livable Mega-Cities—be addressed? Clearly, software is a key factor, but these challenges transcend any single discipline. And so it is of paramount importance to encourage multi-disciplinary work and to create multi-disciplinary communities.

The goal of such communities ideally should be to address significant societal problems that, if left unaddressed, challenge the very fabric of our lives – our safety, security, prosperity, and health. Each of these problems will, without doubt, be addressed, in part, by systems that have pervasive computational elements. Hence, from a software engineering perspective, we need to create technical information infrastructures that can catalyze the development of successful societal-scale infrastructure systems. But the key word here is "catalyze." No one—not software engineers, practitioners, policy makers, scientists, or any other single group—will generate robust solutions to these grand challenge problems on their own. These are "wicked" problems – problems that involve complex interdependencies where solving one aspect may reveal or create other problems. Hence, our success will lie in our ability to provide an infrastructure that will catalyze innovation and creativity across a distributed community, allowing solutions to "grow" as understanding increases and new technologies emerge.

Our typical scientific approaches to problems in the past have been top-down and reductionist, and these methods are notably inadequate to deal with grand challenges.

The solutions to these grand challenge problems will be ultra-large-scale (ULS) systems and no single group will be able to design, deploy, and evolve them. Such systems are beyond precedent. They will be composed of both legacy systems and new technology, but affected by institutions, intellectual communities, and legal, policy, political, and economic constraints. Their creation and sustainment requires cooperation between research and other communities that have not typically worked together before.

To be successful, we need to create an intellectual and technical environment that will foster bottom-up innovation, focusing on cross-disciplinary research coalitions that include software engineers and domain experts as co-equal partners, but also policy makers and social scientists. Workshops are needed to bring together stakeholders to define new research communities leading to new programs. Finally, the multi-agency funding issues must be addressed: cross-disciplinary research on a grand scale does not fall neatly into existing funding models and funding agency missions. Financial incentives for the necessary participants are missing. Thus changes in policy will be just as important as changes in research paradigms. An ambitious campaign of public and industry outreach must be initiated to raise awareness of the importance of these cross-disciplinary efforts.

6.2.2 Timeliness

First and foremost, the requirements were not clearly defined, immediate, or compelling. Moreover, the world has never been as connected as it is now with the Internet, wireless technology, and the power of computational units and sensors. Historically, isolated research communities built customized, proprietary, and siloed systems. These point solutions typically do not scale well. Furthermore, this disconnect that has traditionally existed between software engineering and computer science, and other scientific research communities (and the reward structures within those communities) have seldom fostered an environment conducive to cross-disciplinary research.

But the payoffs, if we succeed, are compelling and urgently needed. An environment must be developed that fosters new understanding of complex systems, leading to revolutionary advances in each of these domains. And the software engineering community cannot do this alone.

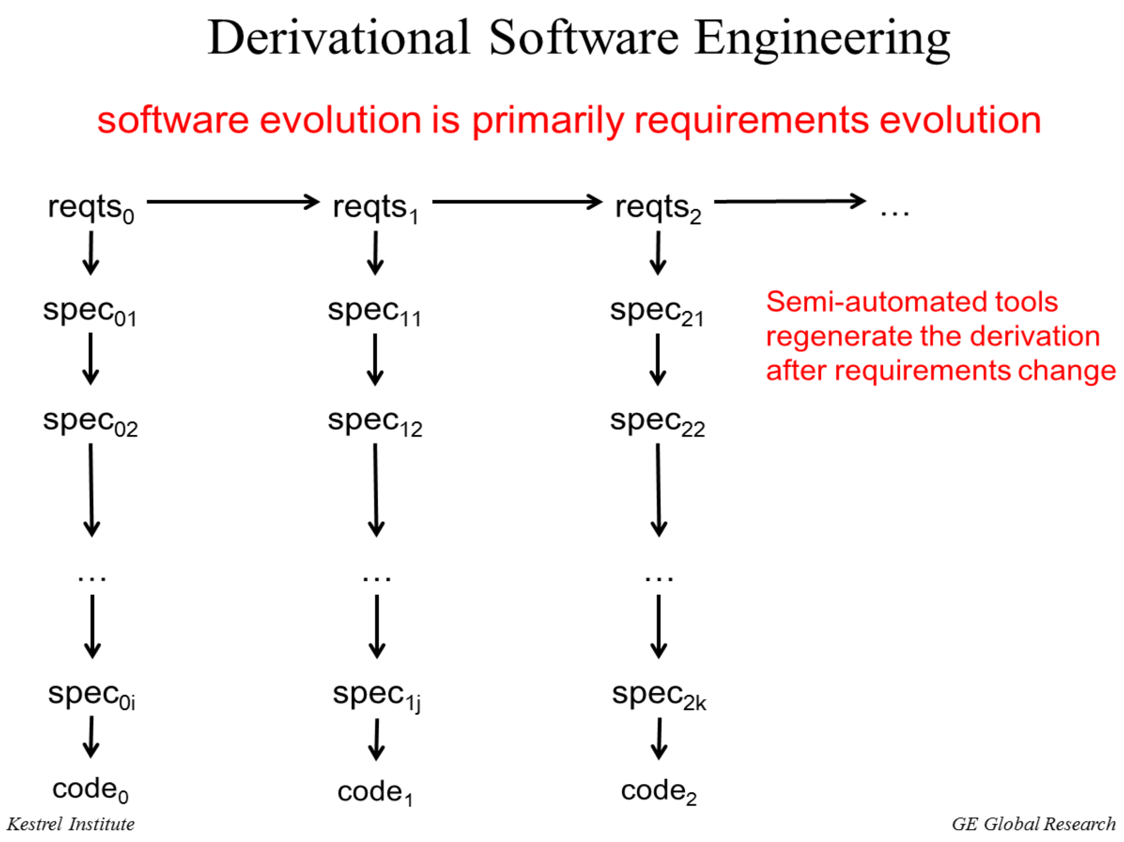
This will be costly and risky. A large and sustained program with funding in the range of $50-100 million per year over 5 to 10 years is necessary. Anything less will not fund the kinds of fundamental society-changing advances that are needed. The risks are numerous: researchers may not focus on issues most critical to standing up a successful national system, but instead continue with “business as usual,” creating small point solutions that do not scale well and don’t talk to one another; the required changes in political structures, economics and incentives may not be made; and industry may not be sufficiently involved.

7. Dependable Software-Intensive Systems

7.1 Automated Programming

The goals of automated programming research and development are to provide the foundation for major improvement in productivity for the development and evolution of software-intensive systems. At the same time, the ‘correct by construction’ paradigm provides the foundation for software dependability: high assurance, high performance, and other “ilities” of software-intensive systems. Automated programming R&D is especially targeted towards a radical improvement in software evolution – which is the major software engineering activity as prescribed by cost and percentage of the lifecycle. Evolution will be accomplished by machine-assisted modification of requirements and specifications. Implementations will be re-derived by replaying derivations up through high-level design, to increasingly higher-level automatic programming systems. Figure 1 illustrates this major improvement in the software lifecycle that will be enabled by automated programming R&D.

Figure 1 - Derivation Software Engineering



Achieving automated programming, although challenging, addresses the core scientific issues of software engineering. Achieving this goal will involve systemizing and capturing in machine-manipulable formalisms, software engineering knowledge that is now only implicit. This systematization will create a true science of software engineering. A conceptual model describing how an automated programming system could work is illustrated in Figure 2.

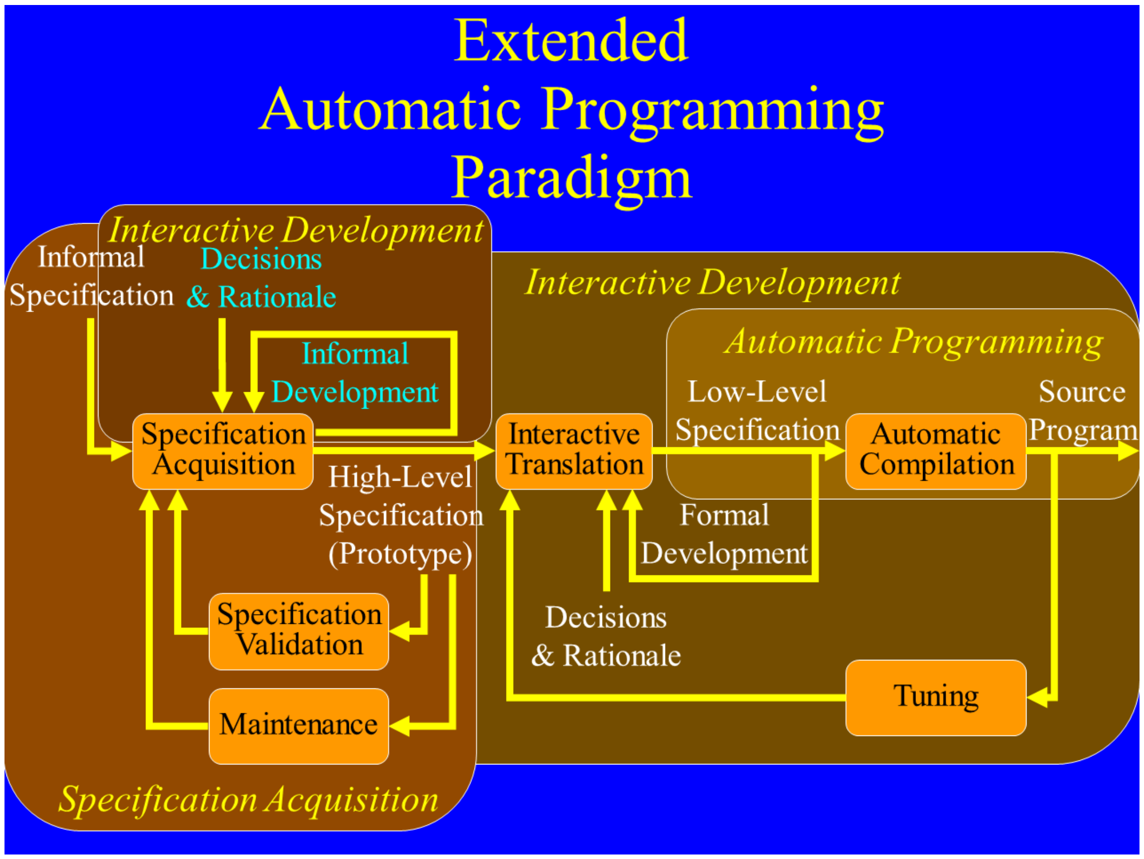


Figure 2 – Extended Automatic Programming Paradigm

The conceptual model represented in Figure 2 depicts a number of synergistically promising approaches:

- Interactive development of requirements and specifications
- Natural language and multi-modal support for interactive requirements development from informal artifacts
- Open-source corpus of software design knowledge
- Interactive derivation of software designs incorporating corpus of design knowledge
- Automated generation of high-assurance and high-performance implementations from designs
- Evolution by revising requirements and specifications and replaying design and implementation derivations

This proposed work builds upon past R&D, such as the Report on a Knowledge-Based Software Assistant[6] (C. Green et al). There have been sufficient advances in foundational and supporting technology development to provide a critical breakout opportunity for automated programming in the next five to ten years. Specifically, several formalisms have demonstrated they represent design knowledge in non-trivial domains. Critical supporting technology has been developed that includes high-performance automated inference support and integrated development environment frameworks, such as Eclipse.

