Evolving Critical Systems

Mike Hinchey
Lero – The Irish Software Engineering Research Centre
University of Limerick, Ireland

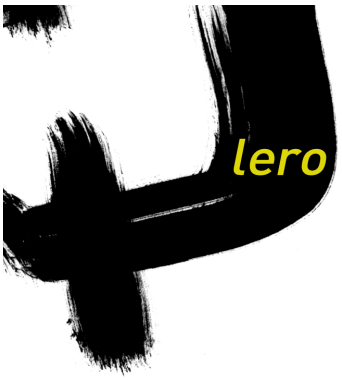
Lorcan Coyle
Lero – The Irish Software Engineering Research Centre
University of Limerick, Ireland

2009

Contact

Address	Lero International Science Centre University of Limerick Ireland
Phone	+353 61 233799
Fax	+353 61 213036
E-Mail	info@lero.ie
Website	http://www.lero.ie/

This work is partially supported by Science Foundation Ireland
under grant no. 03/CE2/I303-1



THE IRISH SOFTWARE ENGINEERING RESEARCH CENTRE

Evolving Critical Systems

Mike Hinchey
Lero – The Irish Software Engineering Research Centre
University of Limerick, Ireland

Lorcan Coyle
Lero – The Irish Software Engineering Research Centre
University of Limerick, Ireland

27 July 2009 DRAFT

The authors would like to thank, for their invaluable comments, suggestions, and feedback, their colleagues in Lero, in particular M. Ali Babar, Goetz Botterweck, Jim Buckley, Andrew Butterfield, Vinny Cahill, Davide Cellai, Siobhán Clarke, Ciaran Clissmann, Simon Dobson, Michael English, Jorge Fox, Benoit Gaudin, Geoff Hamilton, Bill Harrison, Matthew Hennessy, John Hickey, Fergal McCaffery, René Meier, Joseph Morris, Paddy Nixon, Mel Ó Cinnéide, Claus Pahl, Norah Power, Aaron Quigley, Ita Richardson, Kevin Ryan, Hesham Shokry, David Sinclair, and Xiaofeng Wang.

Lero Technical Report Lero-TR-2009-00

Contact

Address .. Lero
International Science Centre
University of Limerick
Ireland

Phone ..... +353 61 233799

Fax ..... +353 61 213036

E-Mail ..... info@lero.ie

Website .. http://www.lero.ie/

This work is partially supported by Science Foundation Ireland under grant no. 03/CE2/I303-1.

1. Introduction

There are few areas of modern life in which software is not an important (though often invisible) component. The software in our lives is increasingly complex; its interaction with the real world means that its requirements are in a state of constant change (Lehman & Fernández-Ramil, 2006). Many non-software products and services, from healthcare to transport, education to business, depend on reliable, high-quality software.

Software engineering is the activity that applies engineering principles to software. It applies systematic, rigorous discipline to the design and development of software, much as civil engineering does to construction. Software engineering improves the quality, reliability and predictability of software systems, by generating knowledge, tools and processes that both facilitate and improve the software development process. These qualities are essential wherever software failure might lead to significant safety, security, or economic losses.

Software systems frequently need to be modified in response to changes in system requirements and in their operational environment (Swanson, 1976). Such modification may involve the addition of new functionality, the adjustment of existing functions, or the wholesale replacement of entire systems. All such change is fraught with uncertainty – software projects involving change frequently fail to meet requirements, run over time and budget, or are abandoned (Rajlich and Bennett, 2000). As the ubiquity and complexity of software increase, a requirement has emerged for **critical software which can successfully evolve** without loss of quality—software that is engineered from the start to be easily changed, extended and reconfigured, while retaining its security, its performance, its reliability and predictability.

2. Evolving Critical Systems

The need is becoming evident for a software engineering research community that focuses on the development and maintenance of Evolving Critical Systems (ECS). This community must concentrate its efforts on the techniques, methodologies and tools needed to design, implement, and maintain critical software systems that evolve *successfully* (without risk of failure or loss of quality).

In order to understand the challenges of ECS it is important to consider the complementary domains of Evolving Systems and Critical Systems.

Evolving systems (Lehman (1980) called these E-type systems) may

- have evolved from legacy code and legacy systems;
- result from a combination of existing component-based systems, possibly over significant periods of time;
- be the result of the extension of an existing system to include new functional requirements;
- evolve as the result of a need to improve their quality of service, such as performance, reliability, usability, or other quality requirements;
- evolve as a result of an intentional change to exploit new technologies and techniques, e.g., service-oriented architectures, or a move towards multi-core-based implementations;
- adapt and evolve at run-time in order to react to changes in the environment or to meet new requirements or constraints.

Most large and complex software systems are evolving systems. The alternative to system evolution is total replacement, often not feasible for cost and other reasons.

Table 1: Cost of one hour of downtime from 2000[1]. Costs are likely to be significantly higher today.

Companies	$/hour
Brokerage Operations	$6,450,000
Credit Card Authorisation	$2,600,000
eBay	$225,000
Amazon	$180,000
Package Shipping Services	$150,000
Home Shopping Channel	$113,000
Catalog Sales Center	$90,000

Critical systems are systems where failure or malfunction will lead to significant negative consequences (Lyu, 1996). These systems may have strict requirements for security and safety, to protect the user or others (Leveson 1986). Alternatively, these systems may be critical to the organization's mission, product base, profitability or competitive advantage. For example, an online retailer may be able to tolerate the unavailability of their warehousing system for several hours in a day, since most customers will still receive their orders when promised. However, unavailability of the website and ordering system for several hours may result in the permanent loss of business to a competitor (Amazon's estimated downtime costs were $180,000 per hour in 2000, cf. Table 1). A brief categorisation of types of critical systems is shown in Table 2.

Table 2: Types of Critical Systems: Many systems have overlapping aspects of criticality, e.g., a system might be both safety-critical and business-critical.

Type of Critical	Implication for Failure
Safety-Critical	May lead to loss of life, serious personal injury, or damage to the natural environment.
Mission-Critical	May lead to an inability to complete the overall system or project objectives; e.g., loss of critical infrastructure or data.
Business-Critical	May lead to significant tangible or intangible economic costs; e.g., loss of business or damage to reputation.
Security-Critical	May lead to loss of sensitive data through theft or accidental loss.

[1] From InternetWeek 4/3/2000 and Fibre Channel: A Comprehensive Introduction, R. Kembel 2000, p.8. based on a 1996 survey done by Contingency Planning Research, available online: http://www.contingencyplanningresearch.com/cod.htm

2.1. Software Engineering and ECS

ECS can be viewed as a special case of the broader software engineering discipline. Similar issues and questions must be addressed within ECS as in other (non-ECS) software engineering research, but with the added (and conflicting) requirements of predictability/quality and the ability to change.

The IEEE Computer Society's "Software Engineering Body of Knowledge" (SWEBOK) characterises the elements and boundaries of the software engineering discipline (Abran et al., 2004). It defines ten Knowledge Areas (KAs) that are recognised as being core to the discipline. ECS can be considered from a similar perspective:

1. *Software Requirements*. When changing a critical system careful consideration must be given to the requirements process – the elicitation, analysis, specification, and validation of requirements. A critical system cannot evolve successfully unless the requirements are correctly elicited and applied.
2. *Software Design*. The design of a critical system will have an important bearing on the cost required for it to evolve successfully. The design includes the software structure and architecture, design quality attributes and evaluation, design notations, and design strategies and methods.
3. *Software Construction*. Software construction for ECS must emphasise the use of appropriate construction processes, the application of coding standards and effective management.
4. *Software Testing*. Testing for ECS offers an opportunity to validate the requirements and design and to assess the performance of the system in an evolving environment. Generation of test suites for evolving critical systems is a particular challenge.
5. *Software Configuration Management (SCM)*. SCM is essential for ECS if the change process is to be managed effectively. This includes planning, identifying the changes required, accounting for changes made, auditing, and release management.
6. *Software Engineering Management (SEM)*. In order to change an ECS in a systematic, disciplined, and quantified manner the best principles in management activities must be applied. The SEM KA includes scope definition, project planning, project enactment, review and evaluation.
7. *Software Engineering Process*. A clear development process will be needed to define the regimen of activities that must occur as a critical system is modified, including process implementation and change, process definition, and assessment.
8. *Software Engineering Tools and Methods*. This KA includes the tools and methods that can assist in software life cycle processes. As such, this KA cuts across the other KAs.
9. *Software Quality*: In order to verify that an ECS has been changed successfully it will be necessary to review its quality. This should include quality assurance, verification and validation, and reviews and audits.

