Software Engineering: A Roadmap

Anthony Finkelstein
Department of Computer Science
University College London
Gower St.
London WC1E 6BT, UK
+44 020 7380 7293
a.finkelstein@cs.ucl.ac.uk

Jeff Kramer
Department of Computing
Imperial College
180 Queens Gate
London SW7 2BZ
+44 020 7594 8271
jk@doc.ic.ac.uk

ABSTRACT
This paper provides a roadmap for software engineering. It identifies the principal research challenges being faced by the discipline and brings together the threads derived from the key research specialisations within software engineering. The paper draws heavily on the roadmaps covering specific areas of software engineering research collected in this volume.

Keywords
software engineering, research, discipline, future, strategy

1 INTRODUCTION

This paper attempts to construct a roadmap for software engineering research. It seeks to identify the principal research challenges being faced by the discipline and to bring together the threads derived from the key research specialisations within software engineering. In doing so it draws heavily on the roadmaps covering specific areas of software engineering research collected in this volume.

Definitions are notoriously difficult but for working purposes and those of this volume - software engineering is the branch of systems engineering concerned with the development of large and complex software intensive systems. It focuses on: the real-world goals for, services provided by, and constraints on such systems; the precise specification of system structure and behaviour, and the implementation of these specifications; the activities required in order to develop an assurance that the specifications and real-world goals have been met; the evolution of such systems over time and across system families. It is also concerned with the processes, methods and tools for the development of software intensive systems in an economic and timely manner.

We do not aim to provide a summary of the overall state-of-the-art in software engineering. The reader interested in a general introduction should refer to the many excellent textbooks that are available. Best known, and a good starting point, are [5], [3] and [2], all of which are reasonably up to date. [1] affords a good start to the broader literature. The research literature on software engineering is readily available. IEEE Transactions on Software Engineering (IEEE-TSE) and ACM Transactions on Software Engineering and Methodology (ACM-TOSEM) are the principal archival journals. There are a large number of specialised journals including for example Automated Software Engineering (ASE), Requirements Engineering Journal (REJ), Software Process Journal (SPJ). IEEE Software plays an important role in bridging between the 'pure' research literature and practitioner-oriented articles. The International Conference on Software Engineering (ICSE) is the flagship conference of the software engineering community; papers in this conference are generally of a high standard and the proceedings reflect a broad view of research across software engineering. The European Software Engineering Conferences (ESEC) and the Foundations of Software Engineering Conferences (FSE), which are held jointly in alternate years, are similar to ICSE though have tended historically to have a slightly more 'theoretical' orientation. There are a large number of specialised conferences and workshops ranging from established meetings such as the International Workshop on Software Specification and Design, International Software Architecture Workshop, International Symposium on Software Testing and Analysis to the 'hot-topic' workshops held in conjunction with ICSE. There are excellent resources on the web (links are provided on the web site associated with this volume). General software engineering announcements are distributed through the community-wide "seworld" mailing list.

The structure of the paper is approximately as follows. In sections 2 and 3 we discuss the changing context of software system development and the changing orientation of software engineering research. In section 4 we make some broad observations about the evolution of the discipline. Then, in section 5, we analyse the key research

challenges and show how these challenges are reflected in the specialised roadmaps that comprise the volume. We finish by drawing some broad and necessarily speculative and personal conclusions about the future of software engineering.

2 CONTEXT

The context of software system development is changing. Systems are rarely developed from scratch; most system development involves extension of preexisting systems and integration with 'legacy' infrastructure. These systems are embedded in complex, highly dynamic, decentralised organisations; they are required to support business and industrial processes which are continually reorganised to meet changing consumer demands. The services that such a system provides must, for the life of the system, satisfy the requirements of a diverse and shifting group of stakeholders. There is a shift towards client and user centered approaches to development and an accompanying shift from a concern with whether a system will work towards how well it will work. Overall, fewer 'bespoke' software systems are being constructed. Instead, generic components are built to be sold into markets. Components are selected and purchased 'off the shelf' with development effort being refocused on configuration and interoperability.