In order to achieve widespread automated programming, the following are required:

- Assemble a maturation of formalisms to represent software design knowledge
- Create a critical mass of machine-manipulable software design knowledge, e.g., an open-source corpus, a jointly co-operative effort of the research and practitioner communities.
- Establish a coordinated effort across, at times, insular communities: requirement engineering, formal methods, software architectures, compilers, etc.

The potential payoffs for automated programming are significant. These include:

- Increased productivity through automation for initial development and evolution
- Software "ilities" that will be a by-product of mechanized development
- Certification evidence addressed as part of the largely mechanized development, rather than as an expensive afterthought activity
- Incorporation of mechanically supported design knowledge and design methods results in software products (or development?) that incorporate best practices

Developing the technology for widely applicable automated programming will be challenging. Costs include continued foundational work, building tool support, and compiling a corpus of mechanized software design and derivation knowledge. These are all part of the core scientific theory development for software engineering – a worthwhile endeavor even if the ROI takes longer than expected. At the same time, we can identify jump-start, near-term, activities that will bring targeted payoffs in the intermediate time frame and serve to mitigate the following risks:

- Achieving a critical mass of design knowledge and mechanized support
- Achieving an integrated effort across different software engineering research communities

We believe there is a unique opportunity for automated programming to revolutionize the industry in the next five to ten years. There is overwhelming evidence that the large software engineering community, and especially the community that develops safety and mission critical software-intensive systems that are commissioned by the government, is clamoring to see this happen. The success of domain-specific automated programming systems and model-driven software development (coupled with "autocoding" technology) demonstrates that the productivity benefits are substantial and will be adopted by software practitioners. The envisioned automated programming systems address critical gaps in current technology, that are of special interest to the Government. They provide:

- The foundation for assurance and other "ilities" such as performance
- Support for general-purpose as well as domain-specific programming

[6] R. Balzer, T. Cheatham, C. Green, D. Luckham, C. Rich, Kestrel Institute, Palo Alto, CA, August 1983; This report presents a knowledge-based, life-cycle paradigm for the development, evolution, and maintenance of large software projects.

- Support for requirements and specification validation and evolution

We now have the computational horsepower and core algorithmic advances that provide sufficient capability for formal automated programming support technology. Automated inference is one of these capabilities, as well as framework support for integrated development environments. Advances in multi-modal user interaction and natural language processing, coupled with large ontologies, provide the capability for interactive mechanized support for deriving formal requirements and specifications from informal and heterogeneous descriptions.

While challenging, the ambitious goal of widespread automated programming can be jump-started with near-term and intermediate action plans that will produce immediate, sustained, incremental benefits.

Jump-start, near-term action plan includes:

- Kick off an open-source catalog of formalized software design knowledge
- Form an Electronic Journal of software design knowledge
- Organize a crosscutting workshop comprising experts across software engineering disciplines

Intermediate action plan:

- Extend current domain-specific/model-based systems with interactive derivation of models
- Demonstrate the feasibility of the full paradigm in a particular domain

The jump-start and intermediate action plans will demonstrate the feasibility of the full paradigm and leverage the open research community. At the same time, it must be emphasized that achieving widespread automated programming will require long-term support of foundational R&D: formalisms for representing software design knowledge; interactive and mechanized tools for requirements, specification, and implementation derivation; mechanized support for software evolution through interactive requirements modification followed by mechanized re-derivation of implementations. This foundational R&D not only will provide long-term economic payoffs, but also will form the core of future scientific knowledge of software engineering – knowledge that is now only implicit, parochial, and unfortunately not always repeatable.

We expect to be able to gauge the success of this endeavor incrementally. The following indicate signs of success:

- Domain-specific automated programming will provide associated artifacts supporting certification of software-intensive systems as a side effect.
- Testing will no longer be considered an exorbitant expense as a result of the domain-specific automated programming side effects.
- There will be a measurable reduction in life-cycle costs for software projects incorporating automated programming.
- Software project costs will become significantly more predictable.
- Software evolution will no longer be delegated to the most junior people.
- Software evolution will become a fluid continuation of software development.

7.2 Dependability Arguments

7.2.1 Goals

We start with two basic premises:

1. A good design divides the system into elements that can each be targeted by appropriate analyses and thus is key to scaling up and targeting realistic systems. This top-down approach can facilitate effective allocation of analysis resources and achieve a lower cost of constructing a dependability case. Yet, little attention has been paid to exploring the relationship between the design and analyses.
2. Software analysis techniques are intended to provide information about the behavior of software, and thus facilitate confidence building, debugging, dependability arguments, etc. A typical validation involves a combination of different tools. Yet, these tools are not designed to make semantic analysis efficient or effective and inter-operate in a usable way.

The goal of this approach is to create a synergistic relationship between bottom-up analyses and top-down design to enable compositional evidence-building and tool interoperability that would enable analysis of large complex systems while quantifying guarantees that the analysis tools provide.

This project is part of a grander goal of providing support for creation and analysis of, quality software systems. To do this, construction of dependability arguments must be facilitated. The problem requires the collaboration of multiple computing sub-disciplines – from requirements engineering (for identifying requirements and determining which ones are critical), to establishing standardized critical requirements methods to create architecture and design of systems. Furthermore, dependability arguments need traceability of critical requirements in the code that may enable the evidence of program coverage produced from various analysis tools to be converted into dependability arguments that can be communicated to stakeholders.

Last but not least, evidence-based dependability arguments are expected to be essential for evolving software systems.

7.2.2 Challenges

The influence of design on analysis needs to be understood, i.e., to determine, among other issues, how properties/requirements vary with design contours and how the structure of design elements influences cost-effectiveness. Next, a technique-independent model combining results of multiple analyses must be developed that describes the evidence merge operation and is capable of handling multiple levels of abstraction, constructed under different assumptions in analyses, while accommodating both under- approximating (e.g., testing) and over-approximating (e.g., most static analyses) approaches.

Additional technical challenges involve determining how to adapt analysis techniques to produce and consume “combinable” results. For example:

- Determine how to compute rich semantic information without a significant increase in cost
- Determine how to exploit existing analysis results, taking into account differing assumptions/abstractions, so that the subsequent tool in the chain not only benefits from the evidence previously collected but is also able to concentrate on the unexplored aspects of the program.

7.2.3 Promising Approaches

If the goal is to produce dependable, cost-efficient software systems, the synergistic relationship between the design and analysis is of paramount importance. The design component must be categorized by requirements and analyses. There are numerous examples of systems that were designed for dependability from the beginning, especially in security and safety engineering. Leveraging techniques and experiences from those areas, and devising a general software design methodology, is one promising approach. Ample research exists concerning requirements traceability that also could be helpful.

The analysis component quantifies dependability through the use of tool interoperability and evidence accumulation, thinking of evidence as positive and negative (and unknown, which needs to be reduced by additional applications of non-redundant tools), merge operations, tool-independent and language-independent representation of assumptions and positive/negative evidence. Thus, rather than trying to do it all, tools should primarily focus on producing “better’’ evidence, i.e., constructed under fewer assumptions, or “more’’ evidence, by reducing unknown program behavior.

Finally, we propose to store and communicate “normalized” evidence using a database.

7.2.4 Timeliness

7.2.5 Potential Payoffs

By factoring out critical parts at the design level, resources can be allocated more effectively and thereby provide an explicit economic incentive for upfront design.

Categorization of requirements will ameliorate the tool-selection process. Presentation of generated evidence can be standardized and thus help the conversion into dependability arguments. This approach should also explicate, quantify, and exploit synergy between different analysis techniques.

Furthermore, a repository of analysis results helps analysis users understand not only the accumulated evidence but also the cost-benefits of using different tools. In addition to creating higher-quality systems and understanding how to build targeted system-appropriate tools, this framework will catalyze research on static and dynamic analysis tools. Not only can their speed and scalability be compared, but also their “equality,” i.e., enabling analyze of a greater percentage of the program or with less assumptions. This proposed language and tool-independent representation is expected to serve as ontology for quality comparison of tools.

7.2.6 Costs and Risk

Design for dependability and analysis is risky and may be costly because:

- The influence of design on analysis is still not well understood.
- As requirements change, the design will also likely change, and this may invalidate previously-established claims about the dependability of the system.
- The top-down design approach is not applicable for legacy systems; the questions of refactoring may have to be investigated instead.

7.2.7 Evidence from tools

Finally, on the synergetic level, we have assumed, possibly incorrectly, that collected evidence coupled with requirements traceability is sufficient for generating dependability arguments. This may not be so, especially for systems that are not well designed.

7.2.8 Action Plans and Jump-Start Activities

On the design level, we need more research projects that involve compartmentalizing critical properties in order to explicate connections between dependability arguments and the design structure of the system.

On the synergetic level, we need to gain experience constructing dependability arguments from analysis-tool-generated evidence, to gain a better understanding of the information that needs to be collected. Furthermore, we need experience constructing dependability arguments from a piece-meal collection of available evidence.

On the analysis level, we need to design a representation for capturing the assumptions and semantic evidence as well as to construct an infrastructure (database) to store this information. Next, techniques and frameworks must be constructed to merge the evidence at different levels of abstraction that may also be computed under different assumptions by different tools. The feasibility of such an endeavor must be documented by configuring dependable tools to generate the evidence. If successful, we will invite subject-matter experts to a tool builders' "summit" to communicate this paradigm. In addition, we propose to kick off a research project focusing on reusing evidence from different tools and leveraging it to analyze changed programs.

The experience gained from reusing evidence for changed programs will help us understand how to construct dependability arguments for changed software.

7.2.9 Evaluation

- Multiple diverse techniques will be able to work together effectively.
- Evidence of behavior coverage can be presented in a technique-/tool-independent form.
- Ability to identify how design makes specific analyses more efficient or more precise for a given set of techniques.
- Industry will accept these methods; construction of dependability cases will become more widespread.
- Techniques and knowledge for evaluating the evidence in a dependability case can be transferred to certification boards.

7.3 An Informal Approach to Automated Programming

7.3.1 Vision

Summary: Instructing computers should be like instructing humans. Consider programming as a process of computers helping humans to clarify and communicate ideas.

Instructing computers has been frustrating because we must use carefully constrained ways of combining constrained vocabulary (search being a notable exception). But when we interact with other humans, we're not nearly so constrained. Interacting with computers could be more like interacting with other humans if we could tackle the problem of informality.

We envision a programming system where a human and a computer work together to iteratively refine informal natural language descriptions (or other informal descriptions) into artifacts with increasingly controlled semantics, producing code that is tested and from which models are developed. The system could then use (and update) repositories of world, domain, and problem-solving knowledge in order to handle ambiguity. In the face of changing requirements or unforeseen failures, the system could revisit earlier choices.