While ECS is related to each of these Knowledge Areas, a tenth Knowledge Area, *Software Maintenance*, is most obviously relevant. Software Maintenance concerns the changing of a software system – the processes and activities concerned with changing software, as well as specific techniques undertaken during maintenance, including program comprehension, re-engineering, and reverse engineering. A more complete mapping of SWEBOK's KAs to ECS is outlined later in Section 4 of this document.

2.2. Related work in Software Evolution

The problem of how to modify software easily without losing quality was widely understood and discussed at the NATO Software Engineering Conference in 1968 (Naur & Randell, 1968). Lehman et al.'s early work on the continuing change process of the IBM OS360-370 operating systems and the work that followed from that led to a large body of research into software evolution and the formulation of eight "Laws of Evolution" (Lehman & Belady 1985, Lehman & Fernández-Ramil 2006). Swanson (1976) identified three types of evolution:

1. *corrective maintenance*, used to overcome processing failure, performance failure, and implementation failure;
2. *adaptive maintenance*, which would overcome change in data environment (e.g., restructuring of a database) and change in processing environment (new hardware, etc); and
3. *perfective maintenance*, which would improve design, which might overcome processing inefficiency, enhance the performance, and the system's maintainability.

Rajlich & Bennett's (2000) staged-life cycle model highlighted the maturity of a software system as being an essential consideration when planning change. More mature software, where many (or all) of the key developers are no longer in place is seen as being harder to evolve than newer software supported by its original developers.

As software evolves in terms of functionality, it often degrades in terms of reliability. While it is normal to experience failures after deployment and the goal of much of software maintenance is to remove these failures, experience has shown that evolution for new functionality and evolution for maintenance can both result in "spikes" of failure (cf. Figure 1). Over time, a traditional system degrades as it evolves and more, rather than fewer, failures are experienced (Lehman, 1996, Parnas, 1994, Rajlich & Bennett, 2000).

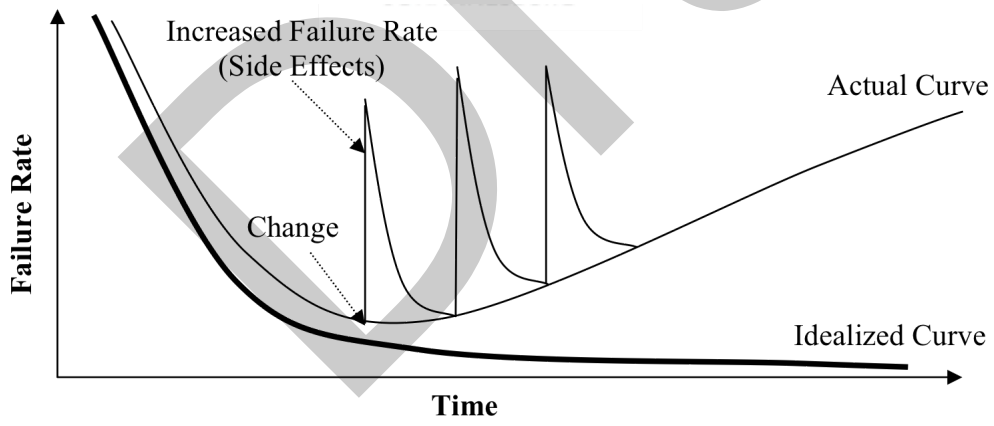


Figure 1: As complex software evolves new defects are introduced, causing the failure rate to spike and increase over time, from Pressman (1997).

Dynamic evolution (sometimes called run-time or automatic evolution) is a special case whereby certain critical systems may need to change during run-time, e.g., by hot-swapping existing components or by integrating newly developed components without first stopping the system (Buckley et al. 2005). This has to be either planned ahead explicitly in the system or else the underlying platform has to provide a means to effectuate software changes dynamically. In terms of the software evolving itself automatically, there are a number of challenges beyond those faced when a human drives the process. Ubiquitous computing systems or autonomic systems are often typified as consisting of large numbers of distributed autonomic, often resource-constrained embedded, systems. These types of systems could be hoped to evolve dynamically but as Baresi et al. (2006) point out, in these domains open-world assumptions about how a piece of software might be used are dominant. Designers cannot fully predict how a system behaves and how it will interconnect with a continuously changing environment. Therefore open assumptions must be built in and software must adapt and react to change dynamically, even if such change is unanticipated.

3. Evolving Critical Systems – PEA+T

Research topics for ECS may usefully be categorized according to the stage of the evolution process to which they are relevant (before, during or after modification). These three stages are referred to here as *Plan*, *Evolve* and *Assess* – shown in Figure 2. A fourth heading, *Tooling*, cuts across the stages of the PEA cycle and concerns the tool support available and required for ECS (in much the same way as the Software Engineering Tools and Methods knowledge area cuts across the other SWEBOK knowledge areas):

- **Plan:** All activities/considerations that can occur *before* the system is next changed, including detecting and weighing the need for evolution and modelling its implications.
- **Evolve:** All activities/considerations that can occur during the modification of the system
- **Assess:** All activities/considerations that can occur *after* the most recent change has been made to the system.
- **Tooling:** Consideration of tools to assist in the evolution cycle. Many of these support the whole cycle, e.g., visualisation and version control, others support aspects, e.g., refactoring support.

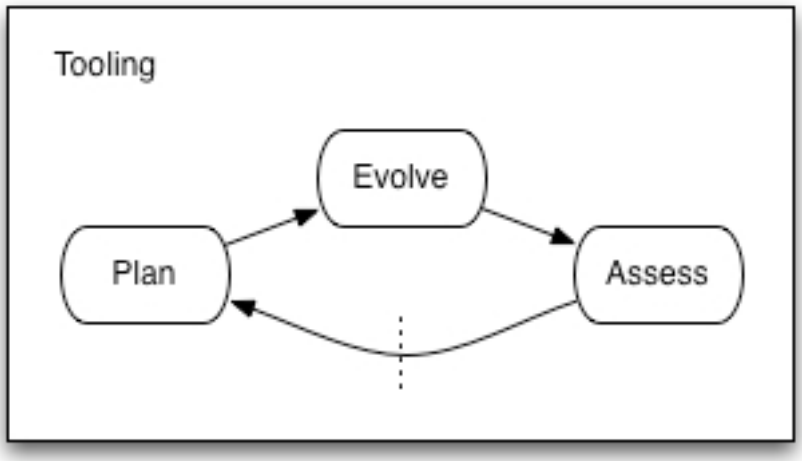


Figure 2: The PEA Cycle for Evolving Critical Systems

The Plan-Evolve-Assess (PEA) model can refer to a whole evolution process or just a single change made as part of a larger development process. After a cycle of change has completed, it may be desirable to undertake another cycle, in which case the outputs of the *Assess* stage will be used as inputs to the *Plan* stage of the next cycle. Continuously evolving autonomic or adaptive systems may be expected to cycle continuously as a control loop (Dobson et al., 2006) and so the separation between the Assess and Plan stages may become blurred.

3.1. A Research Taxonomy for Evolving Critical Systems

We suggest here a breakdown or taxonomy of the software engineering research topics that are most relevant to the ECS domain, arranged in line with the stages of the PEA cycle. The relevant topics are listed with a brief introduction, followed by a sub-topic breakdown.

3.2. Plan

Before undertaking any modification of a critical system it is necessary to consider a number of factors if evolution is to be successful, without being prohibitively expensive. The requirements for change must be carefully captured and applied (**requirements engineering**). It will be necessary to consider the **software development process** used to undertake the change and the system's **architecture design** before modification is undertaken and the intended architecture after modification. **Proscriptive methods** may be used to control the scope of change and **risk management** should be used to assess the causes, likelihood of success and cost of modifications. The classification of the relevant software engineering research topics for the *Plan* stage is shown in Figure 3. The topics are elaborated on below.

