2010 17th IEEE International Conference and Workshops on Engineering of Computer-Based Systems

Evolving Critical Systems: a Research Agenda for Computer-Based Systems

Mike Hinchey and Lorcan Coyle
Lero—the Irish Software Engineering Research Centre
University of Limerick, Ireland
Email: mike.hinchey@lero.ie, lorcan.coyle@lero.ie

Abstract

Increasingly software can be considered to be critical, due to the business or other functionality which it supports. Upgrades or changes to such software are expensive and risky, primarily because the software has not been designed and built for ease of change. Expertise, tools and methodologies which support the design and implementation of software systems that evolve without risk (of failure or loss of quality) are essential. We address a research agenda for building software in computer-based systems that (a) is highly reliable and (b) retains this reliability as it evolves, either over time or at run-time.

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 [6]. Many complex systems, from medical systems to transport and telecommunication control infrastructures, depend on reliable, high-quality software.

The software in computer-based systems frequently need to be modified in response to changes in system requirements and in their operational environment. Such modification may involve the addition of new functionality, the adjustment of existing functions, or the wholesale replacement of entire sub-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 [10]. As the amount and complexity of software increases, a requirement has emerged for critical software, which can evolve without loss of quality — which is engineered from the start to be easily changed, extended and reconfigured, while retaining its security, its performance, its reliability and predictability, and the prior effort that went into these.

With enough effort and expertise it should be possible to evolve any large software system without loss of quality or risk of failure. In order to keep costs reasonable a certain amount of risk might be tolerated, but if the system is critical, and faults cannot be tolerated, the cost of evolution will be high given today's resources. In pathological cases, where the cost of an evolution exceeds the business value of the system by orders of magnitude, it would be unwise to attempt change at all.

The need is becoming evident for a research community that focuses on developing and refining a body of scientific knowledge that supports the development and maintenance of Evolving Critical Systems (ECS). This community must concentrate its efforts on the development and refinement of the necessary techniques, methodologies and tools to support the design, implementation, and maintenance of critical software systems that evolve without risk (of failure or loss of quality). We define ECS more clearly, we outline a research agenda, describe a set of application scenarios, and describe the Plan-Evolve-Assess (PEA+T) cycle for ECS, aligning with a taxonomy of research themes for ECS.

2 Evolving Critical Systems

In order to understand the challenges of ECS it is important to investigate the terms Evolving Systems and Critical Systems:

Evolving (software) 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;
- require the extension of an existing system to include new functional requirements;
- evolve as a result of a focused and intentional change in organization and architecture to exploit newer techniques believed to be beneficial, e.g., service-oriented architectures as a means of restructuring for reuse,

978-0-7695-4005-4/10 $26.00 © 2010 IEEE
DOI 10.1109/ECBS.2010.56

efficiency and flexibility, and a move towards multi-core-based implementations that successfully exploit the processing power of such an architecture;

- require that the system adapt and evolve at run-time in order to react to changes in the environment or to meet necessary constraints on the system that were not previously satisfied and possibly not previously known, and which may not be achievable without a complete "re-think" of the use of technology.

Table 1. Cost of one hour of downtime from 1996/2000[3]. Costs are likely to be significantly higher today.

Companies	cost per 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

The second key characteristic of the ECS research focus is the critical nature of the systems under investigation. Critical systems are systems where failure or malfunction will lead to significant negative consequences. These systems may have strict requirements for security and safety, to protect the user or others. 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.

3 An ECS Research Agenda

An ECS Research Agenda addresses several core research topics in the evolving critical systems field. The central research topic is *designing, implementing, and maintaining critical software systems that*

a) are highly reliable, and

b) retain this reliability as they evolve without incurring prohibitive costs.

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.

An ECS must be described in a manner that enables the developer to understand the necessary functionality of the system and which is expressed in a clear and precise way and yet which offers sufficient flexibility to follow the processes and practices within the organisation or necessitated by the development process. While the need for complete understanding and appropriate processes is in theory true of all computer-based systems, it is particularly great in systems that must evolve, since we must understand the implications of change for other parts of the system and the negative effects that can ensue.

