# New Visions for Software Design & Productivity: Research & Applications

Workshop of the
Interagency Working Group for Information Technology Research and Development (ITRD)
Software Design and Productivity (SDP) Coordinating Group

December 13 - 14, 2001
Vanderbilt University
Nashville, TN

## Workshop Report

**Edited by:**

**Adam Porter**
**Janos Sztipanovits**

**NSF Grant CCR-0138554**

**Supported by:**

National Science Foundation

Vanderbilt University

**New Visions for Software Design & Productivity: Research & Applications**

Workshop of the Interagency Working Group for Information Technology Research and Development (ITRD) Software Design and Productivity (SDP) Coordinating Group
December 13 - 14, 2001
Vanderbilt University
Nashville, TN

## Foreword

The late eighties and nineties were the "Big Bang" of IT. Forged from decades of prior R&D investment and triggered by research breakthroughs in computing hardware, software, and networking, we witnessed a tremendous expansion of IT during the past dozen years. The resulting "IT explosion" proceeded simultaneously in many directions.

The effects of this process are profound:

- The most exciting developments occur at the "wave-front", i.e., at the intersection of IT with other areas. There are many examples for new science and technology fields, such as bioinformatics, nanotechnology, computer-assisted learning, embedded software and real-time systems, and many others that have yielded new disciplines and have the potential to jumpstart important new industries.
- IT is becoming so pervasive that the classical structure of IT research and industry is changing drastically. For example, the tight integration of physical and information processes in embedded systems requires the development of a new systems science, which is simultaneously computational and physical. These advances will ultimately require new educational approaches and project management structures representing radical departures from existing models.

At the heart of the IT-driven transformation of our economy is high-quality software for complex computing and information systems. At stake is the very success and future progress of our technological dominance in the world. Given the difficulties of building quality software quickly and efficiently, the President's Information Technology Advisory Committee (PITAC) recommended that the U.S. "Make fundamental software research an absolute priority." Carrying through with those recommendations by establishing a substantial research initiative on software design and productivity is therefore critical for the security and continued economic viability of the U.S.

The workshop on "New Visions for Software Design and Productivity" provided a forum for scientists, engineers, and users to identify revolutionary thinking about software development techniques that could dramatically increase software productivity without compromising software quality. This workshop was the second event initiated by the Software Design and Productivity (SDP) Coordinating Group of the Interagency Working Group for Information Technology Research and Development (ITRD). The planning workshop was held in Arlington, VA on April 18-19, 2001. The second SDP workshop built upon and expanded the results and insights gained at the planning workshop. Invitations to the meeting were issued based on evaluating White Papers that were received in response to a Call for Position Papers distributed to leading members of the research and development communities. Workshop participants included 64 invited researchers from industry and academia and 14 government researchers. Full material on the SDP workshop is available on http://www.itrd.gov/iwg/pca/sdp/sdp-workshops/Vanderbilt.

*Workshop Co-Chairs:*

Dr. Janos Sztipanovits
Vanderbilt University

Dr. Adam Porter
University of Maryland

*SDP Co-Chairs:*

Dr. Frank Anger
NSF

Dr. Douglas C. Schmidt
DARPA

## Contents

## Executive Summary

The goals of the SDP workshop on "New Visions for Software Design and Productivity" were to:

- Bring together leading-edge researchers and practitioners
- Encourage brainstorming and out-of-box thinking
- Inform the Federal research agenda
- Involve Federal agencies and research community

Of particular interest were technology needs and promising solutions that could revolutionize the way we design and produce software in the coming decades, but that are currently beyond the scope of today's time-to-market and profit-driven research and development (R&D) programs. The SDP workshop included panel discussions, breakout sessions, and plenary discussions. The panels and breakout sessions addressed the following four issues central to software design and productivity:

*1. The Future of Software and Software Research*

Software's rapid penetration throughout all sectors of the economy challenges both the fundamentals underlying software research and how that research is conducted. Today, it's estimated that at least 50 percent of large software projects fail. Moreover, even though new and old software applications often cannot work together seamlessly, investments in legacy software cannot be abandoned. Clearly, new ideas are needed to help us move beyond this situation. For this reason, the workshop included discussions of the following:

- What should software and software development teams look like in the future?
- What should the role of programming languages be?
- How should the software research community evolve, e.g., should it move to more empirical foundations, focus on support tools, divide into multiple domain-based paradigms, develop whole new abstractions, or go in entirely new directions?
- How should we train future software practitioners, who range from scientists to programmers to end-users?

*2. New Software Development Paradigms*

Software is increasingly the universal integrator for large-scale systems, which themselves are network-centric "systems of systems." Paradigms are needed that include careful engineering processes and systematic validation methods. Questions related to these issues that were addressed include:

- In what ways and contexts should software development become more dynamic and fluid versus more rigid and rule-driven?
- What should be the balance between formal and informal methods, engineering and artistry, evolution and rebuild, correct-by-construction and correct-by-consensus?
- What will be the role of open standards and open-source software development, end-user programming, and other radically different development models?
- What effect will legal and societal demands have on software development, testing, and certification paradigms?
- How will distributed and collaborative development environments impact the design, productivity, and quality of software?

*3. Software for the Real World*

Well over 90 percent of all microprocessors are now used for embedded systems. The characteristics of many of these embedded systems are constrained by the physical world. We need principled methods for specifying, programming, composing, integrating, validating software for embedded systems while enforcing the physical constraints and satisfying conventional functional requirements. Questions related to these issues that were addressed include:

- How can a myriad of real-world physical constraints be integrated with functional constraints when designing software?
- How can hardware/software co-design contribute to a solution?
- Models and generators are possible solutions at many levels of software design, but what are other promising approaches?
- How can we handle the legacy systems (and legacy developers) that constitute a substantial portion of today's world?

*4. Software for Large-scale Network-Centric Systems*

Next-generation applications will increasingly be run on networks, posing hard configuration and workload challenges, including latency hiding, partial failure, causal ordering, dynamic service partitioning, and distributed deadlock avoidance. To address these challenges, we need techniques for end-to-end quality of service frameworks, understanding multi-level distributed resource management, and adaptive middleware architectures. Questions related to these issues that were addressed include:

- What are the fundamental design challenges to be solved in this emerging network-centric world?
- How can we effectively specify and enforce end-to-end quality of service requirements across heterogeneous networks, operating systems, and applications?
- What analytical methods and tools are needed to design and implement mission-critical, continuously operational systems?
- When risk management is part of the system, how will that affect software products?

Invited panelists discussed the questions above during the first day plenary sessions. The panel discussions were followed by breakout sessions led by the panel chairs. The breakout discussions allowed all attendees to provide input for the SDP workshop recommendations. The session chairs presented the final recommendations from the breakout sessions during the closing plenary session.

The full documentation of the SDP workshop includes (a) the Executive Summary), (b) Breakout Group Summaries, (c) Outbriefing Presentations of the session chairs, (d) Presentations of the panelists, and (e) Position Statements of the participants. The documentation is accessible on the workshop web site: http://www.itrd.gov/iwg/pca/sdp/sdp-workshops/Vanderbilt

Summarized below are the main findings and recommendations from the SDP workshop.

## Where Are We Now?

There is an inevitable and continuous "software crisis" since the inception of the software field in the mid-1960s. This is due to the fact that end-user demands inevitably exceed technology advances. Software research and better engineering practices clearly have resulted in tremendous productivity increases during the past decades, particularly in business applications. In fact, at this point we often take for granted how pervasive software is in our everyday world: from home appliances, automobiles, streetlights, office security systems, business and personal communications, to banking functions, medical instruments, emergency services, power distribution, scientific discovery, national security, and warfare systems software now pervades our lives.

Ironically, some claim that not much progress has occurred over the past quarter century since today we still cannot reliably build dependable, high-quality, affordable complex software successfully. This critique, however, misses a fundamental point: *we build vastly different systems today that are orders of magnitude more complex than those of even a decade ago*. Since IT is driven by escalating user needs, software research is a continuous struggle to expand the complexity limit for the systems we build. The software R&D community can take credit for advancing the state-of-the-practice in the following areas:

- Due to advances in modern development practices, such as model-based code generation, automated testing suites, higher-level programming languages, and reusable patterns and object-oriented tools, small teams can now create high quality software systems up to 100,000+ LOC (Line Of Code) in a fraction of the time it used to take. As recently as ten years ago it was considered a daunting task to develop software at this scale.
- Comprehensive commercial application frameworks, such as web browsers, user interface generators, and the standard Java class libraries, combined with plug-and-play component technology, such as the CORBA Component Model and J2EE (Java 2 Platform Enterprise Edition) and .NET web services, have enabled huge productivity increases in business desktop and enterprise applications, such as data warehousing, enterprise resource planning, and e-commerce websites.
- Quality-of-service (QoS) support in distributed object computing middleware, such as Real-time CORBA, is beginning to have significant impact on design productivity and dependable end-to-end QoS enforcement for distributed real-time and embedded applications.
- There are some highly successful examples for end-user programmable applications, application-specific model-based front-ends, and program generators that provide strong motivation to proceed in these directions.
- Maturation and adoption of software lifecycle process models are improving the quality in a range of projects. For example, standard processes, such as the Rational Unified Process and eXtreme Programming, can predictably reduce the development time and cost, and increase the quality of large-scale and smaller-scale software systems.

## Future Challenges

Despite periodic economic downturns driven by business cycles, it is clear that IT will continue to gain momentum. The exponential expansion driven by the relentless progress in processor and networking technology is predicted to continue well into the next decade. Most importantly, the profound impacts of IT on application domains and global competitiveness will continue to expand its pervasiveness and transformational effect. The panels and breakout sessions at the SDP workshop identified the following emerging new application challenges that will require further major advances in software technology:

- **System integration**. Perhaps the biggest impact of the “IT explosion” in the last decade has been the emerging role of computing and software as the “universal system integrator.” The potential impact of this on software technologies is twofold. On the one hand, there is an ever-tighter fusion of computing and software with application domains. On the other hand, there is a strong divergence in software technologies, which is becoming richer with each new application direction.

- **Critical infrastructure role.** Large-scale software systems increasingly serve as critical infrastructure for banking functions, medical instruments, emergency services, power distribution, telecommunications, transportation and national security and defense. Since much of this infrastructure can never be shut down entirely, we must devise systems that can monitor and repair themselves and evolve continuously without disruption.
- **Real-time and embedded systems.** A rapidly growing part of applications, such as pacemakers, controllers for power plants, and flight-critical avionics systems, embed intelligence in physical devices and systems. Since these applications are inextricably connected to the physical environment, they must be designed to satisfy physical demands and limitations, such as dynamics, noise, power consumption, and physical size, in a timely manner.
- **Dynamically changing distributed and mobile applications.** As connectivity among computers and between computers and physical devices proliferates, our systems have become network-centric. Distributed network-centric applications are dynamic, continuously change their topologies and adapt their functionality and interaction patterns in response to changes in their environment.

## New Research Directions

Given the strong divergence of major application domains with many unique challenges, it is not surprising that no single technology silver bullet exists. Still, the diverse challenges created by new types of applications—along with the universal demand to push the complexity limit—define the following core research goals to be addressed by the IT R&D community in the next decade:

- **Multi-faceted programming.** A general theme that surfaced throughout the SDP workshop is the need to expand composition from today's hierarchical, modular composition to multi-faceted composition. The increased fusion and deep integration of application domains with computing implies that essential characteristics of systems are strongly influenced - or simply determined - by the software. Consequently, software requirements become multi-faceted, i.e., computation and software architectures must satisfy many functional and physical requirements *simultaneously*. The goal of multi-faceted program composition is separation of concerns in design by enabling the simultaneous use and management of multiple design aspects. The primary challenge in multi-faceted composition is the explicit representation of the interdependence among different design aspects, such as dependability, scalability, efficiency, security, and flexibility, and its use for the automated composition of systems that satisfy different objectives in different contexts. Multi-faceted composition can be and must be integrated with development paradigms working at different levels of abstractions, such as procedural and object-oriented languages or declarative modeling languages.
- **Model-based software development.** Discussions at the workshop have shown wide agreement among participants that source code alone is inadequate for documenting and managing design and maintenance processes. An important research goal is to integrate the use of high-level, domain-specific abstractions into the development process. These high-level, domain-specific modeling languages must be formal enough to be used directly for analysis of designs and for software generation. In addition, tools are needed to facilitate developing and associating models post hoc with large quantities of software generated without them. Model-based software development technologies should serve as a foundation for creating systems that utilize their own models to provide a wide range of new capabilities such as self-monitoring, self-healing, self-adaptation and self-optimization.
- **Composable and customizable frameworks.** Software applications have historically been developed as monolithic implementations of functionality that are hard to understand, maintain, and extend. An important technology that has emerged to alleviate these problems is *object-oriented frameworks*, which contain reusable components that can be composed and specialized to produce custom applications. Frameworks help to reduce the cost and improve

the quality of application software by reifying proven designs and patterns into concrete source code that enables larger scale reuse of software than can be achieved by reusing individual classes or stand-alone functions. Significant extension and improvement of framework-based approaches has emerged as another important research goal. A central research challenge is to devise tools and techniques that can refactor key application domains, such as telecommunications, e-commerce, health care, process automation, or avionics, into reusable frameworks.