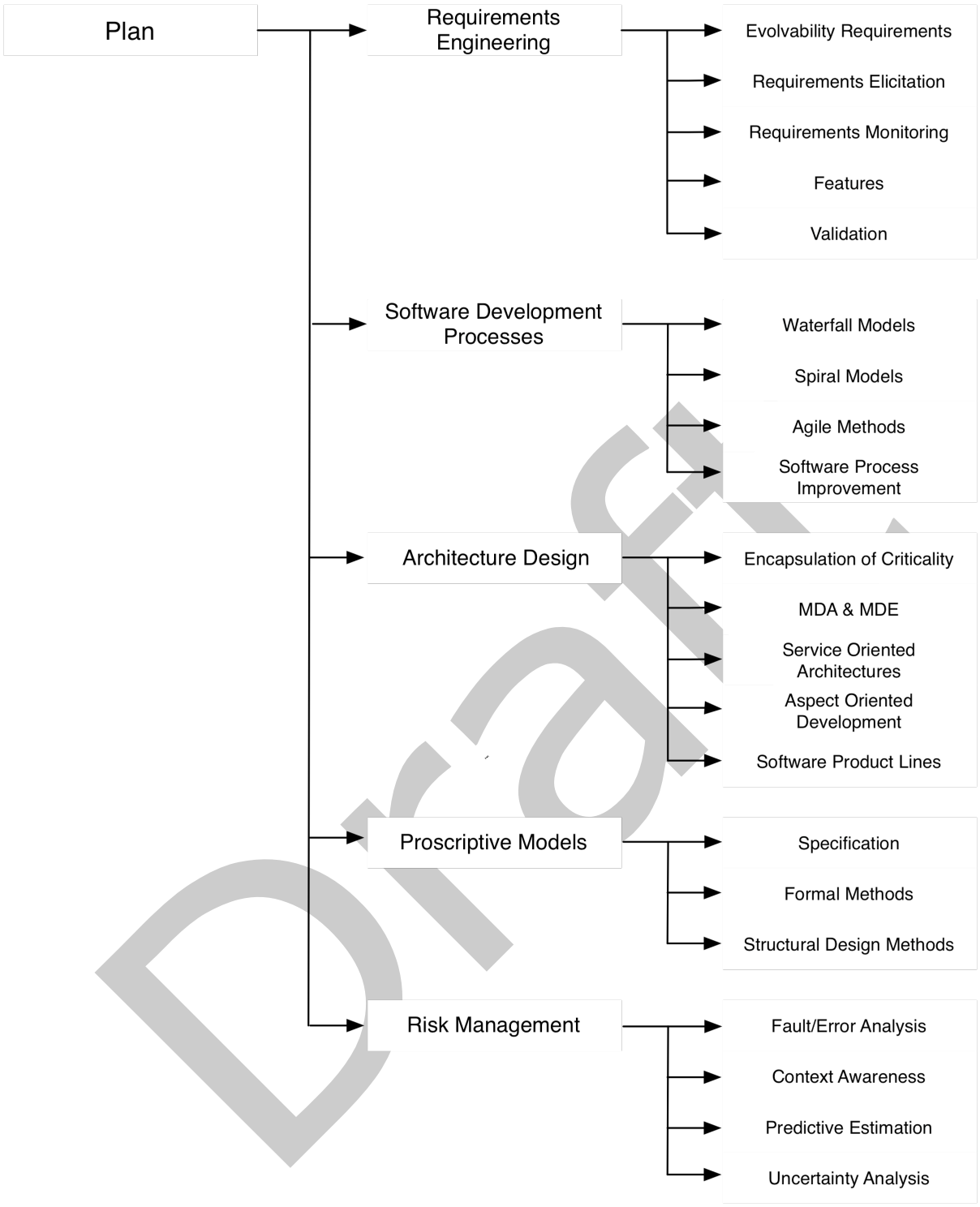


Figure 3: Software Engineering Research Topics under Plan

3.2.1 Requirements Engineering

Van Lamsweerde (2009) identified a number of causes for changes to software systems that are relevant for ECS, including errors and flaws in the requirements document, better customer understanding, new functional features, or improved quality features. A number of environmental changes were also identified, including new classes of users or new usage conditions, new ways of doing things, alternative ways of doing things, new regulations, alternative regulation, organizational changes, new/alternative technology, and new opportunities.

In order to successfully evolve a critical system, it is imperative that we understand *to what the system should be evolving*. We must validate that we are capturing the new system requirements and specifying them correctly (**requirements elicitation**). When requirements specify new or changed features we must ensure that they do not interact adversely with one another or with the existing feature set (**feature interaction and selection**). **Requirements monitoring** can be used to test the extent to which a running system is meeting its requirements (Fickas & Feather, 1995). When developing critical systems we must also be mindful of the fact that it must evolve and this requirement itself should be captured and internalised (**evolvability requirements**).

1) Requirements Engineering
   A) Evolvability Requirements (*cf. Assess 3.A)*
   B) Requirements Elicitation
   C) Requirements Monitoring
   D) Features
      i) Feature Interaction
      ii) Feature Selection
   E) Validation *(cf. Assess 1.A)*

3.2.2 Software Development Processes

If the evolution of software artefacts is to be disciplined, the development processes must be planned, driven and controlled (Lehman & Fernández-Ramil, 2006). **Waterfall** (Royce, 1970), **Spiral** (Boehm, 1988), and **Agile** (Beck et al., 2001) models of software development and evolution have been applied successfully to software evolution. These models lie on a spectrum of regimentation; Planned waterfall-like approaches are very regimented, while feedback-driven Agile methods are more flexible, with Spiral models lying between the two.

An organisation's choice of software process will depend on a number of factors, including the criticality, risk, and regulatory environment of the system. Less regimented processes are becoming more popular due to their speed and efficiency but are often perceived to be less suited to the development and evolution of larger, more critical systems, or to systems whose development must adhere to regulatory guidelines. The development process may itself evolve; a number of steps have been made to formalise **process improvement** in terms of reliability, repeatability, and reducing cost as well as in the assessment of particular software development processes.

2) Software Development Processes
   A) Waterfall Models
   B) Spiral Models
   C) Agile Methods
   D) Software Process Improvement *(cf. Assess 4)*
      i) CMMI *(cf. Assess 4.A)*
      ii) Process Assessment

3.2.3 Architecture Design

The use of well-designed architectures will have an important bearing on the ease with which a critical system can be successfully evolved. The system is afforded a measure of future proofing by encapsulating aspects of the system that are both critical and likely to change (**encapsulation of criticality**). **Model-driven architectures** allow the specification of a system to be defined as a model that is distinct from the implementation; the intent is that the model is defined in such a way that it is easy to change and such changes can be reflected in the implementation (ideally automatically) in preference to undertaking the more expensive process of changing the implementation itself (Hearnden et al., 2004). A critical system may use a **service-oriented architecture** to expose functionalities that can be changed or composed in different combinations to provide new services. **Aspect-oriented development** increases modularity and improves maintainability by separating cross cutting concerns and reducing code tangle. The application of **software product line** approaches can improve reuse and evolvability (Weiss & Lai, 1999).

3) Architecture Design
   A) Encapsulation of Criticality (*cf. Assess 3.B*)
      i) Fault Isolation
   B) Model-Driven Architectures and Model-Driven Engineering
      i) Code Generation
   C) Service Oriented Architectures
      i) Evolving Middleware
      ii) Service/Feature Selection
   D) Aspect-Oriented Development
   E) Software Product Lines (SPLs)
      i) Dynamic Software Product Lines (DSPLs)
      ii) Software Process Lines
      iii) Services and SPLs

3.2.4 Proscriptive Models

Proscriptive models (Heimbigner, 1990) define what is required of the system at the end of an evolutionary cycle rather than the activities that must be performed during the cycle. By **specifying** requirements formally, (including system objectives, domain concepts, functional and non-functional requirements) before a change takes place, it becomes possible to validate the changed system formally against that specification, thus ensuring that the modifications were successfully applied (**formal methods**). **Structural Design Methods** describe the intended system based on a decomposition of the structure of that system, e.g., SSADM, Yourdon, and more recently UML. The structure is used to derive the design of the system and describe its functionality in a way that can later be used to drive the implementation.

