The Art and Science of Software Architecture

Alan W. Brown[1] and John A. McDermid[2]

[1] IBM Software Group
Raleigh, NC, USA
awbrown@us.ibm.com
[2] University of York
Heslington, York, UK
John.McDermid@cs.york.ac.uk

Abstract. The past 20 years has seen significant investments in the theory and practice of software architecture. However, architectural deficiencies are frequently cited as a key factor in the shortcomings and failures that lead to unpredictable delivery of complex operational systems. Here, we consider the art and science of software architecture: we explore the current state of software architecture, identify key architectural trends and directions in academia and industry, and highlight some of the architectural research challenges which need to be addressed. The paper proposes an agenda of research activities to be carried out by a partnership between academia and industry. While challenges exist in many domains, for this paper we draw examples from one area of particular concern: safety-critical systems.

Keywords: Software architecture, Software engineering, Systems engineering.

1 Introduction

Experience developing software-intensive solutions in many domains leads one to conclude that a very significant shift is taking place in delivery of complex software systems[1]. This shift is being driven by several convergent factors, including [1, 2]:

- *End user expectations* – End users want information to be available everywhere, on demand, with no downtime.
- *Cost to create solutions* – To be competitive it is essential that IT organizations take advantage of lower labor rates around the world, integrate components from a variety of suppliers, and reuse across product lines and solution families.
- *Auditing and Compliance* – Increased regulations and oversight are placing additional requirements of adoption of well-documented best practices with mandated control points and delivered artifacts to aid auditing.
- *Speed of change* – The fast pace of business change demands that IT systems can be reconfigured quickly as those needs change.

[1] We refer to “complex software systems” as a shorthand for “large-scale development and support of software-intensive solutions in domains such as defense, aerospace, telecommunications, banking, insurance, healthcare, retail, etc.”.

- *Adaptable business platforms* – Today's distributed solutions platforms must allow optimization around current business needs, and support reconfiguration as those needs change.

As IT organizations reexamine their IT systems and rethink their practices and tools, the subject of systems and software architecture is frequently at the core. In fact, we observe more categorically that architecture is pivotal in the development of complex software systems; the Royal Academy of Engineering and British Computer Society (RAE/BCS) report on Complex IT Systems [3], states that:

"Systems architecture is one of the most significant technical factors in ensuring project success."

Unfortunately, as a central element in a project's success, the problems too often associated with delivery of high quality systems on-time and to budget can also be frequently traced to issues of systems and software architecture. As the RAE/BCS report identifies:

"At present, definition of architecture is relatively ad hoc and, whilst there is good practice, much more needs to be done to codify, validate and communicate the experience and skills of the best systems architects."

Such concerns raise a number of deep questions concerning the art of software architecture as practiced and supported in industry, and the science of software architecture as studied and taught in academia. This provides the motivation for our paper; to see how to gain benefit by combining industrial experience with academic work on software architecture.

We start by considering the role of software architecture in the software development process then outline what we see as the "state-of-the-practice" in industry/commerce and the "state-of-the-art" in academia. To focus this analysis – we believe without loss of generality – we consider the challenges of UK Defence systems, including safety-critical systems. We use this analysis to identify a research agenda – to strengthen industrial/commercial practice, and to focus academic research.

2 State-of-the-Practice in Software Engineering

There is not a "hard" distinction between academic and industrial/commercial research in software architecture – in fact, we shall argue that there is a need for greater links between academic and commercial endeavors – but there is a discernible difference in emphasis. Our aim below is to outline some of the main thrusts in these areas –space does not permit a full exposition of ongoing work, and we focus more on research relevant to safety critical and embedded systems, as this enables us to illustrate some successful industrial-academic collaboration.

2.1 Academic

Taken broadly, the academic view has been to try to bring rigor to architectural definition – either building new approaches or seeking to "strengthen" techniques used in industry. We distinguish here the formal and analytical approaches from those more pragmatic approaches which seek more to underpin the development process.

2.1.1 Analytical Approaches

Widely used software architecture description notations such as UML are often viewed by academic researchers as having weak semantics; some early work on "precise UML" (pUML[2]), sought to provide a sound semantic basis for the notation. In some research programs, the aim was to support analysis of models but in many cases it was just an attempt to remove ambiguity. Arguably such work is bound to be of limited value as the tools effectively define semantics, and the tools evolve faster than the modeling research.

More recently, the focus has been on defining new notations with better defined semantics (see below) or on adapting popular notations to make them analyzable. A common approach has been to identify parts, or aspects of notations, which can be subjected to formal analysis. Perhaps one of the most common has been to use model checking on the state machines in UML see, for example [7], or in StateMate. This sub-problem has been chosen as it is amenable to automated analysis, but it still does not solve the basic problem of semantic ambiguity[3]. Further, there is an issue of scalability; although model checking is becoming more powerful, there are as yet few examples of the being used on (large-scale) real-world problems ([8] is one of the few published examples). Perhaps more critically, such approaches do not address the full expressive power of notations such as UML.

In safety critical systems, where there is a strong motivation to verify programs, the SPARK notation and toolset [9] has become successful [10]. The tools incorporate the long-established principles of program verification by extending a subset of Ada with annotations which allow the expression of pre- and post- conditions. The SPARK Examiner tool is capable of discharging some "routine" proof obligations, e.g. freedom from run-time exceptions, largely automatically, as well as showing correctness against the pre- and post- conditions.

Some work has "lifted" this idea to the architectural level, by defining a state machine subset and adding annotations, e.g. on assumptions of rate of change of external variables when moving between states [11] (this was done on Stateflow, the state machine element of Matlab-Simulink-Stateflow (MSS), which is widely used for modeling control systems). Related work also addressed the rest of the MSS notation, viz control law diagrams, proposing a variant of WP-calculus which deals with differentials (and integrals) as found in control systems [12]. This work has some technical merit, but there are problems of scaling it to complex systems – especially in the interaction of the state machine and control law elements (there is a "state explosion" which hinders automated analysis).