- **Intelligent, robust middleware**. Participants, focusing on large-scale, network-centric systems agreed that to handle the complexity of this system category, we need to develop and validate a new generation of intelligent middleware technologies that can adapt dependably in response to dynamically changing conditions for the purpose of always utilizing the available computer and network infrastructure to the highest degree possible in support of system needs. This new generation of middleware is software whose functional and QoS-related properties can be modified either statically, (e.g., to reduce footprint, leverage capabilities that exist in specific platforms, enable functional subsetting, and minimize hardware/software infrastructure dependencies) or dynamically (e.g., to optimize system responses to changing environments or requirements, such as changing component interconnections, power-levels, CPU/network bandwidth, latency/jitter; and dependability needs).
- **Design of networked embedded systems.** One of the fundamental trends in IT is the increasing impact of embedded computing[1]. Software developments for these systems are more complex due to their physicality. Participants, focusing on programming for the "real world" have identified a rich research agenda to answer challenges in this fundamentally important system category. Identified research goals include extended use of models in the development process, new execution frameworks that support self adaptation and provide guarantees to prevent unsafe behavior, and a deeper understanding on managing the effects of resource limitations by developing lightweight protocols, power constrained services and fault tolerant architectures.
- **Collaborative software development.** As the economics underlying the software marketplace are changing, companies reorganize themselves, redefine their business goals, and develop software differently. Tools and paradigms are needed to support (1) collaborative software development by multi-disciplinary, geographically-distributed development teams; (2) software development teams that will routinely be distributed across time zones and will be comprised of end-users and subject-matter experts (e.g., biology, psychology, medicine, and finance); (3) development processes that help groups strengthen domain and business models and pursue multi-faceted design and value-based development; and (4) support the extension and continuity of these collaborations across the full spectrum of the software lifecycle, from early requirements gathering through post-release maintenance.
- **System/software co-design environments.** As increasing number of components and interactions in our real-world systems become *computational*, system and software design become inextricably combined. Increased design productivity requires the development of new science base for system/software co-design, a wide range of system and software analysis and synthesis tools, and technologies for composing domain-specific system/software co-design environments. In addition, the new design environments must enable software/system developers to quickly change software, place it into the field or into simulators, experiment with the updated software and then repeat the process.

## Concluding Remarks

The ultimate goal of software development is to create quality systems with an acceptable expenditure of time and effort. Software research, which is restricted by cost limitations to remain purely theoretical, i.e. not based on experimental data and not verified in real-life context, will

[1] D. Estrin (ed), *Embedded Everywhere*, NRC Study, 2001

lack scientific credibility and will be considered speculative by practitioners. We need to restructure our national software research strategy to incorporate fully a strong experimental component. Software researchers must be allowed to evaluate their research in realistic settings and apply it to more complex systems. They must also pay more attention to systems that involve people and do more in-depth studies of existing and proposed large-scale systems to understand how and why various techniques behave in various ways.

The last decade has shown that IT has been a crucial—perhaps the most crucial—factor in increasing US economic competitiveness. It is a widely accepted that the tremendous productivity increases in the US economy during the last decade stemmed from aggressive enterprise automation using IT. While the rest of the world's industrialized nations try to catch up, we must recognize that the expansion and transformational impact of IT will not stop at enterprise automation, but will rapidly progress in products, infrastructure, education, and many other areas.

If we want to maintain and increase the economic advantages of our IT prominence, we must increase our investment in understanding the relationship between emerging new application domains and IT technologies.  More importantly, we must, aggressively seek out new methods and tools to explore emerging opportunities and extend our strategic advantage. End-user industry has neither the expertise nor the resources to make these changes themselves, and the dominant software industry may not have the resources or the interest in changing the status quo. Expanded government investment in IT is vital to accelerate this process. We believe that creating and maintaining a vibrant, active IT research community in the US is vital to our long-term economic and national security interests.

## The Future of Software and Software Research

Breakout Group Summary

Prepared by: Adam Porter

Panelists: Adam Porter, Chair

Barry Boehm, Bob Neches, Daniel Jackson, Gabor Karsai

Breakout session participants: Richard Buchness, Laura Dillon, Susan Gerhart, Susan Graham, Alkis Konstantellos, Steven Ray, Rose Gamble, Helen Gigley, Steven Reiss, William Pugh, Eric Dashofy, Philip Johnson, Marija Mikic-Rakic, Kenny Meyer

### Introduction

Software research focuses on methods, techniques, and processes that help to improve the quality and productivity of designing, constructing, implementing, and evolving complex software systems. The software systems being built today differ dramatically from those built 20 years ago. It's therefore natural to wonder how software research has evolved to reflect these changes. This panel examined this issue, looking at how software research will need to change over the medium- and long-term.

### Where Are We Now?

Although we often no longer notice it, software touches us throughout our daily lives. Software now runs in our home appliances, automobiles, streetlights, office security systems, business and personal communications, banking functions, medical instruments, emergency services, power distribution, scientific discovery, national security, and warfare systems. Software was not as pervasive thirty years ago, which by itself bears witness to the vast successes of software. Looking forward, however, it's clear that this progress cannot continue without fundamental advances in software technology. In particular, current software development techniques are reaching a complexity cap that limits their ability to deliver affordable, safe and reliable, easy to change, highly usable systems that are essential to improve our quality of life. Such systems might allow automated traffic management, interconnected medical devices (some of which would work inside the human body), coordinated management of emergency and crisis service, microscale robotic services, and automated battlefield management.

### What Can We Do Well?

To understand exactly what progress must be made to realize these systems, we must first take stock of the current state of software research. When contemporary software practices are compared with those from twenty years ago, it's clear that software research has had a dramatic positive effect. Some indicators of these impacts include:

- **Tremendous productivity gains.** High quality, complex software systems can be built in less time than it used to take. In fact, modern development practices routinely and easily accommodate 100,000+ LOC systems. As recently as 10 years ago this was considered a daunting task. These productivity gains are supported by repeatable best practices. For many classes of systems, developers can use standard processes with predictable schedule and quality characteristics. These gains derive from research in software processes, methods, and tools that are used throughout the software lifecycle. Some examples include:

  - Techniques and languages for requirements capture (e.g., user-centered design, Z, DOORS)
  - Design principles, languages, and notations (e.g., information hiding, object-oriented design, Java, UML)
  - Software architecture styles (e.g., C2, client/server, event-based programming)
  - Domain-specific languages, (e.g., Matlab, Simulink, Splus, Genoa, Lex and Yacc)
  - Techniques and tools for software inspection and testing (e.g., FTArm, Selective regression testing, protocol conformance testing)
  - Software process support (e.g., configuration management systems, CMM, process modeling languages)

▪ **Successful adaptation to new computational models**. As computing hardware has evolved, so have software development techniques. As a result, we can more easily build software for new computing environments, e.g., batch vs. interactive computing, distributed vs. single processor systems, text-oriented vs. graphical interfaces. Software research on the following topics has enabled this ability to adapt:

  - Object-oriented modeling and implementation. (e.g., UML, patterns and pattern languages)
  - Middleware (e.g., DCE, CORBA, COM+, J2EE, .NET)
  - GUI generators (e.g., PowerBuilder)

▪ **Software ubiquity.** Software is now a fundamental part of our national infrastructure. It is as crucial as energy, roads, and clean water. It runs a myriad of everyday devices. It has even become an accepted part of safety-critical applications, such as airplane control, medical technology, and factory automation. The success of safety-critical software is directly enabled by software research including:

  - Software safety (e.g., state charts, fault trees)
  - Analysis and verification (e.g., model checkers, type checkers, Lint, Purify)
  - Software testing (e.g., reliability modeling, test coverage analyzers)

▪ **(Partially) automated solutions for limited domains**. Software developers are increasingly able to reduce development time and effort by generating and reusing code. Design tools can sometimes be used to generate significant portions of systems. For tasks such as database management and network communication, software developers can make use of high-quality, systematically reusable frameworks and components, such as:

  - Model-based development tools (e.g., Promula, Rational Rose, ObjecTime, GME, MatLab)
  - Application frameworks (e.g., JavaBeans, SAP, ACE)

**Why Can't We Declare Victory?**

Despite many advances over the past two decades, software development remains hard. Several factors contribute to the gap between the software we need to build and the software we can afford to build given our level of technology maturity. These factors range from the technological to the social, from the theoretical to the practical, as described below:

▪ **Pervasiveness and criticality of software**. Software is a fundamental part of our national infrastructure. As a result, the amount of software we have to write has increased dramatically. Moreover, the reliability and safety requirements for much of this software have become much more demanding.

- **Rapid pace of change**. Information technology changes at an extremely rapid pace. The same goes for software as well. For example, as stand-alone computing systems have increasingly networked, software development techniques have been developed to support distributed computing environments. Likewise, as embedded devices and sensors became cheaper and more pervasive, development techniques and environments sprung up to support these domains. Moreover, these changes are affecting the economics of software development, once again calling for new software development techniques. Software is becoming a commodity, cutting profit margins and increasing the need for agile—rather than monolithic—lifecycle processes and development methods.
- **Program scale**. The scale and complexity of software systems is growing. In particular, they are being connected at a global scale. This growth motivates the need to manage the massive concurrency created by the large numbers of interconnected systems of systems. Moreover, since the individual systems that comprise these systems of systems cannot be designed simultaneously in a cost-effective manner, integration has become an enormous problem. Due to the challenges associated with integrating large-scale software systems, future systems must often be designed to adapt to changing computational environments and runtime configurations, rather than being built from a clean slate.
- **Poorly understood componentry**. Large software systems are increasingly constructed from existing components. While this reuse enables rapid construction of the initial system, it can create other problems if individual components are not flexible or customizable. For example, developers who use components are often incapable of predicting overall system performance because the components may interact in unforeseen ways. They are also often optimized for average conditions that differ from those in which specific applications will run. Moreover, it may be hard to change a component-based system if component interfaces do not support specific changes in application requirements.
- **Slow adoption of best practices**. Studies repeatedly show that mainstream software development practice lags far behind software research and the best practices adopted by leading software practitioners. In fact, individual technologies often require 20 years or more to successfully transition from research to practice, which is far too slow to meet the growing demands of end-users.
- **Low productivity in software testing**. Since current development technology cannot provide guarantees on the behavior of our software, testing still has a major role in software development. As long as we remain unable to create "correct by construction" development technology or "self-repairing" software, low software testing productivity will continue to be a serious bottleneck in the development process.

**Specific R&D Challenges**

In order to resolve the general forces described above software research must address several specific challenges.

- **Richer computing environments**. The physical elements with which software systems interact and the environments in which they run are becoming increasingly sophisticated. In the future, software will run in a multitude of computing environments ranging from traditional single-CPU desktop environments, to multiple host, networked systems, to mobile systems, to embedded systems, to wearable computers (inside the body as well as on it – cochlear implants, pacemakers, etc.). Some characteristics of these richer environments include:

    - *Highly distributed and concurrent*. Software will increasingly run on multiple host machines communicating across networks and networks of networks.
    - *Real-world sensing, location-aware computing*. Software systems will be aware of their physical surroundings, taking input from real-world sensors and sending output

intended to control the real-world environment. They will also understand their proximity to other computing resources and use this information to guide their behavior.

- *Greater platform heterogeneity*. Software will continue to run over computing platforms characterized by diverse languages, operating systems, and hardware.
- *Component-based system development*. Due to global competition and time-to-market pressures, developers of next-generation of complex systems will increasingly buy and build software components and integrate them to create systems. Software components range from relatively small reusable classes to large-scale frameworks and subsystems.
- *Wide-ranging modalities*. Software systems will routinely receive their input and write their output in non-textual ways including speech, touch, electrical impulses, etc.