4) Proscriptive Models
   A) Specification
      i) System Objectives

ii) Domain Concepts

iii) Functional Requirements

iv) Non-functional Requirements

B) Formal Methods

i) Formal Specification

ii) Validation *(cf. Assess 1)*

(a) Formal Analysis *(cf. Assess 1.B)*

iii) Refinement/Reification

(a) Data Refinement

(b) Operation Refinement

C) Structural Design Methods

3.2.5 Risk Management

As software ages, modification will be increasingly likely to introduce unforeseen errors into a system (Parnas, 1994, Lehman, 1996). This poses significant risks when modifying critical software, as the software *must* retain its quality after changes are applied. When change is necessary due to flaws in a system, an analysis must be performed to diagnose the underlying cause and plan the treatment (Mens & Demeyer, 2001) (**fault/error analysis)**; if a system is self-evolving it needs to be aware of its current context (**context-awareness)** in order to do so. The potential risk and cost of performing evolution should be estimated *a priori*, as should the evolution's likelihood of success and cost (Ramil & Lehman, 2000) (**predictive estimation)**. Underlying these concerns is the need to model the various inherent uncertainties of the system, its operational environment, and the agents of change (Baresi et al., 2006) (**uncertainty analysis)**. Modification of a critical system must occur only when it is certain that such changes will not reduce the quality of the system.

5) Risk Management

A) Fault/Error Analysis

i) Treatment Analysis

(a) Diagnosis

ii) Error Processing

(a) Detection

(b) Diagnosis

B) Context Awareness

C) Predictive Estimation

i) Prediction of Cost/Effort

ii) Process Metrics

D) Uncertainty Analysis

3.2.6 Summary of the *Plan* Phase

We should not feel confident in changing a critical system until the planning stage is complete and we are certain of

- what we are trying to achieve by modifying the system (**requirements engineering**);
- how we will undertake the modification (**software development processes**);
- what we are modifying (**architecture design**);
- what the scope of the modification is (**proscriptive models**);
- and have assessed the possibility and cost of success (**risk management**).

3.3. Evolve

Once we have planned our modifications and are confident of success we are able to undertake the actual modification of the software. We must fully understand the critical system and be able to affect changes safely in order to modify it successfully. Modification may require the reconstitution of the system in a new form (**reengineering**). As these changes are made we must be able to trace the impact of each change throughout the system (**impact analysis**). If faults are present in the system, either pre-existing or introduced by the modification, they must be treated and removed (**fault treatment and error processing**). The classification of the relevant software engineering research topics for the *Evolve* stage is shown in Figure 4.

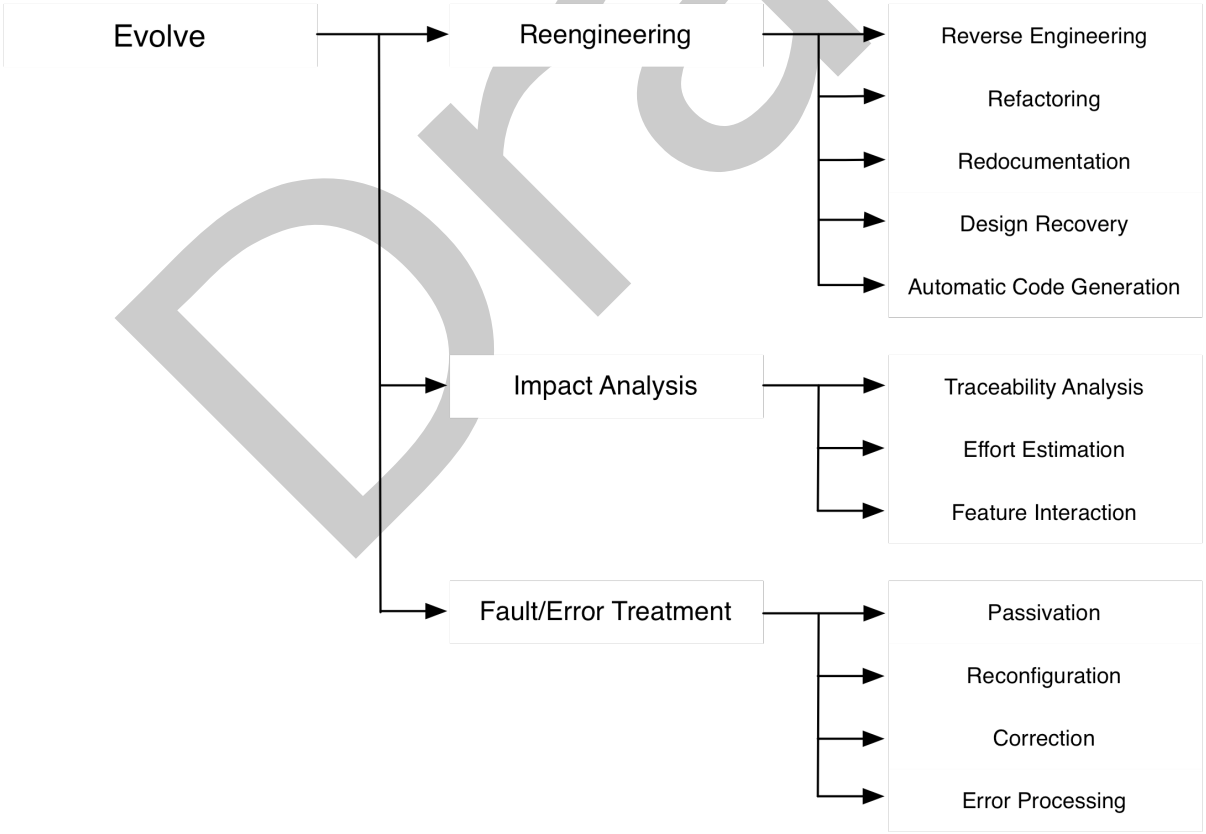


Figure 4: Software Engineering Research Topics under Evolve

3.3.1 Reengineering

Reengineering involves the examination and alteration of an existing system to reconstitute it in a new form (Chikofsky & Cross, 1990). System comprehension may be a significant barrier to such evolution (Rajlich & Wilde, 2002), especially as many long-lived critical systems are not well documented and/or few (or none) of its original developers are available to explain its inner workings (Rajlich & Bennett, 2000). In this case it may be necessary to **reverse engineer**, to reconstruct the software's design (**design recovery**), and/or to **redocument** the system to understand how to make the necessary changes. Once the system is understood it may be **refactored** (Fowler, 1999) into its modified form either manually or automatically (**automatic code generation)**.

1) Reengineering
   A) Reverse Engineering
      i) Concept Location (cf. Evolve 2.A)
   B) Refactoring
      i) Model Transformation *(cf. Plan 3.B)*
      ii) Configuration-Management *(cf. Plan 3.C, Plan 3.E)*
      iii) Automatic Code Generation *(cf. Evolve 2.A.v)*
      iv) Architectural Refactoring
   C) Redocumentation
   D) Design Recovery
   E) Automatic Code Generation

3.3.2 Impact Analysis

As software is modified, seemingly small changes can cause a ripple effect, leading to significant consequences for the modification process. To manage this it is important to be able to identify and trace the impact of each change (**traceability analysis**). Traceability has a significant cost, and can be an additional source of errors, but for ECS it is essential for a number of artefacts, including requirements, assumptions, architecture, risks, design modules, faults, documentation, correctness proofs, tests and source code (Lions (1996) attributed the Ariane 5 disaster to the non-traceability of assumptions and their dependencies).

As evolution is taking place it may be important to monitor and estimate the effort required to complete the cycle completely (**effort estimation**). In systems that evolve in runtime there may be constraints on the amount of time that is allowed for evolution to take place; if analysis reveals that the process is likely to take too long or is likely to fail, the process may be halted or other mitigating steps taken. Amortisation may be used to balance the cost of change against its long-term benefit. This would involve estimating in advance the effects, costs, effort, and time required to undertake an evolution action, as well as its anticipated benefits.

As new features are added to a system, or as existing features are changed, it is possible that they may interact in a detrimental manner (**feature interaction**). While these features in isolation may be correctly designed and implemented, unexpected interactions between the features may occur when these features are integrated into a larger system. It is essential that these potential interactions be detected as early as possible in the development process.