The above discussion only considers the functional properties of programs. Most safety-critical systems are also real-time systems, and it is necessary to be able to demonstrate that they meet (specified) timing properties. This involves both determining the worst case execution time (WCET) of code fragments, and the overall timing properties of the program, which involves analysis of scheduling. There have been mature theories for scheduling for some time, and these have been applied to real-world systems, e.g. [13].

---

[2] There is still a webpage at www.cs.york.ac.uk/puml/ but the site is now moribund.

[3] Work by Mikk in the late 1990s identified over 100 semantic models of StateMate statecharts.

Some time ago, it was possible to determine WECT analytically (formally), by a combination of program static analysis and instruction counting. Again this work reached a level of maturity which allowed it to be applied on real systems, e.g. [14]. Since the mid 1990s, timing analysis has become harder as processors have been optimized for the best average case performance, and worst case program timing is influenced by the interaction of cache and pipeline with the program data. More recently, therefore, research has moved away from pure analytical techniques and is using a combination of static and dynamic (testing) techniques. Work at Rapita[4] shows that it is still possible, however, to produce techniques which are applicable to industrial scale problems and which can estimate timing properties at the architectural level.

In summary, a major thrust of much academic research which can be viewed as being at the "architectural level" has been focused on the use of formal, analytical, techniques. There have been some successes, both intellectual and practical, but there are also significant limitations on the techniques. The formal analysis of functional properties has not proven to scale effectively. Model checking techniques are advancing in power faster than Moore's law (i.e. algorithms are improving as well as the processing hardware), but dealing with all aspects of large-scale complex control systems remains problematic. Approaches to timing properties have been more successful, in impacting real-world systems, but to cope with modern processors it has been necessary to move away from a purely analytical approach to using a mixture of static and dynamic techniques. This must be considered a success, not a failure, but it highlights the difficulties of taking analytical approaches to large-scale problems.

2.1.2 Pragmatic Research

There is less of a clear-cut distinction between what is industrial research and academic research in the more pragmatic areas – indeed there is effective collaboration between Universities and industry – and we focus here on those developments which are clearly influenced by both an academic and an industrial perspective. We consider two aspects of work: that which expands on notations to make them more relevant to their application domains, and that which focuses on making development processes more rigorous and repeatable.

Languages such as UML, despite being multi-faceted, do not address all the properties of interest for real-time safety critical systems; for example they do not express timing properties within the core language. There have been approaches to dealing with these limitations for some time, e.g. real-time extension of UML [15], but these have not been widely taken up. The reason seems to be mainly that the extensions are not sufficiently rich – and thus do not do enough to enable the techniques to solve all the problems encountered in practice.

Some academic work has tried to address this problem by developing wide-spectrum specification languages, and supporting contracts which enable aspects of systems to be specified and built independently, e.g. AIM [16]. Whilst mainly pragmatic, this work has also embraced formal analysis of failure properties, a key aspect of safety-critical software designs, with a technique known as FPTC [17]. Associated work has also considered the use of automated techniques for model transformation (we defer such issues to section 3.2 below).

[4] See http://www.rapitasystems.com/

Perhaps the most significant aspect of this work has been to influence domain-specific languages such as the Architecture Analysis and Description Language (AADL) which started out as a research program funded by the US Army Avionics Command, and which is now an SAE standard[5]. AADL incorporates contracts and notions of fault propagation which are influenced by AIM and FPTC. Whilst AADL was originally an "independent" notation it is now available as a UML profile.

There are many approaches to improving development processes which build on architectural representations; we consider two: product line research and test automation.

Many systems, e.g. airplane engines, evolve as product lines, i.e. the features and properties of one product bear a strong relationship to those of others in the family – and the relationships can be utilized to simplify the design process. Specifically the common parts can be represented in the architecture – which also has explicit representation of variations (alternatives) and options in the system. These concepts have been studied in academia [18], but are also used in industry [19] and are supported commercially[6]. Further work has considered the process for deriving requirements for product lines and architectural flexibility [20] in the context of embedded control systems (aircraft engine controllers). Generally work in this area is quite mature. It is not as widely adopted as might be expected, but that is largely a socio-technical issue, which is outside the scope of this paper.

Perhaps one of the reasons for limited take-up of such ideas is that the real cost of producing critical systems arises from the verification activities, not the development activities. If code generation costs, say, 10% of the development cost and verification is 50% of the cost, then halving code production time is of little benefit if the verification cost is not also reduced. Some tools, e.g. Reactis[7], have been developed to automate some of the verification tasks when using model-based development, but generally these do not consider issues such as product lines. Some research work, e.g. [21], has addressed product lines but there remains much more to be done.

Interestingly, because of the cost of testing, there have been a number of attempts to substitute formal analysis for testing. One of the most interesting developments is ClawZ which verifies Ada against MSS models [22]. This approach has been used on practical applications, e.g. the Eurofighter Typhoon flying control system.

For these approaches – test automation and formal analysis – there are many challenges, including ensuring that the certification authorities accept their validity. Again, space does not permit a full treatment of these issues, but [23] indicates one way of overcoming these difficulties, by providing a logical argument why the techniques proposed are an effective substitute for those mandated by the standards.

2.1.3 Observations

The work outlined in the two previous sections can (perhaps simplistically) be characterized as "science driven" and "problem driven" (or an engineering approach) respectively.

[5] See http://www.aadl.info/

[6] See, for example http://www.biglever.com/index.html

[7] See http://www.reactive-systems.com

The "science driven" approaches have potential, but often fall a long way short of solving industrial problems. It is likely that this is where radical changes to processes will come but the success rate is likely to be low, as the problems of scaling the "science" to industrial scale problems are very great, and perhaps need more investment than can be found in academic projects.