An ECS must be structured in a way that change can be controlled and clear, with fixed core functionality and then features that may be changed, adapt, and even be deleted in order to support the necessary evolution. The architecture of the system must be well understood; it may be the basis for future decisions on changes to be made as part of the evolution process, particularly where the system evolves at run-time.

Criticality is an important component of ECS. 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 [5, 10]. Determining that quality and reliability are not impaired involves continual overview of the development and evolutionary process, ensuring that policies and constraints are met, collecting and recording data and evidence and com-

putation of a range of reliability measures at various points in time and the appropriate analysis thereof.

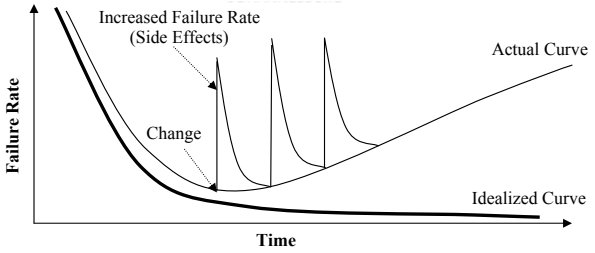

Figure 1. As complex software evolves new defects are introduced, causing the failure rate to spike and increase over time (from [9]).

A research agenda and research community for ECS must be concerned with the creation of the knowledge, the processes, the protocols, and the tools that support the development of critical systems that evolve in a way that is cost-effective, well managed, timely, and ensures that previous levels of quality and reliability are maintained, or in some cases (particularly in the case of adaptive systems) even improved.

4 ECS Scenarios

ECS is important for all application domains, but most of all where large and important computer-based 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. 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. We have chosen ECS scenarios from four different domains of application and outlined them here to highlight its importance.

4.1 Parallel and Multicore Computing

Many developers are faced with the opportunity of increased processing power brought about by the advent of multicore computing. However, quite often, 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 [12]. Writing directly for multicore or trying to port 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. Ideally software should be evolved (automatically if possible) to best utilise the currently available resources. Could critical software be designed to evolve itself in reaction to the dynamic availability of computing resources? How might an ECS 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 ECS in parallel computing?

4.2 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. Our online infrastructures have been found to be very lacking in security and it is believed that a number of compromises to their security have been made by criminal, mercenary, and state-sponsored hackers in recent years [7]. 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). Can we ensure the safety of a networked infrastructure, while maintaining the integrity of data and ensuring responsiveness and availability even in the presence of malicious onslaughts from unknown (possibly automated) sources? How can essential upgrades be made to these infrastructures in such a way that security loopholes are fixed and upgrades made without taking systems out of operation for any period of time (even a second)? What are the critical threats of software based on large networks? How might an ECS predict and prevent the onset of a possible global failure? How can the software evolve without entering into a potentially unstable behaviour?

4.3 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. The swarm can be left in place over long periods of time, during which the mission might be expected to change. It will not always be appropriate or possible for a human to be 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 [13], but more mundane examples

abound, where the software may be too complex or 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 the mission goals in response to changing environmental conditions? How can new mission requirements be incorporated into individual agents' current goals 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?

4.4 Automotive Software

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

- 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, due to the high pressure on vendors for fast time-to-market delivery, 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 the AUTOSAR initiative, along with static code analysis tools to capture design flaws. However, these still fail to address the dynamic and realtime aspects of the infrastructure software. Advances in model-based design technology indicate that embedded software developers can expect more tool support for the whole electronic control unit software spectrum [11].

From the perspective of ECS, how can we enable upgrades to automotive software without the need to return to a dealership? Could updates to software be made when a driver passes a tollbooth? Could that update be installed, safely, while the car is in motion? Can we offer services and possible upgrades to drivers as an option that can be accepted as they pass certain points? Can we undertake these tasks without making any safety/reliability sacrifices?