2) Impact Analysis
   A) Traceability Analysis
      i) Requirements *(cf. Plan 1)*
      ii) Assumptions *(cf. Plan 4.A)*
      iii) Compliance *(cf. Plan 4.A)*
      iv) Risk, Criticality *(cf. Plan 5)*
      v) Source Code Analysis
         (a) Program Slicing
         (b) Refactoring *(cf. Evolve 1.B)*
      vi) Software Architecture *(cf. Plan 3)*
      vii) Design Models *(cf. Plan 3.B)*
      viii) Faults *(cf. Plan 5.A)*
         (a) Fault Tolerance *(cf. Assess 2.B.ii)*
      ix) Test Coverage
      x) Developer or End-user Documentation
      xi) Correctness Proofs *(cf. Plan 4.B)*
   B) Effort Estimation *(cf. Plan 5.C)*
      i) Timeliness Monitoring
      ii) Amortisation
   C) Feature Interaction Analysis *(cf. Plan 1.D)*
      i) Configuration-Management
      ii) Software Product Lines *(cf. Plan 3.E)*

3.3.3 Fault Treatment and Error Processing

If a fault is observed in a critical system its treatment and removal will take high priority. If the fault is critical to the functionality of a live system, it may be necessary to temporarily disable the affected function (**passivation**) or to **reconfigure** the system before **correcting** the fault. When autonomous critical systems encounter errors during operation, they must be capable of identifying, detecting, and recovering from the error, potentially without human assistance (**error processing**).

3) Fault Treatment *(cf. Plan 5.A)*
   A) Passivation
   B) Reconfiguration
   C) Correction
   D) Error Processing *(cf. Plan 5.A)*
      i) Recovery

3.3.4 Summary of the *Evolve* Phase

The Evolve phase covers the mechanics of actually changing a critical system – from system comprehension, reverse engineering, and refactoring/reengineering to tracing the impact of each change made and overcoming and correcting faults.

3.4. Assess

After a critical system has been modified and before it is activated it must be assessed to ensure that quality has not been lost. The modified system must be validated against its requirements to ensure that the changes achieved the desired effects (**incremental verification and validation**). The system can be assessed using a number of **software quality attributes** to measure its characteristics, including safety, dependability, and security. Its architecture can be assessed to ensure that it reflects good principles of software design – this will be important if the system is to continue to be evolvable in the future (**architecture quality metrics**). Finally, at the end of a cycle of evolution we can attempt to refine the evolution process itself by reflecting on the success and cost of the cycle (**software process improvement**). The classification of the relevant software engineering research topics for the *Assess* stage is shown in Figure 5.

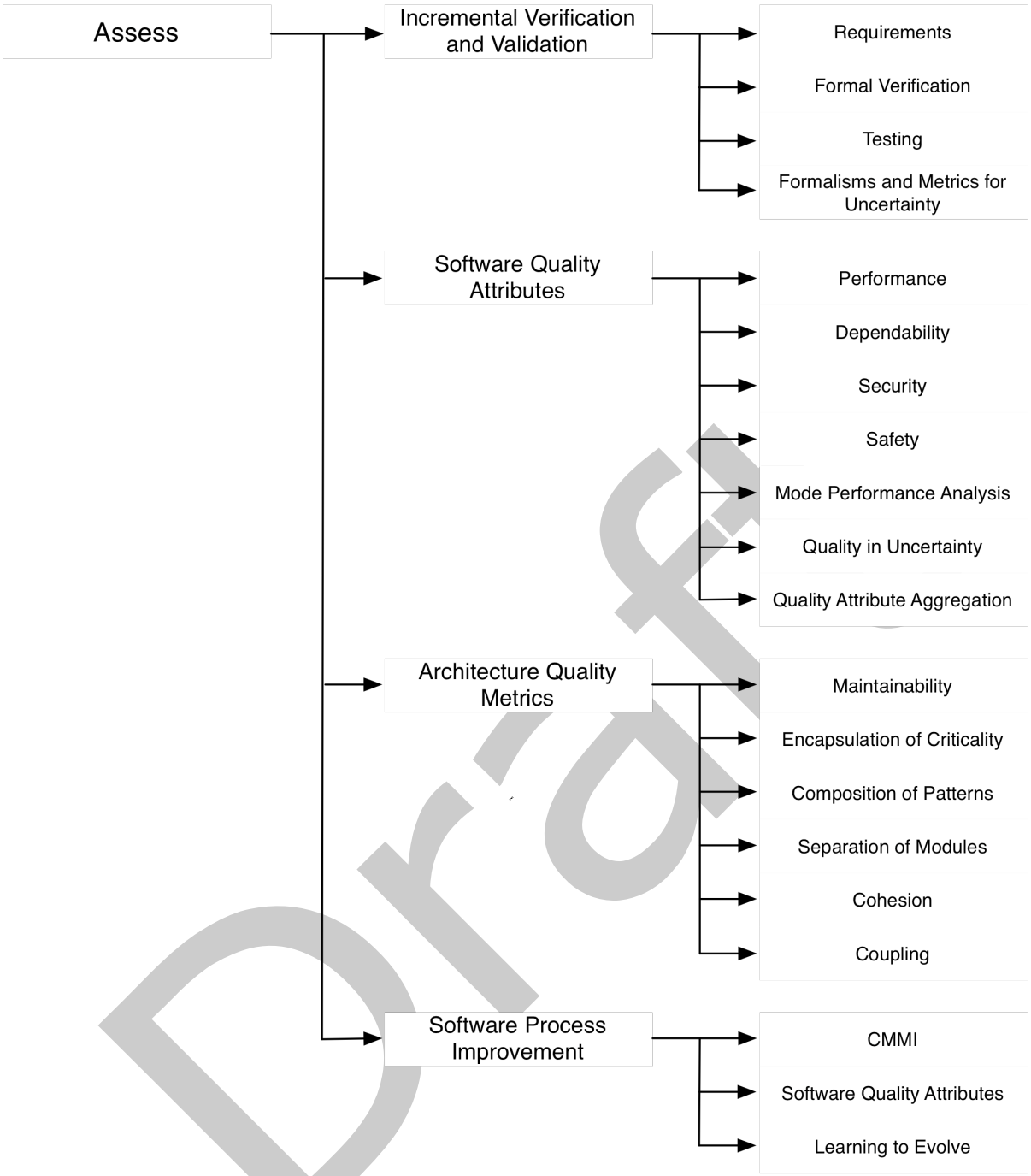


Figure 5: Software Engineering Research Topics under Assess

3.4.1 Incremental Verification and Validation

If evolution of a critical system is to be successful the newly evolved system must be verified and validated against its **requirements**. This can be done using **formal verification** – but one of the key difficulties of formal verification is that its cost tends to match the size of the system rather than the size of the change. An important challenge for ECS is to ensure that the scope of the verification correlates to the magnitude of the change – that it is truly *incremental* (Mens et al., 2005).

Software tests with high levels of code coverage has been shown to improve reliability (Lyu, 2007) and regression testing can be used to prevent old bugs from reappearing after changes have been made (**testing**).

Even if the evolved system is found to meet its requirements, is formally verified, and passes its test suite – it is possible that uncertainty in the operational environment will cause it to fail if this uncertainty is not incorporated into the verification process. This highlights a requirement for **formalisms and metrics to express uncertainty**.

1) Incremental Verification and Validation
   A) Requirements *(cf. Plan 1)*
      i) Consistency
      ii) Completeness
   B) Formal Verification *(cf. Plan 4.B)*
      i) Proof
      ii) Static Analysis
      iii) Dynamic Analysis
   C) Testing
      i) Regression Testing
      ii) Fault Analysis
      iii) Test Coverage
      iv) Automatic Test Case Generation
   D) Formalisms and Metrics for Uncertainty

3.4.2 Software Quality Attribute Measurement

After evolving a critical system its quality attributes should be measured in terms of **performance**, **dependability**, **security**, and **safety** (Barbacci et al., 1995, Leveson, 1986). If a system can operate in a number of different behaviour modalities with different levels of criticality these metrics must be measured and assessed individually for each mode (**mode performance analysis**). Metrics that account for uncertainty will be necessary in dynamic operating environments, as will estimation of the extent to which the system will be robust to unanticipated conditions or faults (Schneidewind, 1996, Schneidewind & Hinchey, 2008) (**assessing quality in the presence of uncertainty**). Given that there will often be trade-offs between desirable quality attributes in a critical system it may be necessary to develop meta-measures that combine these measures to make the most appropriate trade-offs (**quality attribute aggregation**).