Such a programming system would demonstrate competency in a number of ways. For example, given a prescriptive description, it should produce (in interaction with humans) models, tests, user interfaces (UIs), and code. Given bug reports, it would be able to produce tests. Given narratives of hypothetical interactions with a target system, it could determine constraints and tests about that desired system. The programming system could also seek evidence that a given solution system addresses its informally specified requirements.

Applications include end-user programming and professional programming. In end-user programming, such a system would "lower the bar" for programming. The same techniques could also give end-users a "natural language command line," which could help them specify tasks with multimodal input (text, speech, touch, gesture, etc.) or help them accomplish difficult or one-of-a-kind tasks.

Anecdotally, the more powerful a formal approach is, the more difficult users encounter dealing with errors. The informal approach could make errors more understandable, for example, by providing an intuitive context for the error. Regardless, proponents of the formal approach (especially at levels above code) must address debugging: what action to take when the formal reasoning fails or makes unsuitable assumptions.

7.3.2 Difficulty and Interest

Natural language is everywhere, and it is flexible. It is the essence of human communication, in part because of its informality. However, reasoning with informal representations such as natural language is not well studied. Also, the ability to correctly understand ambiguous statements depends on shared models of the world, and currently computers share few models, i.e., they lack human common sense.

But if the reasoning problems are solved, systems developed informally will be better equipped to behave appropriately in changing contexts or in response to complex failures. They will also communicate more effectively with users during execution, e.g., by allowing users to give instructions informally, explaining unexpected behavior understandably, and explaining failures in a way that the human can understand and help resolve.

7.3.3 Promising Approaches

Some approaches presented at this workshop are promising, including a system that can map English requirements specifications into UML models ("RECAA," Tichy and Koerner 2010[7]) and a system ("ProcedureSpace," Arnold and Lieberman 2010[8]) that uses code examples to help clarify informal natural language statements of code purpose. Another promising approach describes incrementally structuring informal input, exemplified by the Business Insight Toolkit at IBM (Ossher et al. 2010[9]).

Efforts in related fields using these methods are also promising. The Human Computer Interaction (HCI) community has begun to use informal methods to improve the usability of programming, such as research on Opportunistic Programming at Stanford and Keyword Programming at MIT. Many researchers are now applying natural language information retrieval techniques to repositories of software artifacts. Earlier approaches used structured natural language (Fuchs 1999, Rathod 2005) or ontological representations (Kaiya and Saeki 2005/2006, Gervasi 2001, Meng 2006) of natural language to move towards informal languages in software engineering.

7.3.4 Timeliness

[7] Position papers are available on the ACM Digital Library (https://dl.acm.org/citation.cfm?id=1882362&picked=prox)

[8] Ibid

[9] Ibid

Many previous attempts have failed to appreciate the challenge and opportunity presented by informality and ambiguity in natural language. In recent years, however, natural language processing (NLP) techniques, ontologies of world knowledge, and open-source repositories have made significant progress, providing a stronger foundation for rapid innovation.

7.3.5 Payoff

Informal methods would reduce premature commitment to formal representations, which would reduce development and modification costs, improve software quality, and speed up development cycles. These methods would also allow customers to become part of the design process by directly connecting to the models that the software engineers have built and manipulating the specification. They would also enable a more integrated style of development, where customers and users participate in a dialogue for writing and clarifying specifications, eliminating the need for programmers to guess.

7.3.6 Costs and Risks

These techniques may not be practical, though formal techniques run similar risks. For some projects it may be less expensive to write the code than to work with a natural language specification. These representations may not be scalable or easily generalized. Finally, flaws in any knowledge base or ontology that the system uses might compromise the result.

7.3.7 Action Plan

In the near term, we can extend the scope of existing approaches (e.g., generate test cases from APIs and bug reports) as well as scale up techniques (e.g., try with other programming languages or in different programmer communities). The psychology of program development could be explored to help determine the perceptions programming researchers have about their programs and how useful they are. In the interim, existing natural-language processing tools need to be improved to allow a more complete coverage of natural language by comparison with the current, sometimes isolated and scattered solution spaces. In addition, existing ontologies (K of 2004) should be enhanced and consolidated to deliver comprehensive coverage of world knowledge.

In the intermediate term, developing small working tools, in particular as plugins for platforms such as Eclipse, will encourage innovation, collaboration, and clarify requirements. In this timeframe, software engineers should initiate a dialogue with industry to identify relevant problems and concrete challenges. Finally, a cohesive community must be formed to bring together nascent work at the intersection of HCI, NLP, and Software Engineering, as well as communities working with informal reasoning and knowledge bases. Finally, common benchmarks and the use of examples could provide a collaborative, common ground for the community to share and verify ideas, and to intensify competition.

7.3.8 Evaluation

Decades from now, we should be able to describe our requirements to a computer, similar to the way we describe them to human software engineers, and have a dialogue with the computer to clarify those requirements and produce working code, without directly working with any formal representation. In the shorter term, the benchmarks suggested in the action plan will demonstrate progress, as will demonstrations in small example domains.

7.4 Differential and Interactive Program Analysis

It is well known that developers and testers spend a lot of time understanding program changes.

They need to understand the impact of changes on the quality of the code (e.g., introducing bugs, worsening the execution time, breaking interface contracts). Existing techniques such as regression testing are expensive, and though they provide evidence of correctness and quality, they do not guarantee it. Formal static analysis techniques do provide guarantees, but their performance may be too slow to apply during software evolution, which forces developers to delay formal analysis until a final testing phase. At this late stage, however, correcting defects may be very time-consuming and costly.

Our goal is to combine static analysis with quick, interactive feedback about the possible impacts of code changes. Defects can therefore be detected and corrected early in the development cycle, improving code quality and developer productivity as the source code evolves.

With the maturation of the state of the art of program analysis, it has become clear that verification is not, or cannot feasibly be, an instantaneous event or demonstration that an entire given system is correct. Instead, verification must be a process that tracks software across its evolution, either on a day-to-day basis or over the longer term of major releases. In this setting, two key insights can be exploited to achieve higher software quality through better verification and providing better, quicker feedback to designers and testers. The first is differential analysis through which a code version can be verified with respect to previous versions. The second is interactive analysis whereby developers can explain key assumptions and justifications supporting their coding decisions.

7.4.1 Vision

Differential static analysis attacks one of the fundamental problems in system verification - static analysis approaches are scalable but ordinarily generate too many false alarms, while dynamic analysis algorithms are non-scalable for large systems. If a differential approach is taken, it may be possible to focus on the set of differential behaviors while assuming that the original version, which has been changed, is "correct." This means that we will provide "relative" guarantees (across a change) rather than "absolute" guarantees. This makes the verification problem focused (hence tractable) while minimizing false alarms.

7.4.2 Potential Payoffs

Interactive analysis supports the observation that a developer, mired in coding, is the person most cognizant of the sequence decisions that has resulted in the current coding state and what the next intended step will be. Therefore, this is the most opportune moment to challenge the developer to provide a justification for these coding decisions. This interactive analysis has the potential to locate errors and also provide a formal documentation trail.

7.4.3 Achieving Goals

Achieving these goals is difficult because testing and analysis algorithms are inherently hard. Moreover, understanding developer intents behind code changes is difficult. However, there is a real opportunity to make analysis tractable by exploiting code deltas. Therefore, achieving these goals is worthwhile and can involve exciting research.

7.4.4 Promising Approaches

Promising research efforts toward achieving these goals include exploiting better algorithms that can exploit changes, differential symbolic analysis of partial code, and exploiting change and interaction history to construct useful feedback to developers. Promising recent work includes exploiting similarity of code using uninterrupted functions for equivalence checking and differential symbolic executions. Work has also been done in the area of interactive static analysis of hard real-time software. An enhanced high-speed analysis algorithm runs in the background of the integrated development environment (IDE), providing continuous feedback to the developer about the worst-case performance of the evolving source code. In addition, the advent of multi-core and experimenting with the power of the "cloud" can enable the development of new and interesting interactive IDE facilities by making version histories more accessible.

7.4.5 Not Attempted Previously

There are various reasons why these research directions have not been pursued. First, for differential analysis, the foremost challenge is to formulate concrete problems that will be solved by these tools. Regression test selection (selecting a subset of tests impacted by a change) is a very well-studied problem, and our proposal is complementary to it. Second, many of the required advances in symbolic analysis capabilities have been fairly recent. These capabilities are essential in order to reason about partial code. With the recent advances in symbolic reasoning, and other advances in supporting technologies such as document generation based on natural language processing as well as hardware acceleration capabilities, it has become far more feasible to support these research directions now.

7.4.6 Costs and Risks

Potential costs and risks include inaccuracies introduced by static analysis due to the complexity of the underlying algorithms and the consequent need to employ over-approximation techniques. In some cases, the effects of a code delta may propagate pervasively, potentially requiring whole-program analysis. In addition, understanding developer's intent and separating expected changes from unexpected ones are challenging. Last but not least, most of the machines sold a decade from now will consist of multiple CPU cores. Programs for these machines will involve a combination of constructs, including parallel loop execution, producer/consumer pipelines, and delegates/continuations. These notations provide an overall sequential/deterministic semantics while hiding under the guise of the powerful task scheduling mechanisms. A 'change' in this setting involves functional, performance, and resource consumption dimensions. Computing the impact of a change is a non-trivial (and yet very important) research agenda.

7.4.7 Timeliness

This research is timely for several reasons. Gone are the days of releasing software in 1-2 years; today many companies are following the so-called continuous software development model, which exacerbates the verification problem. However, by exploiting deltas, there is a real opportunity to contain debugging costs. In addition, significant new demands are being placed on software quality (e.g., software in embedded contexts), and it is important to exploit concurrency/parallelism. Security and privacy issues are also of growing concern. Advances in symbolic analysis (e.g., SMT) are happening now, and many differential program analyses based on Pex, Differential Symbolic Execution, and SymDiff serve as good exemplars. Efficient and parallel/concurrent programming requires these advances, as well.

7.4.8 Action Plans

Proposed concrete plans include

- Understanding how developers characterize changes:
- Creating a community of researchers interested in delta-based static analysis an testing
- Developing analysis infrastructures that allow community-wide effort

- Launching focused research addressing issues such as concurrency and security
- Collecting benchmarks representative of various types of code changes (refactoring, bug fixes, feature additions, performance optimizations, etc.)

7.5 Defining Real Programs for the Masses

Requirements for systems and specifications of system components, including libraries, are important enablers for software development, maintenance, and a host of analyses and aids to development and maintenance. For these benefits to be

realized the languages used to record program requirements and inter-module interface specifications must be informative and clear not only for tools but also for their human readers (system architects, programmers, testers, etc.).

There is considerable excitement about the wide range of potential uses and applications of specifications. These range from previously-known applications such as verification, automated program development using transformational techniques, and automated debugging and fault localization (also known as blame analysis in the programming languages community) to emerging application areas, including end-user programming, semantic code search using specified properties, and globally-distributed development. The increasing number and scale of libraries provided with languages like Java and Python result in intensive usage. This usage makes the need for high-level specifications a crucial productivity-enhancing technology for library understanding and reference purposes.