The resulting systems are composed from autonomous, locally managed, heterogeneous components, which are required to cooperate to provide complex services. They are, in general, distributed and have significant non-functional constraints on their operation. There are a wide range of new, and constantly changing business models relating to the provision of software and software-mediated services resulting from internet and e-commerce technology.

The overall setting is characterised by on the one hand an increasing business dependence on reliability of software infrastructure and on the other hand rapid change and reconfiguration of business services necessitating rapid software development and frequent change to that software infrastructure.

3 ORIENTATION

Reflecting an increased disciplinary maturity the 'orientation' of software engineering research has changed. Software engineering research is being more carefully targeted towards "real" industrial problems. This entails thorough problem analysis. Increasingly software engineering research is aimed towards engineering solutions that are lightweight, in the sense that they make minimal assumptions about the engineering environment in which they are deployed. This is often related to the need for solutions to be simple enough that they can be adopted in practice. Much, but by no means all, software engineering research to date has been characterised by overly complex, heavyweight, solutions that have, understandably, encountered significant resistance from practitioners.

It has taken a long time for researchers to realise that we cannot expect industry to make very large big-bang changes to processes, methods and tools, at any rate without substantial evidence of the value derivable from those changes. This, accompanied again by the increased disciplinary maturity, has lead to a higher "validity" barrier which research contributions must cross. It is readily observable that research that proposes new frameworks, methods and processes are not accepted without positive evidence that they are of use rather than simply airy and unfounded speculation.

Particular attention is being paid to the issue of scalability. It has, in the past, proved all too easy for researchers to ignore or make light of the problems that the shear scale of industrial software systems development gives rise to. Problems that appear simple in paper-and-pencil exercises in the laboratory are often far from simple when dealing with very large amounts of data. The internet too has implications for scalability. In an open internet setting there may be millions of potential users of a software service.

While research methodology remains a potent issue there is some evidence of an increasing acceptance of methodological diversity. Case studies, qualitative studies, experiments, proof and mathematical analysis are being combined judiciously to make a case for research contributions. Research is expected to, and increasingly does, build on the work of others. Using existing standards and building on, rather than in parallel to, proven research contributions characterises the best research. There is however less tolerance for reinventing the wheel.

4 DISCIPLINE

In defining software engineering we described it as a "branch of systems engineering". Unfortunately systems engineering, despite a long history, is less mature than software engineering! Software engineering research is increasingly aware of the interplay between systems context and software and there are attempts to take into account the co-development of hardware and organisational systems with software. We anticipate a further shift in orientation among software engineers towards a broader systems engineering view with software engineers taking a lead in the creation of a truly integrated systems engineering discipline

A traditional theme in software engineering discourse has been "why can't we build software like other engineers build bridges" [or similar traditional engineering product]. This refrain has lead to some productive thinking and has forced software engineers to think hard about achievements, aspirations and the status of our claim to be 'engineers'. It has however lead us to apply inappropriate analogies with, for example, mechanical and other

artefacts, which have fundamentally different characteristics from software. It has also given rise to a perceived sense of inferiority, unjustified by the significant research accomplishments of software engineering. We believe that this self-deprecation has in turn had an adverse effect on the funding and the status of software engineering in computer science. There is some evidence that this traditional theme is finally becoming less frequently heard and that a more robust self-image with respect to other disciplines is emerging. We need to recognize, claim and publicize our many successes.

Software engineering has, to a large extent, historically defined itself in terms of testing and debugging. Most software engineering textbooks start with a discussion of early error detection and removal. Software engineers seem to enjoy talking about errors. Failures such as that of the London Ambulance Service Computer Aided Despatch system and the Ariane 5 receive much attention. While such a focus on failures can be instructive - even construction engineers study bridge failures as a means of learning lessons - it can also be said to lead to a negative orientation in which the absence of bugs rather than the positive presence of quality, however defined, is the most important goal. The changing context of software development in which there is a pressing need to roll out a service rapidly and to change it to meet new business demands, forces a change in this outlook towards a more positive 'holistic' view of the role of software engineering in delivering satisfaction to users.