2) Software Quality Attribute Measurement *(cf. Plan 4.A)*
   A) Performance
      i) Latency
      ii) Throughput
      iii) Capacity
   B) Dependability
      i) Reliability
      ii) Fault Tolerance *(cf. Plan 5.A)*

iii) Availability

C) Security *(cf. Plan 5.A)*

i) Confidentiality

ii) Integrity

iii) Availability

D) Safety *(cf. Plan 5.A)*

i) Hazard Identification

ii) Hazard Analysis

E) Mode Performance Analysis

F) Assessing Quality in the presence of uncertainty *(cf. Plan 1.C)*

G) Quality Attribute Aggregation

3.4.3 Architecture Quality Metrics

In order for a critical system to remain evolvable or maintainable (i.e., to remain in Rajlich & Bennett's (2000) evolution stage of the software life cycle) it is essential to assess whether the system's architecture complies with good design principles. Architecture elements that are badly structured, contain errors, or are incomplete can be considered evolution-critical and should be refactored (Mens & Demeyer, 2001) (**maintainability)**. Core features of a critical system's architecture should be modularised and emphasis placed on their correct design (**encapsulation of criticality**). After any refactoring-focused evolution (or maintenance) cycle it will be necessary to ensure that the new architecture has improved quality, without impacting on functionality or on any other software quality metrics. In order to confirm this, a useful technique may be to identify and ensure the correct application and composition of design patterns (Tsantalis & Halkidis, 2006, Fowler, 1999) (**composition of architecture patterns**), to assess the application of best principles in information hiding (Parnas, 1972) (**separation of modules**), and to ensure the improvement of Object Oriented Design metrics (Chidamber & Kemerer, 1994), where appropriate (**high cohesion, low coupling**).

3) Architecture Quality Metrics *(cf. Plan 3)*

A) Maintainability *(cf. Plan 1.A)*

B) Encapsulation of Criticality *(cf. Plan 3.A)*

C) Composition of Architecture Patterns

D) Separation of Modules

i) Failure Resilience *(cf. Plan 3.A.i)*

E) High Cohesion

F) Low Coupling

3.4.4 Software Process Improvement

Existing process improvements like the SEI's Capability Maturity Model (CMM) and Capability Maturity Model Integration (**CMMI**) have been shown to reduce defect density

(Diaz & Sligo, 1997). Every time a cycle of change has completed there is an opportunity for analysing and improving the process or metrics used (**improved software quality attributes)**. Autonomic systems could incorporate learning algorithms into the cycle to improvement their processes (**learning to evolve**).

4) Software Process Improvement *(cf. Plan 2.D)*
   A) CMMI *(cf. Plan 2.D.i)*
   B) Improved Software Quality Attributes *(cf. Assess 2)*
   C) Learning to Evolve *(cf. Plan, Evolve, Assess)*
      i) Correctness *(cf. Assess 1)*
      ii) Timeliness *(cf. Evolve 2.B.i)*

3.4.5 Summary of the *Assess* Phase

The research topics under *Assess* cover the process of confirming that the modifications were applied successfully (**verification and validation**) and the metrics that may be applied to confirm that system and architecture quality were retained (or improved) after modification (**software quality attributes, architecture quality metrics**). At the end of each cycle there is an opportunity to reflect upon and improve the process of change itself (**software process improvement**).

4. Discussion

The taxonomy outlined in the earlier section represents all the main software engineering research areas for ECS. In Section 2.1 we outlined briefly the relationship between ECS and the software engineering research topics listed in the SWEBOK. Figure 6 shows a mapping between the SWEBOK knowledge areas and the Plan-Evolve-Assess stages of ECS. In order to explore the research agenda for ECS further we outline a number of key research questions for ECS.

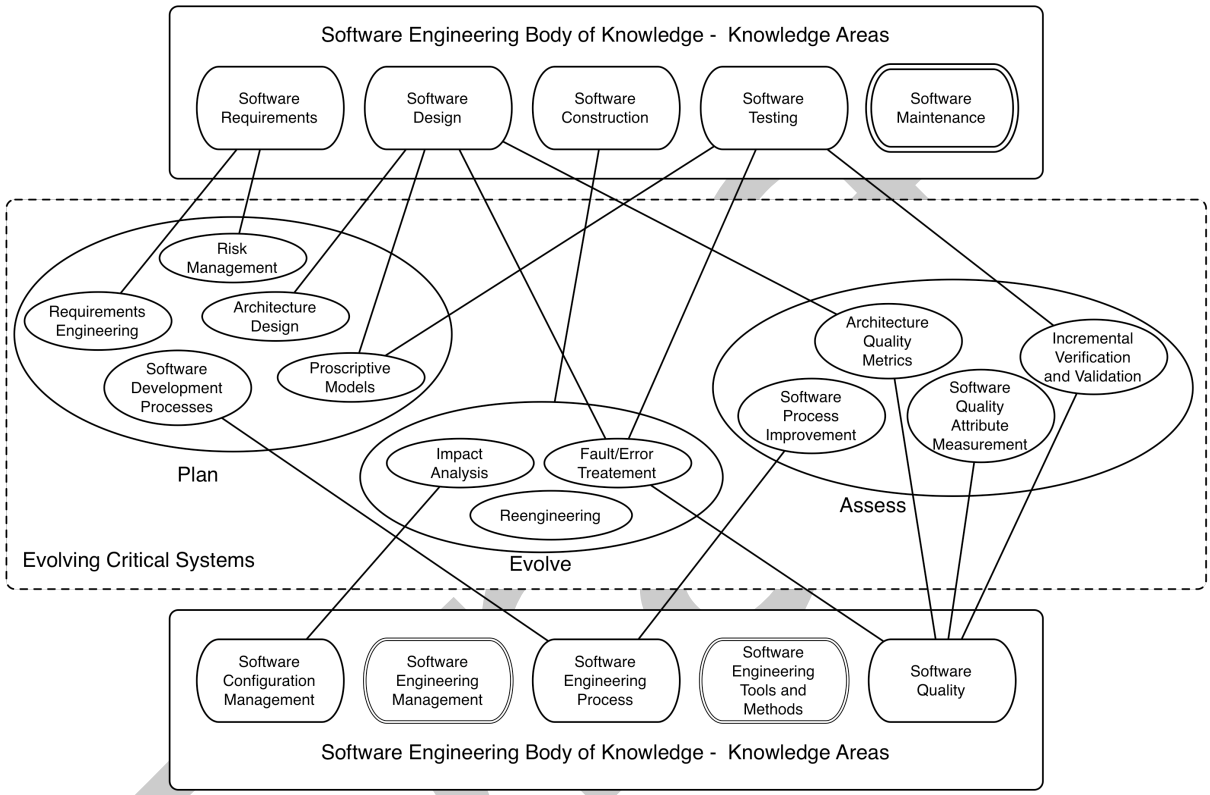


Figure 6: Mapping of ECS to the SWEBOK Knowledge Areas

4.1. Research Questions for ECS

The following are a number of interesting research questions for ECS broken into the Plan-Evolve-Assess stages. Fundamental questions underlying ECS research are:

- How do we design, implement, and maintain critical software systems that
  a) are highly reliable
  b) retain this reliability as they evolve *without* incurring prohibitive costs.
- How can we maintain critical software quality when its teams, processes, methods and toolkits are in a state of constant change?

4.2. 'Plan' Questions

Under the Plan phase the following are important research questions:

- How do we specify what we want an evolution cycle to produce and how will we know that it has been successful?

- What new advances in requirements engineering are appropriate for ECS? How can requirements be captured and elucidated in a manner that allows for subsequent successful change in both the system and the requirements?
- What are the software design methodologies that best facilitate the support and maintenance of ECS?
- What changes to current processes are needed to make agile an appropriate environment in which to develop critical systems?
- What is the best architectural model/technique for ECS? What characteristics are best for successful ECS software architectures?
- How do we effectively model a changing environment?
- How best can we estimate the effort involved in specific evolution exercises?

4.3. 'Evolve' Questions

Under the Evolve phase the following research questions may be considered:

- To what extent can we *automatically derive* evolved critical systems from models?
- In an evolving system how do we keep the various software artifacts (e.g., documentation and source code) in sync? Which artifacts are most important for traceability in an ECS
- How might automated design improvement techniques be used to prepare a critical system for evolution?
- How can we evolve systems where there is considerable uncertainty in the operational environment, where the environment changes in a deterministic, non-deterministic, or stochastic manner?

