Software-Intensive Systems

European Commission -
US National Science Foundation

Strategic Research Workshop

Engineering Software-Intensive Systems

Edinburgh, UK,
22-23 May 2004

Workshop Report and
Recommendations

Report on the EU/NSF Strategic Workshop on

Engineering Software-Intensive Systems

May 22-23, 2004
Edinburgh, GB

Martin Wirsing
Workshop coordinator and editor

Rémi Ronchaud
Organisational coordinator

Preface

This report contains a summary of the results of the workshop on "Engineering Software-Intensive Systems". The workshop was part of a series of EU-NSF strategic research workshops organised by ERCIM under the auspices the European Commission (programme IST-FET) and the US National Science Foundation (CISE-NSF division) to identify key research challenges and opportunities in information technologies. These workshops are intended to facilitate brainstorming and awareness about potential breakthroughs in innovative domains, stimulate research activities and scientific discussions of mutual interest.

The workshop on "Engineering Software-Intensive Systems**"** took place in Edinburgh, Scotland, on May 23-24, 2004, and was held as a co-located event of the International Conference on Software Engineering, ICSE 2004. Participation in the workshop was by invitation only. About 20 leading experts from Europe, the United States and Australia participated in the workshop. The program consisted of presentations and discussions of future R&D directions, challenges, and visions in the emerging area of Engineering Software-Intensive Systems.

As coordinator of the workshop I would like to thank the chairman of ICSE Anthony Finkelstein for hosting the workshop. I am extremely grateful to Rémi Ronchaud from ERCIM for all his invaluable work and effort with the preparation and organisation of the workshop. My warmest thanks go to Axel Rauschmayer for his help in editing this report and to Hubert Baumeister for his useful comments on this report. Finally, I would like to thank all participants for their inspiring contributions.

Munich, December 2004 Martin Wirsing

Summary of Workshop Results

Software has become a key feature of a rapidly growing range of products and services from all sectors of economic activity. Software-intensive systems include large-scale heterogeneous systems, embedded systems for automotive applications, telecommunications, wireless ad hoc systems, business applications with an emphasis on web services etc. Our daily lives depend on complex software-intensive systems, from banking to communications to transportation to medicine.

In the near future, software-intensive systems will exhibit adaptive and anticipatory behaviour; they will process knowledge and not only data, and change their structure dynamically. Software-intensive systems will act as global computers in highly dynamic environments and will be based on and integrated with service-oriented and pervasive computing.

However, actual practice shows that the techniques for engineering software-intensive systems suffer from many severe deficiencies in quality and methodological shortcomings:

- pragmatic modeling languages and techniques have no clean scientific foundations which inhibits the construction of powerful analysis and development tools;
- formal approaches are not well-integrated with pragmatic methods and do not scale up to complex software-intensive systems;
- aspects such as change, adaptation, heterogeneity, quality of service, security, trust, and highly dynamic and unpredictable environments, are important for software-intensive systems, but are not well supported by actual engineering methods.

The strategic research workshop on “Engineering Software-Intensive Systems” was organised in Edinburgh, Scotland, on May 23-24, 2004, with the objective to present and discuss future R&D directions, challenges, and visions in the emerging area of software-intensive systems. About 20 leading experts from Europe, the United States and Australia participated in the workshop and identified research issues and challenges.

The grand challenge is to develop practically useful and theoretically well-founded principles, methods and tools for engineering high-quality software-intensive systems.

Mastering the complexity of software-intensive systems requires a combined effort for foundational research and new engineering techniques that are based on mathematically well-founded theories and approaches. The new methods should support the whole system life cycle including requirements, design, implementation, maintenance, reconfiguration and adaptation. Research is required for:

- developing innovative engineering support for software-intensive systems to ensure required levels of quality and trust;
- putting change and adaptation at all levels of system development;
- developing a science of software-intensive systems;
- bridging the gap between pragmatic development techniques and foundational validation and verification methods.

Table of Contents

Challenges for Engineering Software-Intensive Systems

1. Introduction

Software has become a key feature of a rapidly growing range of products and services from all sectors of economic activity. Software-intensive systems include large-scale heterogeneous systems, embedded systems for automotive applications, telecommunications, wireless ad hoc systems, business applications with an emphasis on web services etc. Our daily lives depend on complex software-intensive systems, from banking to communications to transportation to medicine. Software technology is a driving factor for many high tech products; competence in software technology defines more and more the innovation capability of the whole industry.

On the other hand, software is undergoing a fast technological progress where object-orientation, service-orientation, modeling languages such as UML, programming and mark-up languages such as Java and XML, and CASE tools have considerably influenced the system development techniques of today. Also formal techniques have undergone a steep development during the last years. Based on formal foundations and deep theoretical results, methods and tools have been developed to support specification, design, validation and verification of software systems. Many other formal specification and verification techniques have been applied to non-trivial case studies and are used in practice e.g. for the development of safety critical systems.

However, actual practice shows that the techniques for engineering software-intensive systems suffer from many severe deficiencies in quality and from methodological shortcomings:

- pragmatic modeling languages and techniques have no clean scientific foundations which inhibits the construction of powerful analysis and development tools;
- formal approaches are not well-integrated with pragmatic methods and do not scale up to complex software-intensive systems;
- aspects such as change, adaptation, heterogeneity, quality of service, security, trust, and highly dynamic and unpredictable environments, are important for software-intensive systems, but are not well supported by actual engineering methods.

2. The workshop and its objectives

In May 2004 the workshop on “Engineering Software-Intensive Systems**”** was held as a co-located event of ICSE in Edinburgh, Scotland. The workshop was organised by ERCIM, the European Research Consortium for Informatics and Mathematics with the support of the US National Science Foundation (CISE-NSF division) and the European Commission (programme IST-FET). Participation in the workshop was by invitation only. About 20 leading experts from Europe and the United States participated in the workshop.

The objectives of the EU-NSF workshop were threefold:

- to discuss the state of the art in engineering software-intensive systems, to evaluate potential or partial solutions that have been proposed, and to analyze why some ideas were or were not successful;

- to propose and discuss in a pro-active way innovative approaches and solutions of the problems and challenges of software-intensive systems and to present visionary and explorative perspectives and bold ideas for modeling, programming, constructing, validating, and verifying software-intensive systems;

- to show how pragmatic methods in software-intensive systems engineering can be integrated with and enhanced by the results of foundational research to handle the new problems posed by, among others, the requirements of embedded systems, the different levels of component and system granularity, the heterogeneity of components, the use of distribution, mobility and communication, and the request for appropriate human-interface support.

3. The challenges for software-intensive systems engineering

The ongoing decrease in the cost of microprocessors is fueling a "silent revolution" in computing; far more computing cycles are now devoted to the control of devices, such as anti-lock braking systems and cardiac defibrillators, than to the running of traditional applications found in desktop computers. Software systems are becoming increasingly distributed and decentralized. Applications are composed as dynamic federations of autonomous and evolving components. Examples are emerging in the area of ambient intelligence, pervasive ubiquitous computing, and web services.

In the near future, such control- and software-intensive systems will exhibit adaptive and anticipatory behaviour; they will process knowledge and not only data, and change their structure dynamically. Software-intensive systems will act as global computers in highly dynamic environments. They will be based on and integrated with the service-oriented computing paradigm where services are understood as autonomous, platform-independent computational entities that can be described, published, discovered, and dynamically assembled to develop massively distributed, interoperable, evolvable systems.

These newly emerging software-intensive systems present unique challenges to their developers. On the one hand, they must meet very stringent guarantees of dependability and reliability, both for business as well as safety reasons. On the other hand, their development requires interaction between control system and software engineers, whose differing backgrounds (continuous vs. discrete mathematics) are often a source of confusion and misunderstanding.

Mastering the complexity of software-intensive systems requires a combined effort for foundational research and new engineering techniques that are based on mathematically well-founded theories and approaches. The new methods should support the whole system life cycle including requirements, design, implementation, maintenance, reconfiguration and adaptation. The grand challenge is to develop practically useful and theoretically well-

founded principles, methods and tools for engineering high-quality software-intensive systems.
Research is required for:

- developing innovative engineering support for software-intensive systems ensuring required levels of quality and trust including properties such as reliability, safety, security, correctness, performance, availability, and dependability;
- putting change and adaptation at all levels of system development including change of requirements, technologies and environments, as well as self-adaptation and self-organization of systems;
- developing a science of software-intensive systems;
- bridging the gap between pragmatic development techniques and foundational validation and verification methods.

In the following, these research issues are presented in more detail.

3.1 Developing innovative engineering support for software-intensive systems ensuring required levels of quality and trust.

Software-intensive systems are complex programmable systems exhibiting properties such as adaptive behaviour and dynamically changing structure. Key issues for *implementation* are technical aspects such as mastering size, complexity and change, as well as economic aspects of middleware, business models, and costs. Key drivers for *acceptance* of such systems are satisfaction of user requirements, interoperability of products, systems and applications, and quality issues such as usability, testability, reliability but also security and safety.

Although software engineering methods have matured significantly in the past twenty years, and have also benefited from advances in object-oriented and component-based development methods, they do not support well enough the particular problems of software-intensive systems; problems of actual engineering methods are e.g. insufficient requirements capture and validation, inadequate treatment of quality and trust properties, or insufficient support of compositionality and interoperability.

A promising approach for solving these problems is model-based development: models can serve as a vehicle for communicating information between business process engineers, control engineers and computer scientists, and can also provide an additional basis for pre-implementation validation of requirements and quality properties as well as for automatic generation of source code. However, this paradigm for software-intensive system development is still in its infancy, with many issues requiring further study for its full and radical potential to be realized. These include:

- *Distributed control systems:* Enhancing control-system modeling languages to support controller interaction is essential for the future. A complicating feature is the fact that although controllers are digital and independent, the environment with which they interact is continuous and shared.
- *Capturing and formalizing requirements:* Capturing and debugging requirements is one of the main issues for developing high quality software-intensive systems. Usable formal notations for requirements would enable mechanized support for checking requirements against models. Requirements should include a mixture of discrete and continuous mathematics and be executable, so that they may also be debugged.

- *Compositional design models:* The notion of composition is necessary to deal effectively with designing large complex systems. Compositionality of designs facilitates sharing of artefacts and should support both homogeneous models and heterogeneous models.
- *Techniques and tools for model debugging, validation and verification.* Methods for assessing model correctness need to be developed, including model checking, interactive verification as well as analysis techniques based on typing, program slicing and constraint solving. Techniques lying "midway" between *ad hoc* testing and full formal verification should also be investigated.
- *Certification based on design:* To be able to evaluate the quality of software-intensive systems, it is necessary to develop a certification process based on sound scientific foundations which should make it possible to measure quantitatively how well a system meets its requirements.

Security and *trust* are main challenges for the software-intensive systems of tomorrow. These systems will be more complex, built out of more and more mismatched components than today's systems, and continue to be rife with bugs. Attackers only need to find one bug to exploit, but defenders have to find and fix them all. In addition, the environments in which systems are deployed will be more unpredictable and more malicious. As a consequence, security will be more and more important for software-intensive systems and will have to be built into the system by design. Compositional techniques are needed, e.g. to discover interface mismatches that lead to security flaws or to anticipate emergent abusive behaviour. Security-respecting software design principles have to be developed such as defence in depth, principle of least privilege, or security by default. A further grand challenge is the development of trustworthy software where not only security is considered but also further aspects including reliability, privacy, and usability.

Summarizing, research is needed for constructing very high quality requirements specifications and compositional design models, and for guaranteeing critical functional and non-functional system properties, and in particular for guaranteeing security and trust. Research topics include:

- new methods for capturing, formalising, and validating requirements;
- modelling languages for distributed control supporting compositional design;
- techniques for guaranteeing quality properties such as security, safety, trust, reputation, fault tolerance, behavioural, and real-time properties;
- composing heterogeneous components where heterogeneity e.g. deals with interaction, execution platforms, and legacy code.

3.2 Putting change and adaptation at all levels of system development.

Software systems change or perish in response to changing business needs. In addition, they are required to evolve dynamically as new components are introduced and as existing components are removed or fail. Software systems become more complex and fragile as they age, and more so as network, hardware, and business innovations emerge. In particular, software-intensive systems are subject to continuous and dynamic change of structure, requirements, implementation technology and operating environment. As for engineering

quality and trust, today's software engineering practices do not provide adequate support for system change and evolution of business needs, requirements, platforms, and technologies.

It is therefore necessary to advance beyond the "engineering" metaphor for software development, and focus more on support for change. Specific areas of opportunity are:

- *Self-organising software systems:* The objective of self-organising systems is to minimise the degree of explicit management necessary for construction and subsequent evolution whilst preserving the required properties and operational constraints of the system. Sound approaches are needed to support system composition on the fly and dynamically reconfigurable services, and to help manage change, especially where it is required to take place in a deployed system.
- *Language support for change:* present-day languages focus on static design and offer few mechanisms to support design evolution. It is necessary to develop high-level languages and mechanisms that can express fine and coarse grained evolution, cope with radical changes in design, and support the coexistence of multiple running versions.
- *Tool support for change:* tools to model, analyse, and transform evolving software systems are needed. Such tools would support co-evolution of artefacts at various levels of abstraction, from code through design to requirements. In particular, these tools should be able to track change and help to modify and store development artefacts.
- *Methodologies for supporting change:* although "agile" processes are gaining acceptance, there is still a great deal of scepticism concerning their applicability to conventional projects. Research is needed to determine which software practices are most effective in coping with high rates of change, to reverse engineer the informal processes and to formalise processes e.g. by defining appropriate meta-models.

Summarizing, research is needed to cope with change and evolution at all levels of system development including change of requirements, technologies and environments, as well as self-adaptation and self-organization of systems. Research topics include

- modelling paradigms supporting change;
- methods for easy change through reorganization of code;
- techniques for making systems self-adaptable and self-organizing.

3.3 Developing a science of software-intensive systems.

Several new computing paradigms for software-intensive systems are emerging such as global, pervasive, and service-oriented computing. Their integration into a scientifically well-founded engineering approach for software-intensive systems raises a number of fundamental research questions which should lead to a science of software-intensive systems.

For example in the service-oriented computing paradigm, services are understood as autonomous, platform-independent computational entities that can be described, published, discovered, and dynamically assembled for developing massively distributed, interoperable, evolvable systems. Today, services are being delivered on a variety of computing platforms, mostly through the web, personal digital assistants (PDA), and mobile phones. In future scenarios, it will be possible to develop new applications by federating dynamically available services, without relying on a unique central control authority which would be in charge of

the entire process. Rather, a myriad of distributed processes form loosely coupled organizations that provide dynamically composable services. Federations will be formed by dynamically exposing new services and composing them in a self-organizing manner with the aid of an active and cooperative support infrastructure.

In addition, pervasive computing must deal with inherently noisy, imprecise, and inferred information which may be more or less accurate. Extracting information from real-world data requires a deep understanding of the physical, logical, and social constraints controlling external actions, with software able to reason about richly-connected models of tasks and information. Moreover, for software-intensive systems one will need ways to assess the environment and the contexts surrounding them, and adapt to their changes.

To make software-intensive systems of these kinds possible many scientific problems have to be addressed including:

- effectively supporting fully decentralized (peer-to-peer) software architectures even in fully dynamic scenarios where nodes are mobile and connected via wireless links;
- identifying flexible binding mechanisms to support dynamic and self-organizing component federations, via service discovery, brokering, and negotiation, etc.;
- specifying, verifying, negotiating, and monitoring quality of services,
- specifying and supporting the notion of context-awareness to support context-aware services;
- providing ways to design systems stable under perturbation and able to recognise and react to incorrect decisions, to model dynamic component federations and reason about their properties;
- providing abstraction, refinement, and interoperability concepts for dynamic process and component federations.

More generally, foundational research is needed to develop a comprehensive theory for modeling and analysing software-intensive systems in a systematic system development. Research topics include

- theories of views and abstractions of systems such as data view, process view, distribution view, quality of service view, context view, security view, or deployment view;
- appropriate abstraction, refinement, and implementation relations for analysing and comparing different views;
- integrating and relating different system models, e.g. core models based on explicit messages/shared repositories with models for quality of service, transactions with compensation, and dynamic reconfiguration.

3.4 Bridging the gap between pragmatic development techniques and foundational validation and verification methods.

One of the main problems of quality assurance of software-intensive systems consists of the above mentioned gap between pragmatic development techniques and foundational validation and verification methods. Today, pragmatic system development does not provide powerful, semantically well-founded validation and verification techniques. In particular, actual pragmatic development methods do not provide much feedback to requirements and design decisions although developers need such feedback as early as possible. This problem is partly due to the fact that existing tools for automated and interactive analysis require specialised

expert knowledge, are difficult to use, or do not scale up to the complexity of actual applications.

Promising approaches for bridging this gap are model-based validation, model checking and appropriate interactive verification techniques.

Model-based validation is used for testing and verifying executable abstract models. It consists of developing tests, executable models, and formal specifications of requirements and designs and to validate them e.g. by using testing, model checking, or interactive verification techniques.