5 RESEARCH CHALLENGES

A wholly mature discipline is one that is able to identify "dead problems". Dead problems are those to which effort need no longer be devoted because they have been solved. Computer science is slowly building up a list of such dead problems reinforcing its claim as a mature discipline. By contrast software engineering is still struggling to identify its own dead problems. Any list we constructed of such problems would almost certainly be more controversial than the attempt, which follows, to pick out key research challenges across software engineering. These overall software engineering research challenges are not comprehensive and the determined reader can infer our view of dead problems from it. The questions associated with each challenge are exemplars and individual researchers may derive much more specific research questions. Clearly these challenges are not orthogonal and there are complex relationships binding them together. Many of the most interesting research programmes look at these relationships.

- *Compositionality* - When we compose components what effect does this have on the properties of those components? Can we reason about, and engineer for, the emergent properties of systems composed from components whose behaviour we understand?
- *Change* - How can we cope with requirements change? How can we build systems that are more resilient or adaptive under change? How can we predict the effects of such changes?
- *Non-functional Properties* - How can we model non-functional properties of systems and reason about them, particularly in the early stages of system development? How can these models be integrated with other models used in system development?
- *Service-view* - How can we shift from a traditional product-oriented view of software system development towards a service view? What effects do new modes of software service delivery have on software development?
- *Perspectives* - How can we devise and support new structuring schemes and methods for separating concerns?
- *Non-classical life cycles* - How can we adapt conventional software engineering methods and techniques to work in evolutionary, rapid, extreme and other non-classical styles of software development?
- *Architecture* - How can we represent, reason about and manage the evolution of software architectures? How can we relate software architecture to other parts of the software development process?
- *Configurability* - How can we allow users, in the broadest sense, to use components in order to configure, customize and evolve systems?
- *Domain specificity* - How can we exploit the properties of particular domains (telecommunications, transport) to make any of these challenges easier to address?

The detailed tables that follow look at these challenges set against the detailed challenges or pointers identified by the authors of the individual roadmaps. The tables also serve as a useful quick reference to the volume. A grayed box indicates a clear and straightforwardly identifiable relationship between a 'big' challenge and a more fine-grained one. Also included in the tables are links or cross-references from one set of fine-grain challenges to another.

While the challenges that we have been identified do not subsume all of the issues raised by the roadmaps they appear to subtly impact many of them. Very large proportions of the fine-grained challenges relate to the big challenges. Most of those which do not, relate different areas of research activity as indicated by links. A small proportion of the fine-grained challenges relates to neither big challenges nor other parts of the research agenda. These are, for the most part, technical problems blocking advances in particular areas.

6 CONCLUSIONS

This paper takes a positive view of current progress and future challenges in software engineering. We believe the discipline has delivered and is well set to continue to deliver both practical support to software developers and the theoretical frameworks which will allow that practical support to be adopted, used and extended with confidence. It is well known that software engineering innovations take a surprisingly long time to percolate through to every day use [4]. Despite this lag current software engineering practice is being radically reshaped by object-oriented design methods, CASE tools with powerful code generation, testing and analysis environments, development patterns, incremental delivery based life-cycles, component models and document management environments. All of these have been formed through software engineering research.

A vision of the future of software engineering suggests a setting in which developers are able to wire together distributed components and services (heterogeneous and sourced over the net) having established at an early stage, through rigorous (yet easy-to-use) formal analysis that the particular configuration will meet the requirements (both functional and non-functional). The overall process in which this takes place will have seamless tool support that extends through to change over the system or service life. Each facet of the resulting system or service will be traceable to (and from) the originating stakeholders who will be involved throughout the process.

This vision is in fact an old one! The difference is that making it a reality is now within our grasp. We know what we have to do. What makes our field even more exciting is that, in addition to the steady progress towards our vision, there are also the discontinuities, such as was introduced in the last ten years by the web. The impact of such major innovations cannot be predicted but they certainly offer wonderful new opportunities and challenges.