4.4. 'Assess' Questions

The following research questions apply to the Assess phase:

- How do we ensure continual compliance with regulatory requirements in an ECS?
- How can we ensure that software never evolves into a state where it exhibits unstable behaviour?
- Can we provide useable (quality/reliability) analysis tools and methods to evolve critical systems?
- What metrics are suitable to assess the success of evolution?
- How can we enforce run time policies in the presence of a changing environment?
- Can we incorporate learning into large-scale ECS?

5. Conclusions

We have outlined the ECS research domain, which is concerned with the creation of knowledge, processes, protocols, and tools for the efficient development of *critical systems that evolve*. In developing and maintaining evolving critical systems there are tensions between reliability, predictability, and cost of evolution on the one hand, and the need for the system to evolve on the other. Given these tensions we must ask which processes, techniques and tools are the most cost-effective for evolving critical systems. In fact, there may exist some systems that should not evolve at all because the cost and risk of performing evolution successfully exceeds the value of the system by orders of magnitude. We believe that the community should develop objective criteria to decide whether a given system is in this class, and specific techniques to design such systems where version 1.0 is the final one.

The body of this paper suggests the Plan-Evolve-Assess cycle for critical systems evolution and outlines a taxonomy of research topics for ECS. The taxonomy delineates the boundaries of the ECS field and shows how it relates to widely recognised research topics within the broader software engineering discipline. While we have outlined many issues that may be considered to successfully evolve critical software, it would be costly to apply all of them for any single system. The topics outlined here may be viewed as options for an organisation seeking to evolve critical software rather than a prescriptive regimen.

The paper was concluded with a set of research questions that ECS research should seek to answer. This list is by no means exhaustive and we anticipate that further questions will arise as the field is explored further.

6. References

A. Abran, J. W. Moore, P. Bourque, and R. Dupuis, editors. Guide to the Software Engineering Body of Knowledge (SWEBOK). IEEE Computer Society, 2004. Available online: http://www2.computer.org/portal/web/swebok/htmlformat.

M. Barbacci, T. H. Longstaff, M. H. Klein, and C. B. Weinstock. Quality attributes. Technical Report CMU/SEI-95-TR-021, Software Engineering Institute, Carnegie Mellon University, 1995.

L. Baresi, E. D. Nitto, and C. Ghezzi. Towards open-world software: Issue and challenges. In Software Engineering Workshop, 2006. SEW '06. 30th Annual IEEE/NASA, pages 249–252, April 2006.

K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland, and D. Thomas. The agile manifesto, 2001. Available online: http://agilemanifesto.org.

B. W. Boehm. A spiral model of software development and enhancement. Computer, 21(5):61–72, 1988. ISSN 0018-9162.

J. Buckley, T. Mens, M. Zenger, A. Rashid, and G. Kniesel. Towards a taxonomy of software change: Research articles. J. Softw. Maint. Evol., 17(5):309–332, 2005. ISSN 1532-060X.

G. Canfora and M. Di Penta. New frontiers of reverse engineering. In Future of Software Engineering, 2007. FOSE '07, pages 326–341, May 2007.

S. Chidamber and C. Kemerer. A metrics suite for object oriented design. Software Engineering, IEEE Transactions on, 20(6):476–493, Jun 1994. ISSN 0098-5589.

E. Chikofsky and I. Cross, J.H. Reverse engineering and design recovery: a taxonomy. Software, IEEE, 7(1):13–17, Jan 1990. ISSN 0740-7459.

R. Cohen, K. Erez, D. ben Avraham, and S. Havlin. Breakdown of the internet under intentional attack. Phys. Rev. Lett., 86(16):3682–3685, Apr 2001.

M. Diaz and J. Sligo. How software process improvement helped Motorola. Software, IEEE, 14(5):75–81, Sep/Oct 1997. ISSN 0740-7459.

S. Dobson, S. Denazis, A. Fernández, D. Gaïti, E. Gelenbe, F. Massacci, P. Nixon, F. Saffre, N. Schmidt, and F. Zambonelli. A survey of autonomic communications. ACM Trans. Auton. Adapt. Syst., 1(2):223–259, 2006. ISSN 1556-4665.

S. Dobson, L. Coyle, G. O'Hare, and M. Hinchey. From physical models to well-founded control. Engineering of Autonomic and Autonomous Systems, IEEE International Workshop on, 0:119–124, 2009.

C. Ebert and C. Jones. Embedded software: Facts, figures, and future. Computer, 42(4):42–52, 2009. ISSN 0018-9162.

S. Fickas and M. Feather. Requirements monitoring in dynamic environments. In Requirements Engineering, 1995., Proceedings of the Second IEEE International Symposium on, pages 140–147, Mar 1995.

M. Fowler. Refactoring: improving the design of existing code. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999. ISBN 0-201-48567-2.

D. Hearnden, P. Bailes, M. Lawley, and K. Raymond. Automating software evolution. In Software Evolution, 2004. Proceedings. 7th International Workshop on Principles of, pages 95–100, Sept. 2004.

D. Heimbigner. Proscription versus prescription in process-centered environments. In Software Process Workshop, 1990. 'Support for the Software Process'., Proceedings of the 6th International, pages 99–102, Oct 1990.

B. Jacob, R. Lanyon-Hogg, D. Nadgir, and A. Yassin. A practical guide to the IBM autonomic computing toolkit. Technical report, IBM International Technical Support Organization, 2004.

J. Kramer and J. Magee. The evolving philosophers problem: Dynamic change management. IEEE Trans. Softw. Eng., 16(11):1293–1306, 1990. ISSN 0098-5589.

J. Kramer and J. Magee. Self-managed systems: an architectural challenge. In Future of Software Engineering, 2007. FOSE '07, pages 259–268, May 2007.

M. Lehman. Programs, life cycles, and laws of software evolution. Proceedings of the IEEE, 68(9):1060–1076, Sept. 1980. ISSN 0018-9219.

M. M. Lehman. Laws of software evolution revisited. In EWSPT '96: Proceedings of the 5th European Workshop on Software Process Technology, pages 108–124, London, UK, 1996. Springer-Verlag. ISBN 3-540-61771-X.

M. M. Lehman and L. A. Belady, editors. Program evolution: processes of software change. Academic Press Professional, Inc., San Diego, CA, USA, 1985. ISBN 0-12-442440-6.

M. M. Lehman and J. C. Fernández-Ramil. Software Evolution and Feedback: Theory and Practice, chapter Software Evolution. John Wiley & Sons, 2006. ISBN 0470871806.

N. G. Leveson. Software safety: why, what, and how. ACM Comput. Surv., 18(2):125–163, 1986. ISSN 0360-0300.

J. L. Lions. Ariane 5 flight 501 failure, 1996. Report by the Inquiry Board.

M. Lyu. Software reliability engineering: A roadmap. In Future of Software Engineering, 2007. FOSE '07, pages 153–170, May 2007.

M. R. Lyu, editor. Handbook of software reliability and system reliability. McGraw-Hill, Inc. Hightstown, NJ, USA, 1996. ISBN 0-07-039400-8.

McAfee. McAfee virtual criminology report, 2008. Available online: http://resources.mcafee.com/content/NAMcAfeeCriminologyReport

T. Mens and S. Demeyer. Future trends in software evolution metrics. In IWPSE '01: Proceedings of the 4th International Workshop on Principles of Software Evolution, pages 83–86, New York, NY, USA, 2001. ACM. ISBN 1-58113-508-4.

T. Mens, M. Wermelinger, S. Ducasse, S. Demeyer, R. Hirschfeld, and M. Jazayeri. Challenges in software evolution. In Eighth International Workshop on Principles of Software Evolution, pages 13–22, Sept. 2005.

P. Naur and B. Randell, editors. Software Engineering: Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct. 1968, Brussels, Scientific Affairs Division, NATO, 1968.

P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum, and A. L. Wolf. An architecture-based approach to self-adaptive software. IEEE Intelligent Systems, 14(3):54–62, 1999. ISSN 1541-1672.