5 PEA+T

In order to arrange the core research topics under ECS we divide them among three stages that make up a cycle for evolving critical systems: Plan, Evolve, and Assess (PEA) — shown in Figure 2. Tooling cross-cuts the PEA cycle and refers to tool support for ECS:

- **Plan:** All activities/considerations that can occur before the system is next changed, including deciding on the processes to use, requirements engineering, and risk management.
- **Evolve:** All activities/considerations that can occur as the system is changing, including impact analysis and reengineering.
- **Assess:** All activities/considerations that can occur after the most recent change has been made to the system, including incremental verification and validation, and ensuring software quality attributes are maintained.
- **Tooling:** Consideration of tools to assist in the evolution cycle. Many of these support the whole cycle, e.g., visualisation, version control, others support particular aspects, e.g., refactoring tools and traceability model databases.

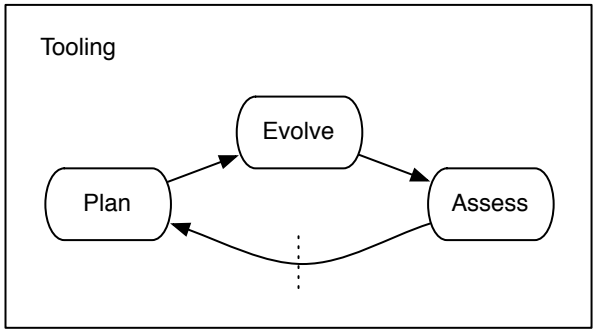

Figure 2. The PEA Cycle for Evolving Critical Systems

The Plan-Evolve-Assess (PEA) cycle can refer to a whole system evolution process or just a single change made as part of a larger evolution process. The PEA cycle has much in common with IBM's Monitor, Analyze, Plan, Execute (MAPE) model for Autonomic Computing [4], and Dobson et al.'s Autonomics Feedback Loop [1]. The MAPE model describes a control loop with four stages, the Monitor stage, which determines the environmental conditions that might drive a change; the Analyze stage, which allows the

autonomic manager to learn about and predict situations; the Plan stage, which provides the mechanisms that structure the action needed to change; and the Execute stage, which executes the changes. MAPE's Monitor, Analyze, and Plan stages correspond to the Plan stage in the PEA cycle, and its Execute stage corresponds to the Evolve stage in the PEA cycle. The PEA Assess stage does not have an analogue in the MAPE model, as MAPE serves primarily as a regulator, although there is scope for assessment during its Analyze stage.

An evolving critical system should not be activated until the PEA cycle has fully completed, i.e., until the Assess stage is completed. After an evolution cycle has completed, it may be desirable to undertake another cycle, e.g., to remove software faults that were tolerable during the current cycle. In any case, the outputs of the Assess stage will likely be used as inputs to help drive the Plan stage of the next cycle. Continuously evolving autonomic or adaptive systems can be expected to cycle continuously as a control loop [1] and so the separation between the Assess and Plan stages may become blurred.

In order to describe the scope of research required to develop a body of knowledge for ECS we have defined a taxonomy of research topics under the PEA+T headings, summarized in Figure 3.

6 Conclusions

Critical computer-based systems are ever more important in our lives, but as they age they will be in need of increasing amounts of effort to evolve them to remain useful [5]. Much work needs to be done to help ease this burden if the community is not to be overwhelmed (cf. Mens et al. [8] for a more detailed list of challenges for software evolution). We propose *Evolving Critical Systems* as an area for research to tackle the challenge and outline four scenarios to highlight some of the important research questions that should be asked of the community. Given that software evolution can be seen as a compromise between cost and risk, the most pressing question to ask is which processes, techniques and tools are most cost-effective for evolving critical systems.

7 Acknowledgments

The authors would like to thanks the invaluable comments, suggestions, and feedback from their colleagues in Lero.

This work was supported, in part, by Science Foundation Ireland grant 03/CE2/I303_1 to Lero—the Irish Software Engineering Research Centre (`www.lero.ie`).