REFERENCES

1. Dorfman, M. & Thayer, R.H. (Eds) Software Engineering, (November 1999), IEEE Computer Society.
2. Ghezzi, C. Jazayeri, M. & Mandrioli, D. Fundamentals of Software Engineering, (January 1991), Prentice Hall.
3. Pressman, R.S. Software Engineering : A Practitioner's, 4th edition (August 1996), McGraw Hill College Div.
4. Redwine S.T. & Riddle, W.E. Software Technology Maturation, *Proceedings of the 8th International Conference on Software Engineering*, 1985, pp 189-200, IEEE Computer Society.
5. Sommerville, I. Software Engineering (International Computer Science Series), 5th edition (November 1995) Addison-Wesley Pub Co.

1 Software Process		Compositionality	Change	NF Properties	Service view	Perspectives	Lifecycles	Architecture	Configurability	Domain specificity	Links
1.1	PML must be tolerant and allow for incomplete, informal, and partial specification										
1.2	PSEE must be non-intrusive. It must be possible to deploy them incrementally.										
1.3	PSEE must provide the software engineer with a clear state of the software development process (from many different viewpoints).										18
1.4	The scope of software improvement methods and models should be widened in order to consider all the different factors affecting software development activities. We should reuse the experiences gained in other business domains and in organizational behavior research.										
1.5	Statistics is not the only source of knowledge. We should also appreciate the value of qualitative observations.										23

2 Requirements Engineering		Compositionality	Change	NF Properties	Service view	Perspectives	Lifecycles	Architecture	Configurability	Domain specificity	Links
2.1	Better modelling and analysis of problem domains, as opposed to the behaviour of software.										
2.2	Development of richer models for capturing and analysing non-functional requirements.										
2.3	Bridging the gap between requirements elicitation approaches based on contextual enquiry and more formal specification and analysis techniques.										10
2.4	Better understanding of the impact of software architectural choices on the prioritisation and evolution of requirements.										
2.5	Reuse of requirements models to facilitate the development of system families and the selection of COTS.										
2.6	Multi-disciplinary training for requirements practitioners.										25

3 Reverse Engineering		Compositionality	Change	NF Properties	Service view	Perspectives	Lifecycles	Architecture	Configurability	Domain specificity	Links
3.1	Teach reverse engineering, program understanding, and software analysis in computer science, computer engineering, and software engineering curricula.										25
3.2	Investigate infrastructure, methods, and tools for continuous program understanding to support the entire evolution of a software system from the early design stages to the long-term legacy stages.										
3.3	Develop methods and technology for computer-aided data and database reverse engineering.										
3.4	Develop tools that provide better support for human reasoning in an incremental and evolutionary reverse engineering process that can be customized to different application contexts.										
3.5	Concentrate on the tool adoption problem by improving the usability and end-user programmability of reverse engineering tools to ease their integration into actual development processes.										

4 Testing		Compositionality	Change	NF Properties	Service view	Perspectives	Lifecycles	Architecture	Configurability	Domain specificity	Links
4.1	Development of techniques and tools that will help component users integrate and test the components with their applications more efficiently and effectively										
4.2	Creation of techniques and tools that can use precode artifacts, such as architectural specifications, for planning and implementing testing activities.										6
4.3	Development of techniques and tools for use in estimating, predicting, and performing testing on evolving software systems.										
4.4	Establishment of effective processes for analyzing and testing software systems.										
4.5	Investigation of methods that use testing artifacts to assist in software development.										