Model checking is an automated technique for formally verifying concurrent systems where the validity of a system property is checked by exploring the full state space of the system. Model checking is successfully used in hardware development and begins now to be used also for validating software properties. This approach is well-suited for finite state control systems including real time and probabilistic systems but it is difficult to use for data-intensive applications. Main difficulties for applying model checking is scalability: to cope with the "state explosion" problem of the analysis of large systems and the infinite state spaces of data-intensive applications, abstraction techniques and combinations with other analysis techniques (such as abstract interpretation, constraint solving and interactive verification) are needed.

Interactive verification techniques have a broad range of data-intensive and control-intensive applications but are only successful for small applications e.g. in e-commerce or in the area of certification of software products. For larger software systems, full code verification is unrealistic. A feasible strategy is to verify design models since models are of considerably smaller size than source code although models of larger systems may also be too large. Then model verification needs the reduction of the model to a small "verification kernel" whose properties hold for the full model.

Summarizing, research is needed for enhancing pragmatic system development with powerful, semantically well-founded validation and verification techniques and to scale up novel automated analysis methods to the complexity of actual applications.

Research topics include

- novel techniques for requirements validation, design testing, early test generation, enhanced model checking, and automated verification;
- methods for automatically assessing model correctness;
- scaling up verification techniques through model transformation techniques and through combination of model checking and interactive verification with each other as well as with other analysis techniques;
- the quest for "disappearing formal methods" by providing user-friendly specification and verification languages, tailoring model checking and verification to modeling languages, and developing novel transformation techniques for hiding details of formal specification and analysis.

Position Statements

Don Batory: Relational query processing as a basis for software design

Software design is a poorly understood art-form. As long as it remains so, our abilities to automate key tasks in software development are fundamentally limited. Much of the focus today is on designing, building, and understanding one-of-a-kind systems. While there may be strong justification for doing so, this orientation has fundamental limitations. Sciences have never been created by studying singleton entities: theories of atomic physics arose out of studies of all kinds of atomic phenomena, not just the study of one kind of atom. Theories of astronomy are based on the study of many stars and galaxies, not just a single star or galaxy. And so on. A science for software design will arise out of the study of many related systems; they will be predictive and constructive theories of how software in a particular domain can be automatically constructed and evaluated. Formalizing well-understood processes in a particular domain for the purposes of mechanization will raise the level of software quality, reduce maintenance costs, improve our ability to certify important properties of the software that is produced. This can be done because we are not solving general problems without constraints (which historically rarely succeed), but instead we are solving well-defined and well-understood subproblems with clear and specific constraints.

One of the most significant results in automated software production is *relational query processing (RQP)*. A data retrieval program is specified declaratively in the SQL language. A parser maps this specification to a relational algebra expression, a query optimizer rewrites this expression into a semantically equivalent expression with better performance characteristics, and a code generator translates the expression into an efficient executable. Query evaluation *programs* are represented algebraically as compositions of relational algebra operators. (This is a classical example of *compositional programming*). This is also one of the few significant examples of *automatic programming* — transforming a declarative specification into an efficient program.

The RQP paradigm has the right look and feel for the basis of a science of software design. The domain of query evaluation programs are defined by algebraic expressions. Identities among operators are the basis for optimizing expressions (and hence program designs) automatically. Different representations of these programs (e.g., cost models and code) are derived automatically from their algebraic definitions. And after being scrutinized for over 25 years, the relational algebra approach has survived the test of time and has been broadened to include many new operators beyond the traditional operators that were identified in the early 1970s.

I believe that the RQP paradigm holds the key to a Science of Software Design. The challenge is to show how it can generalize to other domains and be able to achieve improved software quality, reduced costs and better certification.

Ira Baxter: Design maintenance systems

Software Systems are growing in size, complexity and utility. This means not only increased complexity in initial engineering but increased duration and complexity of maintaining the software in the face of continuously changing functional, performance, context and technological foundations.

Problems: We desperately need software design/lifecycle models that completely merge the design and maintenance phases, so that information obtained during « design » is passed painlessly and seamlessly to « maintenance », and retained accurately. Such knowledge requires we be able to capture problem domain descriptions and corresponding solutions at various levels of abstraction and compose them sensibly and straightforwardly. We need methods to scalably automate analysis and inference over large complex artefacts because people cannot do this reliably. We need techniques enabling engineers to specify system changes explicitly and implement these incrementally, using the captured design knowledge to aid the modification. And we need ways for large teams of engineers to reliably make un-coordinated changes in parallel.

Solutions: We believe the key technologies to achieve this include generalized compiling plus program transformation over domain-specific languages at multiple levels of abstraction, coupled with design decision and rationale capture. Formal semantics will provide the means for gluing specifications in multiple formalisms together in a well-founded whole. Design decisions as chosen-branch of multiply-refinable concepts will be implicitly satisfy functional semantics and explicitly justified by performance specifications. Reasoning costs will be alleviated by applying compiler-like optimizations and parallel computing to symbolic inference processes. Capture of design histories will enable incremental updates using distributive algebraic laws derived from domain semantics. Design information must be captured in databases with long-term transactions to enable long-term changes. (Semantic Designs is attempting to reify some of these ideas in commercial tools).

Ed Brinksma: Experimental methods for formal design

Formal models are to play an increasing role in controlling the quality and complexity of software-intensive systems, as is apparent from the growing prominence of the qualifier "model-driven", as in model-driven architecture, model-driven test generation, etc. Research in computer science has concentrated on what can be achieved on the basis of good models, and not on the issue of how to obtain good models.

In practice, models are often the result of some form of "hacking", and the results of their application correspondingly unreliable. We argue that not formal but experimental and informal methods are the key to obtaining good models that form the basis of formal design activities.

Rance Cleaveland: Model-based development for control software development

The ongoing decline in the cost of microprocessors is fueling a "silent revolution" in computing; far more computing cycles are now devoted to the control of devices, such as anti-lock braking systems and cardiac defibrillators, than to the running of traditional applications found in desktop computers. These newly emerging control- and software-intensive systems present unique challenges to their developers. On the one hand, they must meet very stringent guarantees of dependability and reliability, both for business as well as safety reasons. On the other hand, their development requires interaction between controls and software engineers, whose differing backgrounds (continuous vs. discrete mathematics) are often a source of confusion and misunderstanding. Model-based design is showing promise as a solution to this problem: models can serve as a vehicle for communicating information between engineers and computer scientists and can also provide a basis for pre-implementation validation as well as for automatic generation of source code. However, this paradigm for control software development is still in its infancy, with many issues requiring further study for its full and radical potential to be realized. These include:

- **Distributed control systems.** Enhancing control-system modeling languages to support controller interaction is essential for the future. A complicating feature is the fact that although controllers are digital and independent, the environment with which they interact is continuous and shared.
- **Formalized requirements.** Usable formal notations for capturing requirements, and checking requirements against models, would enable mechanized support for checking requirements against models. Requirements should include a mixture of discrete and continuous mathematics and be executable, so that they may also be debugged.
- **Techniques and tools for model debugging, validation and verification.** Methods for automatically assessing model correctness need to be developed, including ones based on program slicing and constraint solving. Techniques lying "midway" between *ad hoc* testing and full formal verification should also be investigated.

Simon Dobson: Challenges of Pervasive Computing

Pervasive computing allows designers to build information systems that both react and adapt to real-world conditions. At a shallow level this allows us to improve usability by matching services closely to the tasks they are being used to fulfill. At a deeper level, pervasive computing blurs the distinction between atoms and bits to allow actions to simultaneously have physical and informational content. A number of new challenges are presenting themselves.

Firstly, pervasive computing must deal with inherently noisy, imprecise and inferred information which may be more or less accurate. These uncertainties typically cannot be eliminated by further analysis, so systems must be **stable under perturbation** and able **to recognise and react to incorrect decisions** – essentially being designed under an assumption of incorrectness, leading to new ways **to think about, describe and handle exceptional conditions**. Secondly, extracting information from real-world data requires a deep understanding of the physical, logical and social constraints controlling external actions, with

software able to **reason about richly-connected models** of tasks and information. In many cases **the underlying networks and logics remain to be discovered**. Thirdly, mobility and dynamism mean that pervasive computing systems 'meet each other' in unexpected ways. We need to be able to describe these interactions at a high level of abstraction, not simply as shared interfaces. In some ways this **re-opens the field of semantics**, in that we need to consider 'open denotations' of systems that will **meet some parts of their functionality as they evolve**. Finally, pervasive systems need to adapt their behaviour largely autonomously within a design envelope. We have **little understanding of how to describe envelopes or adaptation strategies** – and even less of how to prove properties about them, which is a vital component of both usability and safety.

In many ways pervasive computing re-visits existing concerns with an increased stress being placed on dynamism, rich interconnections, stability and interaction. Addressing these concerns may therefore have a general impact on the development of complex software systems.

Jose Fiadeiro: Physiological and social complexity of software-intensive systems

In our day-to-day, we use the term "complex" in a variety of ways. Many times, we apply it to entities or situations that are "complicated" in the sense that they offer great difficulty in understanding, solving, or explaining. There is nothing necessarily wrong or faulty in them; they are just the unavoidable result of a necessary combination of parts or factors ; their complexity is « physiological ». In other circumstances, complexity derives more from the number and "open" nature of interactions that involve "autonomic" parts. Social systems are inherently complex in the sense that it is almost impossible to predict what properties can emerge and how they will evolve as a result of the interactions in place or the dynamics of the population itself.

This same distinction between physiological and social complexity also applies in software engineering. Software applications can be very complex entities in the sense that they may require an intricate interlacing of parts to provide the solution to a problem; the problem may be simple to understand and formulate but the nature of the parts available to build a solution may be such that the process of construction and the resulting application are complex. On the other hand, even very simple software applications may be required to take part as components of large and intricate systems in which they have to interact in often-unpredictable ways.

Social complexity of software is much more recent but becoming a prevailing trait. Software systems are now pervading key areas for the day-to-day functioning of our society. They are required to participate in heterogeneous networks of physically distributed and dynamically changing locations connected through often-unreliable communication infrastructures. Hence, software engineering is now facing the challenges that more traditional science and engineering disciplines have known for years.

However, much of the social complexity of software-intensive applications is being addressed with concepts and techniques developed over decades to address physiological complexity. A typical example is the use of object-orientated methods and languages for service-oriented

computing. The talk will discuss the difference we see in these two levels of complexity and the forms of formal support that we can provide to address them. Our ultimate goal is to contribute to the effort of developing methods and tools that can address social complexity in software intensive systems from first principles.

Carlo Ghezzi: Dynamic Software Federations

Software systems are becoming increasingly distributed and decentralized. Applications are composed as dynamic federations of autonomous and evolving components. Examples are emerging in the area of ambient intelligence, pervasive ubiquitous computing, and WEB services. Component technology and middleware have maturing to support the kinds of software architectures that are needed in these contexts. Although we are still in the initial stage, the trends towards increasing flexibility, evolution and decentralization will continue.

In future scenarios, it will be possible to develop new applications by federating dynamically available services, without relying on a unique central control authority which would be in charge of the entire process. Rather, a myriad of distributed processes form loosely coupled organizations that provide dynamically composable services. Federations will be formed by dynamically exposing new services and composing them in a self-organizing manner, with the aid of an active and cooperative support infrastructure.

Many problems have to be addressed to make scenarios of this kind possible. A possible research agenda includes:

1. Effectively supporting fully decentralized (peer-to-peer) software architectures even in fully dynamic scenarios where nodes are mobile and connected via wireless links;
2. Identifying flexible binding mechanisms to support dynamic and self-organizing component federations, via service discovery, brokering and negotiation, etc.
3. Specifying, verifying, negotiating, and monitoring quality of services;
4. Specifying and supporting the notion of context-awareness to support context-aware services;
5. Providing ways to model dynamic component federations and reason about properties (such as quality of service and context-dependent behaviour);
6. Supporting distributed workgroups contributing to the virtual marketplace of services;
7. Understanding software business models in highly decentralized marketplaces and supporting federated processes.

Conny Heitmeyer: Verified Software Generation and Composition from Requirements

My vision is that, in the future, software developers in a given domain (e.g, avionics, medical devices, automotive) will specify the requirements of a system or component in a standard, user-friendly, domain-specific language with a sound formal semantics. Specifications in the language will provide the basis for formal analysis and simulation (symbolic execution of the specification). Developers will have available an integrated suite of domain-specific tools to automatically analyze specifications for well-formedness and application properties, (e.g., safety and security properties). Tool feedback will be in terms of the standard language. When a developer has high confidence in the specification's correctness, technology will be

available (e.g., a verified compiler, on-the-fly verification) that transforms large parts of the specification (e.g., the control logic, simple functions) into provably correct, efficient code. Technology will also be available for composing synthesized code with other code (e.g., legacy code, COTS, standard domain-specific software components, code for abstract data types, hand-coded software) so that (not always possible) the composite satisfies the component properties. Developers will also have available technology integrating testing and formal analysis and will use the technology to gain high confidence that software integrating a set of heterogeneous software components satisfies the requirements.

Major problems include: lack of standard requirements specification languages and models, difficult-to-use formal analysis tools, failure to integrate testing and formal verification, lack of standard domain-specific software components, lack of verified compilers.

Major opportunities: Program managers, especially those responsible for safety-critical software, are most interested in tools/techniques/methods (e.g., automatic test case generation) that provides high assurance of a software product's correctness.

Where is more research needed: languages/models for requirements specification; techniques that extract requirements from legacy code; verified compilers; models, tools, and techniques for specialized domains, measures for assessing the quality of a requirements specification or a software component, and a theory of software testing.

Stefan Jähnichen: Abstraction, adaptation and testing of software-intensive systems

In the next ten years, software will continue to play an increasingly dominant role in the development and deployment of new and innovative systems. Its easy producibility, and in particular its reproducibility, make it necessary – for a number of different reasons (cost, investment, training, maintenance) – to use software to make complex mechanical, electrical and even pneumatic components more universal, and thus ultimately make them cheaper to produce.
But there is no such thing as a free lunch, and producing and adapting software turns out to be an extremely complex problem, especially from the point of view of quality. I wish to focus on three aspects of this problem which will attract increased attention in the future (meaning, of course, that additional research will be needed) and on whose mastering competitive advantages and market penetration will ultimately depend.

1. The level of abstraction of our means of description
It is a well-known fact the level of abstraction of today's programming languages is too low (i.e. too detailed and too strongly oriented to machine-processing). The development of mathematically based modelling languages, and in particular the availability of accepted modelling notations (e.g. UML), is a step in the right direction, but it also highlights the dilemma that exists between neat mathematical calculus and application-oriented – and in some cases graphical – notations. A way out of this dilemma is offered by work on model-based architectures, which attempts both to provide neat notations – in semantic terms, too – for developing systems and, based on this, to generate executable code or at least code fragments. This work will acquire increasing importance and, provided we succeed in hiding the generation process itself, will yield directly executable modelling languages.

2. Adaptation and reuse of existing code or code fragments
The vast number of existing software systems makes it impossible to constantly redevelop systems. It is absolutely essential to continuously further develop this software cost-effectively and adapt it to increased quality requirements. This is where the approach using model-based development techniques comes in again. If it proves possible, at least on the level of the modelling languages, to subsequently produce – or, even better, to generate – a behaviour and architecture description, this will enable new functionalities or features/characteristics/properties of the software to be validated for test purposes, at least by simultaneously executing the specification (or specification-based code) and the modified existing code. Here, replacing the existing code is unlikely to be an option, especially for reasons of efficiency.

3. Verification vs. testing
In the near future, I do not expect it to be possible to formally verify arbitrary code artefacts and thus prove their correctness. This means that quality assurance by means of systematic and comprehensive testing will continue to be very important. This is particularly true in cases where software is a major component of technical systems and the interplay of the different components (in their different models) is of great importance. For example: How do I test a vehicle where a garage has installed a new gear box containing new microprocessors and new software?

The importance of software in technical systems will continue to grow even more, forcing software engineers to increasingly assume responsibility for the system, and thus for system integration. To this extent, there are bound to be changes in the area of systems engineering, and research will have to devote substantial resources to addressing the question of system architectures and system integration. And here, too, the dominant issues will be quality assurance and productivity.

Even in the future, the engineering of software-intensive systems will not be marked by quantum leaps in terms of methodology, but rather by the continuous refinement of existing methods – but then that is an essential characteristic of engineering.

Jeff Kramer: Analysis techniques for self-organizing and pervasive systems

Self-organising software systems

Most systems are required to be capable of evolution, many being required to evolve dynamically as new components are introduced and as existing components are removed or fail. The objective is to minimise the degree of explicit management necessary for construction and subsequent evolution whilst preserving the required properties and operational constraints of the system. How can we provide a sound approach to help manage such change, especially where it is required to take place in a deployed system?

System Models and Analysis

Models provide the basis for design and analysis in engineering. In computing, approaches such as model checking have been very successful in checking whether or not a proposed software design satisfies its desired behaviour properties. Furthermore,

models have the potential to support other forms of analysis – such as performance and reliability - and feedback though animation, scenario replay, test generation, conformance testing, and simulation. However, model building (or synthesis) and analysis is difficult and has yet to make a significant impact on practice. How can we facilitate the process of model construction ? What forms of analysis are amenable to tool support and application?