The "problem driven" approaches have often involved interaction between academia and industry – focusing academic research on issues which cause difficulty in practice. Whilst this work may not always produce full solutions it often ameliorates the problems, and produces useful incremental improvements to processes.

Both forms of research are needed – see the later discussions for our views on the balance.

2.2 Industrial/Commercial

Architectural concerns are important to all industrial systems. In any development project there is significant focus on the architecture as a central artifact. However, practical considerations require efficient techniques and tools that provide clear value to the project. All too frequently this leads to shortcuts being taken that have consequences on the long term maintenance and evolution of the system.

Here, we explore several areas of industrial software architecture as practiced today, discuss current practices in use, and highlight several key gaps and challenges.

2.2.1 Architectural Design

Today, a majority of software developers still take a code-focused approach to development, and do not use separately defined abstract (architectural) models at all. They rely almost entirely on the code they write, and they express their model of the system they are building directly in a third-generation programming language (3GL) such as Java, C++, or C# within an Integrated Development Environment (IDE), e.g. the IBM Rational Application Developer, Eclipse, or Microsoft VisualStudio. Any separate modeling of architectural designs is informal and intuitive, and lives on whiteboards, in Microsoft PowerPoint slides, or in the developers' heads. While this may be adequate for individuals and very small teams, this approach makes it difficult to understand key characteristics of the system among the details of the implementation of the business logic. Furthermore, it becomes much more difficult to manage the evolution of these solutions as their scale and complexity increases, as the system evolves over time, or when the original members of the design team are not directly accessible to the team maintaining the system.

The UML is the most frequently used language for visualizing static and dynamic aspects of software-intensive systems [5]. Users of UML for architectural design are supported by well-established methods that offer a set of best practices for creating, evolving, and refining models described in UML. One of the most well-known is the IBM Rational Unified Process (RUP). The RUP describes a development process that helps organizations successfully apply a Model-driven Development (MDD) approach [24].

The UML is one element of a broader initiative aimed at encouraging a model-driven approach. This is most clearly seen in OMG's Model Driven Architecture

(MDA) approach, a set of technologies intended to provide an open, vendor-neutral approach to the challenge of business and technology change [26]. Based upon OMG's established standards such as the Meta-Object Facility (MOF), UML, and XMI, the MDA separates business and application logic from underlying platform technology. The goal of MDA is to offer a conceptual framework for using models and applying transformations between models as part of controlled, efficient software development process [27]. Several commercial and open source software products claim support for part or all of an MDA approach, and are used in many architectural domains most notably in real-time and embedded systems.[8]

In spite of these improvements in model-driven development support, many developers use the UML notation informally as a way to "sketch" the design of new and existing systems [28]. Architectural design can then occur through use of domain-specific languages (DSLs) that embed many architectural patterns and assumptions into the notation and its transformation into a solution specific to the domain. Many such DSLs are available focusing on business domains such as banking, automotive, and telecommunications systems.

The approach to creating and applying DSLs has received additional attention recently as it is one of the cornerstones of Microsoft's Software Factories approach [29]. A Software Factory is a collection of technologies that introduces product line thinking around an application architecture that is defined and refined through domain specific languages. Tooling for this approach is part of Microsoft's latest releases of its VisualStudio product line.

2.2.2 Architectural Analysis

State-of-the-art transformation techniques used in MDA generally cannot be "steered" by dependability issues, and have not been widely applied to architectural models with dependability attributes. Integrating MDA with mechanisms for building dependable systems requires deep and applied knowledge of dependability as well as MDA standards, tools, and techniques.

Models of enterprise architecture (EA) are rarely used to develop assets used downstream. There are several reasons for this. Downstream assets (such as code, documentation for review, and deployment models) are difficult to derive from models using the current state-of-the-art transformation tools, which are typically targeted at single diagrams. An EA model typically provides information spanning several meta-models, and most transformation tools are applicable to a single source meta-model. Deriving downstream assets from EA models may be more feasible – and demonstrably more useful – with mechanisms for integrating different views (from different meta-models) – the results of which could then be applied to transformation. Another difficulty encountered is dealing with non-functional attributes, such as quality-of-service constraints. While mechanisms like the UML Quality-of-Service (QoS) profile can help to enrich EA models with QoS and dependability characteristics, current generation transformation and integration tools must be made aware of these characteristics during the generation and integration process.

[8] See www.omg.org/mda for more details and examples.

Automated architectural analysis techniques often have to be tailored specifically to modeling languages and tool infrastructure; moreover, adapting existing analysis implementations (e.g., fault and failure analysis, timing analysis) to specific architectural modeling languages and tools often requires much repeated work. As a result, much of the currently practiced architectural analysis remains informal and based on inspection techniques supported by architectural heuristics gathered from experiences across several domains.

2.2.3 Architectural Frameworks

Support for a consistent approach to software architecture requires a set of guidelines that identify the key artifacts to be delivered, and constrains the method by which those artifacts are produced. For many organizations, a primary objective is to produce a standardized blueprint describing a complex systems architecture so that decision-makers can then use this report to compare the architecture of alternative system designs and manage the evolution of existing systems. The blueprint must describe a system's architecture well enough to enable detailed analysis, justify procurement to the project's sponsors, and support ongoing management of the in-flight project.

To assist in this, several organizations have formalized such guidelines in the form of a so-called "architectural framework". These guidelines typically are specialized to a specific technology or business domain (e.g., defense systems, or telecommunications systems), or else promote specific architectural views of the system to highlight particular forms of communication and analysis (e.g., business/IT communication, or operational systems management). In the first category we have efforts such as the US Department of Defense Architectural Framework (DODAF) and the UK Ministry of Defence Architectural Framework (MODAF). In the second category we have efforts such as the Zachmann Framework and The Open Group Architectural Framework (TOGAF).