▪ **Integration mechanisms**. Large systems are increasingly composed of other systems. Understanding and facilitating the integration of these "systems of systems" is becoming quite important. In particular, we must develop methods for ensuring system-wide properties, such as quality of service, across these systems of systems.
▪ **Changeability**. Along with a need to integrate systems, we also need better support for modifying components and component interactions. For example, we must be able to upgrade a specific component, remove the current version from the system (wherever and in whatever quantities it may be running), and then replace all instances of the component system without bringing down the whole system.
▪ **Fault tolerance.** We need to devise protocols and mechanisms to create systems that are robust and resilient even if individual components and connected systems have faults and fail in operation. Components must be able to operate even when the components they interact with fail, exhibit poor performance, or provide incorrect input.
▪ **Changing software developers, processes, and tools**. The economics underlying the software marketplace are changing, which is leading to the following changes in how companies organize themselves, define their business goals, and develop software:

- *Multi-disciplinary, geographically distributed development teams.* Software development teams are now routinely distributed across time zones and comprised of end-users and subject-matter experts (e.g., biology, psychology, medicine, and finance). Development teams must employ stronger domain and business models.
- *ncreased focus on time-to-market*. While functionality and correctness are important, many buyers will trade these for software that's available today.
- *Incremental development with frequent change propagation*. Time-to-market pressures and distributed code updating encourages developers to make small changes and frequently update the system, regression testing it, and then propagating the changed code to users.
- *Rapid development and experimentation*. Software developers must be able to quickly change software, place it into the field or into simulators, experiment with the updated software, and then repeat the process consistently.

▪ **Tool support**. With computing power available on a massive scale, professional developers can apply far more powerful support tools, such as:

- *Testing, analysis, and verification tools*. Increased computing power can be leveraged by software developers. Tools include model-checkers, generation and execution tools for regression and functional testing, design critics, program analysis tools, and semantics-aware tools.

  - *Collaboration tools*. Developers must have greater support for collaboration, design, change integration, automated testing, analysis and verification, and requirements and design specification and reuse.
  - *Software process support*. Software development organizations need help in modeling, understanding, and improving their development processes. They also need better support for recording and analyzing performance data, and using it to consistently improve their performance.

- **Ever-persistent legacy components.** Even if our technology improves, software developers often cannot take immediate advantage of technology changes because they cannot afford to throw away their investment in existing software and development tools. Consequently, software written with new technology often must interoperate with that written using older technology. Legacy software requires maintenance.

**Research Agenda**

To meet the R&D challenges described above software researchers must make progress in the following areas:

- **Design multi-objective techniques**. Software researchers have historically emphasized functional correctness above other concerns, such as time-to-market or cost containment. Many techniques are now being redesigned with other metrics in mind. For example, the following new techniques accept and even invite the use of heuristics, preferring practical usefulness over theoretical soundness:

  - *Value-conscious techniques* strive to help developers apply techniques in places and in proportions such that the benefits are likely to outweigh the costs, i.e., in ways that make sense economically. Some researchers, for example, are beginning to study how to quantify the value of making a design decision today or deferring it till later.
  - *People-conscious techniques* take human limitations and strengths into account. For example, formal specification techniques traditionally rely on complicated mathematical notations. Recently, some new specification notations have been designed with the goal of being easily readable by non-computer professionals.
  - *Opportunistically-applied techniques* strategically apply lightweight techniques, in piecemeal fashion, throughout the lifecycle. These techniques can often be heuristic in nature, theoretically unsound, and limited in scope. For example, design refactoring techniques emphasize improving software designs after each coding iteration, rather than designing the system once at the outset.
  - *Non-uniformly applied techniques* can be applied to specific parts of a software artifact, not just to the whole-program. For example, recent work on software analysis has focused on checking of partial, focused models, rather than complete models of system behavior.

- **Conduct large-scale empirical evaluation**. Software development is done in the field. Software research benefits heavily from contact with industrial development. Software researchers must increasingly evaluate their research in realistic settings and apply it to larger systems. They must also pay more attention to systems that involve people and do more in-depth studies of existing large-scale systems.
- **Incorporate domain-qualities into development languages**. Developers are asked to program at higher and higher levels of abstraction. Programming languages research is starting to create languages with high-level, application-specific concepts, such as trust management, security, power management, and timeliness built-in.

- **Collaborative software development**. New techniques and tools are being discovered that make it easier for distributed developers to work simultaneously. For example, some work has been done to better understand how to automatically decompose systems to make it easier for distributed developers to modify a system without undue conflicts. Work is also being done to support distributed testing and profiling of multi-platform systems.
- **Leverage increase in machine power**. Research involving computationally expensive techniques, such as model checking, has not yet had great success in software domains because they have not scaled up to industrial-sized programs. Increases in computing power, however, are making it possible to apply these techniques in broader situations. New work is investigating novel and highly focused ways to apply these techniques that seem to work for industrial-sized programs.
- **Bridge gap between design and code**. Programming at the source code level simply does not scale. Work is being done on tools for modeling and analyzing design level artifacts. Domain-based models and tools (i.e., those that represent and manipulate application-specific information) have had some success in well-understood domains, such as embedded systems.
- **Explore new computing paradigms**. New computing paradigms, such as biological, quantum, and market-based computing, are being developed. Software research is studying these paradigms and developing tools to support programming in them.
- **User-accessible techniques**. Software development by "non-programmers" is increasing and needs to be supported. Techniques are needed to help non-programmers record and negotiate system requirements from multiple points of view and develop, test, and modify programs.
- **Improved SE education and training techniques**. New techniques need to be developed that shorten the learning curve for software technology. These techniques should address multiple levels of IT personnel. They should also focus on drastically shortening learning time to master the new methods and techniques that will be developed to improve software design and productivity.

## Concluding Remarks

Software research has had an enormous, positive effect on our ability to put important, useful software into the field. These successes, however, have only increased the appetite of our consumers, thereby creating new and tougher challenges. To meet these challenges, software researchers have put together a broad and ambitious research agenda. In particular, they need to scale up their research to match the increasing scale of software systems. For instance, future software research will require the following investments from Government funding agencies:

- **Large-scale, multi-domain testbeds**. Software researchers need access to very large software artifacts to conduct research on new techniques to improve software design and productivity:

    - *Software archeology*. Industrial-scale software artifacts are needed to help researchers understand where and how often problems really exist.
    - *Software artifacts*. To validate their research software researchers need more than just source code. They need artifacts that span the software lifecycle, including requirements specs, designs, code, test cases, test case outputs, configuration management system (CMS) logs, and bug databases.
    - *Sponsor open-source software projects as a research enabler*. Funding agencies might be able to partially support some open-source projects (which make many of their artifacts public anyway) on the condition that they construct their artifacts in ways that will be useful to later researchers. Funding agencies could also encourage sponsored projects to instrument their artifacts.

- **Industry pilot projects**. For specific promising research technology, it may be desirable to fund large demonstration projects that involve interactions with industrial development personnel. It may also be desirable to incorporate software research in collaboration with funded research in non-software research, such as physics, biology, medicine, and MEMS.
- **Larger research staff**. Many software research projects are rendered ineffective before they start because funding levels do not allow for programming support staff.
- **Must produce more professionals**. Graduate students have increasingly moved away from software research towards other areas where funding has been more plentiful, such as scientific computing. This has left a shortage of highly trained young researchers in software. If we are to implement the vision of this workshop, we will need to substantially increase the number of graduate students doing software research.

## New Software Development Paradigms

Breakout Group Summary

Prepared by: Janos Sztipanovits, Doug Schmidt and Gregor Kiczales

Panelists: Gregor Kiczales, Chair

Don Batory, Ira Baxter, James Larus, Charles Simonyi

Breakout session participants: Karl Crary, Shriram Krishnamurthi, Paul Hudak, Frank Sledge, Karl Lieberherr, Kurt Stirewalt, Spencer Rugaber, Tzilla Elrad, Benjamin Pierce, Prem Devanbu, Tom McGuire, Doug Smith, Joy Reed, Mike Mislove, Ralph Johnson, Janos Sztipanovits

### Introduction

As Information Technology (IT) has expanded aggressively to all areas of science, technology, and the economy, the pressure to create software for an exploding number and variety of computer applications has increased substantially. The past decades have brought about major improvements in software technology, including advances in architectural styles and patterns, reusable object-oriented frameworks and component libraries, higher-level programming languages, and new development processes (such as extreme programming and open-source). Although these advances have yielded a tremendous increase in software productivity, any individual breakthrough had a limited impact since the rapid growth of user demands quickly surpasses its benefits.

Software productivity and quality depend heavily on *software development paradigms*. A development paradigm is characterized by a collection of design methods, programming languages, computation models, validation and verification techniques, and tools used in the development process, as follows:

- Design methods range from informal, such as flow diagrams, to formal, analyzable representations, such as state-charts.
- Programming languages determine the notations, abstractions, and composition techniques used by programmers to write programs. Determined by differences in composition methods, programming languages support very different styles of programming, such as structured programming, object-oriented programming, or aspect-oriented programming.
- Common examples of computation models are functional, logic, and imperative. The role of validation and verification is to provide assurance that the software will satisfy functional and non-functional requirements in the deployment environment.
- Modeling, analysis, debugging, and testing are important elements of the validation and verification process.

Of course, other elements of development paradigms strongly influence software productivity and quality. In current development paradigms, the predictability of designs is so weak that the cost of validation and verification can exceed 50 to 75 percent of overall development costs. Improving development paradigms is therefore a key factor in improving the productivity of software developers and the quality of their output.

The tremendous cost of maturing and adopting development paradigms motivates standardization and long-term stability. Successful development paradigms evolve over many years and may not become widely accepted for over a decade or longer. Unfortunately, their effectiveness depends largely on how well they support the characteristics of software to which they are applied. For example, the structured programming paradigm works well for moderate sized business applications, but does not scale up effectively to meet the functional and quality of service needs of large-scale distributed, real-time, and embedded systems. Given the aggressive penetration and transformational effect of IT in all conceivable areas, it is not surprising that currently dominating programming paradigms, such as structured programming and object-oriented programming, are challenged from many directions.

The goal of this panel was to identify new trends in the progress of software development paradigms and to discuss their relationship to new application challenges.

## Opportunities and Challenges

By all measures, the pervasiveness of IT is a huge success and a tribute to decades of R&D on hardware and software design. Ironically, however, the same success that has yielded many new opportunities also generates new challenges that make existing programming paradigms rapidly obsolete. Unfortunately, this creates the impression of lack of progress in software – we seem to be in a chronic “software crisis” since the dawn of computer programming. Some of the latest opportunities and related challenges include the following:

- **Increased fusion of software into application domains**. In many application domains, computing has become the key repository of complexity and the primary source of new functionality. For example, over 90% of innovations in the automotive industry come from embedded computing. The increased significance of computing means that unless unique characteristics of the application domain are reflected directly in the programming paradigms, application engineering considerations must be mapped manually onto general-purpose software engineering concepts and tools, which is tedious and error-prone. The difficulty of this manual mapping process motivates the need for carefully tailored capabilities, such as domain-specific languages, modeling tools, and middleware.
- **Software as universal system integrator**. Perhaps the biggest impact of the “IT explosion” in the last decade has been the emerging role of computing and software as the “universal system integrator.” Systems are formed by interacting components. The new trend is that an increasing number of components and interactions in real-life systems are *computational*. Complex information management systems, such as SAP (a popular business software package from Germany), now provide the “IT backbone” that keeps organizations functioning smoothly. Distributed control and process automation systems integrate manufacturing production lines. Flight control and avionics systems keep airplanes flying. The consequences of these changes are twofold. On one hand, there is an ever-tighter fusion of computing and software with application domains. On the other hand, there is a strong divergence in software technology, since the field is becoming much richer with each new application direction.

The increased fusion and deep integration of application domains with computing implies that essential characteristics of systems are strongly influenced - or simply determined - by the software. Consequently, software requirements become multi-faceted, i.e., computation and software architectures must satisfy many functional and physical requirements *simultaneously*.

## Where Do We Have Progress?

The past decade has brought about conspicuous successes in programming paradigms that fueled the IT revolution. Some notable advances are listed below.