Pervasive Computing

Most current access to computing power and the internet is via fixed office or home based PCs. Mobile communicators which integrate voice, video and processing capabilities with wireless communication are likely to replace the current mobile phones in the future. These will not only be used for personal communications and internet access, but will interact with intelligent sensors and actuators embedded in our homes, offices, transportation systems and even within or on the body to form a mobile ubiquitous computing environment. How should such systems be constructed, managed and customized ? What are the QOS, security and confidentiality issues?

Insup Lee: Design techniques for software-intensive systems

To improve the quality of software-intensive systems, it is necessary to develop better design techniques and paradigms. Challenging issues for supporting such design paradigms are as follows:

- Eliciting formal models from informal requirements: The development of most systems starts with informal requirements that specify how the system (consisting of hardware and software), and the user and environment are expected to behave and their interactions. Research is needed to facilitate the elicitation of a design from such informal requirements.
- Design Model validation: The design must be verified and validated to ensure it is correct and consistent with its intended purposes. There is much work to be done on how to validate that models captured in design artifacts are indeed the ones that are intended.
- Multiple uses of modeling artifacts: We need to explore ways to reuse the various design artifacts throughout the other development phases so that investments on design specification and analysis pay off directly in the final product development. The potential promising areas include the use of models for automatic test generation and code generation, as well as the use of logical properties proved at design time for run-time verification.
- Sharing of modeling artifacts: I believe that it is easier (in theory) to share design models than code. This is because models are at higher level of abstraction than code, and thus, tied less with target platforms. There has not been much support and effort with open model (*a la* open source) development so far. Such an endeavor should help to elevate design based on formal methods into main-stream activities.
- Composable design models: The notion of composition is necessary to deal effectively with designing of large complex systems. The composition of design can also facilitate sharing and should be for both homogeneous models or for heterogeneous models. We need to start investigating how to compose designs with different purposes (e.g., one design model for the physical layout of a sensor network and another design model for the protocol used between sensor nodes) to determine interaction between different views and to understand the overall design.

- Certification based on design: To be able to evaluate the quality of software-intensive systems, we need to develop a certification process based on sound scientific foundations. With proper scientific foundations, it should be possible to measure quantitatively how well a system meets its requirements

Luqi: Constructing flexible dependable software-intensive systems

Our ultimate goal is to consistently produce software that is flexible, low cost, and dependable. It must be able to support systems that require data to be processed within strict deadlines. It must also support systems where the software has been designed specifically to control physical devices. Lastly, it must also be able to handle systems that rely on communication between many interconnected modules that can be reconfigured as needs changes. Software is not used in isolation; it is often added into larger systems with unknown properties. It must be able to work with older elements, access key processes from them, and integrate them to perform tasks for the system's users. We work toward these qualities by using modeling, developing systematic engineering methods, and computer automation to ensure that as much of the development as possible follows principles that are known to work, both in prototyping and final designs.

Realization of this ideal requires coordinated advances in much of the current field. For example, in infrastructure, areas for improvement include better integration and communication between different software tools, network communication with predictable delays, and accurate information transfer between contexts with different purposes and different data models. In engineering, areas to focus on include modeling of strategies for design, standards for interoperability, increased automation and diagnostics, methods for effective reasoning support, models for software reconfiguration, and architectures that can handle many different problems with the same design structure. For coordination, we need better control of software risks, techniques to improve requirements, aid for determining design rationale, more reliable approaches to system security, and support for collaboration between groups of human experts and automated systems.

Benefits resulting from these directions include more effective cooperation between military and civilian systems, robust safety-critical systems that can adapt to overcome failures and changing needs, improved software safety, more secure electronic commerce, and many others.

In order to obtain practical results, all of these advances will have to be incorporated into a single coherent system. Ensuring that everyone's numbers and standards match both reality and each other is paramount. People must be trained to both work with this system and improve upon it in order to successfully produce tangible achievements.

Stephan Merz: Development of reliable models and components

As outlined in the motivations for the workshop, the process of software development has undergone significant changes during the past ten years, characterized by concepts such as component-based design, emphasis on expressive modelling languages, integrated development of models and code, agile development processes, architectural patterns etc. Despite of this, software bugs are still commonplace. I believe that more research is necessary concerning the applicability and integration of formal methods with development processes. This will require progress from both sides: state-of-the-art methods of system design are not based on clear and sound semantic foundations. On the other hand, formal methods still have problems of scaling up to realistic systems and they often do not take into account the paradigms of modern software design, such as components or patterns. Much progress has been made in the area of software model checking, based on techniques of abstraction and automatic program analysis, but a better integration of deductive tools based on different technologies is needed to effectively support top-down system development. The second challenge requires more conceptual research into the semantic foundations of software architectures, components, and connectors that give rise to efficient analysis and verification methods. Research in these areas should be accompanied by concrete case studies and should give rise to prototypical implementations, with the objective to give software engineers the benefit of mathematical assurance while presenting models and artefacts at an adequate level of abstraction.

Novel application areas and programming paradigms constitute a second important area for research. Inevitably, many concepts that are being suggested will not survive the test of reality. However, applications such as security-sensitive systems (control of access and/or information flow) or paradigms such as mobility of code or aspect-oriented design appear important enough to warrant additional research on both sides of the divide between software engineering and formal models.

Oscar Nierstrasz: Putting change at the center of the software process

Although software engineering practices have matured significantly in the past twenty years, and have also benefited from advances in object-oriented and component-based development methods, software productivity and quality generally continue to fall short of expectations, and software systems continue to suffer from signs of aging as they are adapted to changing requirements.

These phenomena are not surprising if we consider that the “waterfall” model is today still the predominant model for industrial software development projects, and that software “maintenance” is still undervalued in relation to initial development. What is wrong with this picture is that *change* is not placed in the centre of the software process. We know full well that real software systems must change or perish in response to changing business needs, that software systems become more complex and fragile as they age, and that this cycle is rapidly shortening as hardware and business innovations emerge. We must therefore advance beyond the “engineering” metaphor for software development, and focus more on support for change.

Three specific areas of opportunity are:

1. *Programming languages:* present-day languages focus on static design, and offer few mechanisms to support design evolution. We need research into high-level languages and mechanisms that can express and cope with radical changes in design.
2. *Development tools:* we need tools to model, analyse and transform evolving software systems. Such tools would support co-evolution of artefacts at various levels of abstraction, from code, through design to requirements.
3. *Processes:* although "agile" processes are making inroads, there is still a great deal of scepticism concerning their applicability to conventional projects. Research is needed to determine which software practices are most effective in coping with high rates of change.

Karl Reed: Some issues in the engineering of widely deployed software intensive systems-functional variation

The issue of dynamically linking different versions of components between successive executions of a systems is relevant to component based systems. The idea is not new, and many early systems provide for this capacity. Often discussion refers to functional variation. In principle, this seems to be the opposite of properly engineered software. However in practice modern systems have such large volumes of functionality that users may be surprised by the "miraculous" appearance of hitherto (existing) but unknown functions. From the users perspective, whether or not this functionality was already present or was dynamically included since last execution is immaterial. In practice, users deal effectively with quite large functional variation in this context.
We suggest that there are a relatively wide number of classes of functional variation which can be identified. Some mechanisms for protecting the user from the effects of unexpected functionality variations,are suggested, and as are mechanisms for extending a systems capabilities based upon these variations. We point out that identifying unexpected behaviour is important in safety-critical systems design, and that approaches from fault tolerance may be used to deal with such change. The concept of an operational envelope is proposed as one way of dealing with changes, and whose impact, may need to be accepted or rejected. We conclude by suggesting that the scale of functional variability that can be tolerated my be seen by studying users reaction to systems with large amounts of apparent variability.

Vladimiro Sassone: Context-aware software-intensive systems

Our society increasingly depends on open-ended, global computing infrastructures consisting of millions of individual components. Third-party computation, whereby migrating software and devices execute on networks owned and operated by others, lies at the very heart of the model, and will soon be the norm. Countless `pervasive' devices equipped with limited resources and computational power will roam the network, and support end-to-end applications which far exceed the devices' own capabilities. In such a scenario, the notion of `third-party resource usage' will rise to unprecedented centrality.

From the point of view of their owners, resources will need to be advertised, made available, managed, protected, and priced, while from the user's viewpoint, they will have to be

discovered, explored, acquired, used according to rules, and paid for. The complexity of `digital communities' of such kind is rapidly growing beyond that of other man-made artifacts, and will reach that of natural social systems. The software-intensive mechanisms involved exceed by far our ability to design, comprehend and control, and are by and large the limiting factor to building and deploying innovative applications. Although we are able to analyse and explain the behaviour of each single component and each single component's interface, we are essentially clueless when willing to analyse – and less than ever predict! -- the system as a whole. This is not dissimilar from trying to make a prediction on, say, house prices growth, on the basis of our perfect understanding of the mechanisms at the root of market economies.

As we grow accustomed to rely on such systems for things as precious as human lives, their lack of robustness and vulnerability become unacceptable. We need to design and develop for safety, as we currently design for efficiency and for reuse. The impact of these demands on engineering software intensive systems – and actually on Computer Science as a discipline – is dramatic. Scalability is of course a very present issue, intrinsically accompanied to our specific `globality' hypotheses, but by no means the only one. More generally, ubiquitous software systems are exposed throughout their lifetimes to the great variability of their operating environments, of which they have a very partial knowledge to start with. Change in scale is only one particular case of variations arising from a potentially unbounded number of different sources: new agents' arrivals or service deployment, connections and systems failures, new resource distribution and pricing schemata, and so on. For the purpose of this discussion, we will comprehend them all by the term context-awareness. Software systems will need ways to assess the contexts surrounding them, and adapt to their changes.

Recent approaches to context-awareness rely on powerful middleware, from which the software can `read out' all sort of interesting information about the environment. This appears unrealistic in general, and likely to presume too much of middleware's flexibility. In our intended scenario, we need to stage substantially faster and more feasible reactions to unforeseen events than getting back to the middleware design table. In other terms, middleware mustn't barely support context-awareness; rather, it must support design and programming for context-awareness. I discuss below an approach based on trust.

The idea of third-party resource points towards a model based on negotiation and protection of resource bounds, whereby owners and users dynamically agree on transient resource allocations. This latter is to be realised against resource pricing policies, whereby users agree to pay for their resource usage. As perfect knowledge is a rare commodity in global networks, such allocations will rely on risk assessment based on resource values and on the clients' trustworthiness. A server with a degree of trust in a client may be willing to lease it a resource at a certain price. Lower trust levels may mean higher prices, or refusal to interact altogether. Dually, a client must trust a server before entering negotiations with it, or using potentially harming resources received from it. Risk assessment may be based on a novel concept – yet to be discovered – of `program reputation,' that is a measure of code's past `good behaviour.' Reputation will change in time as a result of interactions as observed by the digital community, and so will systems' trust assessment and, thus, decisions. This gives rise to a number of unresolved issues:

- How do resource owners specify their resource usage policies?
- How do host and client code interact to determine resource requirements and bounds?
- What safety and robustness provisions can be made for both hosts and clients?
- How can the host trust the guest code not to abuse the resources made available?

- What is an appropriate pricing model for resource usage and how does cost negotiation take place? None of these questions are, as yet, resolved.

I advocate an approach to context-awareness based on trust and reputation management. Systems will explore contexts as resource providers, by relying on a notion of context reputation. Reacting to contexts' changes, upon trust evaluation, systems will carry out self-inflicted actions dealing with configuration, healing, maintenance, runtime extension. Such `autonomic' behaviour addresses scalability in a novel way, as a risk-assessed reaction to context variations.

These challenges make the tasks of software design, implementation, validation and maintenance even more troublesome than usual. They require the acquisition of a suitably general conceptual understanding of infrastructures, computational models, and language mechanisms, and breakthroughs are needed at all levels. There should be core computational models which identify flexible and robust notions of context and trust. Foremost, we need to develop a model of trust suitable for global systems based on third-party computation, as those envisaged here. We also need a computationally feasible notion of reputation. For instance, the reputation of entity E could be (an approximation of) the average trust in E over a local fragment of the global network. We then need to identify abstractions suitable to design programming languages, together with development tools to reduce the frequency of bugs, and tools for both qualitative and quantitative analysis. Solid theoretical foundations should ensure safety and security properties. Infrastructure and virtual machine technologies which support resource usage monitoring, global trust evaluation, pricing and negotiation need to be built. Compilation techniques targeted at resource conscious architectures should be investigated. Novel static analysis techniques to alleviate the execution cost of runtime monitoring should be employed.

Joseph Sifakis: Work Directions in Component-based Engineering

Theoretical frameworks for component-based engineering should include satisfactory solutions to two problems: The first is theory for composing heterogeneous components. The second is theory for establishing correctness by construction, to cope with complexity. We discuss work directions and present a general framework for jointly addressing these two problems.

There exist, two specific sources of heterogeneity: interaction and execution.. Heterogeneity of interaction results from the presence of interactions that may be atomic or non atomic, blocking or non blocking. Heterogeneity of execution can be characterized by the way threads are scheduled. Synchronous and asynchronous execution reflect different scheduling policies.

Building systems that by construction meet given properties requires in principle, two types of rules:
• Composability rules allowing to infer that under some conditions, components' properties are preserved when they are integrated in larger systems.
• Compositionality rules allowing to infer a system's properties from its components' properties.

The framework presented adopts a principle of layered description of components, consisting of three layers corresponding respectively to behaviour, interaction and execution models. Using layered descriptions allows the definition of a general associative composition operator as well as composability and compositionality results for deadlock-freedom.

Jeannette Wing: Toward Software Security Design Principles

The biggest challenge for the software engineering community is software design. We know how to teach good programming practice. We know how to write, test, analyze, debug, and verify code. But we do not know how to do any of this for software designs. As a result, practitioners see little value in doing design: they don't know how to do it and they don't know if what they've done is good or not.

The biggest challenge for the software intensive systems of tomorrow is security. Our software is more complex, built out of more and more mismatched components, and continue to be rife with bugs. The environments in which systems are deployed are more unpredictable and more malicious. The attacker only needs to find one bug to exploit. The defender has to find and fix them all. Impossible.

I propose a research agenda that looks at software security design. This agenda had the advantage of not trying to solve the whole software design problem, but one more focused on security. This agenda also has the advantage of moving the security community to focus on a level above the code; in principle we have technical and practical solutions to fixing buffer overruns and other code-level bugs. It is time now to look at design-level vulnerabilities. Attackers already structure their attacks by using the functionality of one component to enable the functionality of another, to enable the next, and so on; this sequence of actions can look benign until the very last step, when the attacker achieves his goal.

Toward the goal of understanding software security design, I suggest we look at (1) compositionality of system components; e.g., when does an unintentionally composed system lead to emergent abusive behavior? and (2) software security design principles; e.g., do we build systems following the End-to-end Argument or Principle of Depth in Defense, and how do we design systems when security properties slice through abstraction boundaries?

Workshop Presentations

Engineering Software-Intensive Systems: Introduction

Martin Wirsing
LMU Munich

EU-NSF Strategic Workshop Series

- **Joint initiative of CISE-NSF and FET-EU**
- **Organized by ERCIM**
- **Goals:**

  Identify key research challenges and opportunities in Information Technologies

- **This workshop is part of the series:**

  SW-intensive systems considered as topic for EC 7th framework programme

Engineering Software-Intensive Systems

Situation

- Daily life depends on complex SW-intensive systems
  - in banking, communication, transportation, medicine, …
- New emerging technologies
  - Global computation systems
    - Internet, Grid, pervasive computing …
  - Embedded systems
    - automotive, avionics, …

Engineering Software-Intensive Systems

- Fast technological progress
  - Modeling languages and CASE tools
  - Object-orientation, programming environments
  - Model checking, proof carrying code, …

- But clash between

Foundational modeling	**Pragmatic SW solutions**
generic/ deep understanding	complete, executable
but	but
too abstract, partial, incomplete	inconsistent, not reliable, not interoperable
…	not well structured …

Engineering Software-Intensive Systems

Challenges:

SW-intensive systems need to

- model correctly data and processes and have adequate system architecture
- ensure quality, i.e. reliability, safety, security, availability, dependability, compliance with the system requirements, …
- support distribution, mobility, heterogeneity and interoperability
- managing change, reconfiguration and adaptation, and
- enhancing the usability
- Integrate and adapt legacy software

EC FP7 Preparation

Foundations of Software-Intensive Systems is a grand challenge

[Report by Hermenogildo, Sifakis, Babagliou]

This includes

- Guaranteeing non-functional properties, such as: security, safety, scalability, resource optimisation, quality of service, efficiency, selfishness etc.
- New, high-level paradigms and languages for programming encompassing distribution, mobility, dynamic evolution, and taking into account non-functional properties.
- New algorithmic techniques for distributed systems, taking into account non-functional properties.