As the world of computing expands, it is becoming increasingly challenging to define engineering requirements given the limitations of computer language. Many programs do not have well-defined or easily formalized correctness criteria. Examples include application domains such as graphics and layout or artificial intelligence-like programs such as those that make recommendations for movies that a user might enjoy. One approach for dealing with uncertainty about correctness is to use probabilistic specification. While non-functional requirements such as performance, security, and privacy are standard in requirements engineering, requirements for usability and maintainability seem difficult to specify.

While the promise and challenges of interface specification for program components have existed since at least the 1960s, a number of modern developments and insights in formal specification are presenting new opportunities with the potential to make a significant impact.

One such insight broadens the classical view of an interface specification as a contract (i.e., pre- and post-conditions written in logic) by using new techniques. The techniques include path properties, which describe behavior across execution traces; models, as used in model-driven development; and test cases, including their more sophisticated modern forms like parametric tests or symbolic predictive analysis. Increasingly, users want specifications that describe non-functional characteristics such as time and memory performance or security- and privacy-related properties.

A second modern development is the increase in sophistication of analysis techniques and more effective approaches for integrating specifications with programming languages. Modern logic solvers and automated theorem proofs have increased tremendously both in sheer power and in the expressiveness of the logical theories they support. This offers increased ability to mine specifications for non-trivial deductive information, as well as to perform tasks such as consistency checking, which are useful during specification development and maintenance. Increasingly expressive type systems also hold the promise of more seamless integration of specification language features into programming languages. Technical type system features like "type states" and "dependent types" allow very rich semantic properties of data and functions to be expressed in types. This approach puts the power of general specification into the programmer's hands in the familiar form of a type system.

We discussed three strategies to make requirement and specification languages more easily understood and usable. The first approach involves the use of abstraction, which allows mathematics to be hidden from specifiers and users to the extent possible. Mathematics would still be used internally in tools that process such specifications, but proofs, problems, and counterexamples should be communicated to users in ways that hide the mathematical details. A second strategy that would make interface specification languages more easily understood by programmers is to add features to the programming language that would be useful for specification, such as dependent types, in particular functional programming features, e.g., closures and expressive value types (like sequences and maps) and type system features. Adding such features would allow programs to have more of the expressiveness needed for specification, and thus would allow specifications to be written directly in the programming language instead of in a special-purpose assertion language. A final strategy is to work on case studies of specifications in many different application domains, to build a vocabulary of modeling concepts that would allow more succinct and helpful specifications in those domains.

For programs written in languages with very large and rich libraries, obtaining appropriate specifications for these libraries from the existing natural language documentation historically has been a major problem, which recent advances in NLP discussed at the workshop may help ameliorate. Another technique for creating such specifications would be to exploit information implicit in the test suites for such libraries, perhaps using data mining of runs of the test suites.

Several concrete activities could jump-start work in this area. First, we could identify challenge problems for requirements and interface specification languages. Second, which is specific to interface specification, we could develop an annotation framework, either for a specific widely used programming language (like Java) or a generic framework that could support various specification language ideas. This would lower the cost of building tools and carrying out new research. A third recommendation is to organize a Specification Languages Summit, possibly as a Dagstuhl meeting or similar focused event, to foster connections across communities that do specification-relevant research, including: requirements engineering, formal methods, programming languages and compilers, and testing. A Specification Languages Summit may also be critical for enlisting researchers for the first two suggested activities.

The ultimate criterion for success in this area is transitioning the requirements and specification language techniques and tools to the private sector. This depends both on reducing the cost of writing specifications (reduce the training requirements for notations and reverse-engineer specifications from existing code or documentation), and improving the payoff from specifications. The main risks involved are organizational risks (a coherent specification community with the diverse expertise required will not crystallize); technical risks (challenges in specification language design and specification analysis will prove too difficult to overcome for mainstream use); and adoption risks (even innovative solutions will require a long time frame to transition to industrial practice).

7.6 Evolution Group

7.6.1 Goals

Many modern software systems will be long-lived and will evolve constantly during their life spans. Designers and developers of these systems should be able to predict how their systems will evolve and be able to manage this evolution in a coherent way. They should be able to estimate costs and monitor the progress of evolution based on knowledge about evolving systems in general and about the specifics of their system.

7.6.2 Challenges

Software evolution is interesting because of the increasing number of long-lived, evolving systems, but it is very difficult to predict and manage. There is no accepted wisdom either about what constitutes normal and expected classes of changes or about what classes cannot be anticipated. Proposed approaches to managing certain forms of evolution, such as Software Product Lines, are not widely used and are not well supported by either formalisms or existing tools.

7.6.3 Promising Approaches

Evolution-aware software processes and practices

- Process models that explicitly address evolution (e.g., Context-Driven Process Orchestration Method (CoDPOM))
- Cost models that adequately assign costs related to evolution (e.g. the concept of technical debt)

Models of software evolution

- Terminology and notations
- Evaluative models and measures of quality

Representations and tools that support the above

- Language features capable of describing and modeling evolution (e.g. program fields)
- Semantics-rich product repositories
- Techniques for analysis, presentation and visualization

7.6.4 Timeliness

There is a dearth of models that either academia or industry find satisfying. Furthermore, most relevant data was hidden within the proprietary bounds of particular institutions. The lack of models has not changed and needs further research, but the blossoming of the open source software community has made interesting data accessible in ways not seen previously. It is now possible to observe how multiple systems evolve over a variety of system scales. The widespread use of IDEs that support plug-ins has simplified the creation and deployment of sophisticated tools making novel tools and methods easier to test.

7.6.5 Potential payoffs

Economic and technical payoffs will be realized when system designers and developers are able to plan for evolution and systems are designed systematically that are capable of supporting predictable future changes.

7.6.6 Costs and risks

The evolving systems of interest will be enormous, making detailed analysis a long and tedious process. Since evolution occurs over time, creativity will be required to evaluate new approaches in a timely manner. The complexity of evolving systems may make it difficult to arrive at concise results.

7.7 Incentives

7.7.1 Goals

We propose to treat software as decentralized systems that undergo continual evolution in highly dynamic environments. This perspective allows us to adapt techniques from economics and evolutionary biology to the study of software, potentially enabling (1) predictive models about software development and evolution and (2) normative models to inform decision-making at many levels of granularity.

7.7.2 Promising approaches

We propose three first-class principles of research study:

1. Change Over Time - Biological systems respond to changes on both evolutionary and ecological time scales; these time scales broadly correspond to pressures towards equilibrium and pressures moving equilibrium forward in economic systems. The forces that can and do govern software change should be studied, including but not limited to evolutionary or economic mechanisms.
2. Decentralized Control - Software systems are largely products of a decentralized environment. Such systems evolve in response to both accidental and intentional changes in the environment from both centralized actors (e.g. upper management) and de-centralized pressures (e.g. technical developments or Application Programming Interface (API) changes). In either case, the mechanisms of decentralized control and software system evolution should be studied.
3. Short-Term vs. Long-Term Tradeoffs - Short- and long-term utility tradeoffs, implicit in management decisions, are explicitly managed in economic systems; evolving biological systems tend to be strictly reactive. Understanding the nature of tradeoffs in software may suggest ways to manage them by proactively leveraging evolutionary or economic mechanisms. How systems, operating at equilibrium under certain economic rules, react when perturbed and how evolutionary systems respond to short-term vs. long-term pressures, should also be explored.

More specifically, economic approaches may also be leveraged to analyze and inform software development decision-making. For example, “technical” debt in software development could be explored to gain an understanding of software management and its evolutionary tradeoffs and determine if it causes a decrease in performance resulting in a loss of market value (if it isn’t economical to fix a bug, should it be classified as a bug?), or develop a notion of what it means for a software system to reach equilibrium. Evolutionary biology suggests approaches and metrics to study the types of selection, mutation, or evolution a system experiences (and at what granularities or timescales) potentially enabling long-term predictions about, for example, scaling behavior and error distribution.

Economics additionally suggests promising approaches to develop normative decision-making models, such as incentive design for effective software development or bidding systems to structure software development decisions (e.g., individuals bid on assignment to teams, teams bid on assignment to projects). It may also be profitable to focus on adapting biological design principles to software, such as the robustness of behavior resulting from multiple heterogeneous drivers, or explore how to adapt micro-biological operations like mutation or crossover at the code level, to revisit the dream of automatic programming.

7.7.3 Challenges

The analogies to economic and biological systems are unlikely to capture all aspects of system development and complexity; an important research challenge is to establish the scope and boundaries of this approach. The forces affecting software change appear to operate at multiple time scales that interact nontrivially, adding additional complexity. Many of the principles of economic and biological systems remain poorly understood, complicating their application to software.

7.7.4 Timeliness

The nature of software systems as complex, evolving entities has only recently become more evident. Software development has been successful without this perspective; the question that must be posed is, “Can software development improve if approached in a different way?” Recent initial work in this area suggests that the proposed approach holds promise.

7.7.5 Action plan, jump-start activities

There is a wealth of existing research on both biological and economic systems under evolutionary pressures that could serve as starting points for new software research. First, test for evolutionary progress in a software system against a null model (where changes are random, not subject to evolutionary forces). Second, adapt measures of economic equilibrium, or biological robustness or selection, to evaluate software development/evolution. Both of these initial approaches suggest the potential utility of a longitudinal case study of the changes that take place in an existing, long-lived software system.

7.7.6 Evaluation

This work may be considered initially successful when new models can be used to gain insight about or influence software evolution. The effectiveness of a predictive model may be tested by predicting future evolution of a long-running system from a past state. Small-scale proof-of-concept evaluations (such as the Liquid project at IBM) may be performed to test new software management techniques. More traditional metrics of software quality or robustness may be used to evaluate new development techniques, such as robust systems or automatic programming or debugging techniques that take advantage of biological principles.

8. Improving Decisions, Evolutions, and Economics

This theme identified a set of common goals that largely unify the three sub-group reports.

The primary goal is to continue to enhance the ability of software engineers, managers, and businesses, to make decisions about software, at many levels and granularities that effectively balance utility and cost. Utility and cost are intended to be broadly interpreted, to include domains such as monetary, societal, and environmental.

A secondary goal is to broaden the number of stakeholders of software-based systems – from very diverse backgrounds, educational levels, cultures, socio-economic groups, etc. – who now benefit from and rely upon these systems, either directly or indirectly. That is, these people must also be part of the "how to make better decisions about software systems" mentioned in the primary goal.

An overarching technical goal is to further increase our ability to build confidence in properties – properties that matter – of software-based systems. These properties represent a variety of perspectives, including usability, analytic properties, emergent properties, properties of the software process, and performance, to name a few. Furthermore, our notion of increased confidence in software-based systems should and must be broadened, beyond traditional characterizations of correctness and must approach characterizations, such as "satisficing" (combining *satisfy* with *suffice*). Confidence increases not only with technical analysis and assessment, but also with experience – perceptions of the approaches and individuals involved with the system. No single property or measure will provide substantive confidence in the overall utility of a system, nor is the overall value of any system (i.e., the utility minus the cost) possible to assess at a single point in time; these must all be assessed over the lifetime of each system. Finally, it is crucial to identify properties *that matter*; in particular, we characterize properties that matter as ones that are not only descriptive, but also provide a basis for improved decision making.