5 Software Maintenance and Evolution		Compositionality	Change	NF Properties	Service view	Perspectives	Lifecycles	Architecture	Configurability	Domain specificity	Links
5.1	The production of new management approaches to evolution, leading to better understanding of the relationships between technology and business.		■								
5.2	How can software be designed so that it can easily be evolved?		■					■			
5.3	More effective tools and methods for program comprehension for both code and data										3
5.4	A better formalism and conceptualisation of 'maintainability'; how do we measure it?			■							
5.5	The development of a service-based model of software, to replace a product view.				■						

6 Software Architecture		Compositionality	Change	NF Properties	Service view	Perspectives	Lifecycles	Architecture	Configurability	Domain specificity	Links
6.1	Software architectures that support dynamic coalitions of software services.		■		■			■			
6.2	New techniques for composing heterogeneous components, and certifying the properties of those compositions.	■						■			
6.3	Software architectures that adapting themselves to their physical setting.							■			17
6.4	Design principles for making architectural tradeoffs between correctness, resource consumption, and reliability.		■					■			
6.5	Self-monitoring systems.							■			

7 Object-oriented modelling		Compositionality	Change	NF Properties	Service view	Perspectives	Lifecycles	Architecture	Configurability	Domain specificity	Links
7.1	To identify appropriate language means for modelling an "aspect" of a system.					■					
7.2	To separate a core modelling language from domain-specific extensions.									■	
7.3	To define the semantics of a high-level, heterogeneous modelling language										10
7.4	To develop means to compose and to refine complex structured models.	■									
7.5	To identify guidelines for an incremental, round-trip software development process						■				

8 Software Engineering for Middleware		Compositionality	Change	NF Properties	Service view	Perspectives	Lifecycles	Architecture	Configurability	Domain specificity	Links
8.1	A large class of distributed systems need not be built from scratch but can exploit middleware to resolve heterogeneity and distribution of the system components.				■						
8.2	State of practice middleware products enable software engineers to build systems that are distributed across a local area network.								■		
8.3	The state of the art in middleware research aims to push this boundary towards Internet-scale distribution, adaptive systems and middleware that can meet reliability and hard real-time constraints.			■							
8.4	The software engineering challenges lie in devising methods, notations and tools for distributed system construction that systematically build and exploit what middleware products will deliver, now and in the future.										7
8.5	Software engineering research can contribute to the further development of middleware, particularly in the areas of version- and configuration management and development environments										18, 19

9 Software Analysis		Compositionality	Change	NF Properties	Service view	Perspectives	Lifecycles	Architecture	Configurability	Domain specificity	Links
9.1	Checking conformance of code to designs is likely to become a central problem for software analysis.										
9.2	Tools that analyze designs in their own right will grow in importance.										
9.3	Abstract design models are the lynchpin for exploiting code analyses in this context: they not only make the analysis results more relevant, but can be used to focus the analysis and extend it.										
9.4	Both powerful tools that can check complex properties and simpler tools that provide rapid but rough results will be useful.										
9.5	Many kinds of analysis will play a role: static and dynamic, sound and unsound, operational and declarative.										

10 Formal Specification		Compositionality	Change	NF Properties	Service view	Perspectives	Lifecycles	Architecture	Configurability	Domain specificity	Links
10.1	Formal specification technology needs to provide *constructive* methods for specification development, analysis, and evolution.										
10.2	Formal specifications need to be fully integrated with other software products and processes all along the software lifecycle.										
10.3	Specification techniques should move from functional design to requirements engineering; higher-level, problem-oriented ontologies must therefore be supported instead of program-oriented ones.										
10.4	The scope of formal specification and analysis must be extended to cover non-functional requirements that play a prominent role in architectural design --such as performance, security, fault tolerance, accuracy, maintainability, etc.										
10.5	Tomorrow's technology will provide lightweight interfaces for multiparadigm specification and analysis.										9