For illustrative purposes, we shall consider MODAF[9]. The MODAF V1.0 set of baseline documentation was published in 2005 [30]. Many of the MOD stakeholders have started to adopt MODAF. There have been a number of notable areas of successful adoption and several MOD organizations have successfully modeled aspects of their architecture using MODAF (e.g., The Director of Equipment Capability (DEC) Command, Control and Information Infrastructure (CC&II), The DEC Intelligence Surveillance Target Acquisition and Reconnaissance (ISTAR), and The Logistics Coherence Information Architecture (LCIA) within the Defence Logistics Organisation (DLO[10])). In addition, many MoD Invitation to Tender (ITT) documents have required MODAF views to be produced as part of the technical proposal response. Consequently, a number of the tool vendors have developed MODAF specific configurations of their modeling tools, including Salamander, Telelogic, Troux Technologies, Artisan Software and IBM.

However, in spite of this success, the use of architectural frameworks such as MODAF is inconsistent and sporadic. For example, MODAF as a standard is not

[9] We believe that to a large degree our experience with other architectural frameworks mirrors the status of MODAF.

[10] The MoD has recently been reorganised, and the DLO is now part of Defence Engineering and Support (DE&S).

completely mature. The MODAF M3 Meta Model is in advance and inconsistent with the MODAF Technical Handbook [31]. Further, initial use and adoption of MODAF has identified a number of areas that need simplification or clarification. In particular, the MODAF Tool Certification Plan [32] needs to be both extended and implemented. Most importantly, despite the fact that different MOD stakeholders have started to model using MODAF, very little architectural analysis is being performed using these models; it is used simply as a documentation approach for architectural designs.

2.2.4 Architectural Styles

Much practical work has taken place over the past few years to understand how various repeating patterns of development can be understood, categorized, and used as the basis for future systems development. As a result, a set of architectural styles has emerged that are used by many organizations as the basis for design decisions, and supported by vendors in their commercial tools, methods and infrastructure offerings [33, 34].

In particular, the need to respond to changing business demands with flexible IT solutions has led many businesses to employ Service Oriented Architectures (SOAs). SOA is an architectural style aimed at more directly representing business processes through choreographed sequences of services realized through reusable components [35, 36, 37]. The service design layer (or "service architecture") is explicitly independent of applications and the computing platforms on which they run. Solutions are designed as assemblies of services in which the description of the assembly is a managed, first-class aspect of the system, and hence amenable to analysis, change, and evolution.

SOAs provide the flexibility to treat elements of business processes and the underlying IT infrastructure as secure, standardized components – typically Web services – that can be reused and combined to address changing business priorities. They enable businesses to address key challenges and pursue significant opportunities quickly and efficiently.

However, as with many such initiatives, the interest in SOA as an architectural style has its challenges. While the application of SOA in commercial business domains is well advanced, issues around performance, reliability, and predictability of such a loosely coupled architectural style remain. Most notably, this is a result of a lack of practical architectural design approaches and verification techniques that address key system properties such as availability, performance, resource usage, timing properties, and so on. Similarly, much of the available commercial infrastructure supporting SOA is largely untested in the kinds of demanding contexts typical of military systems. Some interesting research has been carried out, leading to proof-of-concept demonstrations and prototypes. However, much work remains to see this applied more extensively in commercial practice.

3 A Software Architecture Research Agenda

The breadth of areas of concern to software architecture is daunting. Hence, to be successful, the work in software architecture must focus on key issues which are essential to make progress, and directly bridge the academic-to-commercial gap.

We propose research in software architecture is organized into three areas:

- Management.
- Dependability and Properties.
- Assembly and Integration.

We briefly discuss each area. To be effective, and to enable technology transition, collaborative research between industry and academia needs to produce guidance, such as process guides, and handbooks of best practice, which we generically refer to as Statements of Best Practice.

3.1 Management

Within the area of management, we have identified three research strands. We see these strands as ongoing throughout the life of any software architecture program:

- Acquisition processes.
- Measurement.
- Software development processes.

The acquisition strand is intended to address current acquisition practice to ascertain where it might be changed to better address the challenges of acquiring complex software-intensive equipment, systems and systems-of-systems. This strand provides the greatest potential for early impact. Key research issues include:

- Management of risks and uncertainties associated with the requirements and the technical solution.
- The management of changes to the requirements and / or the technical solution.
- Incremental and evolutionary procurement (closely related to the previous items).
- Through Life Capability Management.
- Estimation of cost and timescale in relation to earned value and other ROI measures.

In order to take effect, any improved acquisition processes will need to be adopted by industry. It is our experience that adoption is difficult to achieve. Accordingly, the research effort should be constructed so as to maximize the chances of successful adoption. In particular,

- The research will be focused on areas where the stakeholders agree that there is a need for change and is willing to change.
- Any changes must be matured, through research and pilot studies, before being rolled out generally.

A corollary is that researchers must seek to engage with the stakeholders to discuss the changes that it is prepared to make; and the research program must be sufficiently flexible to absorb the results of that negotiation, which may alter during the life of the program.

The second priority is measurement: this reflects the importance that we attach to taking an evidence-based approach generally. The measurement strand consists of several activities involving data gathering from projects n industry. This must:

- Provide an improved picture of industry practice, as the basis for additional research activities.
- Assist in the identification and assessment of potential improvements to the processes used within industry.
- Assist in monitoring the effect of any such changes in a well-defined feedback loop.

In addition, the measurement activities must include research into techniques for monitoring project health. This aspect of the research effort will concentrate on metrics that are useful to all stakeholders but which are currently under-researched, such as metrics related to safety or security.

The final strand deals with the processes that are used for software development. Here, we note

- The existing literature is very large. It includes: the CMMI, ISO9001:2000 and TickIT, several methodologies such as RUP, and proprietary processes developed within industry.
- Despite this mountain of advice, many problems on projects arise because of the non-application of known good practice. From the RAE/BCS report on the challenges of complex IT systems [3]:

> *"A significant percentage of IT project failures, perhaps most, could have been avoided using techniques we already know how to apply. For shame, we can do better than this."*

Accordingly, we see the adoption of good practice itself as the central issue and this must be the focus of research. Previous research consists of several studies in the business IT sector, in which researchers have studied software teams in action. This research must be expanded into other domains, determining which practices are readily followed "at the coalface". The outputs from this strand will consist of improved advice regarding the construction of software development procedures within industry.