- **Framework-based design.** One of the most successful approaches to decrease the cost of large-scale applications is to minimize the need to develop new code. To reach this goal, academic researchers and commercial vendors have created application frameworks that provide an integrated set of classes that collaborate to provide a reusable architecture for a family of related applications. By writing small amounts of code using a range of languages, such as Visual Basic, Java, C++, and PERL, users can create complex applications that customize the reusable frameworks. A sophisticated example of large-scale framework reuse is the use of SAP on the top of R3 (a database system), which represents ~4.5 GByte base code. Customization touches approximately 1% or less of the code and requires less than 1% plug-ins. Interestingly, typical applications use only about 10% of the base code. Framework-based design is successful due to the large-scale reuse of an existing and extensively tested code base. The relatively minor, carefully guided extensions cannot upset the overall consistency of the design, which minimizes the level of validation and verification effort.
- **Component-based middleware.** Component-based middleware encapsulates specific building block services or sets of services that can be composed and reused to form larger applications. At the programming language level, components can be represented as modules, classes, objects, or even sets of related functions. Component technologies have achieved significant progress toward providing composability and interoperability in large-scale application domains. Commercial component middleware (such as CORBA, .NET, or J2EE) offers "horizontal" infrastructure services (such as ORBs, interface and server repository, transaction, etc.). "Vertical" models of domain concepts (a shared domain semantics for the components), and "connector" mechanisms between components (message, event, work flow; location transparency, etc.). Current component middleware technologies work well in small-scale applications (such as GUIs) or work well with a few coarse-grained components (such as two- and three-tier client/server business applications).
- **Model-based software engineering.** It has been increasingly recognized that source code is a poor way to document designs. Starting from informal design documentation techniques, such as flow-charts, model-based software engineering is moving toward more formal, semantically rich high-level design languages, and toward systematically capturing core aspects of designs via patterns, pattern languages, and architectural styles. More recently, modeling technologies have expanded their focus beyond application functionality to specify application quality of service (QoS) requirements, such as real-time deadlines and dependability constraints. These model-based tools provide application developers and integrators with higher levels of abstraction and productivity than traditional imperative programming languages provide.
- **Generative programming.** The shift toward high-level design languages and modeling tools naturally creates an opportunity for increased automation in producing and integrating code. The goal of generative programming is to bridge the gap between specification and implementation via sophisticated aspect weavers and generator tools that can synthesize platform-specific code customized for specific middleware and application properties, such as isolation levels of a transaction, backup server properties in case of failure, and authentication and authorization strategies. Early research in this area is now appearing in commercial products that support narrow, well-defined domains, such as the SimuLink and StateFlow tools from MathWorks, which generate signal processing and control applications from high-level models. The productivity increase achieved by generative programming is impressive, e.g., users account for 40-50% increase even using partial solutions.

## Why Can't We Declare Victory?

Years ago, Donald Knuth wrote[2]: "One of the most important lessons, perhaps, is the fact that software is hard…. The creation of good software demands a significantly higher standard of accuracy than those other things to do, and it requires a longer attention span than other intellectual tasks." Knuth's message remains valid today. Our development paradigms do not measure up to the challenges of current and future applications. The lack of high standards of accuracy in our design paradigms makes validation and verification incredibly expensive and inefficient. Additionally, the targets keep changing. For example, the solutions to hard business software problems do not necessarily translate into solutions to hard embedded software problems. Moreover, effective solutions today generate new needs and enable expansion into new domains and new levels of system complexity tomorrow, where we have less experience.

Listed below are some areas of concern today that represent bottlenecks in our ability to exploit IT in key emerging areas:

- **Enabling end-users to create and maintain complex applications.** The meteoric growth in the pervasiveness of IT means that either everyone will become a trained programmer or benefits of programming will become accessible for everyone. Past and current efforts in IT R&D have largely been proceeding along the first trajectory, i.e., "no matter what career path you choose, you'll eventually end up programming…" Unfortunately, this approach does not scale. Economic pressure and solid technical arguments are therefore forcing researchers to understand how to make programming accessible for end-users. There are sporadic examples of the huge impact of end-user programming. From spreadsheets to CAD packages and from circuit simulators to workflow management systems, increasing number of applications are built with end-user oriented APIs. The main impediments to succeeding with end-user programming today are the high cost of creating and maintaining end-user programmable systems. We need technology for creating and evolving end-user programmable applications.
- **Addressing many concerns simultaneously.** Programming must satisfy many different concerns, such as affordability, extensibility, flexibility, portability, predictability, reliability, and scalability. The current trends mentioned earlier, e.g., increased fusion with domains, integration role in physical systems and increasing size, point toward the need to build systems that satisfy these different concerns simultaneously. For example, business applications must provide the required functionality *and* be safe and highly dependable. Likewise, embedded system applications require the satisfaction of many types of physical constraints (e.g., dynamics, power, and size), in addition to being predictable, dependable, safe, secure, and small. While composing software systems from a single (usually functional) point of view is relatively well supported in modern programming paradigms via module interconnection languages and hierarchical composition, we do not yet have scalable solutions for composing systems dependably from multiple points of view.
- **Building and understanding large, fine-grain distributed applications.** One of the fundamental trends in IT is the increasing impact of networking on computing systems. Applications for these systems are not only more complex, they also interact with their environment (including humans) and with each other in demanding and life-critical ways. Many of these distributed applications are dynamic, i.e., they continuously change their shape and interaction patterns as the environment is changing. Design and execution frameworks, which constrain design decisions to bound the behavior of these systems and the related development paradigms, constitute a major opportunity that must be addressed in the future.

Using more general terms, there is little or no science behind software design today. Right now, it is an art-form practiced well by great software engineers, and practiced poorly by most others. As

[2] Keynote address to 11th World Computer Congress (IFIP Congress 89)

long as it remains an art-form, our abilities to resolve the challenges above will remain limited. Our future R&D efforts must therefore move us towards a rigorous science of design, which can serve as a solid foundation for future development paradigms.

**Specific R&D Challenges**

Given the strong divergence of major application domains with many unique challenges, it is not surprising that no single silver bullet exists. Still, the new challenges created by new types of applications, along with the universal need to push the complexity envelope, define the following core research goals that must be addressed by the IT R&D community in the next decade:

- **Composition of domain-specific languages.** As mentioned earlier, domain-specific abstractions are increasingly important in software design. This need is driven by the following major trends:

    - *Increased pervasiveness of computing*, which means that domain engineering and related software design are inextricably combined,
    - *The need for end-user programmability*, which translates into the use of powerful application frameworks that can be customized with domain-specific languages.

  Unfortunately, current technology limitations discourage the wider introduction of domain-specific programming paradigms due to the high cost of developing solid semantic foundations and reasonable tool support required for safe application. We must therefore build a new infrastructure that enables rapid *composition of domain-specific languages* on different levels of abstraction, such as high-level, declarative modeling languages, languages representing heterogeneous models of computations, semantically solid pattern languages, and others.
- **Tool infrastructure supporting reusability in broadly diverse domains.** Increased automation requires the application of a range of analysis and synthesis tools, which are expensive to develop and expensive to learn. Since much tool development occurs in the context of a specific domain or development paradigm, reusability is currently extremely limited due to the many undocumented assumptions and the implicit, shared semantics of the domain. While changing this situation toward increased reusability requires little justification, actually doing it is an exceedingly hard problem since it requires comprehensive capture and explicit representation of the stakeholders meaning and concerns in the many different sub-domains used by individual tools and programming paradigms. Moreover, the selected representation formalisms must provide support for modeling the relationships among sub-domains and the ability to translate these relationship models into some form of "semantic interfaces" among the tools. To make the problem even more challenging, success requires a broad consensus on actually using the results. Aggressive research in this area will bring to bear a culture change, where precise semantic specification will be considered "good hygiene" required for safe software development, rather than a burden.
- **Composition for multi-faceted programming.** As mentioned earlier, composition is a key means to achieve scalability. Modern programming languages support hierarchical, modular composition, which is insufficient in key, emerging directions of computing, such as embedded systems, large-scale distributed applications, and others characterized by the presence of many crosscutting design constraints. Progress in these areas requires a revolutionary change in programming paradigms by introducing a completely new composition strategy: *multi-faceted composition*. Multi-faceted composition enables that:

    1. Programs are specified from different point of views,
    2. Reusable components are developed to support these different views, and

3. Automated composition mechanisms – called program weaving - combine and produce the whole.

Obviously, multi-faceted composition can be and must be integrated with development paradigms working on different levels of abstractions, such as procedural object-oriented languages or declarative modeling languages.

The major challenge in multi-faceted software development is the presence of crosscutting constraints that make the different views interdependent. This results in a tremendous increase in complexity during automated composition. Resources invested in solving automated multi-faceted composition are well spent, however. The alternative - doing it manually - leads to the tedious, error-prone, and expensive state of system integration found in current practice.

- **Framework composition.** Large-scale, dynamic, distributed applications represent very different challenges. While the complexity of the behavior of the individual nodes is limited, the global state and behavior of the overall system comprised of a large number of interacting nodes can be extremely complex. The problem in these systems is not the design of a specific global behavior, which may not even be monitored or known with perfect accuracy, but bounding the behavior in "safe regions" of the overall behavior space. This goal can be achieved by *design and execution frameworks* that introduce constraints in the behavior of and interaction among the distributed components. The constraints must be selected so that they provide the required level of guarantees to prevent unsafe behavior. The development of complex design and execution frameworks is extremely hard today and requires a long maturation process. We need technology that can automate a large part of this process and allow the automated synthesis of frameworks that are highly optimized for particular domain characteristics.

**Concluding Remarks**

Even as computing power and network bandwidth increase dramatically, the design and implementation of application software remains expensive, time consuming, and error prone. The cost and effort stem from the growing demands placed on software and the continual rediscovery and reinvention of core software design and implementation artifacts throughout the software industry. Moreover, the heterogeneity of hardware architectures, diversity of operating system and network platforms, and stiff global competition makes it hard to build application software from scratch and ensure that it has the following qualities:

- *Affordability*, to ensure that the total ownership costs of software acquisition and evolution are not prohibitively high
- *Extensibility*, to support successions of quick updates and additions to take advantage of new requirements and emerging markets
- *Flexibility*, to support a growing range of multimedia data types, traffic patterns, and end-to-end quality of service (QoS) requirements
- *Portability*, to reduce the effort required to support applications on heterogeneous OS platforms, programming languages, and compilers
- *Predictability* and *efficiency*, to provide low latency to delay-sensitive real-time applications, high performance to bandwidth-intensive applications, and usability over low-bandwidth networks, such as wireless links
- *Reliability*, to ensure that applications are robust, fault tolerant, and highly available, and
- *Scalability*, to enable applications to handle large numbers of clients simultaneously.

Creating applications with these qualities in a timely manner depends heavily on software development paradigms. There is an important synergy between development paradigms and application domains. In earlier generations when computing and communication machinery were scarce resources and applications were relatively small scale, development paradigms focused largely on efficient use of hardware/software resources rather than efficient use of developer resources.  As hardware has grown more powerful and applications have grown in complexity, elevating computation models, programming languages, and associated software tools closer to application domains has become increasingly important. As IT continues its rapid expansion in new domains, we expect that software technologies based on reusable frameworks and patterns, component middleware, model-integrated computing, and generative programming will become more pervasive, application-centric, and ultimately much more capable of improving software productivity and quality.

**Software for the Real-World (Embedded Systems)**
Breakout Group Summary

Prepared by: Robert Laddaga

Panelists: Robert Laddaga, Chair

Paul Robertson, Scott Smolka, Mitch Kokar, David Stewart, Brian Williams

Participants: Insup Lee, Kang Shin, John Reekie, Rajeev Alur, Calton Pu, Mitch Kokar

## Introduction

Among the hardest problems software developers will face in the future are those associated with producing software for embedded systems. Embedded systems are systems in which computer processors control physical, chemical, or biological processes or devices. Examples of such systems include airplanes, cars, CD players, cell phones, nuclear reactors, oil refineries, and watches. Many of these embedded systems use several, hundreds, or even thousands of processors. Although most embedded systems today are relatively small scale (e.g., operate with limited memory and use eight-bit processors), they are still hard to program. The trend is toward increasing memory and computational power and significantly increasing functionality. With the added functionality and hardware resources comes an even greater increase in complexity. Each year, the semi-conductor industry produces more processors than there are people on the planet, and the number of processors produced is growing significantly. Almost all of these processors are used in embedded applications, rather than the personal computer, workstation, server, or mainframe systems with which users are traditionally familiar.

The real-world processes controlled by embedded systems introduce numerous hard constraints that must be met when programming embedded processors. Among these constraints are:

- Real-time requirements, such as low latency and bounded jitter
- Signal processing requirements, such as truncation error, quantization noise, and numeric stability
- High availability requirements, such as reliability and fault propagation/recovery across physical and computation system boundaries and
- Physical requirements, such as power consumption and size.