Engineering Software-Intensive Systems

Objectives of this workshop:

- discuss and evaluate the **state of the art** in engineering discuss the state of the art in engineering software-intensive systems, ,
- Identify and **elaborate challenges** for software-intensive systems,
- show how **pragmatic methods can be integrated with foundational research** in software-intensive systems engineering

Agenda

Saturday, May 22

9:00 - 9:30	The ERCIM workshop series by Remi Ronchaud Introduction by Martin Wirsing
9:30 - 10:30	Presentations of challenges and research issues by participants Don Batory, Carlo Ghezzi, Connie Heitmeyer, Luqi
11:00 - 12:30	Presentations of challenges and research issues by participants Jeannette Wing, Vladimiro Sassone, Insup Lee, Jose Fiadeiro, Oscar Nierstrasz, Ira Baxter, Simon Dobson
14:00 - 15:00	Presentations of challenges and research issues by participants Rance Cleaveland, Stefan Jähnichen, Stephan Merz, Joseph Sifakis (by MW), Kevin Sullivan
15:00 - 15:30	Identification of challenges and research topics
16:00 - 18:00	Discussion of topics in working groups

Agenda

Sunday, May 23

8:30 - 9:30	Presentations of challenges and research issues by participants: Jeff Kramer;
	Presentation and discussion of first results of the working groups
9:30 - 10:30	Discussion of topics in working groups
11:00 - 12:30	Discussion of topics in working groups
14:00 - 16:00	Presentations of results of working groups and final discussion

Challenges in Software Intensive Systems Research

Don Batory
Department of Computer Sciences
University of Texas at Austin

This Talk...

- 20 years of observations (mostly by others)
- Central technical problems of software intensive systems
- Central barriers to progress

Hard Technical Problems

- As a **community**, we don't understand:
  - the mathematics of large scale software design
    or know if mathematics are needed...
  - how to go from declarative specifications of systems automatically to their optimized source
  - how to scale proofs of correctness to large systems
  - role of multiple languages in large scale system design
  - how architectural recovery can be supported in an automated way
- But **individually**, members of our community do...
- Technical problems are known, but not main challenge

Have You Noticed...

- Child prodigies are in:
  - art, chess, mathematics, music, golf
- But not in:
  - surgery, politics, engineering, **software engineering (SE)**
- Why?
  - complexity assimilated through experience, time
- Success in SE (if any) will be in the long haul
  - SE is driven by fads
  - ideas that last must survive the test of time

Paraphrasing Our Discipline

- M. Graham: hard thing about SE is that you can't ignore anything
- Dijkstra: need supreme competence in many areas
- We need best and brightest, but future relies on student sacrifice

Paraphrasing Our Discipline

- Good career move?
  - competencies that are needed are not generally appreciated
  - Ph.D. in software design?? program algebras?? program transforms??
  - reward system is not in place
- SE or software-intensive systems is fundamental
  - not sexy
- Most significant advances in SE must be hidden

  "Don't tell them how or why it works;
  it will just scare them..."
- Best and brightest have far easier career paths from which to choose

What to Do?

- Solvay Conferences – premier conferences in physics in early 1900s
  - foundations for quantum mechanics, relativity
- Meetings of excellence
  - by invitation on per area basis
  - akin to Dagstuhl paradigm, but on scale
  - best and brightest focused on key problems
  - advertise for students, funding

Solvay Conferences

- Progress requires close coordination theory and practice
  - will take years
  - time is not on our side
- Importance critical
  - funding is vanishing
  - panama canal, time
- Funding contradiction
  - if short term, people may incorrectly assume certain ideas failed (where they have not)
  - may direct focus elsewhere...
- Technical problems are solvable, in time
- Non-technical problems are greatest barrier to progress

Engineering Software Intensive Systems using Design *Maintenance* Systems

Putting Design permanently into the Process

Ira D. Baxter

What's the Question?

- We can already build very complex software systems
  - Telephone switches, Windows OS, Insurance software
- Bad: Simple models of engineering (e.g., waterfall)
  - Unable to get anything right and move on!
    - Requirements, specification, design, performance
- Worse: Can't maintain it well once delivered
  - No decent institutional memory of artifact structure/rationale
  - Cost of software primarily occurs after delivery
  - Software lifetimes are growing!
- *Question: How to capture and harness design to enhance development/maintenance process?*

State of the Art

- Terminology isn't defined clearly in SE field
  - What's a *design*? What's an *architecture*? When to use which?
  - Hard to make progress without definitions
- Weak "design" tools (UML, StateCharts, ...)
  - focus designer's attention on some *technological principal specification*
  - focus on software technology rather than domain knowledge
- Continuing demand for "Reverse Engineering"
  - Attempt to recover information from an existing artifact
- Vast literature on formal specification and refinement
  - Practically nothing available to harness it
- Considerable work done in AI field on "design"
  - Connecting specifications to implementations
  - Tracking alternative results (non-monotonic backtracking)
  - Ignored almost completely by SE field

A Radical Approach

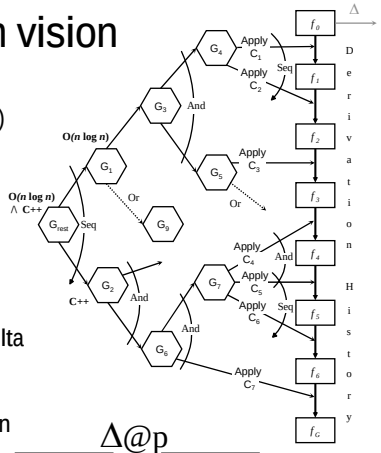
- *Make Design be the engineering product*
  - So that artifact implementation is trivially extracted
    - Engineers never allowed to code directly
  - So that what/how/why questions easily answered
    - Capture "what" knowledge in domain-specific forms
    - Capture "how" knowledge tying domain concepts to technology implementations
- Define desired artifact changes as formal spec deltas
  - "Integrate" deltas to produce complete design
  - Merges "implementation" and "maintenance" phases
- Provide mechanical, scalable support over designs:
  - To navigate
    - Support analysis of existing structure
    - Propose alternative implementation choices
  - To incrementally update using spec deltas
  - To implement most trivial tasks
  - To handle long-term transactions of multiple engineers

How to deliver SE technology

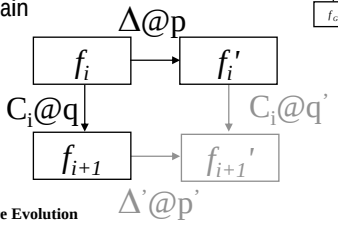
- Organize knowledge around "problem" domains
  - Use to specify functionality/performance in notation of problem expert
  - Index to "how-to" knowledge:
    - analyzers, refinements, implementation tactics
  - Abstraction levels: specification, technology, implementation
- Use Program Transformation to implement knowledge
  - Optimizations and Refinements
    - Inference as rewrites; Classic compiler technology as special case
  - Mixed-initiative implementation: tacticals + interactive guidance
    - Mixed-initiative analysis of result performance
- Capture *transform sequence* + *rationale* as Design
  - Rationale = proof that performance achieved by transform choice
  - Design-updates as transactions
  - Incremental updates using
    - distributive laws on specs, commutative laws on implementations
    - additive domain knowledge

The Design Maintenance System vision

- Transformational Designs
  - Functionality Spec (f_0) + Performance Spec (G_{rest}) + Derivation + Justification + Alternatives
- Scale
  - Metaprogram driven automation
  - Incremental Updates
    - Specification & Technology Δs
    - Δs drive design revision: retain transforms that commute with delta
  - Domain-based specification/implementation
    - Simplify expression of problem
    - Store implementation knowledge with domain
  - PARLANSE: Parallel foundation of DMS



$$\underline{\Delta@p}(\underline{C_i@q}(f_i)) = \underline{C_i@q'}(\underline{\Delta'@p'}(f_i))$$



ICSE 2004: **DMS: Program Transformations for Practical Scalable Software Evolution**

Engineering Challenges for Software Intensive Systems

Rance Cleaveland
Dept. of Computer Science
SUNY at Stony Brook
-and-
Reactive Systems, Inc.

What Have I Been Doing for the Past 15 Years?

- 1989—????: Prof. of Computer Science
  - Process algebra
  - Semantics
  - Model-checking tools
- 2001—????: CEO of Reactive Systems
  - V&V tools for Simulink and Stateflow
  - Semantics of Simulink and Stateflow
  - Testing
  - More testing

State of the Art for SIS

- What the professors might say
  - Model checking
  - Formal specification
  - Middleware
  - Software architecture
- What the industrialists might say
  - Model-based development
  - “Autocoding” (generating code from models)
  - FlexRay / Time-triggered protocol
  - Software architecture

3

Which Industrialists?

- Embedded system developers in automotive, aerospace, defense, etc.
- Characteristics
  - Backgrounds are in non-CS engineering
  - CS info from consultants, trade magazines
  - “We are not in the software industry”

4

Where Work Is Needed

- Better design processes (Where are “blueprints”? “Circuit diagrams”? “Simulation models”?)
- Standardized (mathematical!) design notations
- More nuanced V&V
- *Better tech transfer*

Remember:

- Engineers are not afraid of math.
- They are afraid of wasting time.
- They are wary of computer scientists.

5

“More Nuanced V&V?”

- Currently, SIS builders can:
  - Test (little support from CS community)
  - Formally verify (impractical)
- We need V&V strategies that fall in between these extremes
  - Less thorough but still rigorous
  - Theorems characterizing “gaps”
  - Requirements as test oracles
- Goal: the longer you validate the more confidence you should have

6

Integrating Foundational Influences

Tools! cf.

- Electronic Design Automation in the 1980s
- Control Design Automation in the 1990s

Tools must have sound mathematical foundation to be accepted and used.

Mathematics must have foundation in intuition to be accepted and used.

7

Mathematical Intuitions?

- State machines
- Sequence diagrams
- Architecture
- Interfaces (external and internal)

All the above are already used.
With right mathematics, tools will follow.

Note: structure, operational content

8

Conclusion

- Big gap between research and practice
- More attention needed to improved software design
- Mismatch in expectations about V&V
  - CS researchers: get systems “right”
  - System builders: get systems “right enough”
- Mathematics of structure, operation essential

9

University of Dublin
Trinity College
Dublin 2, Ireland

Simon Dobson

Distributed Systems Group
Department of Computer Science
Trinity College, Dublin IE

simon.dobson@cs.tcd.ie

Software's reflection: Software-intensive challenges from ambient computing

EU/NSF joint workshop on Engineering Software-Intensive Systems, Edinburgh, May 2004

Overview

Ambient systems

- Also called ubiquitous computing, pervasive computing, context-aware computing, …

Software systems sitting in an unusually close relationship to real-world process and actions

- Provide IT support that responds directly to location, co-location, process, history, interest, expertise, … , of users
- Add information and/or services to everyday artefacts

Major international research and commercial topic

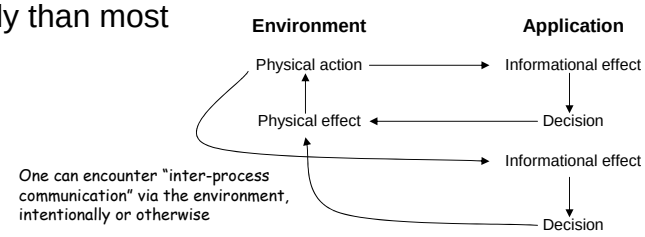
- Europe: GLOSS, Ambiente, E-Gadgets, Tacit, …
- US: MIT Aura and Oxygen, UC Berkeley "motes", GA-Tech, …
- Industry: HP Cooltown, Microsoft Easy Living, IBM, Intel, …

The environment

Potentially every action has both physical and informational significance

- Blurs the traditional distinction between atoms and bits
- All events are potentially mediated by, or significant to, software
- Achieve coherence between both worlds

An ambient system "sees its own reflection" more strongly than most



Stability

Information in ambient systems is inherently noisy, and is manipulated with inherently uncertain methods

- Can't be removed by further analysis

React to "significant" events – and only those events

- The infamous "flickering light" problem…
- The slightly less famous "dude, where's my printout?" problem…

How do we engineer systems that are stable under minor perturbations and will react to events without being swamped by noise?

Stability both singly and under composition

- Adding information and/or resources should not move behaviour unexpectedly

Not exceptional

Any “fact” may be wrong; any decision may need to be un-done

- Can’t open the door half-way if we’re only half-sure who’s there…

In practice we must design under an assumption of incorrectness (or at least of uncertainty)

> What are the correct conceptual and software structures for dealing with inherent uncertainty and incorrectness, where failure is completely un-exceptional?

Build this uncertainty more into the fabric of systems

- Languages with no boolean type?

A richness of models

The information available to ambient systems is extremely rich and inter-connected

- Can often draw inferences as much from a constellation of “facts” as from the “facts” themselves

Constrained by real-world physics and policies

- Spotting the impossible, non-standard logics
- Move further away from the Von Neumann view?

Many traditional issues assume new importance and subtlety – privacy and security most especially

> What are the appropriate logics and approaches for specifying and dealing with richly-connected and subtly interdependent contextual information?

Re-opening semantics

Ambient systems are almost always mobile to some degree

- Meet pieces of functionality as they go along…
- …use dynamically-located resources…
- …but still do something sensible and recognisable

What does a program denote when it meets part of itself as it goes along?

Closely related to stability and richness – in fact, this is probably the unifying “grand challenge”

- Behavioural envelopes within which the system can adapt
- Continuity and compositionality of behaviour
- Denotations are open, but may still have properties to analyse

Joint EU/NSF Strategic Research Workshop on

Engineering Software Intensive Systems

José Luiz Fiadeiro

University of Leicester

A case of "complexity"

in-the-head

mnemonics

result-driven

symbolic information

elementary control flow

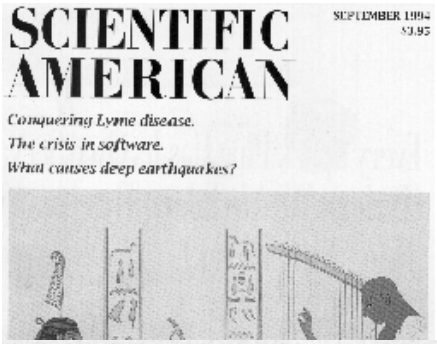
- "**One man and his problem…**" (and **his program**, and **his machine**)
- The Science of **Algorithms** and **Complexity**
- not so much **Engineering** but more of **Craftsmanship** (one of a kind)
- a case for **virtuosi**

A case of "complexity"

in-the-head	**in-the-small**
mnemonics	**I/O specs**
result-driven	**algorithms**
symbolic information	**data structures and types**
elementary control flow	**execute once termination**

- The need for commercialisation…
- **"One man and his problem…"** (and **his program**, but **their machine)**
- The Science of **Program Analysis and Construction**
- Commerce, but not yet **Engineering**

10 years ago, the "software crisis"


SCIENTIFIC AMERICAN

SEPTEMBER 1994
$3.95

Conquering Lyme disease.
The crisis in software.
What causes deep earthquakes?

86

TRENDS IN COMPUTING

Software's Chronic Crisis

W. Wayt Gibbs, staff writer

The U.S. economy, and indeed all society, has plunged into cyberspace. Computers turn up in everything from toasters and aircraft-control systems to the cash register at the supermarket checkout. Yet software remains largely the custom product of a cottage industry. Can it ever be manufactured so that it meets industrial standards of mass production and reliability?

10 years ago, the "software crisis"

5

- *The challenge of complexity is not only large but also growing. […]. To keep up with such demand, programmers will have to change the way that they work. "**You can't build skyscrapers using carpenters**," Curtis quips.*
- *[…] Musket makers did not get more productive until Eli Whitney figured out how to manufacture interchangeable parts that could be assembled by any skilled workman. In like manner, software parts can, if properly standardized, be **reused** at many different scales.*
- *[…]In April, NIST announced that it was creating an Advanced Technology Program to help engender **a market for component-based software**.*

A case of "complexity"

6

in-the-head	in-the-small	in-the-large
mnemonics	I/O specs	**complex specs**
result-driven	algorithms	**system modules**
symbolic information	data structures and types	**databases, persistence**
elementary control flow	execute once termination	**continuous execution**

- **"One man and his problem…"** (but **their programs**)
- The Science of **Software Specification and Design**
- **Engineering**

The case for objects/components

7

- Builds on a powerful methodological metaphor – **clientship**
- Inheritance hierarchies for **reuse**
- Software **construction** becomes like child's play

The case for new mathematics

8

- **Algebraic techniques for structuring specifications**
  - "Putting Theories together to Make Specifications"
  - The theory of **Institutions**
  - The role of **Category Theory**

Yet, in 2003 the crisis was going on

9

- *Computing has certainly got faster, smarter and cheaper, but it has also become much more complex.*
- *Ever since the orderly days of the mainframe, which allowed tight control of IT, computer systems have become ever more distributed, more heterogeneous and harder to manage. […]*
- *In the late 1990s, the internet and the emergence of e-commerce "broke IT's back". Integrating incompatible systems, in particular, has become a big headache. A measure of this increasing complexity is the rapid growth in the IT services industry. […]*

and the "silver bullet" became…

10

- *Computing is becoming a utility and software a service. This will profoundly change the economics of the IT industry. […]*
- *For software truly to become a service, something else has to happen: there has to be a wide deployment of web services. […]*
- *applications will no longer be a big chunk of software that runs on a computer but a combination of web services*

The Economist, May 10, 2003

Web services are…

- *"self-contained, modular applications that can be described, published, located, and invoked over a network, generally the Web"*

  Web Services architecture overview
  – the next stage of evolution for e-business
  IBM www-developerswork

- **"Sexed-up"** components?