3.2 Dependability and Properties

Dependability and properties, especially so-called non-functional requirements (NFRs), are cognate issues which need to be treated together in software architecture, especially in trade-studies and practical architectural analysis.

Currently, individual high-integrity systems (HIS) achieve acceptable levels of safety, albeit at high cost. In TLCM, the crucial issues for HIS are reducing the timescales and costs of procurement and extending capabilities to a full range of properties, including other aspects of dependability, e.g. security. In particular, rapid, low-overhead, assurance techniques are required to demonstrate that a given HIS achieves a specific safety or security level, without the current substantial delays in carrying out labor-intensive evaluation processes. The emphasis of new research should focus on automation of assessment – although any such research will need to draw on other work to address this issue fully.

Particular attention of research in this area should be on Systems-of-systems (SoS) issues, where the problems are much more open-ended. Approaches to safety and security which have been used for HIS don't scale (or don't apply) to SoS. For example, safety and security assessment normally starts by defining the boundary of the system of interest – for SoS the boundary is not known in advance, and may change dynamically, especially in open systems. Conditions of use may vary dynamically (e.g. the changing topology of ad hoc networks of users communicating over different categories of secured links, and changing access rights of those users), and the SoS must remain demonstrably safe/secure. This requires new approaches to dealing with safety and security, with more emphasis on the dynamic management of safety and security issues, i.e. sensing and responding to them at run-time. Further, safety and security are not disjoint – for example compromise of a communications link to a UAV could lead to an unsafe weapon release; new approaches to integrated safety and security analysis are needed. Also, there is a view that all elements in a SoS will require some level of assurance – say SIL 2 in safety terminology; emphasis is needed on developing effective assurance techniques for such levels of integrity.

The scope of the work will be different for the two areas. In HIS, the focus of research should be mainly on improving software engineering economics, for example by developing and justifying "agile high integrity" processes. For SoS, the emphasis of research must be more on identifying means by which dependability can be "engineered in" to products, and dependability can be preserved through dynamic changes in the membership of the SoS. The boundaries are the genuine systems engineering issues, e.g. evaluating the dependability of different system concepts.

3.3 Assembly and Integration

To be effective, architectures need to be multi-faceted and concerned with data flow, control flow, module decomposition, and properties such as throughput, resilience and recovery, and so on. Current architectural design methods are inadequate as they do not deal with the range of properties of interest, allow for the prediction of implementation properties, and so on. For HIS, the Architecture Analysis and Design Language (AADL) and work in the Defense and Aerospace research Partnership (DARP) form starting points, but much needs to be done e.g. on analysis and prediction of properties, effective interface definition to allow ease of integration, and so on. Issues which research needs to address include:

- Architecture evolution.
- Assigning values (figures of merit) to architectures to enable trade-offs.
- Migration of application architectures between computing platforms.
- Scalability in terms of numbers of nodes, and data volumes.
- Open and modular systems (especially SoS), including interface definition.
- Robustness (resilience), recovery and reconfiguration.
- Definition and control of emergent properties.

In practice, however, existing systems often have a brittle, complex structure that does not easily enable a move to the kinds of flexibility required. The IT drivers found by customers for undertaking such a migration include the need for operational and systems agility, interoperability, reuse, streamlining architectures and technology solutions and leveraging legacy systems and existing capability.

The emergence of techniques and technologies for service-oriented architectures (SOA) are directly aimed at providing a design framework to support this kind of flexibility and agility. There remain many research challenges to their widespread use in embedded systems. Most notably, this is a result of a lack of practical architectural design approaches and verification techniques that address key system properties such as availability, performance, resource usage, timing properties, and so on. Similarly, much of the available commercial infrastructure supporting SOA is largely untested in the kinds of demanding contexts typical of military systems. Some interesting research has been carried out, leading to proof-of-concept demonstrations and prototypes. However, much additional research work remains.

Transformation of IT systems of itself will reduce application maintenance and operational cost. However, the bottom-up only approach carries the risk that the new services will embed old ways of doing business, providing flexible systems, but inflexible enterprises. In order for enterprises to become flexible, parallel organizational transformation needs to take place and the complementary supporting governance embedded. Only in this way can the benefits to Operations can be realized (e.g., efficient global footprints, economies of scale, agility - rapid change, regulatory compliance and process optimization).

Integration is concerned with construction of systems from components – whilst minimizing risk and surprises. Several research activities would be valuable. The concerns are, in many ways, the same as for architecture, but from the viewpoint of validating the properties and behavior of the composed system. Issues include:

- Planning and management of integration, including risk management.
- Architectural evaluation (determining figures of merit).
- Testing strategies, especially for SoS which may be configured in an ad hoc and dynamic fashion, maximising what can be predicted from the minimum of testing.
- Configuration control, and evaluation and assessment of differing system configurations.

4 Discussion

We discuss the role of academia and industry in addressing the art and science of software architecture. Simply stated, we believe software architecture holds the pivotal position in helping:

- Academia to educate the next generation of architects in software engineering.
- Industry to produce more robust, scaleable methods and tools that meet users' needs.
- Academia and industry to form a closer partnership to advance the state-of-the-art in software engineering.

4.1 What Can Academia Do?

Software engineers need to understand key aspects of computer science (CS) fundamentals – the analogy is with the need for understanding of physics in the "physical" engineering disciplines, such as mechanical engineering and electronics. Beyond this, they need knowledge and skills in five areas – albeit in different amounts depending on their role in a project.[11] These are:

- Architecture
- Good practice
- Domain knowledge
- Management
- Soft skills

Architecture: Software is an intellectual artifact – producing software is essentially a "pure design" activity. Thus the core of what software engineers need to know is how to architect software systems, and how to tell good architecture from bad. Often this involves understanding the non-functional properties of the architecture – e.g. will it perform fast enough to process all the data, will it be secure (against anticipated attacks), will it manage hardware failures to preserve system safety, and will it be useable by the general public?