Meeting the constraints listed above is hard precisely because they affect embedded systems globally, and because embedded programs contribute globally to these problems. These real world issues have been with us for as long as we have attempted to control physical processes. Therefore, they don't themselves explain the tremendous increase in complexity of embedded software. Two important trends provide some of that impetus: (1) the increasing tendency to use embedded software as the main integrator for complex systems, and (2) the increased dependence on embedded software to supply novel and more intricate functionality. In each case, embedded software is replacing something else. In the first case, the chief integrators of complex systems were the human developers and users. We are now asking our embedded software to handle an increasing amount of that integration burden. In the second case we are replacing mechanical and hydraulic linkages with computer controlled electromechanical systems, because only they can meet the new capabilities and stringent performance requirements that we set for our systems.

The main simplifying technique we have in software development is *abstraction* in the form of functional, object-oriented, or component-based decomposition. To the extent that we can modularize via these decompositions, we can reduce the effective complexity of the software that we write. Complexity increases whenever such modularity is disrupted, in the sense of needing to take account of the internals of other modules while coding a different module. As such cross cutting concerns multiply, the job of software development, validation, and evolution become complicated, and often impossible. Real-time requirements and coordination with physical processes introduce tremendous numbers of such cross cutting concerns, violating modularity with abandon. To address these problems, we need software technologies in several related areas, including multi-layer distributed resource management, multiple decomposition dimensions, self-adaptive software, and tolerant and negotiated interfaces, among others.

In this report, we discuss notable progress in the area of developing software for the real world, then note several of the many things that we can't do now, both from real-world application and technological perspectives. Finally, we describe a number of promising new technologies for real-world software development.

## What We Can Do Well

We have achieved significant progress in real-world software research and development in the following areas:

- **Use of tools—**The past decade has seen significant advances in the maturity of code generators, modeling tools, software validation and model checking tools, and tools for generating test cases. Although there is a significant list of such tools, their use has only recently become pervasive in certain domains of commercial embedded system practice such as the automotive and avionics domains.
- **Use of real-time operating systems—**The use of real-time operating systems saves considerable amounts of work in building embedded systems by implementing reusable support for scheduling, memory allocation, interrupt handling, and peripheral device drivers.
- **Implementation within memory/computation constraints—**We are good at producing software when either memory or computation is highly constrained. In such cases, the scope of the problem is well suited to solution by a single (or small number) of highly skilled programmers, who can produce clever short code sequences that minimize processor cycles or memory use. It is no small achievement that we can produce such systems. Moreover, most embedded systems we have produced to date that can handle these constraints successfully do so largely because, in the past, financial considerations or specific application properties have constrained memory or computational power.
- **Use of fault tolerant protocols—**Highly useful fault tolerant protocols such as the TCP and STCP (Secure TCP) networking protocols have been developed.
- **Scheduling single processors—**We are very good at scheduling single processors, using technologies such as rate monotonic scheduling and related analysis techniques.
- **Testing and validation—**We have made significant progress with both testing and validation. There are tools and techniques for testing, including use of cross products of feature sets, and use of sophisticated statistical techniques along with test generators. There has also been significant progress in using formal methods to validate life- and safety-critical software for limited-sized embedded systems.
- **Implementation of fixed location automation—**We are reasonably good at automating factories, chemical processes, and devices that have fixed locations and limited scopes of activity in the world.

## Why We Can't Declare Victory

Despite the progress reported above, the following are important tasks or capabilities that we need today, but which are beyond the current state of the art:

- **Flexible, complex sensing—**The ultimate goal of embedded computing is to embed intelligence in physical devices. A key element of embedded intelligence is high-level sensing, which is the ability to understand images, audio signals, and other sensor data in terms of real-world object actions and decision problems. We distinguish this intelligent processing from signal processing, which is the low-level extraction of a signal from a noisy environment. What we need is the ability to infer information about objects and their complex relationships to each other, both current and future, from the information in the signals that we process. A similar and related problem is our current inability to fuse information from multiple sensors and data sources in real time into a coherent picture of the world.
- **Control and automation—**We don't know how to automate systems that move about in the real world. We have great difficulty with both navigation and robust behavior in an unknown context. Similarly, when control of a system or process needs to shift, either between automated controllers or between an automated controller and human control, we don't know how to accomplish the transition smoothly. Both these control shift issues make cooperative engagement in warfare and in commercial activity very hard.
- **Readily trusted embedded systems—** The certification processes that we have adopted for life- and safety-critical software are horribly expensive, but without them, our systems are both too fragile and too complex to trust. By making embedded systems "invisible and ultra-stable" we must go beyond simple trust to the point where people are entirely unaware of the controlling software, because our systems will just "do the right things."
- **Adaptability to changing requirements/environment—**Finally, we don't know how to make systems that can adapt to changes in environment and to the different uses to which they are applied.

**Specific R&D Challenges**

The applications and services we can't support today stem from underlying technical shortfalls. The reasons that we can't do the important tasks or provide capabilities listed in the previous section are because the following technologies have yet to be developed, accepted, or widely deployed:

- **Uncertainty and model selection—**We have a very hard time correctly and efficiently representing and computing about uncertainty in the world. Moreover, while we have difficulty estimating parameters of models efficiently, we are often not even sure about what model is correct at the outset. We must therefore either use data to decide about a model and then attempt to estimate parameters, or do both simultaneously.
- **System hardware/software co-design—**We constantly make poor engineering trade-offs because our sequential system design processes over-constrain our solution alternatives. For example, if we design the embedding system hardware first, we may not account for choices in the system that only become apparent during the software design phase, as well as software capabilities based on those choices. We therefore need to develop a credible system hardware/software co-design capability.
- **Testing and validation of large integrated embedded systems—**We can neither effectively test nor validate large integrated embedded systems, because we lack system theory for tightly integrated physical and computational systems. Existing theories are partial and cannot predict essential behaviors. Moreover, the proofs for verification are too large and computationally intractable, and the space of test cases is too large to be feasible. We need to be able to compose verified or tested systems, and be assured that constraints met in the subsystems are guaranteed to be met in the composed system. When testing is needed, only modest testing (in addition to guarantees) should be required.

- **Meeting systemic quality of service (QoS) constraints in integrated systems—**Satisfying systemic QoS constraints (such as limits on resource usage and scheduling or timing issues) in an integrated system present substantial problems.
- **Use of models in integration—**We are relatively good at using substantive models, such as mathematical models, to build a system, but not very good at using models to integrate several systems or modules of code with one another.
- **Automating what we know how to do—**While we know how to manually build systems that have stringent memory or computation constraints, we don't know how to automate the process so that tools can generate these types of systems automatically by refinement from higher-level specifications of the problems to be solved.
- **Distributed dynamic resource allocation—**We can perform centralized resource allocation well but have poor technologies for distributed resource allocation. Distributed resource-based systems may have sub-optimal or pathological allocation or can get into live-lock or deadlock.
- **Fault management—**Diagnosing faults is hard, uncertain, and expensive. For these reasons, we currently don't do a very good job of fault management. We handle fault management through static designs, such as multiple units and voting schemes, by human-in-the-loop intervention, and by shutdown and replacement of faulty units. We need effective techniques for proactive diagnosis and repair of *running* embedded systems.
- **Handling harsh environments cost effectively—**The only approaches that we have successfully applied to handling harsh environments are expensive forms of material and structural over-design and hardening. We haven't explored the use of computational power to make systems adapt to harsh environments and to repair and recover from resultant error, as an approach to dealing with difficult environments more cost effectively.
- **Encoding and measuring functional redundancy—**One of the chief issues we must resolve is how to reuse good designs in embedded systems properly and successfully. Since software itself represents only a small part of the overall embedded system design, we must first understand exactly what we want to reuse. Reusables should range beyond software artifacts to include patterns and architectural styles. Since effective reuse involves modifying or customizing software, we need effective ways to measure functional redundancy and provide descriptions that allow programs to compute functional redundancy. These descriptions will then also be useful in building fault tolerant systems, where we must ensure there is sufficient functional redundancy to overcome subsystem failures.

## Promising Research Strategies

To address the technical shortcomings listed in the previous section—and hence be able to address the service and application needs listed in the section before it—we need focused research and development efforts to mature and transition the following technologies:

- **Model-based software development.** Embedded systems can operate robustly in harsh environments through careful coordination of a complex network of sensors and actuators. Given the increasing complexity of future embedded systems, such fine-tuned coordination is ordinarily a nearly impossible task, both conceptually and as a software engineering undertaking. Model-based software development uses models of a system to capture and track system requirements, automatically generate code, and semi-automatically provide tests or proofs of correctness. Models can also be used to build validation proofs or test suites for the generated code.

  Model-based software development removes much of the need for fine-tuned coordination, by allowing programmers to read and set the evolution of state variables hidden within the physical system. For example, a program might state, "produce 10.3 seconds of 35% thrust," rather than specifying the details of actuating and sensing the hardware (e.g., "signal controller 1 to open valve 12," and "check pressure and acceleration to confirm that valve 12 is open"). Hence a model-based program constitutes a high-level specification of intended state evolutions. To execute a model-based program an interpreter could use a model of a controlled plant to continuously deduce the plant's state from observations and to generate control actions that move the plant to specified states.

Model-based software development research includes the creation of increasingly expressive languages for specifying intended state evolutions and plant behavior, and automated execution methods for performing all aspects of fine-grained coordination. The following are important items in the research agenda for model-based software development:

- Closing the consistency gap between model and code
- Preserving structural design features in code
- Translating informal requirements into formal requirements
- Tracing requirements into implementation
- Integrating disparately modeled submodels
- Enriching formalisms supporting non-functional aspects
- More efficient testing
- Capturing models of distributed embedded systems
- Modeling and using uncertainty
- Understanding and building self-adaptive models
- Seamless extension of embedded languages to:
    - Incorporate rich models of the embedded environment.
    - Shift the role of a program from imperative to advisory
- Managing interactions with fast on-line reasoning, including:
    - State estimation
    - Environment reconfiguration
    - Planning and scheduling
    - Discrete-event control and continuous control
- Automated partitioning of coordination between compile-time and run-time tasks
- Using frameworks for incorporating and reasoning from a rich set of modeling formalisms

Model-based development of embedded software also provides comprehensive support for autonomy in several ways. For example, it simplifies programming for autonomy by offering a simpler model of interaction between the programmer and the environment, as well as delegating reasoning about interactions to the language's interpreter/compiler. It also improves robustness for autonomy by systematically considering a wider set of possible interactions and responses, responding to novel events on line, and employing provably correct algorithms. Moreover, it supports adjustable levels of autonomy by allowing the programmer to delegate the desired level of control authority within the control program.

Key benefits of model-based software development include:

- Lower cost of systems development and certification via streamlined testing, early bug discovery, and powerful validation techniques
- More stable and robust embedded systems
- Greater trust in embedded software via improved understandability, reliability, and certification

▪ **Model selection and estimation.** Research on model selection and estimation aims at producing algorithms and methods to support high-level inference from applying models to sensor data. It consists of techniques for simultaneously estimating model parameters and comparing alternate models. A simplistic view of the world suggests that scientists experiment to determine what parametric models apply to phenomena, and that engineers use those models to build systems, tuning the parameters of the model to the environment. In fact, scientists and engineers often guess as much about the right model as about the parameter settings, and need to do both simultaneously. This is especially true when the engineering task is to interpret sensor data to gain crucial information about an environment. Model selection and estimation can be helpful in:

- Information fusion, by helping to integrate over large, multidimensional information spaces, and distributed, potentially conflicting information sources
- Understanding the behavior of embedded systems, for example by detecting incipient states, which helps to detect masked states, and by detecting hidden states
- Enabling efficient development of deeply networked systems (systems with very large numbers of relatively small components, all of which are networked)
- Enabling adaptive fault management, by adaptive model and parameter selection

The following are important items in the research agenda for model selection and estimation:

- Approximation and optimization techniques to allow tractable computation along with realistic dependency assumptions
- Estimation of performance over large distributed parameter spaces
- Integration of multiple models across different representations – models include constraints, logic, Baysian nets, hidden Markov models, and ordinary differential equations
- How to seamlessly fold methods for model selection and estimation into embedded languages

▪ **Domain specific languages for dynamic planning and scheduling.** Our experience in developing embedded software over the past several decades underscores that static plans or schedules are good only until the first unexpected real-world twist or change. Then it becomes clear that only plans that can be adapted to changed conditions while in use are effective. Both artificial intelligence and operations research have contributed to the study of algorithms and representations for planning and scheduling, but the bulk of that work has supported only static planning and scheduling. Dynamic planning and scheduling creates plans that include assessment of the situation during execution before enacting the plan and replanning in a reactive manner as necessary. Better technologies for building such dynamic plans are needed. Domain specific or embedded languages allow the programmer to express high-level constraints and conditions of computed actions. They involve planning using expected values, and the ability to track large numbers of execution trajectories. They involve highly dynamic techniques using on-line tracking, projection, execution, and re-planning. The following are important items in the research agenda for domain specific languages for dynamic planning and scheduling:

- How to decide which unlikely trajectories to track: while not all trajectories can be tracked, it is not sufficient to track only highly likely trajectories because then catastrophic but unlikely faults will be missed
- How to project forward consequences of traced trajectories to ensure safety
- On-line model checking
- How to fold temporal decision theoretic planning and execution into embedded languages
- How to do temporal decision theoretic planning at a reactive timescale
- How to concurrently plan and execute on line

Temporal decision theoretic embedded languages are a promising way to enable automation of embedded systems and help make them more adaptive and robust. They do this principally through support for dynamic planning and scheduling, and by enabling flexible and dynamic recovery strategies for fault management.