11 Mathematical Foundations of Software Engineering		Compositionality	Change	NF Properties	Service view	Perspectives	Lifecycles	Architecture	Configurability	Domain specificity	Links
11.1	Representing behaviour (including concurrency and duration of activities) and being able to analyse it. There may be many different kinds of behaviour and there is no obvious necessity to have a universal representation - quite the opposite! Engineering tools are the most useful when they are specific, so different classes of problems may demand differing languages to represent them.					■					
11.2	Representing 'ility' properties and devising corresponding engineering theories enabling the use of the 'ility' in design.			■							
11.3	Systematising domain knowledge for the area of application. This is hard, long and often tedious work.									■	
11.4	Defining the specification and refinement patterns/architectures required to encapsulate design choices. This results in support for the 'cookbook' aspects of normal design. The work on product line architectures, if properly driven towards formal engineering systematisation, will contribute enormously to this.							■			
11.5	Better understanding of modularity principles. The only effective method for dealing with the complexity of software based systems is decomposition. Modularity is a property of systems, which reflects the extent to which it is decomposable into parts, from the properties of which we are able to predict the properties of the whole. Languages that do not have sufficiently strong modularity properties are doomed to failure, in so far as predictable design is concerned.	■									
11.6	Improved and specialised analysis tools, many more abstraction (interpretation) tools to address feasibility/tractability of analysis; more and specialised decision procedures for interesting properties (using abstractions to approximate same).										9

12 Software Reliability & Dependability		Compositionality	Change	NF Properties	Service view	Perspectives	Lifecycles	Architecture	Configurability	Domain specificity	Links
12.1	Shifting the focus from software reliability to user-centred measures of dependability in complete software-based systems.										
12.2	Influencing design practice to facilitate dependability assessment.										
12.3	Propagating awareness of dependability issues and the use of existing, useful methods.										25
12.4	Injecting some rigour in the use of process-related evidence for dependability assessment.										
12.5	Better understanding issues of diversity and variation as drivers of dependability.										

13 Software Engineering for Performance		Compositionality	Change	NF Properties	Service view	Perspectives	Lifecycles	Architecture	Configurability	Domain specificity	Links
13.1	To create a well understood formalism, probably based on UML, allowing performance annotations to design models.										7
13.2	To create a methodology which embeds performance questions within the software lifecycle in terms of widely used approaches.										
13.3	To integrate solution tools for performance measures transparently within extended design tools, such as object oriented CASE tools.										
13.4	To develop ways of returning performance results from specialised tools in terms of the design models from which they were derived.										
13.5	To integrate performance modelling measures within a performance monitoring and testing framework in a consistent manner.										4

14 Software Engineering for Real-Time		Compositionality	Change	NF Properties	Service view	Perspectives	Lifecycles	Architecture	Configurability	Domain specificity	Links
14.1	The development of a system architecture that supports the precise specification of the interfaces between components in the value domain and in the temporal domain, such that the components can be developed and tested independently.	■						■			4
14.2	The constructive integration of exisiting prevalidated components into diverse system contexts.	■									
14.3	The systematic validation of ultradependable real-time systems that are used in safety critical applications.			■					■		
14.4	The development of a framework that supports the generic implementation of fault-tolerance without introducing additional complexity into the application software.										12
14.5	The derivation of tight upper bounds for the worst-case execution time of real-time programs.										

15 Software Engineering for Safety		Compositionality	Change	NF Properties	Service view	Perspectives	Lifecycles	Architecture	Configurability	Domain specificity	Links
15.1	Provide readier access to formal methods for developers of safety-critical systems by further integration of informal and formal methods.										10, 7
15.2	Develop better methods for safety analysis of product families and safe reuse of Commercial-Off-The-Shelf software.	■									
15.3	Improve the testing and evaluation of safety-critical systems through the use of requirements-based testing, evaluation from multiple sources, model consistency, and virtual environments.										4
15.4	Advance the use of runtime monitoring to detect faults and recover to a safe state, as well as to profile system usage to enhance safety analyses.										
15.5	Promote collaboration with related fields in order to exploit advances in areas such as security and survivability, software architecture, theoretical computer science, human factors engineering, and software engineering education.										16, 12, 6, 11, 25