Good Practice: All engineers should use good practice, but this is sadly all too rare in software engineering – according to Fred Brooks "in no other discipline is the gulf between typical practice and best practice so large" [40]. Further, good practice is not static, indeed technology moves apace. However, not all the new technology is useful, or stands the test of time. Thus, software engineers need to understand *principles* so they can assimilate new practices and, to some degree, sort out the genuine advances from the "mere fads".

Domain Knowledge: Software engineers need domain knowledge. Programs have a role in the world – either as the control and monitoring element in some embedded system, e.g. in an aircraft engine controller, or as a key enabler in a business or organization, e.g. providing electronic access to patient health records. Most requirements for software systems are incorrect – at least initially. The users or procurers do not fully understand what they want, and also don't write down what is "obvious" – at least to them. To defend against this, and to produce something useful and useable, software engineers need to understand the application domain to be able to validate and complete requirements. It is not normally possible for software engineers to gain domain knowledge in many, disparate, areas – there is simply too much to understand to be expert in say, car braking system design, and on-line reselling. Software may be ubiquitous, but the domain knowledge is not.

Software engineers must obtain domain knowledge to work effectively – computer science knowledge is necessary, but not sufficient, to work in a given domain. An *automotive software engineer* of our acquaintance informs us that he has 20 books on his shelf which he uses regularly – 8 of these are sources of domain knowledge, e.g. the Diesel Engine Handbook.

[11]These ideas are inspired by the work of David Parnas [38], and explored in more detail by one of the authors [39].

Management: Modern software systems are amongst the most complex artifacts produced by man. Developing them requires teams, which need management. The RAE/BCS study found that management was a key success factor, but also a major problem area, for software projects. Managers need to understand what they are managing – otherwise how can they make informed decisions? However they also need to understand management skills and techniques. Traditional engineering distinguishes *repeat* design – making something very similar to what was produced before – from *novel* design, i.e. producing something which is largely unprecedented. Most software projects involve novel, not repeat, design – thus they undertake work for which there is no precedent. Management strategies for dealing with uncertainty, such as incremental development, and approaches to software (project) risk management are therefore keys to success. Software architecture is a control point for managing risk, and a cornerstone of the measures and metrics appropriate for every successful software-based management technique.

Soft Skills: Software engineers may spend much of their day working with computers – but they need to talk to customers, other engineers designing the embedding system, e.g. an aircraft engine, nurses who might use the medical records system, and so on. They need to write manuals, or on-line help, from the *users*' perspective not an internal (design) perspective. They have to work in teams with other software engineers, to ensure that they produce an effective whole. Thus soft skills are crucial to project success. Success of a project is frequently predicated on how well an architect communicates key qualities and properties of the architecture to the rest of the team, and to the broader project stakeholders.

4.2 What Can Industry Do?

Industry needs to produce collections of tools and methods that are more readily consumable by practicing software engineers. While many valuable technologies are available, they need to be more tightly integrated, open to customization, and focused on specific domains and architectural styles.

A collection of tools, practices, guidance, heuristics, and specialized content (patterns, frameworks, domain models, etc.) must be assembled to support an organization as it designs, develops, deploys, manages, and evolves its solutions. Informally we can refer to this as an "architecture-centric workbench". In practice, we can distill certain characteristics in the approaches and capabilities of these workbenches.

As illustrated in Figure 1, such a workbench typically consists of elements in several categories. Here, for illustration purposes we focus on a workbench optimized for SOA architectural styles of solution in business domains such as insurance and banking. The underlying platform for the workbench is a collection of commercial tools acquired from one or more vendors, and integrated through standard techniques such as shared meta-data, import/export across Application Portability Interfaces (APIs), or use of a common plug-in framework such as the Eclipse technology[41]. In the case of many IBM customers it is the IBM Rational Software Development Platform that provides the core set of technologies [42].

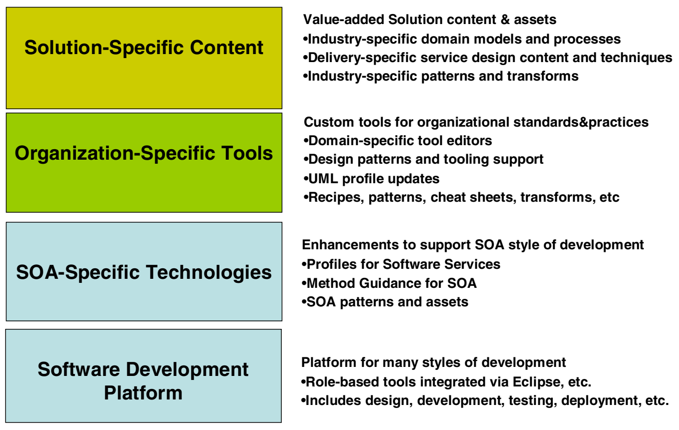
Solution-Specific Content
Value-added Solution content & assets
•Industry-specific domain models and processes
•Delivery-specific service design content and techniques
•Industry-specific patterns and transforms
Organization-Specific Tools
Custom tools for organizational standards&practices
•Domain-specific tool editors
•Design patterns and tooling support
•UML profile updates
•Recipes, patterns, cheat sheets, transforms, etc
SOA-Specific Technologies
Enhancements to support SOA style of development
•Profiles for Software Services
•Method Guidance for SOA
•SOA patterns and assets
Software Development Platform
Platform for many styles of development
•Role-based tools integrated via Eclipse, etc.
•Includes design, development, testing, deployment, etc.