- **Systemic QoS reasoning for embedded software development** provides a bottom up approach to produce reliable components and building blocks. It can help improve automation and certification and help assure the behavior of low level components. It focuses on designing techniques and tools to assist with systemic QoS constraints such as side effects of the body of code, rather than explicit code expressions. Examples are time and space constraints that are not explicitly represented in source code. The following are important items in the research agenda for systemic QoS reasoning for embedded software development:

    - System/software codesign, including software redesign and reconfiguration
    - Reliable device drivers, such as reliable interfaces to unreliable hardware
    - Aspect-oriented software development, including performance monitoring
    - System QoS constraints, addressing time, space, imprecise computation, uncertainty, and fault tolerance issues
    - Trade-off analysis
    - Configurable hardware

- **Self-adaptive software** addresses high level sensing, adaptation, and automation. It is software that monitors itself and repairs or improves itself in response to changes or faults. It effects the repair or improvement by modifying or re-synthesizing its programs and subsystems using feedback-control-system behavior. Examples of uses of self-adaptive software are:

    - Networks of cooperating vehicles
    - Reconfiguration of hardware within vehicles in the air, on land, on the sea, or under the sea
    - Adaptation of control laws for flight surfaces and for submarines
    - Adaptation of numerical codes for optimization or simulation
    - Adaptation of assumptions to track changing conditions during high level sensing (for example for vision or speech)

    The following are important items in the research agenda for self-adaptive software:

    - Investigate ways of ensuring stability
    - Investigate ways of ensuring that the high level system goals are met
    - Investigate how to represent models and monitor models for different classes of systems
    - Investigate ways of synthesizing programs
    - Investigate how to achieve acceptable performance (for example, good enough, soon enough, or in terms of QoS metrics)
    - Architectures and design of self-adaptive software
    - Design languages that incorporate sensing and adaptation ideas

## Concluding Remarks

The bulk of today's computational cycles are expended in controlling electromechanical devices such as aircraft, automobile engines, cameras, chemical plants, hospital patient monitoring equipment, missiles, radar systems, satellites, and watches. In addition to being an information processing system, a personal computer is an embedded system with specialized chips that control communications among its main processor, memory, and peripherals. Each peripheral device, such as a disk, printer, audio card, also has embedded software. The rate at which mechanical controls and linkages are being replaced by software controllers is high and accelerating. This means that more essential features of our lives and economy increasingly depend on the quality and cost of embedded software. We need to know that our devices will continue to work well and will be diagnosable and repairable in a timely and cost effective manner when they fail.

Every year the semiconductor industry manufactures a range of microprocessors and microcontrollers, from 4 bit through 64 bit processors, and in various special sizes. Several years ago the embedded systems industry made the transition from having the bulk of microprocessors being 4 bit processors to having the bulk being 8 bit. We are on the verge of transitioning to the bulk being 16 bit processors, with the transition to 32 bit processors expected quickly thereafter. Memory sizes in these controllers are doubling at roughly the same rate. The increasing size of the most prevalent processor provides a modest measure of the exponential growth of complexity of embedded software applications. Each time we add functionality by applying newly affordable processor and memory resources to an increasing number of roles. However, old approaches of hand coding in assembler programming languages and hand proving behavioral assurance do not scale up either in the complexity or shear number of embedded applications. New technologies for building such applications are therefore absolutely essential, both from a national security and an economic perspective.

Most of the technical approaches advocated above involve techniques for raising the level of abstraction that designers and programmers need to be concerned about. This is the only way we can tackle issues of exponentially increasing complexity. But part of the complexity of embedded software lies in the fact that it is not possible to build nicely insulated layers of abstraction since there are too many interdependencies that cross abstraction boundaries. That is one reason why embedded systems developers have resisted use of abstraction for decades. The promising research strategies described above include different approaches to managing the complexities that cross abstraction boundaries. By employing these strategies we can better build and assure embedded systems for the real world.

## Large-scale, Network-centric Systems

Breakout Group Summary

Prepared by: Rick Schantz, Doug Schmidt

January 16, 2002

Panelists: Rick Schantz, Chair

Dave Sharp, Mike Masters, Prem Devanbu, Priya Narasimhan

Participants: Sally Howe, Kane Kim, Cordell Green, Gary Daugherty, Thuc Hoang, Joe Loyall, Betty Cheng, Jim Hugunin, Martin Rinard, Joe, Cross, Doug Schmidt

### Introduction

Next-generation commercial and military applications will operate in large-scale, network-centric configurations that take input from many remote sensors and provide geographically dispersed operators with the ability to interact with the collected information and to control remote effectors. In circumstances where the presence of humans in the loop is too expensive or their responses are too slow, these systems must respond autonomously and flexibly to unanticipated combinations of events during execution. Moreover, these systems are increasingly being networked to form long-lived "systems of systems" that must run unobtrusively and largely autonomously, shielding operators from unnecessary details (but keeping them appraised so they may react during emergencies), while simultaneously communicating and responding to mission-critical information at heretofore infeasible rates. Examples of these types of systems include (but are not limited to):

- Metropolitan area traffic control systems that process sensor data from 1000s of vehicles
- Coordinated swarms of unmanned air vehicles
- Command and control systems for theater-level battle management
- Supply chain management
- Community analysis of scientific data
- Home power management
- Integrated health care delivery systems and
- Terrorist tracking and identification systems.

In such systems, it is hard to enumerate, even approximately, all possible physical system configurations or workload mixes *a priori*.

An increasing number of these large-scale, network-centric systems include many interdependent levels, such as network/bus interconnects, local and remote endsystems, and multiple layers of software. Desirable properties of these systems include predictability, controllability, and adaptability of operating characteristics for applications with respect to such features as time, quantity of information, accuracy, confidence, and synchronization. All these issues become highly volatile in systems of systems, due to the dynamic interplay of the many interconnected parts. These parts are often constructed in a similar way from smaller parts. While it is possible *in theory* to develop these types of complex systems from scratch, contemporary economic and organizational constraints, as well as increasingly complex requirements and competitive pressures, make it infeasible to do so *in practice*.

To address the many competing design forces and execution-time quality of service (QoS) demands, sustained government R&D investments on comprehensive software methodologies and design-time/run-time environments are required to dependably compose large, complex, interoperable applications from reusable components. Moreover, the components themselves must be sensitive to the environments in which they are packaged. Ultimately, what is desired is to take components that are built independently by different organizations at different times and assemble them to create complete systems that are customized for their requirements and environmental conditions. In the longer run, each complete system often becomes a component embedded in still larger systems of systems. Given the complexity of this undertaking, various tools and techniques are needed to configure and reconfigure these systems in layers so they can adapt to a wider variety of more globally scoped situations, without changing their basic design and implementation.

**Where are We Now?**

During the past decade IT researchers, practitioners, and end users have benefited from the availability of commodity priced hardware (such as CPUs and storage devices) and networking elements (such as IP routers). More recently, the maturation of programming languages (such as Java and C++), operating environments (such as POSIX and Java Virtual Machines), and enabling middleware (such as CORBA, Enterprise Java Beans, and .NET) is helping to commoditize many software components and architectural layers as well. The quality of such commodity software has generally lagged behind hardware, and more facets of middleware are being conceived as the complexity of application requirements increases, which has yielded variations in maturity and capability across the layers needed to build working systems. Nonetheless, recent improvements in frameworks, patterns, design tools, and development processes have encapsulated the knowledge that enables common off-the-shelf (COTS) network-centric software to be developed, combined, and used in an increasing number of large-scale real-world applications, such as e-commerce web sites, consumer electronics, avionics mission computing, hot rolling mills, command and control systems, backbone routers, and high-speed network switches.

Over the past decade, various technologies have been devised to alleviate many complexities associated with developing software for network-centric systems. Their successes have added a new category of systems software to the familiar operating system, programming language, networking, and database offerings of the previous generation. In particular, some of the most successful emerging technologies have centered on *middleware*, which is systems software that resides between the applications and the underlying operating systems, network protocol stacks, and hardware. The primary role of middleware is to

- Functionally bridge the gap between application programs and the lower-level hardware and software infrastructure in order to coordinate how parts of applications are connected and how they interoperate.
- Enable and simplify the integration of components developed by multiple technology suppliers.
- Provide a common reuseable accessibility for functionality and patterns that formerly were placed directly in applications, but in actuality are application independent and need not be developed separately for each new application.

Middleware architectures are composed of relatively autonomous software objects that can be distributed or collocated throughout a wide range of networks. Clients invoke operations on target objects to perform interactions and invoke functionality needed to achieve application goals. When implemented properly, middleware can help to:

- Shield software developers from low-level, tedious, and error-prone platform details, such as socket-level network programming.

- Amortize software lifecycle costs by leveraging previous development expertise and capturing implementations of key patterns in reusable frameworks, rather than rebuilding them manually for each use.
- Provide a consistent set of higher-level network-oriented abstractions that are much closer to application requirements in order to simplify the development of distributed and embedded systems.
- Provide a wide array of reusable developer-oriented services, such as logging and security that have proven necessary to operate effectively in a networked environment.

Some notable successes in the middleware domain include:

- Java 2 Enterprise Edition (J2EE), CORBA, and .NET, which have introduced advanced software engineering capabilities to the mainstream IT community and which incorporate various levels of middleware as part of the overall development process, albeit with only partial support for performance critical and embedded solutions.
- Akamai et al, which legitimized a form of middleware service as a viable business, albeit using proprietary and closed, non-user programmable solutions.
- Napster, which demonstrated the power of having a powerful COTS middleware infrastructure to start from in quickly (weeks/months) developing a very capable system, albeit without much concern for system lifecycle and software engineering practices, i.e., it is one of a kind.
- WWW, where the world wide web middleware/standards led to easily connecting independently developed browsers and web pages, albeit also the world wide *wait*, because there was no system engineering or attention paid to enforcing end-to-end quality of service issues.

The Global Grid, which is enabling scientists and high performance computing researchers to collaborate on grand challenge problems, such as global climate change modeling, albeit using architectures and tools that are not aligned with mainstream IT COTS middleware.

**Why Can't We Declare Victory?**

In certain ways, each of the successes listed above can also be considered a partial failure, especially when viewed from a complete perspective. In addition, other notable failures come from Air Traffic control, late opening of the Denver Airport, lack of integration of military systems causing misdirected targeting, and countless number of smaller, less visible systems which are cancelled, or are fielded but just do not work properly. More generally, connectivity among computers and between computers and physical devices, as well as connectivity options, is proliferating unabated, which leads to society's demand for network-centric systems of increasing scale and demanding precision to take advantage of the increased connectivity to better organize collective and group interactions/behaviors. Since these systems are growing and will keep growing their complexity is increasing, which motivates the need to keep application programming relatively independent of the complex issues of distribution and scale (in the form of advanced software engineering practices and middleware solutions). In addition, systems of national scale, such as the US air traffic control system or power grid, will of necessity be incremental and developed by many different organizations contributing to a common solution on an as yet undefined common high-level platform and engineering development paradigm.

Despite all the advances in the past decades, there are no mature engineering principles, solutions, or established conventions to enable large-scale, network-centric systems to be repeatably, predictably, and cost effectively created, developed, validated, operated, and enhanced. As a result, we are witnessing a complexity threshold that is stunting our ability to create large-scale, network-centric systems successfully. Some of the inherent complexities that contribute to this complexity threshold include:

- Discrete platforms that must be scaled to provide seamless end-to-end solutions
- Components are heterogeneous yet they need to be integrated seamlessly
- Most failures are only partial in that they effect subsets of the distributed components
- Operating environments and configurations are dynamically changing
- Large-scale systems must operate continuously, even during upgrades
- End-to-end properties must be satisfied in time and resource constrained environments
- Maintaining system-wide QoS concerns is expected.