16 Software Engineering for Security		Compositionality	Change	NF Properties	Service view	Perspectives	Lifecycles	Architecture	Configurability	Domain specificity	Links
16.1	Integrating security considerations smoothly into early life-cycle activities: uniform application of cost-benefit analyses to both functional, and security requirements; unified modelling approaches to integrate the engineering of both functional requirements and security requirements.										2
16.2	The development of architectures and designs that are easier to adapt to rapidly evolving security policies, and approaches to facilitate the integration of security features into legacy systems.										
16.3	The invention of cogent, flexible economic models of adversary behaviour which can underly the rational design of software copy-protection and watermarking techniques in different application contexts.										
16.4	Better techniques for formulating desirable security properties, and the development of scalable, predictable, static and dynamic verification tools to evaluate the security of software systems.										9
16.5	The development of automated, robust, flexible infra-structures for post-deployment system administration, that can adapt to the organization's confidentiality, non-repudiation, and trust-delegation requirements.										8

17 Software Engineering for Mobility		Compositionality	Change	NF Properties	Service view	Perspectives	Lifecycles	Architecture	Configurability	Domain specificity	Links
17.1	Mobility challenges old assumptions and demands novel software engineering solutions including new models, algorithms, and middleware.										
17.2	Coordination mechanisms must be developed to bridge effectively a clean abstract model of mobility and the technical opportunities and complexities of wireless technology, device miniaturization, and code mobility.										
17.3	Logical mobility opens up a broad range of new design opportunities, physical mobility forces consideration of an entirely new set of technical constraints, and the integration of the two is an important juncture in the evolution of software engineering as a field.										
17.4	Key concepts shaping software engineering research on mobility are the choice of unit of mobility, the definition of space and location, and the notion of context management.										

17 Software Engineering for Mobility		Compositionality	Change	NF Properties	Service view	Perspectives	Lifecycles	Architecture	Configurability	Domain specificity	Links
17.5	Middleware is the most likely vehicle by which novel perspectives on mobility grounded in clean formal models and effective technical solutions will make their way into industrial practice.										8

18 Software Engineering Tools and Environments		Compositionality	Change	NF Properties	Service view	Perspectives	Lifecycles	Architecture	Configurability	Domain specificity	Links
18.1	The development of methodologies, formalisms, and tool and environment support for separation, extraction and integration of concerns										
18.2	Linguistic and tool support for morphogenic software: software that is malleable for life, sufficiently adaptable to allow context mismatch to be overcome with acceptable effort, repeatedly, as new, unanticipated contexts arise.										
18.3	The development of new methodologies, formalisms, and processes to address non-traditional software lifecycles, and the tool and environment support to facilitate them.										
18.4	The development of new methodologies, formalisms, processes and tool and environment support to address the engineering of software in new, challenging domains, such as pervasive computing and e-commerce.										17
18.5	The adoption or adaptation of XML, Enterprise Java Beans and sophisticated message brokering for integration of both tools and commercial applications.										21, 8

19 Software Configuration Management		Compositionality	Change	NF Properties	Service view	Perspectives	Lifecycles	Architecture	Configurability	Domain specificity	Links
19.1	Functionality & efficiency. The basic core functions of SCM systems, like versioning, data management or workspace support, are still semantically weak, they lack usability, adaptability and so on.										
19.2	PDM vs. SCM. Any serious industrial product now includes a significant amount of software. We need tools covering consistently and homogeneously the development and evolution of products containing software and other kind of components.										
19.3	Process support. Many tools include some process support; SCM in nothing else than one of these. There is a need to make all these (heterogeneous) process fragments interoperate with minimum redundancy, to cover consistently the complete process spectrum of a company.										1
19.4	Web support. The Web opens new possibilities for remote access, but the concepts and mechanism for managing remote concurrent engineering are missing. On the other hand, managing the evolution of web pages and other web artifacts raises new challenges.										21
19.5	Interoperability and architecture. Advanced SCM service have to be componentized, and tailored solutions will be proposed. The SCM core is likely to be part of the kernel on which the many company tools cooperate and interoperate. It will become part of an architecture and interoperability kernel.										20

20 Databases in Software Engineering		Compositionality	Change	NF Properties	Service view	Perspectives	Lifecycles	Architecture	Configurability	Domain specificity	Links
20.1	Separation of concerns between database system and software engineering applications.										
20.2	Application of database technology to provide the infrastructure for the integration of distributed software engineering teams.										
20.3	Provision of intelligent querying functionality for heterogeneous information sources supported by powerful meta-information management.										
20.4	Leverage of integral elements of the transaction concept such as consistency control and recovery to software engineering (environments).										
20.5	Application of database component technology allowing the deployment of database services in a requirement-driven manner.										