Yet a case of "complexity"?

in-the-head	in-the-small	**in-the-large**
mnemonics	I/O specs	complex specs
result-driven	algorithms	system modules
symbolic information	data structures and types	databases, persistence
elementary control flow	execute once termination	continuous execution

- "One man and his problem…" (but their programs)
- **"One man and everybody's problems…"**

A case of "complexity"

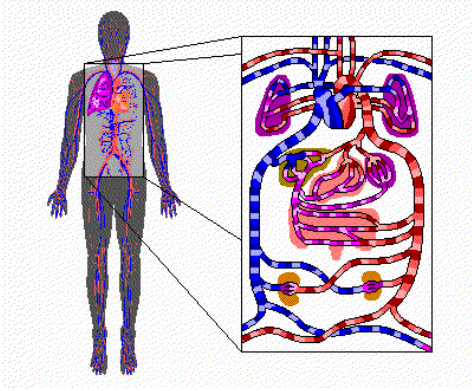
in-the-head	in-the-small	in-the-large	**in-the-world**
mnemonics	I/O specs	complex specs	**evolving**
result-driven	algorithms	system modules	**sub-systems & interactions**
symbolic information	data structures and types	databases, persistence	**separation data computation**
elementary control flow	execute once termination	continuous execution	**distribution & coordination**

Same complexity?

- **"Physiological" complexity**
  - derives from the need to account for problems/situations that are "complicated" in the sense that they offer great difficulty in understanding, solving, or explaining
  - there is nothing necessarily wrong or faulty in them; they are just the unavoidable result of a necessary combination of parts or factors
- **"Social" complexity**
  - derives from the number and "open" nature of interactions that involve "autonomic" parts of a system;
  - it is almost impossible to predict what properties can emerge and how they will evolve as a result of the interactions in place or the dynamics of the population itself.

Same Science & Engineering?

- **"Physiological" complexity**
  - server-to-server, static, linear interaction based on identities
  - compile or design time integration
  - **contracts of usage**
- **"Social" complexity**
  - dynamic, mobile and unpredictable interactions based on properties
  - "late" or "just-in-time" integration
  - heterogeneity of components
  - **quality and trust**



Challenges

- Shift the focus to interactions among autonomous entities
  - Programmable
  - Interfering
  - "Context-aware"
  - Evolvable
- Lift them to the requirements level
  - Policies that control evolution and self-organisation
- Decouple them from specific platforms
  - Develop dynamic binding mechanisms

Dynamic software federations

Carlo Ghezzi
Dipartimento di Elettronica e Informazione
Politecnico di Milano
carlo.ghezzi@polimi.it

The old world

- Product
  - monolithic
  - centralized
  - static, closed
- Process
  - single authority
  - pre-planned
  - monolithic

Achievements

- Product
  - monolithic — modular
  - centralized — distributed
  - static, closed — controlled dynamic binding
- Process
  - single authority — static task decomposition
  - pre-planned — pre-planned evolution
  - monolithic — spiral

Challenges (product, process)

- Systems built by federating dynamically discovered components
  - Open world, mobility
  - The “network as a bazaar” metaphor
    - Different types of resources, different QoS
- Self-organizing vs pre-planned systems

Challenges (product, process)

- No centralized control over available resources and system development
- Resources may be transient
- Mobile users/developers
- Ad hoc scenarios
- New kinds of business and process models

Problem scale

- From in-the-tiny
  - sensor networks
    - huge numbers of autonomous cooperating devices
- To in-the-large
  - web services

Towards a research agenda

- Support fully decentralized (peer-to-peer) software architectures even in fully dynamic ad-hoc scenarios where nodes are mobile
- Identify flexible binding mechanisms to support dynamic and self-organizing component federations, via service discovery, brokering and negotiation, etc.
- Specify, verify, negotiate, and monitor quality of services
- Specify and support the notion of context-awareness to develop context-aware services

Towards a research agenda

- Support distributed workgroups contributing to the virtual marketplace of services;
- Understand software business models in highly decentralized marketplaces supporting federated processes

Conclusions

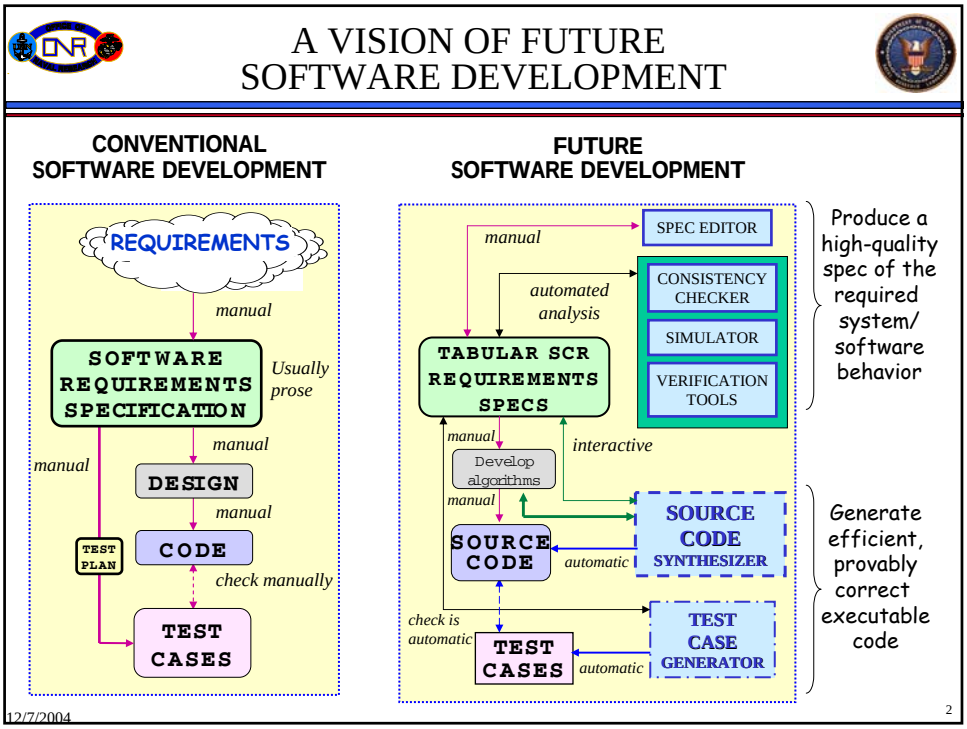
- We are moving towards unprecedented degrees of flexibility, dynamicity, and decentralization *at all levels*
- New challenges to correctness/ reliability, security, performance
- Crucial to understand how we can build on previous approaches and where new ones are needed

ENGINEERING SOFTWARE-INTENSIVE SYSTEMS: RESEARCH ISSUES AND CHALLENGES

CONSTANCE HEITMEYER
NAVAL RESEARCH LABORATORY
WASHINGTON, DC 20375

MAY 2004

A VISION OF FUTURE SOFTWARE DEVELOPMENT



WHAT IS THE BIGGEST CHALLENGE FOR SOFTWARE ENGINEERING?

- The greatest challenge is NOT software design
- Rather, it is capturing the software requirements
  - A good software requirements spec describes the set of all acceptable implementations
    - It excludes no acceptable implementations
    - It includes no unacceptable implementations
  - Fred Brooks refers to the software requirements as the *essence*, i.e., the required externally visible behavior of the system or software component
- Wide agreement on the criticality of correct requirements
  - Yet, well-founded technology supporting the construction of correct requirements is lacking
- Requirements acquisition remains a very hard problem
  - New methods and techniques for capturing and documenting system and software requirements are urgently needed
  - While scenarios can be effective in eliciting requirements, we need new approaches to representing scenarios
    - Message sequence charts are insufficient

MORE CHALLENGES

- New spec languages for capturing the system/software requirements
  - These languages should produce declarative specs that
    - Are readable
    - Limit implementation bias
    - Scale to large systems
    - Are decomposable into parts
  - Methods and techniques for constructing good specs
  - Criteria for evaluating specs
- Models for refining and debugging the spec
  - Some useful models
    - Formal analysis models
    - Simulation/animation models
    - Environmental models
    - Special-purpose models, e.g., security models
  - Methods and techniques for constructing good models
  - Criteria for evaluating models
- Usable tools for debugging the specs and reasoning about their properties
- Compilers for transforming a specification into provably correct, efficient, executable code
- Techniques for extracting requirements specs from legacy code

EVEN MORE CHALLENGES

- Need domain-specific languages, methods, and techniques
  - Example domains: avionics, automotive, medical devices, spacecraft
  - Our experience is with NASA's safety-critical systems and mission-critical military systems (e.g., a security-critical, software-based crypto device)
- Need to integrate tools to work together
  - Tools include "pragmatic" ones, such as those that do symbolic execution and animation, as well as tools for formal verification
  - Our experience: Different tools detect different classes of errors
- Need techniques to manage the complexity inherent in specifying, validating, verifying, and certifying large software systems
  - Abstraction
  - Decomposition and Composability
- Need techniques for composing hetergeneous components to form a software system that satisfies its specification
  - Code synthesized from specs
  - COTS
  - Legacy code
  - Code implementing an abstract data type
  - Manually-constructed code
- Need integrated approach to testing and verification

Challenges for Software Intensive Systems

ITEA Technology Roadmap

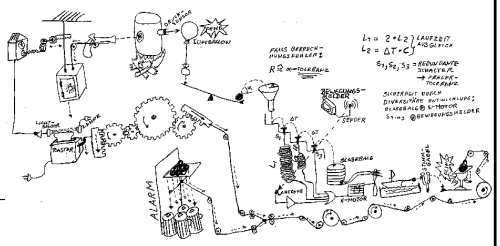
Prof. Dr. Stefan Jähnichen

Fraunhofer Institut Rechnerarchitektur und Softwaretechnik

Basic features of software intensive systems

In years to come, *software-intensive systems* will incorporate four basic features:

- they will be dynamic evolutionary systems,
- they will exhibit adaptive and anticipatory behaviour,
- they will process knowledge and not only data,
- they will allow the user to stay in control



Fraunhofer Institut Rechnerarchitektur und Softwaretechnik

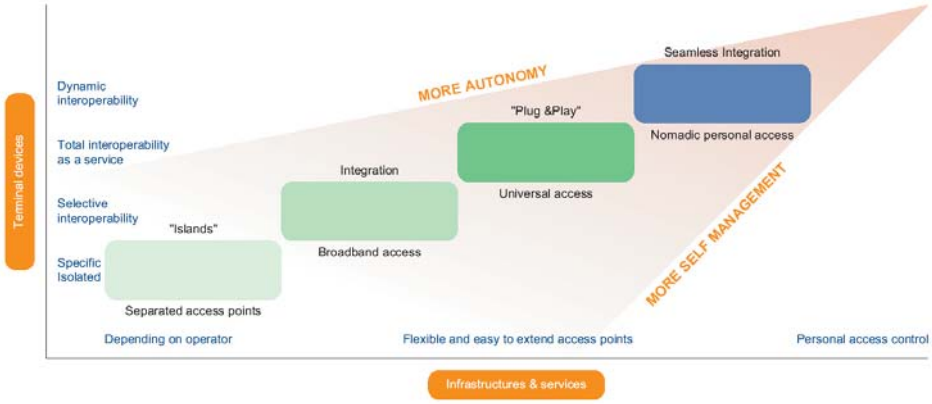
Key drivers

There are two kinds of keys to the development and deployment of these systems:

- Key drivers for *acceptance* are
  - interoperability of products, systems and applications
  - the “-ilities”: usability, testability, reliability … but also security and safety
- Key issues for *implementation* are:
  - mostly technical: mastering size, complexity and adaptiveness
  - mostly economic: middleware business models and costs.

Fraunhofer FIRST Institut Rechnerarchitektur und Softwaretechnik
Norbert Pieth

Roadmap to convergence of software intensive systems



Source: ITEA's Technolopgy Roadmap for Software-Intensive Systems

Fraunhofer FIRST Institut Rechnerarchitektur und Softwaretechnik
Norbert Pieth
Σ!

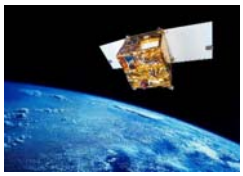
Application domains

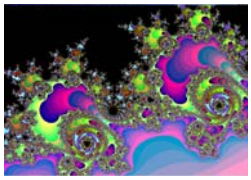
- Home entertainment (Personalized information services)
- Virtual companies and virtual business networks (Network and application management)
- Mobile and personalized Information services (Seamlessly location based and shared services)
- Web services for combination of private and business application (Intermediation and broker services)

Fraunhofer Institut Rechnerarchitektur und Softwaretechnik
Norbert Pieth

Challenges for software intensive systems / System engineering

- Embedded software systems
  - (re)configuration, functional upgrade, maintenance, replacement at runtime, integration of technologies
- Evolutionary system
  - permutative and adaptive system design
- Software family architectures
  - Model driven architecture, product line engineering for software component reuse, generative programming paradigms
- Knowledge based software engineering
  - Automation in verification and validation of software requirements using formal descriptions, automatic model checking, advanced testing strategies





Fraunhofer Institut Rechnerarchitektur und Softwaretechnik
Norbert Pieth

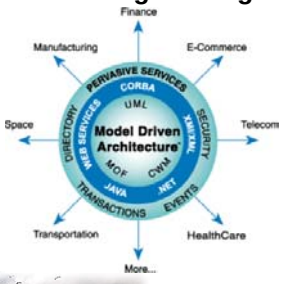
Challenges for software intensive systems / Software engineering

- Component markets and software suppliers
  - Localisation and modularisation of software components, agent based separation of concerns, Variability Engineering
- Reuse of engineering elements for software life cycle extension
  - distributed code libraries, design patterns, model based software development
- Flexible middleware systems for network applications
  - autonomous services and devices, distributed communication and computation schemas, network self-configuration and management

Fraunhofer Institut Rechnerarchitektur und Softwaretechnik
Norbert Pieth

Challenges for software intensive systems / Software engineering

- Domain specific description languages
  - high-level specifications, feature models, UML unifying the development for many target platforms
- Self-organising software agents
  - Intelligent problem separation and solving techniques, dynamically reconfigurable services, system composition on the fly, adaptive software components

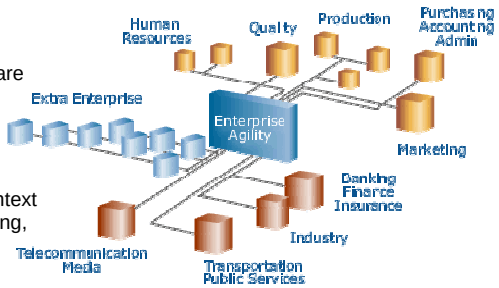


Fraunhofer Institut Rechnerarchitektur und Softwaretechnik
Norbert Pieth

Challenges for software intensive systems / Engineering process support

- Standardisation, Integration and interoperation of engineering tools
  - standardises data and metadata exchange formats
- Distributed and collaborative engineering
  - Communication and collaboration platforms, WIKI-systems, social software initiative
- Knowledge based process management
  - Experience and best practice management, knowledge transfer, context aware decision support, software mining, model mapping, Ontologies

ISO International Organization for Standardization



Fraunhofer Institut Rechnerarchitektur und Softwaretechnik

Fraunhofer Institut Rechnerarchitektur und Softwaretechnik

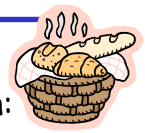
Engineering Software Intensive Systems

"Three half-baked proposals for software development, software organisation and an application"

Jeff Kramer

Imperial College, London.