Fig. 1. A Workbench for Service-Oriented Solutions

The core technologies support a wide collection of practices and architectural styles. Extension and customization allows this core to be adapted. In particular, SOA-specific technologies are applied as encoded in patterns and templates, method guides, and profiles for service design that extend the core tooling base.

Then, solution delivery teams within an organization, or external systems integrators and partners, further customize the platform with their own techniques, technologies, patterns, transforms, and so on. These are specific to an organization's ways of working, and relevant to their particular business practices and domain. For example, IBM Global Business Services has defined a set of proprietary practices to provide consistency in the way it delivers services to its clients. It has a collection of SOA-based design techniques, such as the Service-Oriented Modeling and Architecture (SOMA) method [43], that have been distilled from experiences of practitioners on a wide variety of projects. Customized tooling for SOMA has been created as an extension to the IBM Rational Software Development Platform specifically to support those practitioners by automating many of the SOMA techniques.

Finally, domain content is provided to populate the workbench to improve efficiency of delivering solutions, and offer some measure of consistency across solutions in the same domain. Typically, the tools are augmented with domain models and libraries of common patterns in areas such as retail banking, insurance healthcare, and so on. An example of such a domain model is the Insurance Application Architecture (IAA), a detailed set of content models for several aspects of insurance [44].

So this set of technologies, this layering of capabilities, contains tooling, methods, and content. It provides organizations with a domain-specific platform that can be used to deliver service-oriented solutions specific to their business. With some variation, this approach is being used in many organizations to assemble a technology platform appropriate for developing, delivering, and evolving service-oriented solutions.

4.3 What Can Academia/Industry Do Together?

In practice, by working, together academia and industry can add a significant focus on transition into practice, leading to important improvements and up-scaling in the skills, processes, and practices in use. This will be supported in appropriate industry standards and technologies, and a recognized transition path for promising software systems engineering research into relevant programs.

These objectives will be achieved through a focus on 4 key areas; Advancing key technologies, leveraging the community, transitioning into practice, and learning from experience.

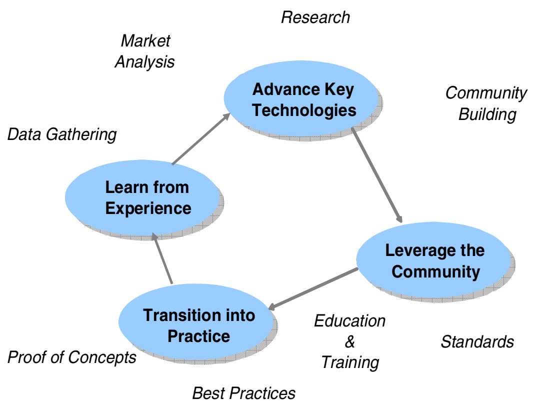

Fig. 2. Academic/Industrial Collaboration

Advance Key Technologies: Identify and advance emerging technologies to address significant and pervasive software systems engineering problems, and develop these technologies to improve software engineering practices in industry. Work with the research community to help create and identify new and improved practices by creating cooperative research and development agreements with industry and academia prove out new and emerging technologies.

Leverage the Community: Work closely with the broad software systems engineering community to seek out best practice and to raise the quality of acquired and delivered software-intensive systems. Work through the global community of software systems engineers to amplify the impact of the new and improved practices by encouraging and supporting their widespread adoption, with the aim of raising the "average" practices much closer to best practices. In addition, this will be supported through the packaging and deliver of a variety of education and training courses and technology practices based on matured, validated, and documented solutions.

Transition into Practice: Work with leading-edge software developers and acquirers to apply and validate new and improved practices. Assist the stakeholders to address specific software systems engineering and acquisition challenges, e.g. clearance of critical aircraft systems, by applying these practices. Transition activities will be primarily funded through contracted additional services with a range of different stakeholders.

Learn from Experience: Build a base of empirical data by undertaking studies, analysis and surveys aimed at establishing a realistic understanding of the current state-of-the-practice for software systems engineering within the systems and software supplier community. In addition to the value of this base data to the wider software systems engineering community, it will also be used to establish appropriate measures for assessing impact of technologies as they transition into practice, and for adjusting the research activities undertaken by in response to that feedback. Where practical this data will also be used to help compare practices with those in other sectors, and to identify opportunities to learn from experience in these other sectors.

5 Summary and Conclusions

It is tempting to believe that software development is easy. You gain an understanding of the problem that needs to be addressed by talking with people familiar with that domain, and then design a solution to meet those needs and deploy it in the customer's environment. Unfortunately, complexity and scale get in the way to make the task of software development a lot more challenging.

Software engineers turn to software architecture as a cornerstone for managing complexity and scale in software development. An architecture is an abstraction of a physical system that allow engineers to reason about that system by ignoring extraneous details while focusing on the relevant ones. All forms of engineering rely on architectures as essential to understanding complex real world systems. Architectures are used in many ways: predicting system qualities, reasoning about specific properties when aspects of the system are changed, and communicating key system characteristics to its various stakeholders.

Here, we have considered the current state of software architecture, identified key architectural trends and directions in academia and industry, and highlighted some of the architectural research challenges which need to be addressed. The paper has proposed a detailed agenda of research activities to be carried out by a partnership between academia and industry. It is our firm belief that this combination of strengths from the two communities will be the basis for future progress in the art and science of software architecture.