To address these complexities we must create and deploy middleware-oriented solutions and engineering principles as part of the commonly available new, network-centric software infrastructure that is needed to develop many different types of large-scale systems successfully.

**Specific R&D Challenges**

An essential part of what is needed to alleviate the inherent complexities outlined above is the integration and extension of ideas that have been found traditionally in network management, data management, distributed operating systems, and object-oriented programming languages. The payoff will be reusable middleware that significantly simplifies the development and evolution of complex network-centric systems. The following are specific R&D challenges associated with achieving this payoff:

- ***Demand for end-to-end QoS support, not just component-level QoS*** – This area represents the next great wave of evolution for advanced middleware. There is now widespread recognition that effective development of large-scale network-centric applications requires the use of COTS infrastructure and service components. Moreover, the usability of the resulting products depends heavily on the properties of the whole as derived from its parts. This type of environment requires *predictable*, *flexible*, and *integrated* resource management strategies, both within and between the pieces, that are understandable to developers, visible to users, and certifiable to system owners. Despite the ease of connectivity provided by middleware, however, constructing integrated systems remains hard since it requires significant customization of non-functional QoS properties, such as predictable latency/jitter/throughput, scalability, dependability, and security. In their most useful forms, these properties extend end-to-end and thus have elements applicable to

    - The network substrate
    - The platform operating systems and system services
    - The programming system in which they are developed
    - The applications themselves and
    - The middleware that integrates all these elements together.

Two basic premises underlying the push towards end-to-end QoS support mediated by middleware are that:

1. Different levels of service are possible and desirable under different conditions and costs and
2. The level of service in one property must be coordinated with and/or traded off against the level of service in another to achieve the intended overall results.

- ***Adaptive and reflective solutions that handle both variability and control*** – It is important to avoid "all or nothing" point solutions. Systems today often work well as long as they receive all the resources for which they were designed in a timely fashion, but fail completely under the slightest anomaly. There is little flexibility in their behavior, i.e., most of the adaptation is pushed to end-users or administrators. Instead of hard failure or indefinite waiting, what is required is either *reconfiguration* to reacquire the needed resources automatically or *graceful degradation* if they are not available. Reconfiguration and operating under less than optimal conditions both have two points of focus: individual and aggregate behavior. Moreover, there is a need for interoperability of control and management mechanisms needed to carry out such reconfiguration. To date interoperability concerns have focused on data interoperability and invocation interoperability across components. Little work has focused on mechanisms for controlling the overall behavior of the end-to-end integrated systems. "Control interoperability" is needed to complement data and invocation interoperability if we are to achieve something more than a collection of independently operating components. There are requirements for interoperable control capabilities to appear in the individual resources first, after which approaches can be developed to aggregate these into acceptable global behavior through middleware based multi-platform aggregate resource management services.

To manage the broader range of QoS demands for next-generation network-centric applications, middleware must become more adaptive and reflective.

*Adaptive middleware* is software whose functional and QoS-related properties can be modified either:

- *Statically*, *e.g.,* to reduce footprint, leverage capabilities that exist in specific platforms, enable functional subsetting, and minimize hardware/software infrastructure dependencies or
- *Dynamically*, *e.g.,* to optimize system responses to changing environments or requirements, such as changing component interconnections, power levels, CPU/network bandwidth, latency/jitter; and dependability needs.

In mission-critical systems, adaptive middleware must make such modifications dependably, *i.e.,* while meeting stringent end-to-end QoS requirements.

*Reflective middleware* goes further to permit automated examination of the capabilities it offers, and to permit automated adjustment to optimize those capabilities. Reflective techniques make the internal organization of systems–as well as the mechanisms used in their construction–both visible and manipulable for middleware and application programs to inspect and modify at run-time. Thus, reflective middleware supports more advanced adaptive behavior and more dynamic strategies keyed to current circumstances, *i.e.,* necessary adaptations can be performed autonomously based on conditions within the system, in the system's environment, or in system QoS policies defined by end-users.

▪ ***Toward more universal use of standard middleware*** – Today, it is too often the case that a substantial percentage of the effort expended to develop applications goes into building *ad hoc* and proprietary middleware substitutes, or additions for missing middleware functionality. As a result, subsequent composition of these *ad hoc* capabilities is either infeasible or prohibitively expensive. One reason why redevelopment persists is that it is still often relatively easy to pull together a minimalist *ad hoc* solution, which remains largely invisible to all except the developers. Unfortunately, this approach can yield substantial recurring downstream costs, particularly for complex and long-lived network-centric systems.

  One of the most immediate challenges is therefore to establish and eventually standardize middleware interfaces, which include QoS attributes. It is important to have a clear understanding of the QoS information so that it becomes possible to:

   1. Identify the users' requirements at any particular point in time and
   2. Understand whether or not these requirements are being (or even can be) met.

  It is also essential to aggregate these requirements, making it possible to form decisions, policies, and mechanisms that begin to address a more global information management organization. Meeting these requirements will require flexibility on the parts of both the application components and the resource management strategies used across heterogeneous systems of systems. A key direction for addressing these needs is through the concepts associated with managing adaptive behavior, recognizing that not all requirements can be met all of the time, yet still ensuring predictable and controllable end-to-end behavior.

- ***Leveraging and extending the installed base*** – In addition to the R&D challenges outlined above there are also pragmatic considerations, including incorporating the interfaces to various building blocks that are already in place for the networks, operating systems, security, and data management infrastructure, all of which continue to evolve independently. Ultimately, there are two different types of resources that must be considered:

   1. Those that will be fabricated as part of application development and
   2. Those that are provided and can be considered part of the substrate currently available.

While not much can be done in the short-term to change the direction of the hardware and software substrate that's installed today, a reasonable approach is to provide the needed services at higher levels of (middleware-based) abstraction. This architecture will enable new components to have properties that can be more easily included into the controllable applications and integrated with each other, leaving less lower-level complexity for application developers to address and thereby reducing system development and ownership costs. Consequently, the goal of next-generation middleware is not simply to build a better network or better security in isolation, but rather to pull these capabilities together and deliver them to applications in ways that enable them to realize this model of adaptive behavior with tradeoffs between the various QoS attributes. As the evolution of the underlying system components change to become more controllable, we can expect a refactoring of the implementations underlying the enforcement of adaptive control.

**Research Strategies**

The following three concepts are central to addressing the R&D challenges described above:

- *Contracts and adaptive meta-programming* – Information must be gathered for particular applications or application families regarding user requirements, resource requirements, and system conditions. Multiple system behaviors must be made available based on what is best under the various conditions. This information provides the basis for the contracts between users and the underlying system substrate. These contracts provide not only the means to specify the degree of assurance of a certain level of service, but also provide a well-defined, high-level middleware abstraction to improve the visibility of adaptive changes in the mandated behavior.
- *Graceful degradation* – Mechanisms must also be developed to monitor the system and enforce contracts, providing feedback loops so that application services can degrade gracefully (or augment) as conditions change, according to a prearranged contract governing that activity. The initial challenge here is to establish the idea in developers' and users' minds that multiple behaviors are both feasible and desirable. The next step is to put into place the additional middleware support–including connecting to lower level network and

operating system enforcement mechanisms–necessary to provide the right behavior effectively and efficiently given current system conditions.

- *Prioritization and physical world constrained load invariant performance* – Some systems are highly correlated with physical constraints and have little flexibility in some of their requirements for computing assets, including QoS. Deviation from requirements beyond a narrowly defined error tolerance can sometimes result in catastrophic failure of the system. The challenge is in meeting these *invariants* under varying load conditions. This often means guaranteeing access to some resources, while other resources may need to be diverted to insure proper operation. Generally collections of such components will need to be resource managed from a system (aggregate) perspective in addition to a component (individual) perspective.

Although it is possible to satisfy contracts, achieve graceful degradation, and globally manage some resources to a limited degree in a limited range of systems today, much R&D work remains. The research strategies needed to deliver these goals can be divided into the seven areas described below:

1. ***Individual QoS Requirements*** – Individual QoS deals with developing the mechanisms relating to the end-to-end QoS needs from the perspective of a single user or application. The specification requirements include multiple contracts, negotiation, and domain specificity. Multiple contracts are needed to handle requirements that change over time and to associate several contracts with a single perspective, each governing a portion of an activity. Different users running the same application may have different QoS requirements emphasizing different benefits and tradeoffs, often depending on current configuration. Even the same user running the same application at different times may have different QoS requirements, *e.g.,* depending on current mode of operation and other external factors. Such dynamic behavior must be taken into account and introduced seamlessly into next-generation distributed systems.

   General negotiation capabilities that offer convenient mechanisms to enter into and control a negotiated behavior (as contrasted with the service being negotiated) need to be available as COTS middleware packages. The most effective way for such negotiation-based adaptation mechanisms to become an integral part of QoS is for them to be "user friendly," *e.g.,* requiring a user or administrator to simply provide a list of preferences. This is an area that is likely to become domain-specific and even user-specific. Other challenges that must be addressed as part of delivering QoS to individual applications include:

   - Translation of requests for service among and between the various entities on the distributed end-to-end path
   - Managing the definition and selection of appropriate application functionality and system resource tradeoffs within a "fuzzy" environment and
   - Maintaining the appropriate behavior under composability.

   Translation addresses the fact that complex network-centric systems are being built in layers. At various levels in a layered architecture the user-oriented QoS must be translated into requests for other resources at a lower level. The challenge is how to accomplish this translation from user requirements to system services. A logical place to begin is at the application/middleware boundary, which closely relates to the problem of matching application resources to appropriate distributed system resources. As system resources change in significant ways, either due to anomalies or load, tradeoffs between QoS attributes (such as timeliness, precision, and accuracy) may need to be (re)evaluated to ensure an effective level of QoS, given the circumstances. Mechanisms need to be developed to identify and perform these tradeoffs at the appropriate time. Last, but certainly not least, a theory of effectively composing systems from individual components in a way that maintains application-centric end-to-end properties needs to be developed, along with efficient implementable realizations of the theory.

2. ***Run-time Requirements*** – From a system lifecycle perspective, decisions for managing QoS are made at design time, at configuration/deployment time, and/or at run-time. Of these, the run-time requirements are the most challenging since they have the shortest time scales for decision-making, and collectively we have the least experience with developing appropriate solutions. They are also the area most closely related to advanced middleware concepts. This area of research addresses the need for run-time monitoring, feedback, and transition mechanisms to change application and system behavior, *e.g.,* through dynamic reconfiguration, orchestrating degraded behavior, or even off-line recompilation. The primary requirements here are *measurement*, *reporting*, *control*, *feedback*, and *stability*. Each of these plays a significant role in delivering end-to-end QoS, not only for an individual application, but also for an aggregate system. A key part of a run-time environment centers on a permanent and highly tunable measurement and resource status service as a common middleware service, oriented to various granularities for different time epochs and with abstractions and aggregations appropriate to its use for run-time adaptation.

   In addition to providing the capabilities for enabling graceful degradation, these same underlying mechanisms also hold the promise to provide flexibility that supports a variety of possible behaviors, without changing the basic implementation structure of applications. This reflective flexibility diminishes the importance of many initial design decisions by offering late- and run-time-binding options to accommodate actual operating environments at the time of deployment, instead of only anticipated operating environments at design time. In addition, it anticipates changes in these bindings to accommodate new behavior.

3. ***Aggregate Requirements*** – This area of research deals with the system view of collecting necessary information over the set of resources across the system, and providing resource management mechanisms and policies that are aligned with the goals of the system as a whole. While middleware itself cannot manage system-level resources directly (except through interfaces provided by lower level resource management and enforcement mechanisms), it can provide the coordinating mechanisms and policies that drive the individual resource managers into domain-wide coherence. With regards to such resource management, policies need to be in place to guide the decision-making process and the mechanisms to carry out these policy decisions.

   Areas of particular R&D interest include:

   - *Reservations*, which allow resources to be reserved to assure certain levels of service
   - *Admission control mechanisms*, which allow or reject certain users access to system resources
   - *Enforcement mechanisms* with appropriate scale, granularity and performance and
   - *Coordinated strategies and policies* to allocate distributed resources that optimize various properties.