S/W Development: system models and analysis

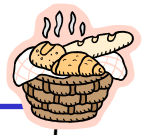


- **Models** provide sound basis for design and analysis.
  - Check properties of a proposed software design:
    - Behaviour, Performance, Reliability …
  - Provide feedback though counterexamples, animation, scenario replay, test generation, simulation…
- **Model building** (or synthesis) and analysis is difficult and has yet to make a significant impact on practice.
  - *How can we facilitate the process of model construction ?* …from requirements? ..for designs?
  - *What forms of analysis are amenable to tool support and practical application? …domain specific modelling?*

Organisation: self-organising software systems

- Most systems are required to be capable of evolution, many being required to evolve dynamically as new components are introduced and as existing components are removed or fail.
- The objective is to minimise the degree of explicit management necessary for construction and subsequent evolution whilst preserving the required properties and operational constraints of the system.
  - *How can we provide a sound approach to self-manage such change, especially where it is required to take place in an executing, deployed system?*
  - cf. autonomic computing, self-healing, adaptive computing, …

An application - pervasive computing



- Mobile communicators which integrate voice, video and processing capabilities with wireless communication are likely to replace the current mobile phones. These will not only be used for personal communications and internet access, but will interact with intelligent sensors and actuators embedded in our homes, offices, transportation systems and even within or on the body to form a mobile ubiquitous computing environment.
  - *How should such systems be constructed, managed and customized ?*
  - *What are the QOS, security and confidentiality issues?*

Challenges in Engineering Software Intensive Systems

Insup Lee
Department of Computer and Information Science
School of Engineering and Applied Sciences
University of Pennsylvania

May 22, 2004

Embedded Systems

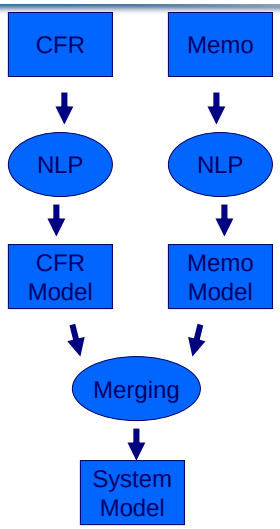
- An embedded system is a system
  - that interacts with (or reacts to) its environment, and
  - whose correctness is subject to the physical constraints imposed by the environment.
- Difficulties
  - Increasing complexity
  - Decentralized and networked
  - Safety critical
  - Resource constrained
    - Non-functional: power, size, etc.
- Development of "invisible" embedded software

Model-based approach

- Formal models from informal requirements
  - NLP to models
  - Restricted specifications
- Model validation
- Sharing of modeling artifacts

4/27/2004

Natural Language Documents



The portion of Hepatitis B testing regulations we examined includes two documents:

- **Code of Federal Regulations 21CFR610.40**
- **Guidance Memo**

CFR and Memo documents are translated into formal models. Currently, this is done manually we are working on using natural language processing (NLP) techniques to automate it.

The multiple models are merged into a single system model, which can be analyzed using standard formal methods techniques such as reachability analysis, model checking.

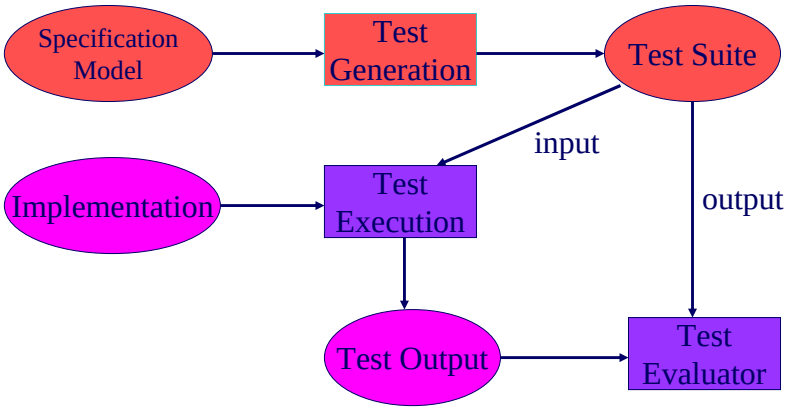
4/27/2004

Narrowing the gap between specification and implementation

- Problem:
  - Gap between an abstract model and the implementation
  - Scalability challenge (software size and complexity)
  - Validation and certification
- Approaches
  - Test generation from specification
  - Model-based code generation
  - Run-time checking/verification
  - Etc.

4/27/2004

Model-based testing



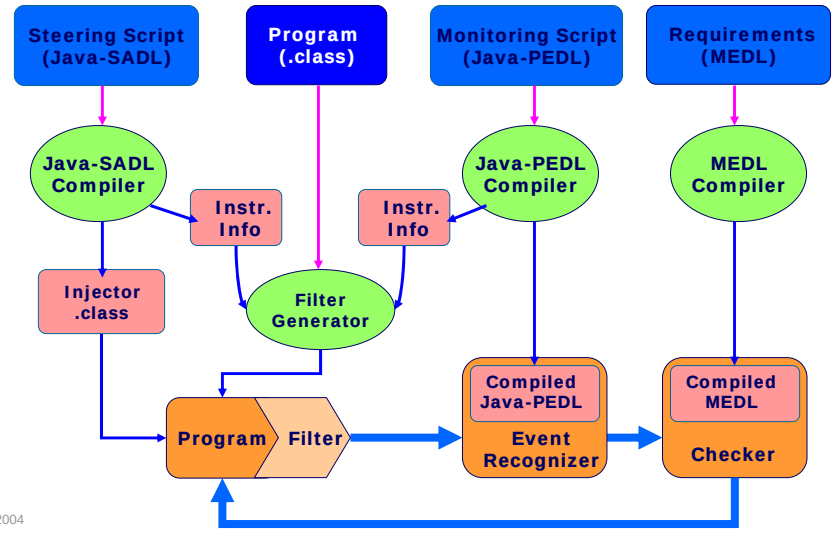
4/27/2004

Run-time verification

- Run-time monitoring and checking w.r.t. formal specification
- Ensures the runtime compliance of the current execution of a system with its formal requirement
  - **detect** incorrect execution of applications
  - **predict** error and **steer** computation
  - **collect** statistics of actual execution
- Complementary methodology to formal verification and program testing
- Prevention, avoidance, and detection & recovery

4/27/2004

Java-MaC



4/27/2004

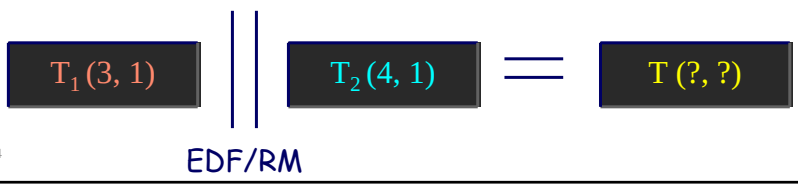
Compositionality

- Composition is necessary for large complex systems
- Composition of homogeneous models
  - Concurrent processes
  - Scheduling components (hierarchical schedulers)
  - Resource models
- Composition of heterogeneous models
  - Composition of models with different purposes (e.g., physical layout of sensor networks, model for nodes movement, communication protocols)

Compositional Real-Time Guarantees

- Real-Time Composition
  - Combine multiple real-time requirements into a single real-time requirement guaranteeing schedulability

  - Example: periodic task model T(p,e)



Certification based on models

- Conformance to models
  - Static, dynamic
  - Incremental
- Metrics, insurable software

4/27/2004

Models and Components

Stephan Merz

MOSEL project

INRIA Lorraine & LORIA, Nancy

Research background

- Formal development methods
  - B, TLA$^+$, action systems
  - systems on chip, protocols, information security
- Verification technology
  - theorem proving
  - model checking
  - combination via predicate abstraction

State of the Art

- System development
  - components, reusability, aspects
  - focus on architecture
  - rich modeling languages
- Formal development methods
  - closed systems, (re)start from scratch
  - mostly “flat” models
  - focus on specific classes of systems

Challenges

- Development of reliable components
  - model-driven development process
  - disappearing formal methods
  - system integration, test, and maintenance
- Wide-spectrum notations and tools
  - common semantic basis: event systems
  - application-specific profiles
  - back-ends: verification, performance, code generation, . . .

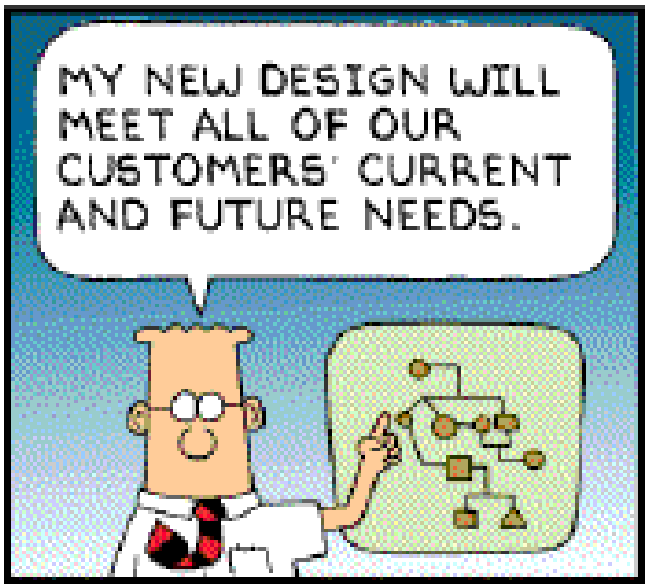
Putting change at the center of the software process

Oscar Nierstrasz

www.iam.unibe.ch/~scg

Software Composition Group

University of Bern



Plus ça change …

Real systems are characterized by continuous change

- Law of *continuing change*
- Law of *increasing complexity*
- 50-75% of development effort is *"maintenance"*
- 60% of "maintenance" is *new functionality*
- Modern methods lead to longer-lived systems, hence *more maintenance*

Status quo

Focus is still on *short-term goals*

Dangerous metaphors:

- Software "engineering"
  - *Process stops when product is ready?*
- Software "architecture"
  - *Buildings are much harder to modify*
- Software product line
  - *"Mass customization" a better metaphor?*
- Software "components"
  - *Hardware breaks you don't maintain it*
- Software "maintenance"
  - *"Maintenance" is really continuous development*

Research directions

Put *change* at the center of the software process, programming languages and tools

Methods

- Empirical research into “best practices” to support change
- Reverse engineer the informal processes
- Tie forward, reverse and re-engineering
- Meta-models for describing processes

Tools

- Tools to model, analyse and transform evolving software systems
- Support co-evolution of artifacts at various levels of abstraction, from code, through design to requirements
- Integrating tools that modify and store code; tracking change

Programming Language

- Bury file-based languages — languages to describe living systems
- Mechanisms to support coexistence of multiple running versions
- Mechanisms to support both fine and coarse-grained evolution
- Express design concepts and design evolution

Some issues in the engineering of widely deployed software intensive systems-functional variation

by Assoc. Prof. Karl Reed,FACS, FIE-Aust., MSc,ARMIT
Chair IEEE-Computer Society Tech. Council on Software Engineering
Governor, IEEE-Computer Society(1997-1999,2000-2002),
Director, Computer Sys. & Software Engineering Board, ACS,
Department of Computer Science & Computer Engineering, La Trobe University

The Problem..

1. **We have a system consisting of a collection of interacting (statically or dynamically) bound components in which either a single component or sub-system can be replaced with another…**
2. **If the functionality represented by the replaced "part" is changed,**
   - **A/What mechanisms can be provided to allow this to be propagated to the user level?**
   - **B/ How shall we (or should we) control the functionality-variation at the user level?**
   - **(Normally I'd be arguing that we need systems stability, and that this is a measure of quality???)**

Why are surprises important?

-Will customers pay for software intensive systems if they continue to be difficult to use

If not, there'll be no research funding!!

Normalised Time to Breakeven-No. learning times T/T_c	Productivity Increase Required
1	50%
1.5	30%
2.00	20%
2.5	15%
3.33	10%
7.00	5%

Agenda

1. Dynamically "varying" systems of (autonomous) dynamically linked components can lead to variant *functional* of *non-functional* behavioural variation.

2. If *functional* variation is allowed, the ability to somehow control it, and determine what is both ***acceptable*** and ***allowable*** is needed

3. **In terms of what is *acceptable*,** we can make use of a number of properties of real systems (Shaw 1999) where the internal states/transitions are quite "fuzzy", and where a user actually accepts a range of outcomes (hence, varying functionality). In fact, humans may not work with precise systems often.

   -Also, humans work with large systems whose **apparent ambiguity** (Reed, 2000) can appear as *functional* variation

4. **What is *allowable*** .. An (functional) operational envelope could be defined in principal. Functionality out-side this envelope could be rejected- BUT RECORDED, AND PRESENTED TO THE USER, WHO COULD ADD IT TO THEIR OPERATIONAL ENVELOPE (or a systems administrator could)

5. Pt.4. Constitutes a "controlled mutant adoption" process.

6. Pt.3. May involve a kind of reverse HCI,or a human-centred fault tolerant approach.

Surprise(we already deal with functional variation)….!!! *(nsf report on s/w research 1998)*

"F1. Current software has too many surprises. The sources of surprise are poorly understood."

F2. Key sources of software surprise include immature or poorly integrated software domain sciences, construction (product) principles, and engineering processes.

Sources of surprises... Real and apparent ambiguity in the means of representation of systems, e.i. Languages *(cf 3 pages of c++ with 3 pages of government regulations)*

Real and apparent unpredictability in behaviour...

"Teenagers have less trouble with PC software because they are adept at playing computer games" *Charles Wright, editor Melbourne Age "green pages" computer section 2000*

"Building 'bots' that play computer games with near human competence is not that hard" US researcher in AI….

LARGE-SCALE (UBIQUITOUS) SOFTWARE SYSTEMS CANNOT HAVE TO MANY 'SURPRISES'

WHAT ARE SURPRISES --- WHO KNOWSABOUT THEM AND WHAT CAN WE DO ABOUT THEM?

what are they?

*-A **"surprise"** (for our purposes) is some behaviour of a system's which causes or could cause a user to make an error[1], or excessive stress and discomfort in resolving the behaviour*

-Occur inherently in Lehmann's E-type systems

-Occur WHEN developers BELIEVE they are building E-type systems

***Examples..**introduction of new expense claim s/w suddenly impacts the work-loads, stress and financial security of 100's of people*

…no logical relationship between functions in s/w and semantics of menus they are in

..almost un-usable web-sites

..SCS system failures are better known

-INCLUDE some un-expected functionality (that requires some effort to interpret?)

WHAT ARE SURPRISES --- WHO KNOWSABOUT THEM AND WHAT CAN WE DO ABOUT THEM?

Who knows about them?

-The SCS community places great effort on identifying "unexpected behaviours"and controlling their impact.

-Those working on systems with adaptive behaviour and user-error recovery.

-Fault-tolerant community (already been mentioned)

-Games technologists (one of the Fraunhofer Institut's has looked at this)

-Extreme Programmers and iterative developers "think" they are dealing with "surprises"

-Product-line strategists

-lateral thinkers (here, the "surprise" is an unrecognised "use" which allows functional substitution

What can we do about them?

-Elevate their definition and management to a high-level design feature rather than an implementation problem to be avoided

-Those working on systems with adaptive behaviour and user-error recovery.

-Study the behaviour of people working in/with systems whose behaviour changes

-at the design level- deal with the adoption and control of autonomous functional variation

adoption of autonomous functional variation of due to some type of component "change"

Where functionality has been removed..

-Maybe analogous to a fault-tolerant situation (however, the system could actively seek a replacement component)

-User may agree that they can manage without this (operational envelope).

-User may seek-out a replacement function--rejecting the original

-Fault-tolerant community (already been mentioned)

-Games technologists (one of the Fraunhofer Institut's has looked at this)

-"Grace-full degradation" of OS in the 70's..

Where functionality has been changed

-Need to define "conformant" and "non-conformant" changes, e.g., a data representational change

-Need to recognise unacceptable (I.e. unadoptable at the system level) change (op-envelope issue)

-Can we define functionality extractors-correctors? (recognise and correct a functional miss-match?- a re-use - "glue-code" problem?)

Dealing with changing function..

Formal approaches….

-Component-contract specifications specify..

1. *What semantic variation in service-component that can be tolerated*
2. *Semantic definition of service's actual functionality*

-Semantic reasoning about aggregated functionality, propagating the changes upwards until either they reaches the user, OR violate some semantic constraint.

Informal Approaches.. (may be an operational approach?)

-Examine examples of functional variation visible to users,

-Using these, develop rules for specifying functional variations at the component level, and for their transmission upwards (an operational approach to the formal)

Suggest there is actually a lot of data and examples to examine..

e.g. behaviour of pc-desk top s/w applications present to the user as having unpredictable changes in functionality

relationship between knowledge, experience,skill and tools is relevant

Operational Envelope approach…

Assumes mechanisms for adopting functional variation and propagating it through a design

An (functional) operational envelope could be defined in principal. Functionality out-side this envelope could be rejected- BUT RECORDED, AND PRESENTED TO THE USER, WHO COULD ADD IT TO THEIR OPERATIONAL ENVELOPE (or a systems administrator could)

Constitutes a "controlled mutant adoption" process.

May involve a kind of reverse HCI,or a human-centred fault tolerant approach.

Use approaches from security (although of conceptual value only)--perhaps consider form of intrusion (e.g. audit trail monitoring)

Adaptive user profile generation could be part of this

Needs..

mechanisms for describing the added functionality to users and administrators

roll-back mechanisms

Research Agenda (one component)

1. Study the way humans work with large systems with **apparent ambiguity**
2. Develop the operational envelope approach..
- Develop a “controlled mutant adoption” process.

Context-Aware Software-Intensive Systems
An autonomic approach

Vladimiro Sassone

University of Sussex, UK

Engineering Software-Intensive Systems (22.05.04)

A Forthcoming Computing Paradigm

- Ubiquitous Computing: migrating software executing on networks owned and operated by others.
  Countless 'pervasive' devices equipped with limited resources and computing power will support end-to-end applications which far exceed their own capabilities.