21 Software Engineering and the Internet		Compositionality	Change	NF Properties	Service view	Perspectives	Lifecycles	Architecture	Configurability	Domain specificity	Links
21.1	The field of software engineering can heavily profit from markup languages, since they provide validation of structure, display-independent formats and sophisticated linking capabilities.										18
21.2	Web integration of all types of project documents, with full support for notations, code and diagrams.										18
21.3	Creation of tools that take advantage of the semantic richness of markup languages in order to provide sophisticated analysis and verifications.										
21.4	Exploitation of sophisticated hypertext linking mechanisms to express complex interconnections among different documents of the software process.										
21.5	Identification of key aspects of documents, process descriptions, and all other types of data objects that are relevant to the field of software engineering and that can benefit from standardization.										1
21.6	Creation of tools that provide useful, non-trivial services through the comparison and integration of information deduced from data objects relevant to the field of software engineering and not created to work together.										

22 Software Economics		Compositionality	Change	NF Properties	Service view	Perspectives	Lifecycles	Architecture	Configurability	Domain specificity	Links
22.1	Principles, models, methods and tools for reasoning about and dynamic management of software development as an investment activity.										24
22.2	Models for reasoning about benefits and opportunities in software development as well as costs and risks.										24
22.3	Principles, models, methods and tools for dealing with uncertainty, incomplete knowledge, and market forces, including competition and change, in software development.										24
22.4	Principles, models, methods, and tools for resolving multi-attribute decision issues in software design and development.										24
22.5	Integration of economic considerations into software design and development methods.										6, 7

23 Empirical Studies of Software Engineering		Compositionality	Change	NF Properties	Service view	Perspectives	Lifecycles	Architecture	Configurability	Domain specificity	Links
23.1	Empirical study play a fundamental role in modern science, helping us understand how and why things work, and allowing us to use this understanding to materially alter our world.										
23.2	Defining and executing studies that change how software development is done is the greatest challenge facing empirical researchers.										
23.3	The key to meeting this challenge lies in understanding what empirical studies really are and how they can be most effectively used - not in new techniques or more intricate statistics.										
23.4	If we want empirical studies to improve software engineering research and practice, then we need to create better studies and we need to draw more credible conclusions from them.										
23.5	Concrete steps we can take today include: designing better studies, collecting data more effectively, and involving others in our empirical enterprises.										24

24 Software Metrics		Compositionality	Change	NF Properties	Service view	Perspectives	Lifecycles	Architecture	Configurability	Domain specificity	Links
24.1	Software metrics should provide information to support quantitative managerial decision-making during the software lifecycle.										22
24.2	Good support for decision-making implies support for risk assessment and reduction.										22
24.3	Traditional metrics approaches, often driven by regression-based models for cost estimation and defects prediction, provide little support for managers wishing to use measurement to analyse and minimise risk.										22
24.4	The future for software metrics lies in using relatively simple existing metrics to build management decision-support tools that combine different aspects of software development and testing and enable managers to make many kinds of predictions, assessments and trade-offs during the software life-cycle.										
24.5	Handle the key factors largely missing from the usual metrics approaches (causality, uncertainty, and combining different, often subjective, evidence) using causal modelling (for example Bayesian nets), empirical software engineering, and multi-criteria decision aids.										22

25 Software Engineering Education		Compositionality	Change	NF Properties	Service view	Perspectives	Lifecycles	Architecture	Configurability	Domain specificity	Links
25.1	Identifying distinct roles in software development and providing appropriate education for each.										
25.2	Instilling an engineering attitude in educational programs.										
25.3	Keeping education current in the face of rapid change.										
25.4	Establishing credentials that accurately represent ability.										