References

1. Friedman, R.: The World is Flat: A Brief History of the 21st Century, Farrar, Straus and Giroux (2005)
2. Bhagwati, J.: In Defence of Globalization. Oxford University Press, Oxford (2004)
3. The Challenges of Complex IT Projects: The Royal Academy of Engineering, and British Computer Society (April 2004), ISBN 1-903496-15-2
4. Computer Weekly Article (2004)
5. Rumbaugh, J., Booch, G., Jacobsen, I.: The UML Reference Manual. Addison-Wesley, Reading (2004)

6. Ministry of Defense: Defense Technology Strategy for the Demands of the 21st Century, UK MoD (2006), www.science.mod.uk
7. Automatic verification of a behavioural subset of UML statechart diagrams using the SPIN model-checker. Formal Aspects of Computing 11(6), 637–664 (1999)
8. Damm, W., et al.: Formal Verification of an Avionics Application using Abstraction and Model Checking. In: Redmill, F., Anderson, T. (eds.) Towards System Safety, Springer, Heidelberg (1999)
9. Barnes, J.G.P.: High Integrity Software: The SPARK Approach to Safety and Security. Addison Wesley, Reading (2003)
10. King, S., Hammond, J., Chapman, R., Pryor, A.: Is Proof more Cost-Effective than Testing? IEEE Transactions on Software Engineering 26(8) (2000)
11. Iwu, F., Galloway, A., McDermid, J.A., Toyn, I.: Integrating Safety and Formal Analysis using UML and PFS, RE&SS (2007)
12. Blow, J.R.: Use of Formal Methods in the Development of Safety-critical Control Software. DPhil thesis, Dept of Computer Science, University of York. YCST-2003-08 (2003)
13. Bate, I.J., Burns, A., Audsley, N.C.: Putting Fixed Priority Scheduling Theory into Engineering Practice for Safety Critical Applications. In: Proceedings of 2nd Real-Time Applications Symposium (1996)
14. Eccles, M.A.: STAMP Tool Assessment. BAe-WSC-RP-R&D-0031, BAe Warton (July 1995)
15. Douglass, B.P.: Real-Time UML: Developing Embedded Objects for Embedded Systems. Addison-Wesley, Reading (1998)
16. Radjenovic, A., Paige, R.F.: Architecture Description Languages for High Integrity Real-Time Systems. IEEE Software 23(2), 71–79 (2006)
17. Wallace, M.: Modular Architectural Representation and Analysis of Fault Propagation and Transformation. In: FESCA'05. Formal Foundations of Embedded Systems and Component-Based Software Architectures (2005)
18. Bosch, J.: Design and Use of Software Architectures. Addison-Wesley, Reading (2000)
19. Hotz, L., Wolter, K., Krebs, T., Deelstra, S., Sinnema, M., Nijhuis, J., MacGregor, J.: Configuration in Industrial Product Families: The ConIPF Methodology (2006), see http://www.conipf.org
20. Stephenson, Z., McDermid, J.A.: Deriving Architectural Flexibility Requirements in Safety-Critical Systems. IEE Proceedings on Software 154(4) (August 2005)
21. Stephenson, Z., Zhan, Y., Clark, J., McDermid, J.: Test Data Generation for Product Lines - A Mutation Testing Approach. In: Nord, R.L. (ed.) SPLC 2004. LNCS, vol. 3154, Springer, Heidelberg (2004)
22. Arthan, R., Caseley, P., O'Halloran, C., Smith, A.: ClawZ: Control Laws in Z. In: Liu, S., McDermid, J.A., Hinchey, M.G. (eds.) Proceedings of ICFEM 2000, IEEE Computer Society, Los Alamitos (2000)
23. Galloway, A., Paige, R.F., Tudor, N.J., Weaver, R.A., Toyn, I., McDermid, J.A.: Proof versus testing in the context of Safety Standards. In: 24th Digital Avionics Systems Conference (2005)
24. Kruchten, P.: Rational Unified Process. Addison Wesley, Reading (2002)
25. Selic, B.: The Pragmatics of Model-Driven Development. IEEE Software 20 (September 2003)
26. OMG: MDA Guide, Version 1.0.1 (2003), www.omg.org
27. Frankel, D.: Model Driven Architecture: Applying MDA to Enterprise Computing. Wiley Press, Chichester (2003)

28. Fowler, M.: Comments on UML sketching (2005), www.fowler.com
29. Greenfield, J., Short, S., Cook, S., Kent, S.: Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools. Wiley, Chichester (2004)
30. MODAF Tools Policy Statement of Position: MoD (2006)
31. MODAF M3 Meta Model V1.0: MoD (April 2006)
32. MODAF Tool Certification Plan: V1.0, MoD (April 2006)
33. Shaw, M., Garlan, D.: Software Architecture: Perspectives on an Emerging Discipline. Addison-Wesley, Reading (1998)
34. Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns. Addison-Wesley, Reading (2001)
35. Krafzig, D., Banke, K., Slama, D.: Enterprise SOA. Prentice-Hall, Englewood Cliffs (2005)
36. Bieberstein, N., et al.: Service-Oriented Architecture (SOA) Compass: Business Value, Planning, and Enterprise Roadmap. IBM Press (2005)
37. Herzum, P., Sims, O.: Business Component Factory: A Comprehensive Overview of Component-Based Development for the Enterprise. Prentice-Hall, Englewood Cliffs (2002)
38. Parnas, D.L.: Software Engineering Programmes are not Computer Science Programmes. IEEE Software (November/December 1999)
39. McDermid, J.A.: Tailoring Software Engineering Education: One Size Does Not Fit All. IEE (2006)
40. Brooks, F.: The Mythical Man-Month: 20th Anniversary edn. Addison-Wesley, Reading (2004)
41. Carlson, D.: Eclipse Distilled. Addison-Wesley, Reading (2005)
42. Brown, A.W., Delbaere, M., Eeles, P., Johnston, S., Weaver, R.: Realizing Service oriented Solutions with the IBM Software Development Platform. IBM Systems Journal 44(4), 727–752 (2005)
43. Johnston, S.K., Brown, A.W.: A Model-driven Development Approach to Creating Service-oriented Solutions. In: Dan, A., Lamersdorf, W. (eds.) ICSOC 2006. LNCS, vol. 4294, pp. 624–636. Springer, Heidelberg (2006)
44. IBM Insurance Application Architecture. http://www.ibm.com/industries/financialservices/doc/content/solution/278918103.html