Challenges

Scalability, Variability, Context-Awareness

- System Complexity is rocketing beyond our ability to design, comprehend and control. It approaches that of biosystems (e.g. economic systems).
- We do: Understand the behaviour of components in isolation.
  We don't: Understand the global behaviour of interacting components.
- Not likely to change soon: Need to "design for autonomy."

Autonomous Systems

- exhibit context-dependent behaviour to fulfill specific goals, possibly in complete isolation, and based on previously gathered information.

Examples

Complex biosystems
Beagle2 probe on Mars
Wireless ad-hoc networks
Electronic controlled transport systems
Health monitoring systems
...

- characterised by a degree of independence in making decisions and adapting to unforeseen environmental conditions. Often entail collaborative or competing aspects, self-organisation, emergent behaviour.

Autonomy as a Design Principle

- Need to design systems which build models of the world, gather evidence, learn, and progressively increase confidence in their own autonomous decisions.
- More ambitiously, want to apply concepts and tools from the realm of autonomic systems for the design of systems which must be highly adaptable during their lifetime.

Examples

access control systems
privacy and security systems
traffic control systems ...

- Not a bio-inspired approach: Biological systems are fundamentally non-linear, and thus largely unpredictable. They exhibit splendid properties of self-organisation, context-awareness, adaptability and autonomy, but work by trial and error, on evolutionary timescales.

Autonomy as a Design Principle

- Need to design systems which build models of the world, gather evidence, learn, and progressively increase confidence in their own autonomous decisions.
- More ambitiously, want to apply concepts and tools from the realm of autonomic systems for the design of systems which must be highly adaptable during their lifetime.

Examples

access control systems
privacy and security systems
traffic control systems . . .

- A CS approach: Provide solid foundations for autonomic systems, via a system-and-theory integrated approach: make abstract models, use them for predictions, embed the in middleware and programs.

Two paradigmatic examples

Examples

Third-party resource usage
negotation of lease for resources
pricing policies based on context-awareness
code reputation . . .

London congestion charges
traffic monitoring
pricing per time
car reputation . . .

The first steps

- Understanding autonomy at its foundations. Initially, a model for processes to explore their surroundings via risk-assessment techniques (e.g. trust and reputation).

  More generally, we look at context-awareness as the central notion to equip systems with the tools for autonomy.

Basic Enabling Theories

Concurrency & Mobility
Game Theory & MicroEconomics
Trust & Reputation Theory

Immediate Challenges

Integrate several theories
Yield models & validation mechanisms
Design languages & middleware

Example: A model of trust

$$
\begin{array}{rlr}
N, M ::= & \epsilon & \text{(empty)} \\
& \mid N \mid N & \text{(net-par)} \\
& \mid a\{\, P \,\}_{\alpha} & \text{(principal)} \\
& \mid (\nu n)\, N & \text{(new-net)}
\end{array}
\qquad
\begin{array}{rlr}
P, Q ::= & \mathbf{0} & \text{(null)} \\
& \mid Z & \text{(sub)} \\
& \mid P \mid P & \text{(par)} \\
& \mid (\nu n)\, P & \text{(new)} \\
& \mid\, !P & \text{(bang)}
\end{array}
$$

$$
\begin{array}{rlr}
Z ::= & p \cdot \tilde{u}(\tilde{v}) \,.\, P & \text{(output)} \\
& \mid \phi :: p \cdot \tilde{u}\langle \tilde{v} \rangle \,.\, P & \text{(input)} \\
& \mid Z + Z & \text{(sum)}
\end{array}
$$

Communication

$$
\frac{\beta \vdash \phi \qquad \alpha' = \alpha + [b \cdot \tilde{l} \triangleright \tilde{m}] \qquad b : \tilde{m} \odot p : \tilde{x} = \sigma}{a\{\, p \cdot \tilde{l}(\tilde{x}) \,.\, P' \,\}_{\alpha} \mid b\{\, \phi :: a \cdot \tilde{l}\langle \tilde{m} \rangle \,.\, Q \,\}_{\beta} \rightarrow a\{\, P\sigma \,\}_{\alpha'} \mid b\{\, Q \,\}_{\beta}}
$$

Conclusion

Next Step

A lot of work needed

Refine these ideas until they make sense

Find pilot projects, apply for fundings

Work Directions in Component-based Engineering

Joseph Sifakis
Verimag Grenoble
Report by M. Wirsing

Challenges (ARTIST)

- **Heterogeneity**
  - Communication mechanisms, execution speed, granularity of computation, variety of supports
  - Unified model of computation for heterogeneous integration

    Parallelism, time, resource management, performance
- **Complexity**
  - Validation effort grows exponentially with the number of components integrated
  - Replace a posteriori validation methods with incremental validation
    - ***Composability:*** preserve functionality and quality across integration process
    - ***Compositionality:*** infer the system properties from its component properties
- **Intelligence**

  Means for improving quality (robustness and perfomance) of systems
    - ***Reflexivity:*** capacity to analyze its own state
    - ***Adaptability:*** capacity to adapt behaviour according to robustness and performance objectives

Component-based Engineering

Problems of theoretical foundations

- Theory for composing heterogeneous components
- Theory for establishing correctness by construction

Sources of heterogeneity: interaction and execution

- ***Heterogeneity of interaction*** results from the presence of interactions that may be atomic or non atomic, blocking or non blocking.
- ***Heterogeneity of execution*** can be characterized by the way threads are scheduled. Synchronous and asynchronous execution reflect different scheduling policies.

Approach

Layered description of components

- Three layers: behaviour, interaction and execution models
- general associative composition operator
- composability and compositionality results for deadlock-freedom.

Software Security

Jeannette M. Wing
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA USA

Engineering Software Intensive Systems
NSF/EU Advanced Joint Workshop, Edinburgh, Scotland
22-23 May 2004

A New Research Focus: Software Security

- Security
  - It's about software, not the network.
- Software engineering
  - Forget trying to solve the general problem.
  - Solve it for one class of properties.
  - Choose that class today to be one that is critical, timely, and of societal benefit.
  - For example, security!

A Research Agenda for the Community

- Software design for security
  - What will the buffer overrun problem of tomorrow be?
- Trustworthy software
  - It's not just security, but reliability, privacy, usability, ...
- Security metrics
  - Computer Research Associates Grand Challenge #3: Make security risk management on par with quantitative financial risk management.
    - CRA Grand Challenges on Trustworthy Computing, November 16-18, 2003 http://www.cra.org/grand.challenges

Secure By Design

Anticipating tomorrow's attacks

- Above the level of code, beyond buffer overruns

What we need:

- Compositional techniques
  - To discover interface mismatches that lead to security flaws, e.g., the old Netscape+DNS problem:
    if $n2a(X) \cap n2a(Y) \neq \varnothing$
    then $\exists x \in n2a(X)\ \exists y \in n2a(Y)$ s.t. connect(x, y)
  - To anticipate emergent abusive behavior, e.g., spam, Google Bombs, ballot-stuffing e-voting "bots"
- Software design principles with security in mind
  - E.g., Defense in Depth, Principle of Least Privilege, Secure by Default
  - Something akin to Abadi and Needham's crypto protocol design principles

Secure by Design: MS03-007 Windows Server 2003 Unaffected example from David Aucsmith

Defense in Depth

The underlying DLL (NTDLL.DLL) was not vulnerable	Code made more conservative during the Security Push	Check Precondition
Even if it was vulnerable	IIS 6.0 not running by default on Windows Server 2003	Secure by Default
Even if it was running	IIS 6.0 doesn't have WebDAV enabled by default	Secure by Default
Even if it did have WebDAV enabled	Maximum URL length in IIS 6.0 is 16KB by default (> 64KB needed for exploit)	Tighten precondition, Secure by Default
Even if the buffer was large enough	Process halts rather than executes malicious code, due to buffer overrun detection code (-GS)	Tighten Postcondition, Check Precondition
Even if there was an exploitable buffer overrun	Would have occurred in *w3wp.exe* which is now running as `network service'	Least Privilege

Trustworthy Software

- Reliability
  - Focus on correctness
  - Goal: Checking interface mismatches for design-level vulnerabilities.
- Security
  - Focus on authorized access
  - Goal: Design with security in mind.
- Privacy
  - Focus on authorized use, perhaps after release
  - Goal: Identify a mathematical structure for privacy that Lampson's access matrix is for security.
- Usability
  - Humans are often the weakest link.
  - Goal: Balance between convenience and control.

Security Metrics: Quantitative Security Analysis

CRA Grand Challenge: Within 10 years, develop quantitative information-systems risk management that is at least as good as quantitative financial risk management.

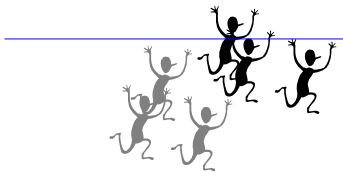
- Computing Research Associates Grand Challenges on Trustworthy Computing, November 16-18, 2003 http://www.cra.org/grand.challenges

Questions the CIO Cannot Answer:

- How much risk am I carrying?
- Am I better off this year than last?
- Am I spending the right amount of money on the right things?
- How do I compare to my peers?
- What risk transfer options do I have?

Trustworthy ~~Security~~ Axiom

Good guys and bad guys are in a never-ending race!



Model Driven Development for Software-Intensive Systems: First Results

Ira Baxter, Connie Heitmeyer, Insup Lee,
Stephan Merz, Martin Wirsing

Content

- **State of art**
- **Core Definitions**
- **Challenges**

State of the art

- Good development techniques
  including analysis, verification and code generation available at programming level, not at higher levels.
- Model checking available for RT and hybrid automata
- Lack of standards for RT components
- Static deployment of components
- Static predictability of resources

Core Definitions

Requirements

Required external (observable) behavior of the system

Model

Sound abstraction of reality

Specification

All behavioral information

Challenges

- Techniques for constructing
  very high quality requirements specifications
  including verification, validation, feedback, tradeoffs, …
- ***"Semantic component algebra"***
  Techniques for
  - decomposing systems into modules,
  - constructing systems from components in compositional way
  - transforming systems of components for optimisation and change
- ***"Disappearing formal methods"***

  Develop reasoning, analysis and transformation techniques at the level of domain-specific languages such that formal methods are just used for proving the correctness of the domain-specific techniques.

Challenges

- *SWIS design and requirements validation*
  Scaling up formal methods and tools to SWIS design and requirements including
  - model driven test case generation
  - exploitation of compositional structure for verification
- *"Power proof"*
  Develop a high level language for proofs
- Experimental challenges:
  Define a *suite of case studies for reference developments*
  from different SWI application domains

EU-NSF Workshop on Engineering Software-Intensive Systems

Report by
Don Batory
University of Texas at Austin
In cooperation with C. Heitmeyer, M. Wirsing

EU-NSF Strategic Workshop Series

- **Joint initiative of CISE-NSF and FET-EU**
- **Organized by ERCIM**
- **Goals:**

  Identify key research challenges and opportunities in Information Technologies

Software-Intensive Systems

Situation

- Daily life depends on complex SW-intensive systems
  in banking, communication, transportation, medicine, …
- New emerging technologies
  Global computation systems
  Internet, Grid, pervasive computing, service-oriented computing …
  Embedded systems
  Automotive, avionics, …

Definition

Software-intensive systems are programmable systems that include

- dynamic evolutionary systems,
- exhibit adaptive and anticipatory behaviour,
- process not only data but knowledge,
- are under user control

Application domains

- Automotive systems
- Medical devices and services
- Virtual companies and virtual business networks (Network and application management)
- Mobile and personalized Information services (Seamlessly location based and shared services)
- Web services for combination of private and business application (Intermediation and broker services)

Challenges of Engineering Software-Intensive Systems

Ludwig Maximilians Universität München

Challenge:

Create adaptable systems (including self-adaptable systems)
in which software interacts with devices, sensors, humans
ensuring required levels of quality and trust

This requires

- ***requirements elicitation and documentation***
- ***design for change*** at all levels of abstraction including
  - requirements for change, architecture for change, models for change
  - methods and techniques for reorganization of code
  - static and dynamic adaptation
- ***methods for composition*** including
  - Composition of heterogeneous components where heterogeneity e.g. deals with communication and execution platforms and with synthesized code, COTS, legacy code, hand-crafted code, code that implements abstract data types, etc.
  - Compositional reasoning techniques for critical properties (e.g., behavioral, security, safety, fault tolerance, real-time)

Challenges of Engineering Software-Intensive Systems

Ludwig Maximilians Universität München

- ***User-centred techniques for software development*** - i.e., user-friendly languages, analysis techniques, simulation/animation, techniques, etc.
  - Domain-specific
  - Hide details of formal analysis

- **Experimental challenges:**
  Define a ***suite of case studies for reference developments***
  from different SWI application domains (e.g., automatic, avionics, cell phones, medical devices, web services, ...)

- Close gap between

Foundational modeling	**Pragmatic SW solutions**
generic/ deep understanding	complete, executable
science of design	
but	but
too abstract, partial, incomplete	inconsistent, not reliable, not interoperable
…	not well structured …

Appendix A: Workshop Agenda

Saturday, May 22

9:00 - 9:30 The ERCIM workshop series by Remi Ronchaud
Introduction by Martin Wirsing

9:30 - 10:30 Presentations of challenges and research issues by participants
Don Batory, Carlo Ghezzi, Connie Heitmeyer, Luqi

11:00 - 12:30 Presentations of challenges and research issues by participants
Jeannette Wing (Report by M. Wirsing), Vladimiro Sassone, Insup Lee,
Jose Fiadeiro, Oscar Nierstrasz, Ira Baxter, Simon Dobson

14:00 - 15:00 Presentations of challenges and research issues by participants
Rance Cleaveland, Stefan Jähnichen, Stephan Merz,
Joseph Sifakis (Report by M. Wirsing), Kevin Sullivan

15:00 - 15:30 Identification of challenges and research topics

16:00 - 18:00 Discussion of topics in working groups

Sunday, May 23

8:30 - 9:30 Presentations of challenges and research issues by participants: Jeff Kramer;
Presentation and discussion of first results of the working groups

9:30 - 10:30 Discussion of topics in working groups

11:00 - 12:30 Discussion of topics in working groups

14:00 - 16:00 Presentations of results of working groups and final discussion

Appendix B: Participants and CVs

Organisation

Martin Wirsing, LMU Munich, Germany wirsing@lmu.de
Remi Ronchaud, ERCIM, France remi.ronchaud@ercim.org

Participants

Europe

Simon Dobson, Trinity College, Dublin, Ireland simon.dobson@cs.tcd.ie
Carlo Ghezzi, Politecnico di Milano, Italy carlo.ghezzi@polimi.it
Stefan Jähnichen, FhG First & TU Berlin jaehn@cs.tu-berlin.de
Jeff Kramer, Imperial College, London, GB jk@doc.ic.ac.uk
Jose Fiadeiro, University of Leicester jose@fiadeiro.org
Stephan Merz, INRIA, Nancy, France Stephan.Merz@loria.fr
Oscar Nierstrasz, Bern, Switzerland oscar.nierstrasz@acm.org
Vladimiro Sassone, University of Sussex, GB vs@susx.ac.uk

USA

Don Batory, University of Texas at Austin batory@cs.utexas.edu
Ira Baxter, Semantic Designs idbaxter@semanticdesigns.com
Valdis Berzins, US Naval Postgraduate School, Monterey berzins@cs.nps.navy.mil
Rance Cleveland, SUNY, Stony Brook rance@cs.sunysb.edu
Constance L. Heitmeyer, Naval Research Laboratory, Washington DC, heitmeyer@itd.nrl.navy.mil
Insup Lee, University of Pennsylvania, Philadelphia lee@cis.upenn.edu
Luqi, US Naval Postgraduate School, Monterey luqi@cs.nps.navy.mil
Kevin Sullivan, University of Virginia sullivan@cs.virginia.edu

Australia

Karl Reed, La Trobe University, Victoria kreed@cs.latrobe.edu.au

Contributions by

Ed Brinksma, Twente, Netherlands brinksma@cs.utwente.nl
Joseph Sifakis, Verimag, Grenoble Joseph.Sifakis@imag.fr
Jeanette Wing, Carnegie Mellon University, Pittsburgh wing@cs.cmu.edu

Don Batory
Professor
Department of Computer Sciences
University of Texas at Austin
batory@cs.utexas.edu

Don Batory holds the David Bruton Centennial Professorship at The University of Texas at Austin. He received a B.S. (1975) and M.Sc. (1977) degrees from Case Institute of Technology, and a Ph.D. (1980) from the University of Toronto. He was an Associate Editor of *IEEE Transactions on Software Engineering* (1999-2002), Associate Editor of *ACM Transactions on Database Systems* (1986-1992), a member of the *ACM Software Systems Award Committee* (1989-1993; Committee Chairman in 1992), Program Co-Chair for the *2002 Generative Programming and Component Engineering Conference*. He has given numerous tutorials on "Product-Line Architectures, Generators, and Reuse", and is an industry-consultant on product-line architectures. His research interests include generative programming, domain-specific languages, software architectures, product-lines, and databases.