As a final goal, we should establish new incentive models – broadly construed and across all stakeholders and properties – that enhance the balance of utility and cost over the lifetime of software systems. A specific challenge is to ensure that these incentive models are, at the very least, not in conflict with one another. This observation goes hand-in-hand with the ever-increasing role of evolution in software. With an ever-increasing number of diverse users, with the Internet, and the like, change is the new constant. Finding ways to balance the many pertinent and diverse pressures on software systems, over time, is vital.

In the article "No Silver Bullet — Essence and Accident in Software Engineering"[10]. (*Proceedings of the IFIP Tenth World Computing Conference*: 1069–1076.) Fred Brooks observed that there is a difference between essential complexity and accidental complexity in software systems; this gap may influence current research. Essential complexity derives from the nature of the problem itself; accidental complexity arises during the realization of software to solve the problem but is not core to the problem itself. Many approaches to improving software engineering intend to decrease the gap between these forms of complexity. At some abstract level, this gap suggests that software engineering has boundaries that are rarely considered: Best practice represents an "upper bound" and essential complexity represents a "lower bound."

[10] Frederick P. Brooks, Jr., Information Processing '86, H. J. Kugler, ed., Elsevier Science Publishers B.V.

Finding more concrete notions of these bounds could change the way in which we – as well as other stakeholders – think about, assess, and improve not only specific software-based systems but also software engineering research results and approaches. Methods, tools, languages, processes could then be considered and assessed in a shared framework.

8.1 Software Data Analysis

8.1.1 Problem description

Everyday software engineering tasks and activities rely on stakeholders making a variety of decisions, ranging from developers making decisions about the implementation of software to managers deciding its release time. These decisions depend on the skills and experience of the people, the availability and access to relevant information, and their ability to understand it. The amount of data generated during the evolution of software systems is staggering. For example, the Mozilla browser project has 10+ million lines of code, almost 200,000 commits, and more than 500,000 bug reports (http://www.ohloh.net/p/mozilla/). Too much data also leads to information overload. It is thus important to determine the amount of information that is both necessary and sufficient to optimally design a software engineering task. Stakeholders, given the right amount of data and the support they need, can then make informed decisions.

8.1.2 Solution

The solution is multifaceted. First, we need to study and understand how humans use data and information to make decisions in software development. Next, traditional software analysis must be augmented with data analysis techniques and integrated into software development processes, practices, and knowledge, from fields such as analytics, business intelligence, data analysis, data mining, prediction models, empirical studies, economics, etc. Additionally, the best ways to present data and information to decision makers must also be defined.

8.1.3 Goals

The goal of this research is to provide analysis skills to stakeholders, define methodologies, and build tools that enable them to determine the support necessary to cope with and reduce the complexity of today's software systems.

8.1.4 Challenges

The nature of software engineering data is unique: it is heterogeneous, incomplete, evolving, and it deals in specifics that are typically not readily generalizable. While other fields that rely on data analysis have common data models for their domain, a software data model does not exist. We believe it is challenging but necessary to design one. Today there exists only a superficial understanding of how software engineers and managers use data to make everyday decisions.

8.1.5 Promising Approaches

Several research communities have investigated how software data can support software engineering tasks; for example, representative venues that promote such work are MSR, PROMISE, SSBSE, ESEM, RSSE, and SUITE. This work has helped identify the problems and challenges, and demonstrated the importance of data analysis in software engineering.

8.1.6 Timeliness

The open-source world provides access to software data at a scale not encountered before. At the same time, more and more industrial, proprietary data are also available. The early work from the communities mentioned above has given software engineering researchers a better understanding of data analysis techniques. Furthermore, society and businesses are interested in becoming more data-driven.

8.1.7 Potential Payoffs

Developers and managers will be able to make more informed and confident decisions, which in turn will lead to better products and practices. The existing solutions to most software engineering tasks will be enhanced and they will rely less on intuition and more on data-supported decisions. That will enhance the ability of people to better understand the software engineering products and processes.

8.1.8 Action Plans, Jump-Start Activities

- Create mechanisms (e.g., funding) that facilitate collaboration among software engineers, data analysts, and human-centered researchers.
- Support data-centered empirical research, i.e., fund activities that result in data collection, sharing, and benchmarks. While other data-driven fields, such as information retrieval, have government-supported venues and competitions, such as the TREC series, nothing comparable exists in software engineering to date.
- Create training programs for software engineers in data analysis.
- Create new positions in software businesses, e.g., special software analysts who have the skill, experience, and knowledge to run data analysis.
- Promote migration to industry: deployment and customization of research tools.

8.1.9 Costs and risks

The main costs reside in training software engineering researchers and practitioners in data analysis. The technical costs relate to data acquisition, cleaning, integration, and maintenance. The main risks are inherent to data analyses, such as the data correlation and causality fallacy.

8.1.10 Evaluation

The success of this research can be measured by increased data sharing and availability of benchmarks, which are necessary to assess the performance of the research. In addition, a wide range of analysis techniques and methodology for software engineering will be available, which stakeholders can use to make data-supported decisions. The implementation of successful research endeavors in industry and open-source is both a goal and a validation of success.

9. Advancing Our Discipline and Research Methodology

9.1 Challenge in Software Engineering Research

Abstract: Software engineering is diverse, and the research methods must be correspondingly diverse. We have not been articulate about what our methods are and how to select a method appropriate to the content and maturity of the topic. A portfolio of research methods will identify the methods, requirements, the criteria for validating results, and opportunities for improvement. Assembling the portfolio will allow us to compare methods, improve evaluation of research and education of researchers, and make decisions about future investment in the portfolio.

9.2 Motivation

Software engineering (SE) has a number of familiar research methods, ranging from exploratory methods to define the problem to be researched, constructive methods that help traverse the possible space of solutions, and empirical methods to assess the quality of the achieved results. However, the methods are not always explicit and clearly defined, and the software community is not always reflective about the methods and the selection. As a result, method selection is not without flaws: the chosen problem might be irrelevant; the proposed solutions might not work in practice; the context for assessment is not given.

The consequence of not identifying and using a proper research method often results in a lack of clarity, making it difficult to evaluate and communicate research proposals, papers, and results to government agencies, program committees, industry, and students. Many senior members, such as Walter Tichy, often complained in the past that "we need numbers," but even with numbers the studies often do not have enough context to be repeatable or transferable. Method selection must be much more rigorous.

SE is diverse, ranging from designing via executing to maintaining processes and programs. Consequently, we need a variety of research methods. To demonstrate the need for varied research methods, consider two possible SE research questions one might ask. (1) Does Distributed/Global Software development affect quality? Contrary to popular belief, the answer is no, at least not for a particular company. The chosen method for answering this question was to use empirical methods, as demonstrated by case studies at Microsoft as reported by Bird et al., 2008. (2) Can we show that a particular program terminates? We employ formal methods to answer this question. Cook demonstrated in 2006 that most system programs do terminate by inferring proper measurement functions despite the general knowledge that proving program termination is in general easily done.

Looking at these very different SE research questions raises the following questions:

- Does the chosen method match the posed question?
- Is the chosen method appropriate at its current stage of maturity?
- Is the chosen method appropriate for the type of data available?
- Does the method yield results of the appropriate level of generality?

Case studies, as performed by Bird et al., cannot support an "always" conclusion; the successful application of a sound formal method, however, is sufficient to show that a particular program always terminates. On the other hand, formal methods cannot be used to answer the research question on distributed development.

9.3 SE Research with Impact

The good news is that SE research and its research methods mature. Software design, a core area of software engineering, includes the development of Design Patterns, which has proven to be an influential component. Design Patterns were proposed in an OOPSLA workshop in 1987 by Beck and Cunningham. They took a small sample of GUI applications, written in Smalltalk, and examined their control flow. The authors determined that the code, in different applications, used particular control flow structures to achieve similar behavior. In essence, Beck and Cunningham were doing status-based research, but of course, they did not describe it as such. In 1994, the ***Gang of Four*** book on design patterns appeared, popularizing the idea. Status oriented research in design patterns exploded, capturing new domains like enterprise systems, security, and parallelism. At the same time, the research methods for design patterns broadened. Formal method researchers captured proof obligations for patterns; empirical researchers studied the effectiveness of design pattern usage in program construction. In the end, the accumulated confidence in the area resulted in widespread practical adoption; for instance, both ASP.NET and Ruby on Rails follow the MVC design pattern. Looking back at the immense research in Design Patterns, it becomes apparent that this line of research had an immense impact. However, the initial research was not very systematic. It did not make the research question explicit; it did not mention the method that it used, it did not provide much context, and it did not evaluate its findings. (For further research with impact, consult http://www.sigsoft.org/impact/)

9.4 The Quest for building a Research Method Portfolio

Providing a portfolio of research methods should enable SE researchers to engage in more research that has real impact. In more detail, it will establish guidelines to categorize methods according to the:

- Type of problem/question
- Type of desired result/answer
- Evaluation criteria
- Ground rules
- Condition criteria
- Costs

The last three sections of this paper describe in more detail three popular SE research methods along these axes: Empirical SE, Social Sciences, and Formal Methods. Each method description details how to improve the selection process.

This guideline provides a framework in order to more effectively perform and communicate SE research methods; it is not intended to impose rigid guidelines regarding method selection. An explicit description of the research portfolio will serve the field in various ways:

1. Establishing explicit methods guidelines will help researchers, especially students, to expeditiously plan and execute their projects
   - Improving the education of Ph.D. students
   - Appropriately pairing research methods to projects
2. Establishing explicit method selection criteria will improve scientific evaluation and review.
3. Clarifying questions and their anticipated answers may make results more relevant to industry.
4. Articulating an overview of the research portfolio will help guide investment in research methods and result in a more balanced portfolio.

9.5 Recommendations

Based on these findings the following approach is recommended:

- Begin building the description of our portfolio of research methods. This requires:
    - Participation from the software engineering industry
    - Guidance from a small steering committee to establish a consistent framework
    - Establishing incentives for participation
- Develop materials to teach the methods and the selection process. Describe the criteria used to identify appropriate research questions

The remaining three sections describe the future of Empirical SE, Social Sciences, and Formal Methods in more detail.

10. The Future of Empirical SE Research

One of the largest problems that we identified is that it is difficult to convince practitioners and fellow researchers of our results. Ultimately, the goal of software researchers is to improve the state of the practice in software engineering, but if practitioners mistrust the results, they will be unlikely to adopt new techniques or change their behaviors according to the findings. The following discussion suggests some reasons why they remain doubtful.