References

[1] 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.

[2] M. Hinchey and L. Coyle. Evolving critical systems. Technical Report Lero-TR-2009-00, Lero—The Irish Software Engineering Research Centre, 2009.

[3] 2000. 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.

[4] 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.

[5] 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.

[6] M. M. Lehman and J. C. Fernández-Ramil. *Software Evolution and Feedback: Theory and Practice*, chapter Software Evolution. John Wiley & Sons, 2006.

[7] Mcafee virtual criminology report, 2008. Available online: `http://resources.mcafee.com/content/NAMcAfeeCriminologyReport`.

[8] 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.

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

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

[11] H. Shokry and M. Hinchey. Model-based verification of embedded software. *Computer*, 42(4):53–59, 2009.

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

[13] 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.

<table>
<tr><th colspan="5">Plan</th></tr>
<tr><th>Requirements Engineering</th><th>Software Development Processes</th><th>Architecture Design</th><th>Proscriptive Models</th><th>Risk Management</th></tr>
<tr><td>Evolvability Requirements</td><td rowspan="2">Waterfall Models</td><td>Encapsulation of Criticality</td><td rowspan="2">Specification</td><td rowspan="2">Fault/Error Analysis</td></tr>
<tr><td rowspan="2">Requirements Elicitation</td><td rowspan="2">Spiral Models</td><td rowspan="2">MDA & MDE</td></tr>
<tr><td rowspan="2">Requirements Monitoring</td><td rowspan="2">Formal Methods</td><td rowspan="2">Context Awareness</td></tr>
<tr><td rowspan="2">Agile Methods</td><td>Service Oriented Architectures</td></tr>
<tr><td rowspan="2">Features</td><td rowspan="2">Aspect Oriented Development</td><td rowspan="2">Predictive Estimation</td></tr>
<tr><td rowspan="2">Structural Design Methods</td></tr>
<tr><td>Validation</td><td>Software Process Improvement</td><td>Software Product Lines</td><td>Uncertainty Analysis</td></tr>
</table>

<table>
<tr><th colspan="3">Evolve</th></tr>
<tr><th>Reengineering</th><th>Impact Analysis</th><th>Fault/Error Treatment</th></tr>
<tr><td>Reverse Engineering</td><td rowspan="2">Traceability Analysis</td><td>Passivation</td></tr>
<tr><td>Refactoring</td><td rowspan="2">Reconfiguration</td></tr>
<tr><td>Redocumentation</td><td rowspan="2">Effort Estimation</td></tr>
<tr><td>Design Recovery</td><td rowspan="2">Correction</td></tr>
<tr><td>Automatic Code Generation</td><td rowspan="2">Feature Interaction</td></tr>
<tr><td></td><td>Error Processing</td></tr>
</table>

<table>
<tr><th colspan="4">Assess</th></tr>
<tr><th>Incremental Verification and Validation</th><th>Software Quality Attributes</th><th>Architecture Quality Metrics</th><th>Software Process Improvement</th></tr>
<tr><td rowspan="2">Requirements</td><td>Performance</td><td>Maintainability</td><td rowspan="2">CMMI</td></tr>
<tr><td>Dependability</td><td>Encapsulation of Criticality</td></tr>
<tr><td rowspan="2">Formal Verification</td><td>Security</td><td>Composition of Patterns</td><td rowspan="2">Software Quality Attributes</td></tr>
<tr><td>Safety</td><td>Separation of Modules</td></tr>
<tr><td rowspan="2">Testing</td><td>Mode Performance Analysis</td><td>Cohesion</td><td rowspan="2">Learning to Evolve</td></tr>
<tr><td rowspan="2">Formalisms and Metrics for Uncertainty</td><td>Quality in Uncertainty</td><td rowspan="2">Coupling</td></tr>
<tr><td>Quality Attribute Aggregation</td></tr>
</table>

Figure 3. A Research Taxonomy for Evolving Critical Systems (a more detailed breakdown is given in [2]).