   Moreover, policy decisions need to be made to allow for varying levels of QoS, including whether each application receives guaranteed, best-effort, conditional, or statistical levels of service. Managing property composition is essential for delivering individual QoS for component based applications, and is of even greater concern in the aggregate case, particularly in the form of layered resource management within and across domains.

4. ***Integration Requirements*** – Integration requirements address the need to develop interfaces with key building blocks used for system construction, including the OS, network management, security, and data management. Many of these areas have partial QoS solutions underway from their individual perspectives. The problem today is that these partial results must be integrated into a common interface so that users and application developers can tap into each, identify which viewpoint will be dominant under which conditions, and support the tradeoff management across the boundaries to get the right mix of attributes. Currently, object-oriented tools working with middleware provide end-to-end syntactic interoperation, and relatively seamless linkage across the networks and subsystems. There is no *managed* QoS, however, making these tools and middleware useful only for resource rich, best-effort environments.

   To meet varying requirements for integrated behavior, advanced tools and mechanisms are needed that permit requests for *different* levels of attributes with different tradeoffs governing this interoperation. The system would then either provide the requested end-to-end QoS, reconfigure to provide it, or indicate the inability to deliver that level of service, perhaps offering to support an alternative QoS, or triggering application-level adaptation. For all of this to work together properly, multiple dimensions of the QoS requests must be understood within a common framework to translate and communicate those requests and services at each relevant interface. Advanced integration middleware provides this common framework to enable the right mix of underlying capabilities.

5. ***Adaptivity Requirements*** – Many of the advanced capabilities in next-generation information environments will require adaptive behavior to meet user expectations and smooth the imbalances between demands and changing environments. Adaptive behavior can be enabled through the appropriate organization and interoperation of the capabilities of the previous four areas. There are two fundamental types of adaptation required:

   - Changes beneath the applications to continue to meet the required service levels despite changes in resource availability and
   - Changes at the application level to either react to currently available levels of service or request new ones under changed circumstances.

   In both instances, the system must determine if it needs to (or can) reallocate resources or change strategies to achieve the desired QoS. Applications need to be built in such a way that they can change their QoS demands as the conditions under which they operate change. Mechanisms for reconfiguration need to be put into place to implement new levels of QoS as required, mindful of both the individual and the aggregate points of view, and the conflicts that they may represent.

   Part of the effort required to achieve these goals involves continuously gathering and instantaneously analyzing pertinent resource information collected as mentioned above. A complementary part is providing the algorithms and control mechanisms needed to deal with rapidly changing demands and resource availability profiles and configuring these mechanisms with varying service strategies and policies tuned for different environments. Ideally, such changes can be dynamic and flexible in handling a wide range of conditions, occur intelligently in an automated manner, and can handle complex issues arising from composition of adaptable components. Coordinating the tools and methodologies for these capabilities into an effective adaptive middleware should be a high R&D priority.

6. ***System Engineering Methodologies and Tools*** – Advanced middleware by itself will not deliver the capabilities envisioned for next-generation embedded environments. We must also advance the state of the system engineering discipline and tools that come with these

advanced environments used to build complex distributed computing systems. This area of research specifically addresses the immediate need for system engineering approaches and tools to augment advanced middleware solutions. These include:

- *View-oriented or aspect-oriented programming techniques,* to support the isolation (for specialization and focus) and the composition (to mesh the isolates into a whole) of different projections or views of the properties the system must have. The ability to isolate, and subsequently integrate, the implementation of different, interacting features will be needed to support adapting to changing requirements.
- *Design time tools and models*, to assist system developers in understanding their designs, in an effort to avoid costly changes after systems are already in place (this is partially obviated by the late binding for some QoS decisions referenced earlier).
- *Interactive tuning tools*, to overcome the challenges associated with the need for individual pieces of the system to work together in a seamless manner
- *Composability tools*, to analyze resulting QoS from combining two or more individual components
- *Modeling tools for developing system performance models* as adjunct means (both online and offline) to monitor and understand resource management, in order to reduce the costs associated with trial and error
- *Debugging tools*, to address inevitable problems.

7. ***Reliability, Trust, Validation, and Certifiability*** – The dynamically changing behaviors we envision for next-generation large-scale, network-centric systems are quite different from what we currently build, use, and have gained some degrees of confidence in. Considerable effort must therefore be focused on validating the correct functioning of the adaptive behavior, and on understanding the properties of large-scale systems that try to change their behavior according to their own assessment of current conditions, before they can be deployed. But even before that, longstanding issues of adequate reliability and trust factored into our methodologies and designs using off-the-shelf components have not reached full maturity and common usage, and must therefore continue to improve. The current strategies organized around anticipation of long life cycles with minimal change and exhaustive test case analysis are clearly inadequate for next-generation dynamic systems with stringent QoS requirements.

## Concluding Remarks

In this age of IT ubiquity, economic upheaval, deregulation, and stiff global competition it has become essential to decrease the cycle-time, level of effort, and complexity associated with developing high-quality, flexible, and interoperable large-scale, network-centric systems. Increasingly, these types of systems are developed using reusable software (middleware) component services, rather than being implemented entirely from scratch for each use. Middleware was invented in an attempt to help simplify the software development of large-scale, network-centric computing systems, and bring those capabilities within the reach of many more developers than the few experts at the time who could master the complexities of these environments. Complex system integration requirements were not being met from either the *application perspective*, where it was too difficult and not reusable, or the *network or host operating system perspectives*, which were necessarily concerned with providing the communication and endsystem resource management layers, respectively.

Over the past decade, middleware has emerged as a set of software service layers that help to solve the problems specifically associated with heterogeneity and interoperability. It has also contributed considerably to better environments for building network-centric applications and managing their distributed resources effectively. Consequently, one of the major trends driving researchers and practioners involves moving toward a multi-layered architecture (applications, middleware, network and operating system infrastructure), which is oriented around application composition from reusable components, and away from the more traditional architecture, where applications were developed directly atop the network and operating system abstractions. This middleware-centric, multi-layered architecture descends directly from the adoption of a network-centric viewpoint brought about by the emergence of the Internet and the componentization and commoditization of hardware and software.

Successes with early, primitive middleware have led to more ambitious efforts and expansion of the scope of these middleware-oriented activities, so we now see a number of distinct layers of the middleware itself taking shape. The result has been a deeper understanding of the large and growing issues and potential solutions in the space between:

- Complex distributed application requirements and
- The simpler infrastructure provided by bundling existing network systems, operating systems, and programming languages.

There are significant limitations with regards to building these more complex systems today. For example, applications have increasingly more stringent QoS requirements. We are also discovering that more things need to be integrated over conditions that more closely resemble a volatile, changing Internet, than they do a stable backplane.

One problem is that the playing field is changing constantly, in terms of both resources and expectations. We no longer have the luxury of being able to design systems to perform highly specific functions and then expect them to have life cycles of 20 years with minimal change. In fact, we more routinely expect systems to behave differently under different conditions, and complain when they just as routinely do not. These changes have raised a number of issues, such as end-to-end oriented adaptive QoS, and construction of systems by composing off-the-shelf parts, many of which have promising solutions involving significant new middleware-based capabilities and services.

In the brief time we met at the SDP workshop, we could do little more than summarize and lend perspective to the many activities, past and present, that contribute to making middleware technology an area of exciting current development, along with considerable opportunity and unsolved challenging problems. This breakout group summary also provides a more detailed discussion and organization for a collection of activities that SDP workshop participants believe represent the most promising future R&D directions of middleware for large-scale, network-centric systems. Downstream, the goals of these R&D activities are to:

1. Reliably and repeatably construct and compose network-centric systems that can meet and adapt to more diverse, changing requirements/environments and
2. Enable the affordable construction and composition of the large numbers of these systems that society will demand, each precisely tailored to specific domains.

To accomplish these goals, we must overcome not only the technical challenges, but also the educational and transitional challenges, and eventually master and simplify the immense complexity associated with these environments, as we integrate an ever growing number of hardware and software components together via middleware.

**List of Participants**

| First Name | Last Name | Affiliation |
|---|---|---|
| | | |
| Rajeev | Alur | University of Pennsylvania |
| Don | Batory | University of Texas, Austin |
| Ira | Baxter | Semantic Designs |
| Barry | Boehm | University of Southern California |
| Richard | Buchness | Boeing |
| Betty | Cheng | Michigan State University |
| Karl | Crary | Carnegie-Mellon University |
| Joseph | Cross | Lockheed-Martin |
| Eric | Dashofy | University of California, Irvine |
| Gary | Daugherty | Rockwell Collins |
| Premkumar | Devanbu | University of California, Davis |
| Laura | Dillon | Michigan State University |
| Tzilla | Elrad | Illinois Institute of Technology |
| Kathi | Fisler | Worcester Polytechnic Institute |
| R. | Gamble | University of Tulsa |
| Susan | Gerhart | Embry-Riddle Aeronautical University |
| Susan | Graham | University of California, Berkeley |
| Cordell | Green | Kestrel Institute |
| Paul | Hudak | Yale University |
| Jim | Hugunin | Xerox PARC |
| Daniel | Jackson | MIT |
| Ralph | Johnson | University of Illinois, Urbana-Champagne |
| Philip | Johnson | University of Hawaii |
| Samuel | Kamin | University of Illinois, Urbana-Champagne |
| Gabor | Karsai | Vanderbilt University |
| Gregor | Kiczales | University of British Columbia/Xerox |
| Kane | Kim | University of California, Irvine |
| Mieczyslaw | Kokar | Northeastern University |
| Alkis | Konstantellos | European Commission, IST |
| Shriram | Krishnamurthi | Brown University |
| Robert | Laddaga | MIT |

| | | |
|---|---|---|
| James | Larus | Microsoft |
| Insup | Lee | University of Pennsylvania |
| Karl | Lieberherr | Northeastern University |
| Joe | Loyall | BBN |
| Tommy | McGuire | University of Texas, Austin |
| Marija | Mikic-Rakic | University of Southern California |
| Michael | Mislove | Tulane University |
| Priya | Narasimhan | Carnegie-Mellon University |
| Bob | Neches | University of Southern California |
| Dewayne | Perry | University of Texas, Austin |
| Calton | Pu | Georgia Tech |
| William | Pugh | University of Maryland, College Park |
| Joy | Reed | Armstrong Atlantic State University |
| John | Reekie | University of California, Berkeley |
| Steven | Reiss | Brown University |
| Robert | Riemenschneider | SRI International |
| Martin | Rinard | MIT |
| Paul | Robertson | Dynamic Object Language Labs |
| David | Sharp | Boeing |
| Kang | Shin | University of Michigan |
| Massoud | Sinai | Boeing |
| Douglas | Smith | Kestrel Institute |
| Scott | Smolka | Reactive Systems, Inc. |
| David | Stewart | Embedded Research Solutions |
| Kurt | Stirewalt | Michigan State University |
| Brian | Williams | MIT |
| | | |
| **Government Invitees** | | |
| | | |
| Abdullah | Aljabri | NASA |
| Frank | Anger | NSF |
| Daniel | Dvorak | NASA |
| Helen | Gigley | NCO/ITRD |
| Helen | Gill | NSF |
| Thuc | Hoang | DOE |
| Sally | Howe | NCO/ITRD |
| Michael | Masters | U.S. Naval Surface Warfare Center |
| Kenny | Meyer | NASA |
| Vijay | Raghavan | DARPA |

| | | |
|---|---|---|
| Steven | Ray | NIST |
| Spencer | Rugaber | NSF |
| Doug | Schmidt | DARPA |
| Frank | Sledge | NCO/ITRD |
| | | |
| **Program Committee** | | |
| | | |
| Benjamin | Pierce | University of Pennsylvania |
| Adam | Porter | University of Maryland, College Park |
| Rick | Schantz | BBN |
| Charles | Simonyi | Microsoft |
| Janos | Sztipanovits | Vanderbilt University |
| Don | Winter | Boeing |
| | | |
| **Volunteers** | | |
| | | |
| Aditya | Agrawal | Vanderbilt/ISIS |
| Mark | Briski | Vanderbilt/ISIS |
| Michele | Codd | Vanderbilt/ISIS |
| Brandon | Eames | Vanderbilt/ISIS |
| Jeff | Gray | Vanderbilt/ISIS |
| Jung-Min | Kim | U. Maryland |
| Lorene | Morgan | Vanderbilt/ISIS |
| Tal | Pasternak | Vanderbilt/ISIS |
| Jason | Scott | Vanderbilt/ISIS |
| Jonathan | Sprinkle | Vanderbilt/ISIS |
| Eric | Wohlstad | UC Davis |