10.1 Asking the Right Questions and Providing Useful Answers

One of the complaints voiced by practitioners, funding agencies, and researchers alike is that the right questions are not asked. Prior to embarking on an empirical study or experiment, it is imperative to ensure that the answer to the question posed would actually be useful to practitioners and other researchers. Communication between research and practice is paramount and it must flow both ways. Are practitioners asked about their problems? Are studies in a context that practitioners would agree with? Further, how are empirical results conveyed – in the form of a paper, or are other channels employed? Additionally, sometimes the results of the research are too technical and only understandable by Ph.Ds. We must ensure not only that the results are clear, concise, and comprehensible by the typical software developer, but also that the result is actionable in some way.

10.2 Replication

The results of an empirical finding, when replicated, serve to confirm the finding and give credence to the outcome. One of the complaints of practitioners, when they read papers or are presented with findings, is that the context in which the study was performed may be very different from their own, making the finding difficult to relate to. One way to overcome this problem is by replicating important findings in our field. If an empirical result proves true across a variety of domains and processes, then it is probably a fairly general phenomenon. In addition, if a result is confirmed in some settings and not supported in others, that can improve our understanding of when to leverage the result.

This addresses one important aspect of replications in our field. When a study is replicated, the researchers should think carefully about what type or replication is performed and seek out a context that will complement the existing body of knowledge surrounding the empirical result. For example, a set of replications may be performing the same study on a family of products, where the process is held consistent but the projects themselves vary in size and purpose. In addition, current empirical papers may not be adequately addressing the contextual question. Empirical papers should describe the context of the study in detail so that consumers of the research can make informed decisions about what the result means to them and how it may apply to their own situation.

Unfortunately, it is generally believed by both paper writers and reviewers that replications do not provide high value because they are not considered novel. Although it may be a difficult and long-term goal, changing this thinking to value replications will encourage such studies. Creating incentives for researchers could spark a change in this attitude. Funding agencies should be encouraged to accept replication proposals, or researchers’ grant proposals should be required to include a section on the value of replication.

10.3 Data Sharing

Sharing of data and tools is a problem in our field because it can hinder or halt research. With regard to the point made above, sharing may enable other researchers to replicate studies on different systems and in different domains. Sharing of tools may lower entry barriers for new graduate students and even established researchers endeavoring a foray into the field. There are valid reasons that researchers do not share their tools or data today. It may represent a risk to one's future publication prospects; it may require additional effort to make data or tools appropriate for sharing; and in the corporate world, it may require dealing with competitive threats. We note that in other fields, the controversies surrounding sharing of data and tools deal with "when" to share rather than "whether" sharing should occur.

Clearly, researchers will share if there is a valid incentive. Funding agencies should require grant proposals to include a plan for sharing data and tools that result from the research being funded. In addition, since the process of data collection may not, of itself, be considered a research endeavor, funding agencies could create initiatives funding non-researchers to gather a corpus of data for the purpose of providing it to the research community at large. Such an effort would benefit the community by carefully selecting both the types of data and the range of software projects. In addition, much research activity would be focused on a small set of (carefully chosen) projects to provide a more comprehensive view of the interplay between different results (similar to what we observe about Eclipse today) There are other ways to incentivize researchers as well. For instance, the Mining Software Repositories (MSR) working conference extends the page limit by one page for authors who are willing to share. Although small, this is a first step in acknowledging the importance of this issue.

10.4 Research in the Large

There are few studies that examine non-trivial systems over time. Again, this may be because such research does not have the "aha" appeal required for publication in top venues. However, most research grants today do not provide enough financial support and do not last long enough for the types of large-scale, long-term baseline studies that would provide convincing evidence of trends in software production. Once our community becomes accepting of such work, researchers will embark on it. We need to investigate how to observe and measure creation and evolution of decisions, assumptions, and rationales at scale. In addition, techniques and mechanisms for assembling evidence should be standardized to determine the confidence level required to build a system, or to predict how well that system will meet its requirements.

To summarize, federally sponsored industry/academia experimental trials will be critical to understand the current state of the practice and the effects of trial improvements, and in determining whether the software engineering community is aware of the trial results.

10.5 Too Much Focus on Generalizability and Positive Results

The current thinking says that all findings must be generalized and that only positive results are useful. In reality, it is unlikely that many principles and hypotheses will be universally true for all software projects. In contrast, we must accept that negative results that do not generalize to all contexts are useful, provided that a) the question being answered actually matters and b) the context is provided in enough detail so that the combination of positive and negative results helps others understand the particular phenomenon under study. When preparing a research paper for submission, researchers need to be aware of their inconsistencies, acknowledge such results, and possibly even investigate the differences further. As a community, we also must be accepting of inconsistent results provided they contribute value to the body of knowledge.

10.6 Costs and Risks

Sharing data, replicating experiments, and performing long-term, large-scale studies are both time consuming and financially expensive. Now, more than ever, we need to evaluate and scrutinize the questions being asked so that resources are not wasted. Perhaps our community should provide a venue for researchers to submit large-scale research proposals in order to receive feedback to improve studies before they begin. This approach is not without risks because broadcasting research plans increases the possibility of plagiarism.

10.7 Evaluation

Metrics must be established to assess progress. The following list will help determine whether there is improvement in the state of research and practice in empirical software engineering:

- The creation of more substantial bodies of research surrounding ideas.

  When an important question is answered or a seminal result emerges, that result should both spark new, related questions and it should be replicated to the degree that researchers and practitioners can be convinced of the veracity of the findings.

- Better communication between practitioners and researchers (in both directions).

  We will know that we are having an effect when practitioners discuss their practice and problems and there is an increased bidirectional flow of ideas, information, and results between research and practice.

- Adoption of technology by practitioners that is validated by empirical studies.

  As research is both adopted and validated by industry, practitioners will begin to be convinced that improvement is underway.

- Improved software is developed faster and cheaper.

  Ultimately, the goal of all software development research is to improve developer productivity and software quality in terms of defects, security, fault tolerance, etc. If the state of practice improves on these fronts as a result of our research, then success can be declared.

11. The Future of Formal Methods research

Formal methods research applies logical and mathematical analysis to determine properties of software systems. Pioneering work in formal methods was concerned primarily with systematic proof ("formal verification") of properties of programs, typically treated in isolation from other software development activities, such as testing. Recent research in formal methods is now broader and better integrated, to such an extent that automated formal methods are increasingly hidden in tools for a variety of reasons, from performance analysis to test case generation.

A bedrock of formal methods research is establishing (often by formal proof) the mathematical and logical basis of the reasoning used to establish properties, so the evidence presented in formal methods research often includes proofs of soundness and a careful analysis of limitations (e.g., assumptions about a program under analysis that must be verified by other means). In addition, formal methods researchers present evidence of the relevance and practicality of the methods they develop, which may range from simple demonstrations of application in early stages to benchmarks, competitions, and larger case studies of techniques used in practical applications.

While formal methods research initially focused on a few critical properties, the current blossoming of formal methods research applies to a wide variety of analysis, wherever a well-defined logical property can be identified. Applications include a variety of design and specification notations in addition to program code, and include not only analysis per se, but exploration of the design space and synthesis of software with desired properties. The underlying logical and mathematical foundations of formal methods, together with careful analysis of the assumptions on which they depend, allow formal methods to produce particularly strong conclusions. Even when an analysis based on formal methods does not strictly adhere to its principles in order to render the analysis more useful in practice, the resulting assumptions can be verified in other ways.

Ease of use and transparency are key components of formal methods research for transitioning to mainstream software development. Making warnings and hints transparent and hiding formal support methods to improve tools routinely used by developers (e.g., test case generators) unburdens programmers. The greatest impact is achieved when a formal method, initially designed to check a property of a given software artifact, is able to explore the design space and produce an artifact with the desired property, or synthesize the artifact from a higher-level description.

Advances that make formal methods attractive and therefore used routinely by developers (sometimes without even knowing it) also impact education. Relevance of formal methods is apparent to students who, for example, view them as an aid to test case generation in test-driven development, or use them to quickly explore alternative design decisions.

11.1 Goals

Software engineering is at an inflection point; formal methods are producing practical tools that are perceived as useful by developers for many kinds of software, and no longer limited to critical systems or correctness properties. The goal should be to move towards routine use, including invisible analysis embedded in tools. Opportunities should be explored to design formal methods based on the analysis of given software artifacts to design exploration and synthesis of artifacts with desired properties.

11.2 Challenges

It is challenging to definitively define problems that are, on the one hand, clearly relevant to pertinent properties (not limited to correctness), and on the other, also amenable to efficient analysis. Early on in formal methods research, the focus was on critical properties and systems to achieve a very high level of assurance and justify a large expenditure of effort. A number of advances have enabled a shift in focus and the current renaissance of formal methods research. First, we should not underestimate the contribution of computational power, which has greatly expanded what is feasible. Algorithmic advances have multiplied the expansion of raw computational power, so that many analyses that would once have required an off-line, intensive analysis can be completed almost instantaneously, transforming the user experience of formal methods-based tools.

Partly owing to the limited domains of applicability of early formal methods research, early research methods were compartmentalized. Only recently have advanced technologies such as shared infrastructure, components, and representations allowed greater exploitation of techniques in a variety of applications, as well as more direct comparison of techniques leading to rapid improvements.

11.3 Potential Payoffs

The potential payoffs for a broad portfolio of formal methods research are potentially enormous. Beyond better assurance of a variety of properties, effective formal methods with tool support accelerate and leverage development efforts. This is evident as formal methods are incorporated into tools like test case generators, static performance analysis, and bug finders. Larger impacts, changing not only how efficiently and dependably we can create software systems but even what software systems we are capable of producing, are evidenced as some techniques move from analysis to synthesis and design exploration.

It would be risky to over-commit to one or two "silver bullet" formal methods. The risk is best controlled by investing in a broad portfolio of complementary techniques.

11.4 Timeliness

We are at the early stages of a renaissance in formal methods research. In addition to the algorithmic and hardware enablers noted above, common input languages (e.g., SMT-LIB), benchmarks, and competitions have greatly accelerated progress. It is important that future investments in formal methods research support not only a broad spectrum of approaches, but also interoperability and infrastructure development, so that researchers can incrementally build on each other's work, experiment with novel applications of (off-the-shelf) formal analyses, and understand the applicability and relative strengths and weaknesses of alternative techniques.

11.5 Evaluation

If we are successful, we should see widespread, routine use of formal methods by software developers – including both conscious use of formal techniques and "invisible" use through tools that incorporate formal methods. We expect formal methods-based techniques will be developed to improve the software development process directly and indirectly for faster time to market and higher-level debugging, creativity leveraged by rapid exploration of design alternatives, in addition to judicious use of formal techniques to improve dependability.

11.6 Progress through Research

It is an opportune time for investment in a broad spectrum of formal methods research, including a common infrastructure for interoperability, comparison, and reuse of formal methods-based tool components. Recent blossoming of the field has greatly increased the power and usability of formal methods techniques and fundamentally shifted the research from a narrow focus on dependability to supporting a wide variety of development activities. It has also transformed those activities as the underlying formal techniques become powerful, fast, and transparent enough to move from analysis of a manually produced artifact to aiding exploration and synthesis.