Ira Baxter
Chief Technology Officer
Semantic Designs, Inc.
idbaxter@semanticdesigns.com

Dr. Baxter worked from 1970 until 1985 in industry where he designed metacompilers, time-sharing and network operating systems. He received his Ph.D. in Computer Science in 1990 from the University of California at Irvine, focusing on design reuse using transformational methods. Dr. Baxter spent several years with Schlumberger, working on a PDE-solver generator for CM-5 supercomputers, and has consulted for Rockwell International on industrial automation software engineering tools since 1994. In 1996, he founded Semantic Designs (SD) to build DMS, scalable program transformation tools to bring automation to software maintenance. Through SD, he consults on automated software analysis, transformation and domain-specific synthesis methods. Dr. Baxter is the principal designer and compiler implementer of SD's PARLANSE, the parallel programming language underlying DMS. His interests lie in program transformation, AI to support design, parallel programming and operating systems.

Ed Brinksma
Professor
University of Twente
H.Brinksma@ewi.utwente.nl

Ed Brinksma holds the chair of Formal Methods and Tools at the University of Twente in the Netherlands. His work concentrates on the application of formal methods to reactive systems, ranging from fundamental contributions to industrial applications, as well as methodological issues. In the past he has contributed to areas such as communication protocol specification, specification-based test generation, stochastic process algebra, and guided model checking. His current interests include testing theory for real-time systems, modelling and analysis of hybrid systems, and real-time scheduling synthesis.

Ed served as an editor for IEEE Transactions on Software Engineering, and is on the editorial boards of the Springer International Journals of Software Tools for Technology Transfer (STTT) and Software and System Modeling (SoSym). He is a founding member of the steering committee of the TACAS conference, and has served on the steering committees of PSTV/FORTE, ETAPS and PAPM. Ed's research group participates in a great number of (inter)national research projects with both academic and industrial partners, including the European IST projects AMETIST (timed systems) and ARTIST (embedded systems).

Rance Cleaveland
Professor and CEO
SUNY at Stony Brook and Reactive Systems, Inc.
cleaveland@reactive-systems.com

Rance Cleaveland has published over 100 research papers in the area of formal modeling and verification techniques for concurrent and distributed systems. His research interests include process algebra; model checking; verification and validation tools; formalized approaches to software engineering; and operational semantics. He is one of the original developers of the Concurrency Workbench, an automatic verification tool in continuous distribution since 1988, and Reactis®, a commercial testing and validation tool for embedded software. He is also a co-founder the Tools and Algorithms for the Construction and Analysis of Systems (TACAS) conference and has served on over 40 program committees for technical conferences. Cleaveland received his PhD in Computer Science in 1987 from Cornell University and served on the Computer Science faculties at North Carolina State University and SUNY at Stony Brook. He is currently on leave from Stony Brook to work full-time as CEO of Reactive Systems, Inc., a company he co-founded.

Simon Dobson
Lecturer
Department of Computer Science
Trinity College, Dublin IE
simon.dobson@cs.tcd.ie

Dr Simon Dobson has a research career spanning nearly fifteen years working in government, academia and industry. He began his research at the University of York working on programming environments for scalable high-performance computers. He then spent five years on the staff of the CLRC Rutherford Appleton Laboratory working on high-performance distributed systems. He moved to Trinity College Dublin in 1997 where he has worked on pervasive computing, context-aware systems, programming languages, semantics and type theory. From 2001 to 2003 he was CEO of Aurium, a start-up company he co-founded to develop context-aware software for the travel industry. He serves on the ICT Ireland committee on the Commercialisation of R&D, and has served on a number of expert advisory committees for the EU. He holds a BSc and DPhil in computer science, and is a Chartered Engineer.

José Luiz Fiadeiro
Professor
University of Leicester
Department of Computer Science
jose@fiadeiro.org

José joined the University of Leicester in November 2002 as Professor of Software Science and Engineering. He held previous academic positions at the Technical University of Lisbon and the University of Lisbon, and visiting research positions at Imperial College, King's College London, PUC–Rio de Janeiro, and the SRI International. He became chairman of the IFIP WG 1.3 (Foundations of System Specification) in January 2004. José's research interests are in software specification formalisms and methods for complex software systems. His main contributions have been in the formalisation of specification and program design techniques using modal logics, and of their underlying modularisation principles using category theory. His most recent work has focused on software architecture, including the semantics of architectural connectors and the impact of coordination mechanisms in software evolution. He is now focusing on the methodological and scientific challenges raised by service-oriented computing.

Carlo Ghezzi
Professor
Dipartimento di Elettronica e Informazione
Politecnico di Milano, Milano, Italy
carlo.ghezzi@polimi.it

Carlo Ghezzi is a Professor of Software Engineering at Politecnico di Milano, where he has been Department Chair and member of the Board of Directors. He is now a member of the Academic Senate and the Rector's delegate for Research. He held positions at the University of Padova, the Universities of California at Los Angeles and Santa Barbara, the University of North Carolina at Chapel Hill, ESLAI (Argentina), the Technical University of Vienna and the University of Klagenfurt (Austria), the University of Lugano (Switzerland).
He collaborates with industry and governmental offices; he was the Italian representative in ESPRIT (IV FP).
Ghezzi's research has been focusing on various aspects of software engineering: from programming languages supporting reliable, reusable, and evolvable software to formal methods for specification and rapid prototyping, software quality control, formalization and automation of the software process, languages and architectures for highly distributed applications. He is the co-author of 5 books and nearly 120 papers published on international journals or presented at international conferences. He is the Editor in Chief of the ACM Transactions on Software Engineering and Methodology.

Constance Heitmeyer
Head, Software Engineering Section
Center for High Assurance Computer Systems
Naval Research Laboratory
heitmeyer@itd.nrl.navy.mil

Constance Heitmeyer, who heads a group of software engineering and formal methods researchers at NRL, is the author of more than 100 technical papers and reports on software requirements, real-time computing, formal specification and verification, and computer security. She is chief designer of the SCR (Software Cost Reduction) toolset, a suite of tools for specifying and analyzing software requirements that has been transferred into software development practice. Currently, she is co-program chair for the 2nd International Conference on Hardware/Software Co-Design (MEMOCODE 2004) and co-program chair of the Experience Reports Track at the 27th ICSE. She has served on the steering committees of the IFIP 2.9 Working Group on Software Requirements, the CUE initative, and the International Requirements Engineering Conference series. She has presented numerous invited talks at universities and international conferences and has served as an associate editor of ACM TOSEM, SoSyM, the Real-Time Systems J, and the Requirements Eng. J.

Stefan Jähnichen

Director
Fraunhofer FIRST

Professor
Research group on Software Engineering - FR 5-6
Faculty for Electrical Engineering and Computer Science
Technische Universität Berlin

jaehn@first.fraunhofer.de

Professor Stefan Jähnichen, born in 1947, received his Ph.D. (Dr.-Ing.) in electrical engineering from the Technical University Berlin in 1974. Since 1998 he is managing and scientific director of the Fraunhofer Institute for Computer Architecture and Software Technology FIRST (former GMD FIRST). As a full professor he is furthermore leading the research group on software engineering at the Department of Electrical Engineering and Computer Science of the Technical University of Berlin. Stefan Jähnichen has a solid experience in software engineering especially in programming languages and compilers which he contributes to many national and international comitees such as the IFIP Working Group 2.4 in System Programming Languages, the German Techonology Cooperation with Latin-America (Brazil, Argentina, Chile and Mexico) or the Scientific Advisory Board (Fachkolleg) for Informatics of the German Research Foundation (DFG). Stefan Jähnichen is chief editor of the German research journal Informatik: Forschung & Entwicklung as well as author of 10 books and approximately 50 papers in refereed conference proceedings and journals.

Jeffrey Kramer
Head, Department of Computing
Imperial College London
j.kramer@imperial.ac.uk

Professor Jeff Kramer is Head of the Department of Computing at Imperial College. His research interests include requirements engineering, software architectures and analysis techniques, particularly as applied to concurrent and distributed software. He was a principal investigator in the various research projects which led to the development of the CONIC environment for configuration programming and the Darwin architectural description language. His current research work is on behaviour analysis, the use of models in requirements elaboration and architectural approaches to self-organising software systems.

Jeff Kramer is a Chartered Engineer, Fellow of the IEE and Fellow of the ACM. He was program co-chair of the 21st ICSE (International Conference on Software Engineering) in Los Angeles in 1999, Chair of the Steering Committee for ICSE from 2000 to 2002, associate editor and member of the editorial board of ACM TOSEM from 1995 to 2001 and is currently associate editor and member of the editorial board of IEEE TSE. He is co-author of a recent book on Concurrency, co-author of a previous book on Distributed Systems and Computer Networks, and the author of over 150 journal and conference publications.

Luqi
Professor
Computer Science Department
Naval Postgraduate School
luqi@nps.edu

Prof. Luqi, IEEE fellow, Professor of Naval Postgraduate School. Her research interest and expertise in the past 20 years includes system automation, hardware and software integration, software-intensive system modeling, safety-critical system design, computer-aided prototyping, real-time and embedded systems, signal processing, specification languages, requirements engineering, and software architecture. She funded software engineering automation center and works with faculty and graduate students on hundreds of real world projects on software intensive systems with the IEEE Technical Achievement Award in this area. She has served on many editorial boards, including IEEE Software, Expert, and Transactions on Software Engineering, and many conferences and workshops. She has published over 200 refereed publications and has supervised hundreds of MS and Ph.D. students.

Insup Lee
Professor
University of Pennsylvania
Department of Computer and Information Science
lee@cis.upenn.edu

Insup Lee is Professor in the Department of Computer and Information Science at the University of Pennsylvania. He received a Ph.D. degree in Computer Science from Univeristy of Wisconsin, 1983. His research interests include, embedded systems, real-time computing, formal methods, wireless network, and software engineering. He has developed programming concepts, language constructs, and operating systems for real-time systems. In recent years, he has developed specification, analysis, and testing techniques based on real-time process algebra (ACSR). In addition, he has devoloped a hierarchical specification language for hybrid systems (CHARON). Based on CHARON, he is currently developing techniques for automatic code generation and test generation. Furthermore, he is currently working in proactive sensor networks. He also has benn developing the run-time monitoring and checking framework (MaC) that can be used to assure the correctness of a running system through monitoring and checking of safety and QoS properties. The prototype MaC system has been implemented in Java and is currently being ported to Real-Time Java.

Stephan Merz
Directeur de Recherches
INRIA Lorraine & LORIA
Stephan.Merz@loria.fr

1987	Diplom, Technische Universität, Munich
1992	Ph.D., Ludwig-Maximilians-Universität, Munich
1993/94	postdoctoral stays at IRIT, Toulouse and at Systems Research Center, Digital Equipment Corporation, Palo Alto, CA
1995-98	research grant, state of Bavaria, Germany
1998-2002	assistant professor, Ludwig-Maximilians-Universität, Munich
2002	habilitation degree, Ludwig-Maximilians-Universität, Munich
2002-	directeur de recherches (senior researcher), INRIA Lorraine, Nancy, France

Oscar Nierstrasz
Professor of Computer Science
Institute of Computer Science (IAM)
University of Bern
oscar@iam.unibe.ch

Oscar Nierstrasz is a Professor of Computer Science at the Institute of Computer Science (IAM) of the University of Bern, since 1994, where he leads the Software Composition Group. Prof. Nierstrasz is the author of over seventy publications and co-author of the book Object-Oriented Reengineering Patterns (Morgan Kaufmann, 2003).

The Software Composition Group carries out research in diverse aspects of how to make systems more flexible with respect to changing requirements. Current research is focussed on (i) programming languages and mechanisms to support the flexible composition of high-level, component-based abstractions, and (ii) tools and environments to support the understanding, analysis and transformation of software systems to more flexible, component-based designs.

Karl Reed
Assoc. Prof.
Department Computer Scicnce and Computer Engineering
La Trobe University
kreed@cs.latobe.edu.au

Karl Reed is a pioneer of software engineering education in Australia, and is recognised as national spokesperson on industry policy, advising State and Federal Governments . Associate Professor Reed has held positions, including Senior Visiting Fellow in the Faculty of Business at the Royal Melbourne Institute of Technology University. He is a Fellow and an Honorary Life Member of the Australian Computer Society, and Director of its Computer Systems and Software Engineering Board. He was consultant editor to Australasian Computer World from 1978 to 1995. He was an Honorary Visiting Professor at the University of Middlesex from 2000 to 2002, and a Guest Scientist at the Fraunhofer Insiittute for experimental Software Engineering (2003-2003). He was a Governor of the IEEE-Computer Society f (1997-2000,2000-2002), and is currently the Chair of the IEEE-CS' Technical Council on Software Engineering (2000-2002,2002-2004). Reed is currently in the Computer Science and Computer Engineering Department at La Trobe University, and was the Director of the Amdahl Australian Intelligent Tools Program from 1989 to 1996. His research interests include software testing, the nature of software engineering, Compilable restricted natural languages, Web-page design, high-level software re-use, design isomorphisms, process comparison software architecture,emailusage and industry policy.

Vladimiro Sassone
Professor
University of Sussex
vs@susx.ac.uk

My research activity concerns the semantics of concurrency and mobility. A central theme thereof regards methods for safe resource management in mobile, distributed systems, aimed at laying the foundations of robust, high-level programming paradigms for global ubiquitous computing. Recently my interests expanded to include trust management and reputation systems.
I am the Coordinator of the EU funded project ``MyThS: Models and Types for Security in Distributed Systems" (2002--2005), the Principal Investigator of the EPSRC ``Third-Party Resource Usage for Pervasive Computing" (2003--2006), and the Director of the EU Marie Curie Research Training Centre ``DisCo: Foundations of Distributed Computation" (2002--2005). After completing my PhD in 1994, I held a Marie Curie Mobility Fellowship. I was since employed in Aarhus, Pisa, London, Catania, Copenhagen and Sussex.

Jeannette Wing
Professor and Associate Dean
Department of Computer Science
Carnegie Mellon University
wing@cs.cmu.edu

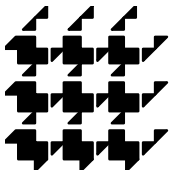
Dr. Jeannette M. Wing is a Professor of Computer Science at Carnegie Mellon University. She is the Associate Dean for Academic Affairs for the School of Computer Science and the Associate Department Head for the Computer Science Ph.D. Program. She received her S.B. and S.M. degrees in Electrical Engineering and Computer Science in 1979 and her Ph.D. degree in Computer Science in 1983, all from MIT.

Professor Wing is the author or co-author of over 80 refereed journal and conference papers, has presented over 160 invited and conference talks, and is or was on the editorial board of seven journals. She is a member of the National Academies of Science's Computer Science and Telecommunications Board and the Microsoft Trustworthy Computing Academic Advisory Board. She was a member of the DARPA Information and Science Technology Study (ISAT) Group and the National Science Foundation Scientific Advisory Board. Professor Wing is an ACM Fellow and an IEEE Fellow.

Martin Wirsing
Professor
Institut für Informatik
Ludwig-Maximilians-Universität München
wirsing@lmu.de

Martin Wirsing is a Full Professor and Chair for Computer Science at Ludwig-Maximilians-University of Munich, where he is also the head of the research group on Programming and Software Technology. .He received his master, diploma and PhD degree in Mathematics from the Universities of Paris 7 and Munich in 1971,1974 and 1976, and his habilitation degree in Computer Science from Technical University of Munich in 1984.
Martin Wirsing is the editor of more than 15 books and has published more than 150 scientific papers. His current research interests comprise software engineering for distributed mobile systems and for hypermedia applications, object-oriented software development based on formal methods, design and semantics of concurrent Java programs.
Martin Wirsing is member of the scientific committees of INRIA France, LORIA (Nancy, France), and the Institute of Informatics of Hanoi (Vietnam). He is member of the editorial board of several scientific journals and book series including Theoretical Computer Science and Journal of Computer Science and Technology. He was and is involved in many national and international projects sponsored among others by the European Commission, the Deutsche Forschungsgemeinschaft and the German Ministry for Education and Research. In the current Global Computing Initiative of the EC he is coordinating the project AGILE.

FET - Future and Emerging Technologies

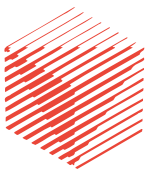


DIMACS — Center for Discrete Mathematics & Theoretical Computer Science

European Research Consortium for Informatics and Mathematics

ERCIM

www.ercim.org



This workshop is part of a series of strategic workshops to identify key research challenges and opportunities in Information Technology. These workshops are organised by ERCIM, the European Research Consortium for Informatics and Mathematics, and DIMACS the Center for Discrete Mathematics & Theoretical Computer Science. This initiative is supported jointly by the European Commission's Information Society Technologies Programme, Future and Emerging Technologies Activity, and the US National Science Foundation, Directorate for Computer and Information Science and Engineering.

More information about this initiative, other workshops, as well as an electronic version of this report are available on the ERCIM website at http://www.ercim.org/EU-NSF/