C. Pancake. Debugger visualization techniques for parallel architectures. In Compcon Spring '92. Thirty-Seventh IEEE Computer Society International Conference, Digest of Papers., pages 276–284, Feb 1992.

D. L. Parnas. On the criteria to be used in decomposing systems into modules. Commun. ACM, 15(12):1053–1058, 1972. ISSN 0001-0782.

D. L. Parnas. Software aging. In ICSE '94: Proceedings of the 16th international conference on Software engineering, pages 279–287, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press. ISBN 0-8186-5855-X.

D. L. Parnas. Software fundamentals: collected papers by David L. Parnas, chapter: On the design and development of program families, pages 193–213. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001. ISBN 0-201-70369-6.

R. S. Pressman. Software Engineering: A Practitioner's Approach, 4 ed. McGraw-Hill, 1997.

V. Rajlich and K. H. Bennett. A staged model for the software life cycle. Computer, 33(7):66–71, 2000.

V. Rajlich and P. Gosavi. Incremental change in object-oriented programming. IEEE Softw., 21(4):62–69, 2004. ISSN 0740-7459.

V. Rajlich and N. Wilde. The role of concepts in program comprehension. In Program Comprehension, 2002. Proceedings. 10th International Workshop on, pages 271–278, 2002.

J. F. Ramil and M. M. Lehman. Metrics of software evolution as effort predictors - a case study. In ICSM '00: Proceedings of the International Conference on Software Maintenance (ICSM'00), page 163, Washington, DC, USA, 2000. IEEE Computer Society. ISBN 0-7695-0753-0.

W. W. Royce. Managing the development of large software systems. In IEEE WESCON, pages 1–9. TRW, August 1970.

N. Schneidewind. Reliability and risk analysis for software that must be safe. In Software Metrics Symposium, 1996., Proceedings of the 3rd International, pages 142–153, Mar 1996.

N. Schneidewind and M. Hinchey. Why predicting outliers in software is a good thing to do! In ICECCS '08: Proceedings of the 13th IEEE International Conference on Engineering of Complex Computer Systems, pages 91–97, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3139-7.

N. F. Schneidewind. Measuring and evaluating maintenance process using reliability, risk, and test metrics. IEEE Trans. Softw. Eng., 25(6):769–781, 1999. ISSN 0098-5589.

H. Shokry and M. Hinchey. Model-based verification of embedded software. Computer, 42(4):53–59, 2009. ISSN 0018-9162.

H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. D. Dobbs Journal, 30(3), Mar. 2005.

E. B. Swanson. The dimensions of maintenance. In ICSE '76: Proceedings of the 2nd international conference on Software engineering, pages 492–497, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

W. Truszkowski, M. Hinchey, J. Rash, and C. Rouff. Autonomous and autonomic systems: a paradigm for future space exploration missions. Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on, 36(3):279–291, May 2006. ISSN 1094-6977.

N. Tsantalis and S. T. Halkidis. Design pattern detection using similarity scoring. IEEE Trans. Softw. Eng., 32(11):896–909, 2006. ISSN 0098-5589.

A. van Lamsweerde. Requirements Engineering: From System Goals to UML Models to Software Specifications. Wiley & Sons, 2009.

P. Wang, M. C. González, C. A. Hidalgo, and A-L. Barabási. Understanding the spreading patterns of mobile phone viruses. Science, pages 1071–1076, April 2009.

D. M. Weiss and C. T. R. Lai. Software product-line engineering: a family-based software development process. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999. ISBN 0-201-69438-7.

1. Appendix: ECS Scenarios

ECS is important for all application domains, but most of all where large and important software systems persist over a period of years, and must be upgraded or enhanced without being rewritten from scratch; or must continue to operate as specified in changing environmental conditions. Examples of important application domains include medical devices, financial services, and telecommunications. Such domains require reliable, flexible software of the highest standard, capable of being changed without having to go back to the drawing board for each new version or iteration. After having outlined the ECS research field we now explore the application of ECS by outlining scenarios from five different domains.

1.1. Parallel and Multicore Computing

Many developers are faced with the opportunity of increased processing power brought about by the advent of multicore computing. However, in many cases, software based on serial execution is ported as-is to the multicore environment. It may indeed run more quickly than on a single processor, but it fails to exploit more than a fraction of the potential power (Sutter, 2005). Writing directly for multicore or porting serial execution software to multicore is difficult; one of the reasons for this is because the parallel environment is susceptible to subtle variations in processor speed, load balancing, memory latency, the sequence and timing of external interrupts and communications topology (Pancake, 1992). Ideally software should be evolved (automatically if possible) to best utilise all available resources.

Some key questions in this application domain are

- Could critical software be designed to evolve itself in reaction to the dynamic availability of computing resources?
- How might a critical system be refactored according to different levels of parallelisation?
- How might security and reliability be guaranteed in the presence of automatically generated code?
- How can we evolve existing software to exploit multicore and parallel architectures fully?
- What verification and validation approaches are necessary for critical systems in parallel computing?

1.2. Critical Network Protection

Networks can describe many important technological infrastructures, including the Internet, national or international power grids, peer-to-peer systems, wireless sensor and delay-tolerant networks. There is evidence that the security of our online infrastructures has been breached a number of times (McAfee, 2008). Network dynamics can allow a local failure/threat to quickly develop into a global failure (e.g., large scale black-outs or e-mail virus outbreaks) (Cohen et al., 2001, Wang et al., 2009).

Key questions in this domain include the following

- Can we maintain the integrity of networked data and ensure the responsiveness and availability of network infrastructure in the presence of malicious onslaughts from unknown (possibly automated) sources?
- How can essential upgrades be made to these infrastructures (e.g. to fix security loopholes, to make upgrades) without taking systems out of operation for any period of time?
- How might a critical network predict and evolve itself to prevent the onset of a possible global network failure?

1.3. Autonomous Environmental Monitoring

Performing environmental monitoring using swarms of autonomous mobile agents is attractive when it is too dangerous or expensive to send humans to do the task (Oreizy, 1999, Truszkowski et al., 2004). The swarm can be left in place over long periods of time, during which its mission may be expected to change. It will not always be appropriate or possible for a human to be involved in the evolution cycle – the system's environment can change quickly leading to new requirements for the software. The classic example of spacecraft operating far from earth is most obvious, but more mundane examples abound, where the software may be too complex or the change too urgent to wait for a team of humans to undertake the required evolution. In this case, software that is capable of identifying required changes and (at least partially) evolving itself would become more valuable. This combination of evolutionary stimuli raises difficult problems for ECS.

- How might a software system evolve its mission goals in response to changing environmental conditions (e.g., detection of a new toxin in a watercourse (Dobson et al., 2009))?
- How can new mission requirements be incorporated into the current goals of individual agents when the complete picture of the environment is distributed across the swarm?
- How might agent redundancy and significant communication interruptions affect the risk, timeliness, and correctness of evolution?

1.3.1 Automotive Software

Automotive manufacturers and software vendors face a number of problems when developing and evolving critical systems, including

- high demand for customization, driven by market competition and end-user preferences;
- subsystems, supplied by multiple vendors that are developed using different processes and different engineering technologies;
- software applications running on these individual subsystems that are inherently incompatible, due to the diversity of vendors' development cultures, and
- unreliability of these software applications, in turn resulting in high costs incurred by manufacturers for vehicle recalls and maintenance.

The current state of practice in developing embedded-infrastructure software uses component-based architectures such as those promoted by the AUTOSAR initiative, along with static code analysis tools to capture design flaws. However, these still fail to address the dynamic and real-time aspects of the infrastructure software (Shokry & Hinchey, 2009).

Automotive software raises some interesting questions about the management of software evolution:

- How can upgrades to automotive software be enabled without the need to return to a dealership?
- Can we undertake these tasks without making any safety/reliability sacrifices?

1.4. Healthcare Management

We envisage a situation where many different providers keep health records electronically across multiple jurisdictions using different formats and different access protocols. We would like this information to be available, for example, to paramedics in Germany treating someone from USA in an accident situation.
Such a scenario raises questions such as

- How can we be sure that the information is available, up-to-date and accurate and that information regarding conditions and medication is available immediately to medical professionals, yet kept confidential during transmission?
- If changes in legislation or policy in (at least) one jurisdiction force evolution of key components within the system, can we ensure that these requirements will still be satisfied, along with the new ones?