12. The Future of Social Sciences in SE Research

The focus of this section is to recommend improvements to software engineering research by learning from social sciences research which was selected because:

- Software engineering is largely a human activity. Although technology plays a critical role in software engineering, the technology is rapidly and continuously changing. Because human changes are evolutionary, studying the human can have a long-lasting impact.
- Social sciences (such as psychology, sociology, communication, and economics) are generally more mature disciplines with respect to studies involving humans. Thus, by learning from research methods from social sciences, we can jump-start our own studies involving humans.

However, learning from social sciences is not trivial for two main reasons. First, researchers cannot be expected to be experts in both software engineering and social science methods. Second, characterizing the methods used by social scientists is an enormous task because the fields of social sciences are so broad. The challenge is to identify how software engineering researchers can become aware of, find, and utilize research methods borrowed from social sciences.

12.1 Recommendations:

- The software engineering research community has made progress in the last few years in evaluating software engineering innovations, especially concerning human-based evaluations. However, too few human-based studies were performed early in the research process. Such formative evaluations can help the software engineering community solve more realistic problems and help uncover novel problems that were previously unknown. Thus, we recommend that more emphasis be placed on human-based formative studies.
- Qualitative research methods, while a staple of the social sciences, are not often used or understood in software engineering research. Advantages of qualitative research methods include generating new hypotheses and developing theories of cause and effect. Researchers should consider using qualitative methods throughout the research cycle.
- A fundamental principle of social research methods is managing variability between individuals. By contrast, software engineering research that use human subjects often use one type of individual but generalize to other individual types. For instance, experiments are often conducted on students but the results are generalized to include professionals; experiments run on open source developers are generalized to include closed-source developers. While some may view this as a problem, we recognize it as a necessary way to run human-centric studies in a cost-effective manner. The generalizations should be more systematically and predictably applied in order to be meaningful. Future studies should investigate the similarities and differences between individuals, such as between students and professional software engineers.
- The use and reuse of social science research results depends on the ability to generalize and reason about those results. However, this is difficult because of individual human differences. This problem can somewhat be alleviated by thoroughly justifying and explaining the context in which a study is performed. Software engineering research often tries to apologetically explain away context as “threats to validity.” Instead, future research should embrace, justify, and explain the context in which the study was performed.

- Because most software engineering researchers are not experts in social sciences, infrastructure is needed to help such researchers take advantage of the accumulated knowledge in this area. The reviewing process is a viable, concrete area to examine; how can a reviewer judge the validity of a human-based evaluation? Reviewers' ignorance of social science methods often results in faulty heuristics, such as rejecting studies with fewer than 10 people (this is faulty because there are good studies with 2 people and poor studies with thousands of people). One way to combat this is to give reviewers checklists based on what the software engineering community believes, as a whole, constitutes a good study. For example, we ask reviewers to verify that "the study methodology used matches the authors' claims." Future software engineering reviewers should be given easily accessible resources for evaluating studies using social science research methods.

13. The NITRD Program

The Networking and Information Technology Research and Development (NITRD) Program is the Nation's primary source of Federally funded revolutionary breakthroughs in advanced information technologies such as computing, networking, and software.

A unique collaboration of Federal research and development agencies, the NITRD Program seeks to:

- Provide research and development foundations for assuring continued U.S. technological leadership in advanced networking, computing systems, software, and associated information technologies
- Provide research and development foundations for meeting the needs of the Federal government for advanced networking, computing systems, software, and associated information technologies
- Accelerate development and deployment of these technologies in order to maintain world leadership in science and engineering; enhance national defense and national and homeland security; improve U.S. productivity and competitiveness and promote long-term economic growth; improve the health of the U.S. citizenry; protect the environment; improve education, training, and lifelong learning; and improve the quality of life.

Federal IT R&D, which launched and fueled the digital revolution, continues to drive innovation in scientific research, national security, communication, and commerce to sustain U.S. technological leadership. The NITRD agencies' collaborative efforts increase the overall effectiveness and productivity of these Federal R&D investments, leveraging strengths, avoiding duplication, and increasing interoperability of R&D products.

13.1.1 The NITRD Program focuses on the following research areas:

- Big Data (BD)
- Cyber Security and Information Assurance (CSIA)
- Health Information Technology Research and Development (Health IT R&D)
- Human Computer Interaction and Information Management (HCI&IM)
- High Confidence Software and Systems (HCSS)
- High End Computing (HEC)
- Large Scale Networking (LSN)
- Software Design and Productivity (SDP)
- Social, Economic, and Workforce Implications of IT and IT Workforce Development (SEW)
- Wireless Spectrum Research and Development (WSRD)

14. Appendix A - List of Attendees

Al-Ghuwairi	Abdel-Rahman	New Mexico State Univ.	aghuwair@cs.nmsu.edu
Allaho	Mohammad	Pennsylvania State Univ.	mya111@psu.edu
Arnold	Kenneth	MIT Media Lab	kcarnold@alum.mit.edu
Atlee	Joanne	University of Waterloo	jmatlee@uwaterloo.ca
Avrunin	George	University of Massachusetts	avrunin@math.umass.edu
Bae	Gigon	KAIST	ggbae@se.kaist.ac.kr
Bagheri	Hamid	University of Virginia	hb2j@Virginia.EDU
Balzer	Robert	Teknowledge	balzer@teknowledge.com
Baquero	Alegria	University of California	abaquero@uci.edu
Baresi	Luciano	Politecnico di Milano	baresi@elet.polimi.it
Begel	Andrew	Microsoft Research	andrew.begel@microsoft.com
Bird	Christian	Microsoft Research	cbird@microsoft.com
Boehm	Barry	University of Southern California	boehm@usc.edu
Bronish	Derek	Ohio State University	derekbronish@gmail.com
Brooks	Ruven	University of Wisconsin	RuvenBrooks@att.net
Bruch	Marcel	Darmstadt University of Tech.	bruch@cs.tu-darmstadt.de
Brun	Yuriy	University of Washington	brun@cs.washington.edu
Bryant	Barrett	University of Alabama	bryant@cis.uab.edu
Bultan	Tevfik	University of California	bultan@cs.ucsb.edu
Burnett	Margaret	Oregon State University	burnett@eecs.oregonstate.edu
Buse	Raymond	University of Virginia	buse@cs.virginia.edu
Cabral	Isis	University of Nebraska	isis.cabral@gmail.com
Cadar	Cristian	Imperial College	c.cadar@imperial.ac.uk

Cantu	Joe	ESC/HIG	joe.cantu@randolph.af.mil
Carter	Angela	NCO/NITRD	carter@nitrd.gov
Chechik	Marsha	University of Toronto	chechik@cs.toronto.edu
Cho	Hyunsik	KAIST	hscho@se.kaist.ac.kr
Choi	Jinho	KAIST	jhchoi@se.kaist.ac.kr
Clarke	Lori	University of Massachusetts	clarke@cs.umass.edu
Clause	James	University of Delaware	clause@udel.edu
Clune	Thomas	NASA GSFC	Thomas.L.Clune@nasa.gov
Damian	Daniela	University of Victoria	damian.daniela@gmail.com
Dang	Zhe	Washington State University	zdang@eecs.wsu.edu
D'Ippolito	NicolÃ¡s	Imperial College London	srdipi@doc.ic.ac.uk
Drager	Steven	Air Force Research Lab	steven.drager@rl.af.mil
Dwyer	Matthew	University of Nebraska	dwyer@cse.unl.edu
Easterbrook	Steve	University of Toronto	sme@cs.toronto.edu
Ebnenasir	Ali	Michigan Technological Univ.	aebnenas@mtu.edu
Elbaum	Sebastian	University of Nebraska	elbaum@cse.unl.edu
Erwig	Martin	Oregon State University	erwig@eecs.oregonstate.edu
Esfahani	Naeem	George Mason University	nesfaha2@gmu.edu
Fisler	Kathi	WPI	kfisler@cs.wpi.edu
Forrest	Stephanie	University of New Mexico	dcosper@cs.unm.edu
Fu	Frank	University of Iowa	fermat1217@gmail.com
Gabriel	Richard	IBM Research	rpg@dreamsongs.com
Garlan	David	Carnegie Mellon University	garlan@cs.cmu.edu
Gethers	Malcom	College Of William & Mary	mgethers@cs.wm.edu
Goodenough	John	SEI/CMU	jbg@sei.cmu.edu

Gopalakrishnan	Ganesh	University of Utah	ganesh@cs.utah.edu
Gray	Jeff	University of Alabama	gray@cs.ua.edu
Grechanik	Mark	Accenture Technology Labs	mark.grechanik@accenture.com
Greenspan	Sol	National Science Foundation	sgreensp@nsf.gov
Griswold	William	University of California	wgg@cs.ucsd.edu
Harmon	Trevor	NASA Ames Research Ctr.	trevor.w.harmon@nasa.gov
Hemmati	Hadi	Simula Research Laboratory	hemmati@simula.no
Hoebel	Louis	GE Research	hoebel@ge.com
Holloway	Seth	University of Texas	sethh@mail.utexas.edu
Holmes	Reid	University of Waterloo	rtholmes@cs.uwaterloo.ca
Holotescu	Oana Iulia Casandra	Politecnica Univ. of Timisoara	casandra@cs.upt.ro
Huang	Jeff	HKUST	smhuang@ust.hk
Inverardi	Paola	Universita' Dell'aquila	paola.inverardi@univaq.it
Johnson	Ralph	University of Illinois	rjohnson@illinois.edu
Jones	James	University of California	jajones@ics.uci.edu
Julien	Christine	University of Texas	c.julien@mail.utexas.edu
Kajko-Mattsson	Mira	KTH	mira@dsv.su.se
Kang	Eunsuk	MIT	eskang@csail.mit.edu
Kazman	Rick	SEI/CMU & U. of Hawaii	kazman@hawaii.edu
Kim	Miryung	University of Texas	miryung@ece.utexas.edu
Kim	Sunghun	HKUST	hunkim@gmail.com
Kimmell	Garrin	University of Iowa	gkimmell@cs.uiowa.edu
Kirby	James	Naval Research Laboratory	james.kirby@nrl.navy.mil
Kishida	Kouichi	SRA	k2@sra.co.jp
Klein	Mark	SEI / CMU	mk@sei.cmu.edu

Koerner	Sven	Karlsruhe Institute of Tech	sven.koerner@kit.edu
Krishnamurthi	Shriram	Brown University	sk@cs.brown.edu
Krka	Ivo	Univ. of Southern California	krka@usc.edu
Kulkarni	Vinay	Tata Consultancy Services	vinay.vkulkarni@tcs.com
Lahiri	Shuvendu	Microsoft Research	shuvendu@microsoft.com
Le	Wei	University of Virginia	weile@virginia.edu
Le Goues	Claire	University of Virginia	legoues@cs.virginia.edu
Leavens	Gary	Univ. of Central Florida	leavens@eecs.ucf.edu
Lee	Seulki	KAIST	leesk@se.kaist.ac.kr
Liu	Yan	Pacific Northwest National Lab.	yan.liu@pnl.gov
Lopez	Nicolas	University of California	nlopezgi@uci.edu
Lowry	Michael	NASA Ames	Michael.R.Lowry@nasa.gov
Lucier	Ernest	NCO/NITRD	lucier@nitrd.gov
Luginbuhl	David	Air Force Off. of Scientific Res.	david.luginbuhl@afosr.af.mil
Lutz	Robyn	Iowa State Univ./JPL	rlutz@cs.iastate.edu
Marcus	Andrian	Wayne State University	amarcus@wayne.edu
Massacci	Fabio	University of Trento	angeli@disi.unitn.it
Mcmillan	Collin	College Of William & Mary	cmc@cs.wm.edu
Mikkonen	Tommi	Tampere U of Tech	tommi.mikkonen@tut.fi
Mockus	Audris	Avaya Labs Research	audris@avaya.com
Moore	Michael	ESC/HIG	michael.moore@randolph.af.mil
Munson	Ethan	University of Wisconsin	munson@uwm.edu
Murphy	Gail	University of British Columbia	murphy@cs.ubc.ca
Murphy-Hill	Emerson	North Carolina State Univ.	emerson@csc.ncsu.edu

Nakakoji	Kumiyo	Software Research Associates, Inc.	kumiyo@sra.co.jp
Nguyen	Hien	New Mexico State Univ.	hinguyen@cs.nmsu.edu
Nguyen	ThanhVu	University of New Mexico	nguyenthanhvuh@gmail.com
Niu	Nan	Mississippi State University	niu@cse.msstate.edu
Northrop	Linda	SEI / CMU	lmn@sei.cmu.edu
Notaro	Robert	Law School Admission Council	rnotaro@lsac.org
Notkin	David	University of Washington	notkin@cs.washington.edu
Nuseibeh	Bashar	Lero/The Open University	Bashar.Nuseibeh@lero.ie
Orso	Alessandro	Georgia Institute of Tech.	orso@cc.gatech.edu
Osterweil	Leon J.	University of Massachusetts	ljo@cs.umass.edu
Ostrand	Thomas	AT&T Labs	ostrand@research.att.com
Ozkaya	Ipek	SEI / CMU	ozkaya@sei.cmu.edu
Park	Sangmin	Georgia Tech	sangminp@cc.gatech.edu
Payton	Jamie	University of North Carolina	payton@uncc.edu
Perino	Nicolo	University of Lugano	nicolo.perino@usi.ch
Picco	Gian Pietro	University of Trento	gianpietro.picco@unitn.it
Podgurski	Andy	Case Western Reserve Univ.	podgurski@case.edu
Porter	Adam	University of Maryland	aporter@cs.umd.edu
Poshyvanyk	Denys	College Of William & Mary	denys@cs.wm.edu
Posnett	Daryl	University of California	dpposnett@ucdavis.edu
Rajan	Hridesh	Iowa State University	hridesh@iastate.edu
Reiss	Steven	Brown University	spr@cs.brown.edu
Richardson	Debra	University of California	djr@uci.edu
Riche	Taylor	University of Texas	riche@cs.utexas.edu
Roach	Steve	UTEP	sroach@utep.edu

Roman	Catalin	Washington University	roman@wustl.edu
Rosenblum	David	University College	d.rosenblum@cs.ucl.ac.uk
Sadowski	Caitlin	University of California	supertri@cs.ucsc.edu
Sangwan	Raghvinder	Penn State University	rsangwan@psu.edu
Sarma	Anita	University of Nebraska	asarma@cse.unl.edu
Scacchi	Walt	University Of California	wscacchi@ics.uci.edu
Schaefer	Wilhelm	University of Paderborn	wilhelm@uni-paderborn.de
Schiller	Todd	University of Washington	tws@cs.washington.edu
Schulte	Wolfram	Microsoft	schulte@microsoft.com
Schulte	Eric	University of New Mexico	schulte.eric@gmail.com
Schumann	Johann	SGT Inc, NASA Ames	Johann.M.Schumann@nasa.gov
Shaw	Mary	Carnegie Mellon University	kari@cs.cmu.edu
Shewmaker	Andrew	UC Santa Cruz	shewa@soe.ucsc.edu
Siami Namin	Akbar	Texas Tech University	akbar.namin@ttu.edu
Sillito	Jonathan	University of Calgary	sillito@ucalgary.ca
Smith	Douglas	Kestrel Institute	smith@kestrel.edu
Sridharan	Manu	IBM T. J. Watson Research Ctr	msridhar@us.ibm.com
Stanley	Joan	NCO/NITRD	stanley@nitrd.gov
Stauts	Matthew	University of Minnesota	stauts@cs.umn.edu
Strasser	Kyle	University of California	kstrasse@uci.edu
Stump	Aaron	University of Iowa	astump@acm.org
Sullivan	Kevin	University of Virginia	sullivan@cs.virginia.edu
Sun	Yu	University of Alabama at Birmingham	yusun@uab.edu
Taivalsaari	Antero	Nokia	antero.taivalsaari@nokia.com
Tamburrelli	Giordano	Politecnico Di Milano	tamburrelli@elet.polimi.it

Taylor	Richard	University of California	taylor@ics.uci.edu
Thummalapenta	Suresh	North Carolina State University	sthumma@ncsu.edu
Tichy	Walter	Karlsruhe Institute of Tech.	tichy@kit.edu
Tracz	William J	Lockheed Martin IS&GS	will.tracz@lmco.com
Treude	Christoph	University of Victoria	ctreude@gmail.com
Turner	Hamilton	Virginia Tech	hamiltont@gmail.com
Vajda	Andras	Ericsson	andras.vajda@ericsson.com
van der Hoek	Andre	University of California	andre@ics.uci.edu
Venet	Arnaud	CMU / NASA Ames Res. Ctr	arnaud.j.venet@nasa.gov
Visser	Willem	Stellenbosch University	willem@gmail.com
Walkingshaw	Eric	Oregon State University	walkiner@eecs.oregonstate.edu
Wang	Chao	NEC Labs	chaowang@nec-labs.com
Wang	Xiaoyin	Peking University	wangxy06@sei.pku.edu.cn
Wasserman	Anthony	Carnegie Mellon Silicon Valley	tonyw@sv.cmu.edu
Weber-Jahnke	Jens	University of Victoria	jens@uvic.ca
Weiss	David	Iowa State University	weiss@cs.iastate.edu
Weyuker	Elaine	AT&T Labs - Research	weyuker@research.att.com
Wilson	Justin	Washington University	jrwilson@go.wustl.edu
Wing	Michael	Critterscape	wing@swcp.com
Wolf	Alexander	Imperial College	a.wolf@imperial.ac.uk
Wolff	Roger	Carnegie Mellon University	roger.e.wolff@gmail.com
Xie	Tao	North Carolina State Univ.	taoxie@gmail.com
Xu	Guoqing	Ohio State University	xug@cse.ohio-state.edu
Ye	Yunwen	SRA Key Technology Lab	ye@sra.co.jp
Young	Michal	University of Oregon	michal@cs.uoregon.edu

Zervoudakis	Fokion	University College	f.zervoudakis@cs.ucl.ac.uk
Zheng	Yongjie	University of California	zhengy@ics.uci.edu
Zhou	Minghui	Peking University	zhmh@sei.pku.edu.cn
Zimmermann	Thomas	Microsoft Research	tzimmer@microsoft.com
Ziv	Hadar	University of California	ziv@ics.uci.edu

15. Appendix B – Acknowledgements

The FoSER workshop was collaboratively organized by the U.S. National Coordination Office (NCO) for Networking and Information Technology Research and Development (NITRD) Software Design and Productivity (SDP) Coordinating Group, in cooperation with the organizers of the ACM SIGSOFT 18th Symposium on the Foundations of Software Engineering (FSE-18) and ACM SIGSOFT. Funding from the NITRD NCO and the National Science Foundation (NSF) made this workshop possible. This workshop would not have been possible without the invaluable contributions of the FoSER 2010 Executive Committee (EC) and Program Committee (PC). The EC provided advice and consent on all key decisions about the organization of this workshop, and EC members also served on the PC. Serving on the EC were Kevin Sullivan as Workshop Committee Chair, University of Virginia; Joanne Atlee, University of Waterloo; Sol Greenspan, National Science Foundation; David Notkin, University of Washington; Gruia-Catlain Roman, Washington University in St. Louis; and Wolfram Schulte, Microsoft Research. The PC also included Margaret Burnett, Oregon State University; Richard Gabriel, IBM Research; Carlo Ghezzi, Politecnico di Milano; Gregor Kiczales, University of British Columbia; Mike Lowry, NASA Ames Research Center; Robyn Lutz, Iowa State University; Greg Morrisett, Harvard University; Linda Northrop, Carnegie Mellon Software Engineering Institute; Bashar Nuseibeh, The Open University; David Weiss, Iowa State University; and Laurie Williams, North Carolina State University. Finally, special thanks and recognition go to Gruia-Catalin Roman, general chair of FSE 18, whose vision and commitment to do something memorable and especially valuable with the FSE-18 workshop program made FoSER 2010 possible, and to Kevin Sullivan, who planned and organized FoSER 2010 in response to this challenge.

16. Appendix C - Abbreviations and Acronyms

ACM - Association of Computing Machinery (ACM)

API - Application Programming Interface (API)

ASP.NET - a web application framework developed and marketed by Microsoft

CoDPOM - Context-Driven Process Orchestration Method

ESEM - Empirical Software Engineering and Measurement

EUP - end-user programming (EUP)

FoSER - Future of Software Engineering Research (FoSER)

FOSS - Free and Open Source Software (FOSS)

FSE - Foundations of Software Engineering (FSE)

Github - Free public repositories, collaborator management, issue tracking, wikis, downloads, code review, graphs

GNU - “GNU's Not Unix”

GoF - Gang of Four (authors of the Design Patterns)

GUI - Graphical User Interface

HCI – Human Computer Interface

IDEs - Integrated Development Environments

MSDN - Microsoft Developer Network

MSR - Mining Software Repositories

MVC - Model-View-Controller is a fundamental design pattern for the separation of user interface logic from business logic

NLP - Natural language processing (NLP)

NRC - National Research Council (NRC)

OOPSLA - Object-Oriented Programming, Systems, Languages, and Applications

PCA - Program Component Area

PCAST - President's Council of Advisors on Science and Technology (PCAST)

PROMISE - **PR**edict**O**r **M**odels **I**n **S**oftware **E**ngineering

ROI - return on investment (ROI)

RSSE - Recommendation Systems for Software Engineering

Ruby on Rails - open source web application framework for the Ruby programming language

SDP CG - Software Design and Productivity (SDP) Coordinating Group (CG)

SE - Software Engineering (SE)

SIGSOFT - Association of Computing Machinery (ACM)
Special Interest Group in Software Engineering http://www.sigsoft.org/impact/

SMT-LIB - The Satisfiability Modulo Theories Library

SourceForge - Find, Create, and Publish Open Source software for free

SSBSE - Symposium on Search Based Software Engineering

SUITE - Search-Driven Development – Users, Infrastructure, Tools and Evaluation

UIs - user interfaces (UIs)

ULS - Ultra-Large-Scale (ULS) Systems

V&V - Verification and Validation (V&V)